Jerônimo C. Pellegrini

Programação Funcional e Concorrente com Scheme

notas de aula

UFABC - Universidade Federal do ABC
Santo André
http://alepho.info/jp
Versão 134

Escrito em LaTeX.

Versão Preliminar

[15 de novembro de 2018 at 14:30]

UMA NOTA AO LEITOR

Comecei a escrever estas notas de aula porque não havia texto em Português que apresentasse a linguagem Scheme[1], especialmente textos que tragam tanto aplicações práticas como exemplos de construção de características de linguagens (úteis em um curso de Projeto de Linguagens ou Paradigmas de Programação, por exemplo).

Concordo parcialmente com Shriram Krishnamurti [Krio8], que questiona o ensino de "paradigmas" de programação: "'Paradigmas' de programação são um legado moribundo e tedioso de uma era já passada. Projetistas de linguagens modernas não os respeitam, então porque nossos cursos aderem de modo escravocrata a eles?" A ênfase deste texto em construção de características de linguagens é uma abordagem alternativa aos cursos de "Conceitos de Lingugagens de Programação" onde tais características são descritas teoricamente. A construção destas caracterśiticas diretamente pelo aluno foi pedagogicamente muito eficaz nas ocasiões em que ministrei a disciplina de Paradigmas de Programação.

Este texto, antes de mais nada, ilustra o fato declarado no parágrafo de abertura da Introdução da especificação de Scheme[2]:

> Linguagens de programação devem ser projetadas não empilhando recursos uns sobre os outros, mas removendo as fraquezas que fazem recursos adicionais parecerem necessários. Scheme mostra que uma quantidade muito pequena de regras para formar expressões, sem restrições para seu uso, bastam para formar uma linguagem de programação prática e eficiente que é flexível o suficiente para suportar a maioria dos paradigmas de programação em uso hoje.

Trata-se, no entanto, de um rascunho – o estilo é inconsistente e há diversos trechos faltando (muitas seções estão em branco, e demonstrações informais de corretude não aparecem onde são necessárias). Em particular, ainda não há nada a respeito de verificação e inferência de tipos. Além disso, há seções e capítulos ainda não desenvolvidos ou que sofrerão grandes mudanças: o próximo padrão de Scheme (R^7RS) está sendo desenvolvido

1 E mesmo em Inglês, há poucos textos que abordam toda a extensão da linguagem

2 Tradução livre. O original é *"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today."*

Versão Preliminar

por dois grupos de trabalho – o WG1, encarregado de criar um núcleo simples, minimalista e limpo, ideal para fins educacionais e para demonstrações conceituais e o WG2, que deverá criar a parte "pesada" da linguagem, tornando-a uma ferramenta prática de uso geral. O R^{7}RS provavelmente será anunciado em 2012, trazendo muitos elementos novos. Aos poucos tenho incorporado partes do novo padrão, à medida que são votadas pelos grupos de trabalho – mas estas decisões não são finais, e o novo padrão ainda precisa ser ratificado.

Alguns novos procedimentos já constam no texto (e no Apêndice com o resumo da linguagem). Partes do texto que certamente serão profundamente influenciadas incluem:

- *Concorrência:* o texto usa o semipadrão SRFI-18, mas a API para concorrência no R^{7}RS poderá ser diferente. Neste texto são desenvolvidas *mailboxes*, semáforos e outras abstrações que poderão ser padronizadas de forma diferente – e neste caso a API usada no texto será modificada para refletir o padrão;

- *Macros não higiênicas:* o grupo de trabalho dois (WG2) votou pode deliberar no futuro sobre a inclusão de macros com renomeação explícita. Estas são descritas apenas superficialmente neste texto;

- *Rede:* implementações de Scheme diferem muito na API para comunicação via TCP e UDP. O WG2 votou por deliberar sobre isto. O pouco que há neste texto *não* é portável (funciona apenas no Chicken Scheme);

- *Outros:* há outros itens pendentes de definição pelos grupos de trabalho, que poderão ser inseridos ou modificados: registros, números aleatóreos, sistema de exceções, argumentos opcionais em procedimentos, asserções, aritmética rápida para tipos estáticos (e vetores para tipos homogêneos), operações com bits, tratamento de Unicode, API para data/tempo, dicionários, indagações ao ambiente, tratamento de diretórios em sistemas de arquivos, argumentos de linha de comando, memoização, parâmetros, casamento de padrões, operações em portas, expressões regulares, API simples para acesso a sistemas POSIX e outros ainda.

Além disso, há um Capítulo planejado que tratará de inferência de tipos.

Há trechos de programas em outras linguagens que pretendo mover para Apêndices ou tutoriais separados em outro meio, externo a este texto.

Versão Preliminar

SUMÁRIO

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Parte I.

Treinamento Básico em Scheme

Versão Preliminar

Versão Preliminar

1 ELEMENTOS BÁSICOS

1.1 PARADIGMAS DE PROGRAMAÇÃO

Há diversas maneiras de conceber o que é "programar um computador", cada uma delas usando diferentes conceitos e abstrações. O nome dado tradicionalmente a cada uma destas diferentes concepções de programação é "paradigma de programação".

No paradigma *imperativo*, a programação se dá por sequências de comandos, incluindo comandos que modificam o estado interno da memória do computador (de maneira semelhante àquela como os computadores funcionam internamente, e talvez seja este o motivo do paradigma imperativo ser o primeiro usado na prática[1]).

No paradigma *orientado a objetos*, programas são conjuntos de objetos de diferentes classes que interagem através de mensagens que ativam métodos nos objetos destino. Smalltalk [Lew95][2] é a linguagem onde o paradigma de orientação a objetos se mostra de maneira mais pura e clara.

No paradigma *lógico* programas são cláusulas lógicas representando fatos, relações e regras. Prolog [CM03] é o principal representante da família das linguagens lógicas.

Dentro do paradigma funcional, um programa é um conjunto de *funções*, semelhantes (mas não sempre) às funções em Matemática. Há diversas linguagens que podem representar o paradigma funcional, como ML, Haskell e Scheme. Estas são, no entanto, radicalmente diferentes apesar de compartilharem a noção de "programas como conjuntos de funções". Escrever programas em linguagens funcionais é, então, escrever funções. Na terminologia da programação funcional funções *recebem* argumentos e *retornam* valores. O ambiente de programação toma *expressões* e calcula seus valores aplicando funções.

Estes não são os únicos "paradigmas" possíveis, mas certamente são os mais conhecidos. Pode-se falar também de programação com vetores (em APL [Rei90]), ou do modelo Actor [Agh85] para programação concorrente como paradigmas de programação, e há a programação com pilha em Forth [Bro04] (e mais recentemente Factor [PEG10]) ou ainda de programação com tabelas em Lua [Ier06] (embora Lua incorpore também elementos de outros paradigmas).

1 A primeira linguagem de programação de alto nível desenvolvida foi o FORTRAN, que é essencialmente imperativo; antes dele usava-se linguagem de máquina – uma forma também imperativa de programação

2 E não Java ou C++, que são linguagens onde paradigmas se misturam.

Versão Preliminar

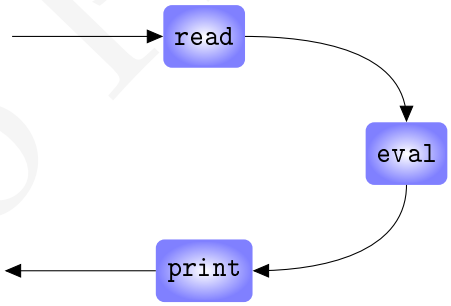
1.1.1 Paradigmas?

Embora esta visão seja amplamente difundida e normalmente reforçada por livros e cursos, Shriram Krishnamurti a questiona [Kri08]: "'Paradigmas' de programação são um legado moribundo e tedioso de uma era já passada. Projetistas de linguagens modernas não os respeitam, então porque nossos cursos aderem de modo escravocrata a eles?"[3] Ainda assim, é interessante notar que algumas das linguagens mais elegantes e simples espelham de maneira bastante direta algum paradigma.

Este texto aborda o paradigma *funcional* de programação usando a linguagem Scheme [Dyb09; AS96; Fel+03; MR07; Kri07], e ilustra a construção das características importantes de linguagens de programação, inclusive algumas das características marcantes de linguagens orientadas a objetos e lógicas.

1.2 O AMBIENTE DE PROGRAMAÇÃO SCHEME

Muitas linguagens oferecem um dispositivo chamado "REPL" (Read-Eval-Print Loop). O REPL é um programa que lê expressões ou trechos de programa, "avalia" (ou "executa") e mostra o resultado. Python, Ruby, Common Lisp, Haskell e a vasta maioria das implementações de Scheme oferecem um REPL. Na prática, a experiência de usar o REPL é semelhante à de interagir com um computador usando linha de comando em um terminal[4].



A maneira de iniciar o interpretador Scheme e começar a interagir com o REPL depende de qual implementação de Scheme é usada.

Neste texto, a entrada do usuário para o REPL será sempre em negrito; a resposta do REPL será sempre em itálico:

3 No original, *"Programming language 'paradigms' are a moribund and tedious legacy of a bygone age. Modern language designers pay them no respect, so why do our courses slavishly adhere to them?"*

4 O leitor perceberá que o ciclo de desenvolvimento em Lisp e outras linguagens ditas "dinâmicas" é diferente do tradicional "editar-compilar-executar", típico de C, C++, C#, e Java.

Versão Preliminar

```
10
```

A resposta é:

```
10
```

No exemplo acima, o primeiro "10" é a expressão enviada pelo usuário; o segundo "10" foi a resposta do ambiente Scheme. Normalmente um interpretador Scheme apresentará um símbolo, como >, ao aguardar pela entrada do usuário. Omitiremos este símbolo neste texto.

Se o REPL receber mais números ou strings, ele os "avalia" e retorna o resultado, que para números e strings é o próprio número ou a própria string:

```
10
10
2.5
2.5
"Uma string"
"Uma string"
```

```
0
0
```

O REPL aceitará e avaliará um trecho de programa Scheme também. A expressão `(+ 3 4 5)` em Scheme significa[5] "chame o procedimento `+` com argumentos `3` `4` e `5`":

```
(+ 3 4 5)
12
```

O REPL tomou a expressão digitada e enviou ao interpretador, que devolveu o resultado `12` ao REPL, que por sua vez o mostrou na tela. Trechos de programas Scheme são chamados de *formas*.

A expressão `(+ 3 4 5)` é uma *aplicação de função*: "+" é a função que soma números; "3", "4" e "5" são seus argumentos. Na notação usual a expressão seria `3+4+5` ou `soma(3, 4, 5)`.

Embora a notação prefixa pareça estranha em um primeiro contato, ela permite tratar com regularidade todo tipo de função. A notação infixa é normalmente usada apenas para as quatro operações aritméticas, como em $a + b * c - 4$; no entanto, ao usar funções como

5 Todas as variantes da linguagem Lisp usam notação prefixa.

Versão Preliminar

seno, tangente, logaritmo e outras, usa-se uma notação diferente, que é muito próxima da prefixa usada em Lisp:

```
sen(x)     (sen x)
tan(x)     (tan x)
log(n)     (log n)
f(x, y, z) (f x y z)
```

As duas diferenças são a ausência de vírgulas em Lisp e o nome da função, que em Lisp fica dentro dos parênteses.

Além do tratamento uniforme, a notação prefixa elimina a necessidade de definição de precedência de operadores: usando notação infixa, $a + b/c \neq (a + b)/c$. Usando notação prefixa, a expressão `(/ (+ a b) c)` só pode ser interpretada de uma única forma: "divida a soma de a e b por c"[6].

Sequências de objetos Scheme delimitadas por parênteses são chamadas de *listas*. Quando o interpretador recebe uma lista, tenta interpretá-la como a aplicação de um procedimento. Por exemplo, se uma lista tem os objetos \, 4 e 5, será interpretada como a aplicação do procedimento \ (divisão) aos parâmetros 4 e 5.

```
(/ 4 5)
0.8
```

1.2.1 Tipos de dados

Em Scheme objetos podem ser de vários tipos:

- Caracteres, strings, números e booleanos são usuais na maioria das linguagens de programação.
- Pares, usados para implementar listas, são tipos nativos em Scheme
- Vetores são tipos nativos em Scheme.
- Vetores de bytes (sequências de bits) são tipos nativos.
- Procedimentos são tipos nativos.
- Portas de entrada e saída são tipos nativos.
- O objeto fim-de-arquivo é um único elemento de um tip nativo.

6 Ler as funções e procedimentos usando verbos como "divida:" ou "some:" ajuda a ambientar-se com a notação prefixa.

Versão Preliminar

- A lista vazia é o único elemento de seu tipo.
- Símbolos (nomes de variáveis) são tipos nativos.

O padrão Scheme define que cada objeto deve ser de exatamente um destes tipos, e há procedimentos que verificam os tipos desses objetos:

```
boolean?   bytevector?
char?      eof-object?
null?      number?
pair?      port?
procedure? string?
symbol?    vector?
```

Por exemplo,

```
(string? "Something rottens in the state of Denmark")
#t
(number? 10)
#t
(boolean? #f)
#t
(string? 10)
#f
```

Além desses predicados, há outros que testam subconjuntos desses tipos:

```
(positive? -1)
#f
(integer? -2)
#t
(rational? 2/3)
#t
```

Versão Preliminar

1.2.2 Símbolos e Variáveis

Variáveis são posições da memória do computador onde nossos programas armazenam valores. Para que possamos descrever o que queremos, precisamos nos referir a estas posições, dando-lhes nomes. Em Scheme, um nome de variável é um *símbolo*[7].

É importante notar que um símbolo *não* é uma variável. Ele é um *nome* que pode ser vinculado a uma variável, e este nome pode ser manipulado como se fosse um objeto como qualquer outro.

Um símbolo também *não* é uma cadeia de caracteres. Um interpretador Scheme, ao ler uma sequência de caracteres entre aspas, reconecerá que se trata de uma string:

```
(string? "Oh brave new world")
#t
```

Mas uma sequência de caracteres, sem aspas, não é uma string. É um *nome*, ou *símbolo*:

```
(string? (quote abcde))
#f
(symbol? (quote abcde))
#t
```

É possível transformar símbolos em strings e vice-versa, com os procedimentos `string->symbol` e `symbol->string`.

```
(string->symbol "brave")
brave
(symbol->string (quote world))
"world"
```

Exemplos de símbolos são `a`, `b`, `uma-palavra`, `+`, `-`, `>`. Em Scheme o nome de uma variável pode ser qualquer sequência de caracteres iniciando com uma letra ou um caracter especial dentre `! $ % & * + - . / : < = > ? @ ^ _ ~`. Os caracteres não iniciais podem ser também dígitos. Símbolos em Scheme não são apenas formados por caracteres alfanuméricos; também incluem os símbolos `+`, `/`, `*`, `-` e vários outros. No entanto, evitaremos criar símbolos iniciando em `+d` ou `-d`, onde `d` é um dígito, que podem ser confundidos com números com sinal.

Quando o ambiente Scheme avalia um símbolo, imediatamente tentará retornar o *valor* da variável com aquele nome. Se não houver valor, um erro ocorrerá.

7 Um símbolo é semelhante a um nome de variável em outras linguagens de programação, exceto que em Lisp os símbolos são objetos de primeira classe (podem ser usados como valor de variável, podem ser passados como parâmetro e retornados por funções).

Versão Preliminar

```
um-nome-qualquer
Error: unbound variable: um-nome-qualquer
```

Para que o Scheme retorne o símbolo e não seu valor, é necessário "citá-lo":

```
(quote um-simbolo)
um-simbolo
```

Uma lista iniciando com `quote` é uma *forma especial* em Scheme; formas especiais tem regras específicas para avaliação. A forma especial `quote` retorna seu argumento, *sem que ele seja avaliado*.

Para criar uma variável dando-lhe um nome e valor há a forma especial `define`:

```
(define um-simbolo 1000)
(quote um-simbolo)
um-simbolo
um-simbolo
1000
```

Se `define` não fosse forma especial (ou seja, se fosse procedimento), o interpretador tentaria avaliar cada elemento da lista, e `(define um-simbolo 1000)` resultaria em erro, porque ainda não há vínculo para `um-simbolo`.

Símbolos são objetos de primeira classe, como números e strings; por isso podemos armazenar um símbolo em uma variável:

```
(define var (quote meu-simbolo))
var
meu-simbolo
```

Qualquer expressão pode ser definida como valor de uma variável:

```
(define pi 3.1415926536)
(define phi 1.618033987)
(define um-valor (* 2 (/ pi phi)))
um-valor
3.88322208153963
```

O valor de `(* 2 (/ pi (phi)))` foi calculado *antes* de ser associado ao nome `um-valor`.

Uma forma curta para `(quote x)` é `'x`:

```
'este-simbolo-nao-sera-avaliado
este-simbolo-nao-sera-avaliado
```

Versão Preliminar

1.3 ABSTRAÇÃO DE PROCESSOS COM FUNÇÕES (PROCEDIMENTOS)

O mecanismo oferecido por quase todas as linguagens de programação para a abstração de processos é a *função*.

Em Scheme, funções são chamadas de *procedimentos*[8].

Na Seção 1.2.2 usamos o símbolo `*` para multiplicar ϕ por 2 simplesmente porque *o valor deste símbolo é o procedimento de multiplicação*:

```
/
#<procedure C_divide>
+
#<procedure C_plus>
*
#<procedure C_times>
```

A resposta do REPL, "`#<procedure ...>`", é uma representação usada por uma implementação de Scheme para estes procedimentos. Esta representação normalmente varia de uma implementação a outra.

Podemos definir outro símbolo para realizar a mesma operação feita pelo procedimento `+`:

```
(define soma +)
(soma 4 5 6)
15
```

Também é possível determinar que o símbolo `+` não tenha mais o valor "procedimento soma", mas que ao invés disso tenha como valor o procedimento subtração:

```
(define soma +)
soma
#<procedure C_plus>
(define + -)
(+ 10 2)
8
```

Como o procedimento de soma original foi guardado na variável `soma`, é possível devolver seu valor à variável `+`:

8 Em outras linguagens da família Lisp, funções são sempre chamadas de "funções"; apenas na tradição de Scheme a nomenclatura é diferente.

Versão Preliminar

```
(define + soma)
(+ 10 2)
12
```

Embora o exemplo anterior pareça um tanto inusitado e aparentemente sem propósito, a redefinição dos valores de procedimentos padrão como `*` e `+` é útil em muitas situações: é possível definir um novo procedimento `+` que some, além dos tipos já existentes, números complexos, intervalos, vetores ou quaisquer outros objetos para os quais a soma possa ser definida. Também é possível trocar o procedimento padrão por outro mais eficiente.

A respeito dos procedimentos `*` e `+`, é interessante observar que podem ser usados sem argumentos:

```
(+)
0
(*)
1
```

O mesmo não é possível com `-` e `/`.

1.3.1 Definindo novos procedimentos

Os procedimentos usados nas seções anteriores a esta existem em qualquer ambiente Scheme, e são chamados de *procedimentos primitivos*. Além destes, podemos criar novos procedimentos usando os procedimentos primitivos; estes são chamados de *procedimentos compostos*.

Em Scheme procedimentos compostos podem ser criados usando expressões `lambda`:

```
(lambda (arg1 arg2 ...)
  ;; corpo do procedimento
  forma1
  forma2
  ...)
```

Quando uma forma `lambda` é avaliada, um procedimento é criado (mas não aplicado – trata-se apenas de sua descrição), e retornado. O exemplo abaixo é um procedimento que recebe dois argumentos (`a` e `b`) e calcula $(ab)^{-1}$:

```
(lambda (a b) (/ 1 (* a b)))
```


Versão Preliminar

Quando este procedimento é aplicado, a forma é avaliada e seu valor é retornado. A *aplicação* deste procedimento consiste de sua descrição entre parênteses – o avaliador Scheme o aplicará porque é o primeiro elemento de uma lista:

```
( (lambda (a b) (/ 1 (* a b))) 10 15 )
1/150
```

O que enviamos ao REPL foi uma lista:

- O primeiro elemento da lista é `(lambda (a b) (/ 1 (* a b)))`, um procedimento;
- Os dois outros elementos são 10 e 15, e serão usados como argumentos para o procedimento.

Do ponto de vista puramente funcional, faz sentido ler a forma `(lambda (a b) (/ 1 (* a b)))` como "troque `a` e `b` por `(/ 1 (* a b))`".

A lista de argumentos da forma `lambda` pode ser vazia. O procedimento a seguir é uma função constante (sempre retorna o número 42):

```
(lambda () 42)
#<procedure (?)>
( (lambda () 42) )
42
```

1.3.2 Exemplo: juros compostos

A seguinte expressão é um procedimento que realiza cálculo de juros compostos. O valor total é $v(1+i)^t$, onde v é o valor do principal, i é a taxa de juros e t é o número de parcelas.

```
(lambda (v i t)
  (* v (expt (+ 1.0 i) t)))
```

O procedimento `expt` realiza exponenciação.

A definição de um procedimento não é útil por si mesma. Ele pode ser aplicado, como mostra o exemplo a seguir:

Versão Preliminar

```
((lambda (v i t)
   (* v (expt (+ 1.0 i) t)))
      1000 0.01 12)
```
1126.82503013197

No exemplo acima, o REPL recebeu uma forma cujos elementos são:

- Primeiro: `(lambda (v i t) (* v (expt (+ 1.0 i) t))`
- Segundo: 1000
- Terceiro: 0.01
- Quarto: 12

O primeiro elemento foi avaliado e o resultado é um procedimento:

```
(lambda (v i t)
  (* v (expt (+ 1.0 i) t)))
```
#<procedure (? v i t)>

Em seguida, os outros elementos foram avaliados e passados como argumento para o procedimento.

Outra coisa que pode ser feita com um procedimento é defini-lo como conteúdo de uma variável (ou seja, dar a ele um nome) para que não seja necessário digitá-lo novamente mais tarde:

```
(define juros-compostos
  (lambda (v i t)
    (* v (expt (+ 1.0 i) t))))
```

Agora podemos usar este procedimento. O próximo exemplo mostra seu uso para verificar a diferença entre duas taxas de juros (2% e 1.8%):

```
(juros-compostos 5000 0.02 12)
```
6341.20897281273
```
(juros-compostos 5000 0.018 12)
```
6193.60265787764

Como um procedimento em Scheme é um valor qualquer (assim como números, caracteres e strings), nada impede que se faça a cópia de uma variável cujo conteúdo seja um procedimento:

Versão Preliminar

```
juros-compostos
#<procedure (juros-compostos v i t)>
(define jc juros-compostos)
(jc 9000 0.03 24)
18295.1469581436
```

Este último exemplo mostra também que uma taxa de juros de 3% ao mês é suficiente para dobrar o valor de uma dívida em dois anos.

1.3.3 Primeiro modelo de avaliação para expressões

Para determinar como um interpretador Scheme tratará uma aplicação de procedimento composto usamos um *modelo de avaliação*.

- Se a expressão é uma constante, o valor retornado será a própria constante;
- Se a expressão é um símbolo, e este símbolo estiver associado a uma variável, o valor da variável será retornado;
- Se a expressão é uma lista, há dois casos:
  - Se o primeiro elemento da lista for o nome de uma forma especial (por exemplo, `lambda`, `quote` ou `define`), esta expressão será tratada de maneira específica para cada forma especial (`lambda` cria um procedimento, `define` vincula símbolos a variáveis, `quote` retorna seu argumento sem que ele seja avaliado);
  - Em outros casos, o interpretador avaliará cada elemento da lista. Depois disso, verificará se o primeiro elemento é um procedimento. Se não for, o usuário cometeu um erro. Se o primeiro elemento for um procedimento, ele será aplicado com os outros como argumentos. Para avaliar a aplicação de um procedimento composto, o ambiente Scheme avaliará o *corpo* do procedimento, trocando cada parâmetro formal pelo parâmetro real.

Este modelo de avaliação é útil para compreender o processo de aplicação de procedimentos, mas não é necessariamente a maneira como ambientes Scheme realmente fazem a avaliação de formas.

Como exemplo, avaliaremos `(juros-compostos valor 0.02 meses)`, presumindo que `valor`=500 e `meses`=10:

```
(juros-compostos valor 0.02 meses)
```


Versão Preliminar

Todos os argumentos são avaliados. `valor` e `meses` são trocados por seus valores, 500 e 10; o número 0.02 é avaliado e o resultado é ele mesmo; já `juros-compostos` resulta em um procedimento quando avaliado:

```
( (λ (v i t) (* v (expt (+ 1.0 i) t))) 500 0.02 10 )
```

Agora, como o primeiro elemento da lista é um procedimento, trocamos toda a lista pelo corpo do procedimento, mas trocando seus parâmetros (v, i, t) pelos valores que seguem na lista (v $\rightarrow$ 500, i $\rightarrow$ 0.02, t $\rightarrow$ 10):

```
(* 500 (expt (+ 1.0 0.02) 10))
```

Não há mais procedimentos compostos. Aplicamos então os procedimentos primitivos – começamos por `+ 1.0 0.02`, depois a exponenciação e a multiplicação:

```
(* 500 (expt 1.02 10))

(* 500 1.21899441999476)

609.497209997379
```

Este é o valor da expressão após ter sido avaliada.

1.3.3.1 *Transparência referencial*

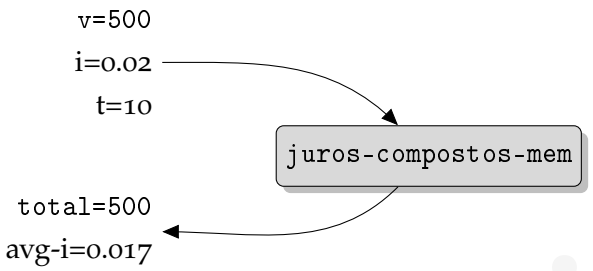
Enquanto usarmos este modelo de avaliação, um procedimento sempre retornará o mesmo valor quando chamado com os mesmos argumentos. Esta propriedade, que chamamos de *transparência referencial*, é de importância fundamental em programação funcional: ela garante que podemos trocar procedimentos por outros equivalentes. Podemos usar o procedimento `juros-compostos` em qualquer lugar, e o resultado dependerá *apenas* dos valores passados para seus argumentos.

Poderíamos ter construído o procedimento `juros-compostos` com apenas dois argumentos, `valor` e `meses`. A taxa de juros seria lida de um banco de dados. Mas neste caso, o resultado da chamada `(juros compostos 200 12)` não dependeria apenas dos valores dos argumentos (neste caso 200 e 12), mas também do valor da taxa de juros no banco de dados.

Esta não é a única forma de quebrar transparência referencial. Presuma que tenhamos desenvolvido um procedimento `juros-compostos-mem`, que calcula juros compostos mas que tem "memória": ele retorna o valor pedido pelo usuário, e também retorna a média das taxas de juros usadas nas chamadas ao procedimento (é possível um procedimento retornar mais de um valor – isso será tratado em outros Capítulos).

Versão Preliminar

Por exemplo, suponha que o procedimento tenha sido chamado com taxas de juros iguais a 0.013 (uma vez) e 0.018 (uma segunda vez). Quando for chamado uma terceira vez, agora com taxa de juros igual a 0.02, ele retornará o valor pedido e também a média das taxas de juros (a média de 0.013, 0.018 e 0.02 é 0.17):



Como a média pode mudar a cada chamada, o procedimento *não* é referencialmente transparente. Não desenvolvremos este procedimento, porque seria necessário usar diversos conceitos ainda não abordados. (O Exercício 76 no Capítulo 3 pede a construção do procedimento).

No Capítulo 3, quando tratarmos de mutação de variáveis, este modelo de avaliação não nos servirá mais e o trocaremos por outro.

1.3.4 Sem transparência referencial: `display`, `newline`

Procedimentos que realizam entrada e saída de dados não são referencialmente transparentes. Dois exemplos importantes são os procedimentos `display`, que escreve objetos na saída padrão, e `newline`, que pula uma linha (ou "escreve uma quebra de linha") na saída padrão.

```
(display ''Isto é uma string'')
Isto é uma string
display (+ 10 20))
30
```

1.3.5 Exemplo: números pseudoaleatóreos

Nosso próximo exemplo de procedimento é um gerador de números aleatóreos que nos poderá ser útil na construção de jogos (onde o ambiente do jogo deve se comportar de maneira aleatórea – por exemplo, ao misturar as cartas de um baralho).

Versão Preliminar

Há muitos métodos para a geração de números aleatóreos. Usaremos um método é de fácil compreensão e implementação. Cada número aleatóreo x_{n+1} será gerado a partir do número anterior x_n da seguinte maneira:

$$x_{n+1} = (ax_n + b) \mod m$$

onde a, b e m são parâmetros fixos do gerador. Um gerador de números aleatóreos que use este método é chamado de *gerador por congruência linear*. Este tipo de gerador é adequado para usos simples, mas não deve ser usado em Criptografia, por exemplo.

Implementaremos então um gerador de números aleatóreos com $a = 1103515245$, $b = 12345$ e $m = 2^{32}$. Em Scheme podemos usar aritmética modular com o procedimento `modulo`.

```
(define linear-congruencial
  (lambda (x a b m)
    (modulo (+ (* a x) b) m)))

(define next-random
  (lambda (x)
    (linear-congruencial x
                         1103515245
                         12345
                         (expt 2 32))))
```

O procedimento `linear-congruencial` recebe os parâmetros do gerador e um número; o procedimento `next-random` é uma barreira de abstração sobre `linear-congruencial`.

Precisaremos de um número inicial a partir do qual possamos gerar números aleatóreos. Este número é chamado de *semente*. É comum usar o relógio interno do computador, por exemplo, como semente para o gerador de números aleatóreos.

Podemos também gerar números aleatóreos de ponto flutuante, dividindo o valor obtido de `next-random` por $2^{32} - 1$, que é o maior número aleatóreo que poderemos gerar:

```
(define next-real-random
  (lambda (x)
    (/ (next-random x)
       (- (expt 2 32) 1))))
```

O procedimento `next-real-random` gerará números entre 0 e 1. Este não é o melhor método, mas nos servirá.

É fácil obter números aleatórios de ponto flutuante em outros intervalos, bastando que multipliquemos o aleatório entre 0 e 1 pelo tamanho do intervalo que quisermos. No

Versão Preliminar

entanto, a geração de números aleatóreos inteiros em um intervalo arbitrário não é tão simples.

Se quisermos um número entre 0 e k poderíamos tentar gerar um número $x \in [0, 2^{32})$ usando `next-random` e usar $x \mod k$, mas infelizmente este método não garante que a distribuição dos números gerados continua uniforme quando k não é divisor de 2^{32}. Trataremos disso na seção sobre repetições.

Pode parecer estranho que um gerador de números aleatóreos tenha a propriedade de transparência referencial – afinal de contas, queremos que o resultado deste procedimento seja imprevisível! No entanto, esta imprevisibilidade (que aliás é limitada) é garantida pela escolha da semente a partir da qual os próximos números são gerados. Esta sim, pode ser obtida a partir de forma imprevisível: para aplicações simples pode-se usar o relógio interno do computador, por exemplo. O fato de a semente determinar completamente toda a sequência de números nos permite fazer algo bastante útil: podemos reproduzir o comportamento de um programa que usa números aleatóreos, desde que guardemos a semente usada para o gerador.

Congruência linear não é um bom método para geração de números pseudo-aleatóreos; este método foi escolhido para inclusão neste texto pela sua simplicidade e facilidade de implementação. O leitor interessado em geração de números aleatóreos encontrará uma breve introdução à geração de números aleatóreos para fins de simulação no livro de Reuven [RK07] Uma exposição mais completa é dada por Gentle [Gen03]. Knuth [Knu98a] também aborda o tema. Geradores de números aleatóreos para Criptografia devem satisfazer requisitos diferentes; o livro de Stinson [Sti05] traz uma exposição básica do assunto. O Exercício 24 pede a implementação do algoritmo Blum-Micali para geração de números pseudo-aleatóreos para Criptografia (é uma tarefa mais difícil do que a implementação descrita acima).

1.4 VARIÁVEIS LOCAIS

Da mesma forma que os argumentos de um procedimento podem ser usados apenas dentro do corpo do procedimento, é possível criar variáveis temporárias acessíveis apenas dentro de um trecho de programa Scheme usando a forma especial `let`.

```
(let ((a 3)
      (b 4))
  (* a b))
```

12

```
a
```


Versão Preliminar

```
ERROR: unbound variable a
```

Tanto `a` como `b` são visíveis apenas dentro da forma `let` (dizemos que este é o *escopo* destas variáveis) – por isso não pudemos usar `a` fora do `let` e o REPL nos mostrou uma mensagem de erro.

A forma a seguir cria duas variáveis, `nome` e `sobrenome`, que são visíveis apenas dentro de seu corpo:

```
(let ((nome "Gustav")
      (sobrenome "Klimt"))
  (string-append nome " " sobrenome))
Gustav Klimt
nome
Error: unbound variable: nome
```

O procedimento `string-append` usado neste exemplo recebe várias strings e retorna a concatenação delas.

`Let` realmente precisa ser uma forma especial (ou seja, o interpretador não pode avaliar toda a lista que começa com `let`). Se não fosse assim, e `let` fosse um procedimento, o interpretador tentaria avaliar cada elemento da lista, mas:

- `((nome "Gustav") (sobrenome "Klimt))` não poderia ser avaliada porque ao avaliar `(nome "Gustav")` o interpretador não encontraria vinculo para o símbolo `nome`;
- `(string-append nome " " sobrenome)`, ao ser avaliada, resultaria em erro (porque o `let` ainda não teria criado os vínculos para `nome` e `sobrenome`.

A forma geral do `let` é

```
(let ((nome1 valor1)
      (nome2 valor2)
      ...)
  ;; nome1, nome2 etc são acessíveis aqui
  )
```

As formas `valor1, valor2, ...` são avaliadas antes de terem seus valores atribuídos às variáveis `nome1, nome2, ...` e os valores resultantes são associados aos nomes `nome1, nome2, ...`. É possível aninhar `lets`:

Versão Preliminar

```
(let ((nome "Gustav")
      (sobrenome "Klimt"))
  (let ((nome-completo
          (string-append nome " " sobrenome)))
    nome-completo))
```

Tivemos que usar dois `let`s aninhados porque não podemos escrever

```
(let ((nome "Gustav")
      (sobrenome "Klimt")
      (nome-completo (string-append nome " " sobrenome)))
```

já que uma definição de variável em um `let` não pode fazer referência às variáveis anteriores no mesmo `let` (ou seja, neste exemplo `nome-completo` não poderá usar `nome` e `sobrenome`). Isso acontece por diferentes razões. A forma `let` *poderia* ser executada em paralelo: as formas vinculando variáveis seriam executadas em paralelo, e não teríamos como saber qual delas terminaria primeiro. Além disso, o `let` é normalmente implementado de forma muito simples usando `lambda`, mas sem permitir o uso de nomes no mesmo `let`.

Para evitar uma quantidade muito grande de `let`s aninhados, há a forma especial `let*`, que é semelhante ao `let` mas permite que a definição de uma variável faça referência a outra, definida anteriormente no mesmo `let*`:

```
;; A definição de nome-completo usa as variáveis nome e sobrenome,
;; definidas antes no mesmo let*:
(let* ((nome "Gustav")
       (sobrenome "Klimt"))
       (nome-completo
         (string-append nome " " sobrenome)))
    nome-completo))
```

Em qualquer momento há diversas variáveis acessíveis em um programa Scheme. Por exemplo,

Versão Preliminar

```
(define saudacao "Olá, ")

(define cria-nome
  (lambda ()
    (let ((nomes '("Gustav" "Pablo" "Ludwig" "Frida"))
          (sobrenomes '("Klimt"
                        "Mahler"
                        "Picasso"
                        "van Beethoven"
                        "Kahlo")))
      (let* ((indice-nome (next-integer-random last-random
                                               (length nomes)))
             (indice-sobre (next-integer-random indice-nome
                                               (length sobrenomes))))
        (string-append
           (list-ref nomes indice-nome)
           " "
           (list-ref sobrenomes indice-sobre))))))
```

A variável `saudacao` é visível em todo o programa, porque foi declarada no nível base[9] As variáveis `nomes` e `sobrenomes` não são visíveis fora do primeiro `let`. Já `indice-nome` e `indice-sobre` são acessíveis apenas dentro do segundo `let`.

A forma `let` na verdade pode ser vista como uma variante de `lambda`. Nosso primeiro exemplo de `let`,

```
(let ((a 3)
      (b 4))
  (* a b))
12
```

poderia ser reescrito como uma aplicação de procedimento:

9 *"Top level"* em Inglês.

Versão Preliminar

```
( (lambda (a b)
    (* a b))
  3 4)
```
12

1.5 CONDIÇÕES

Para tomar decisões e escolher uma dentre duas ou mais formas a avaliar, Scheme oferece algumas formas especiais. A mais simples delas é o `if`. A forma geral do `if` é

```
(if teste forma1 forma2)
```

Durante a avaliação de uma forma `if`, o interpretador primeiro avaliará a forma `teste`. Se o valor resultante for diferente de `#f`, `forma1` será avaliada, e em caso contrário `forma2` será avaliada.

```
(define maximo 20)
(if (> 15 maximo) 'muito-longe 'perto)
```
perto

Como o `if` é usado justamente para escolher qual forma será avaliada, ele deve ser uma forma especial (caso contrário tanto `forma1` como `forma2` seriam avaliadas).

Além dos procedimentos para comparação numérica `=`, `<`, `>`, `<=`, `>=`, há procedimentos para comparar objetos. Dois deles são importantes:

- `eqv?` retorna `#t` para números, caracteres, booleanos ou símbolos iguais (dois símbolos são iguais se são representados pela mesma string). Para objetos compostos (por exemplo strings, listas, vetores) `eqv?` verificará se a localização dos objetos é a mesma na memória; se for, retornará `#t`;
- `equal?` faz as mesmas verificações que `eqv?`, mas também compara objetos compostos como listas, vetores e strings um elemento por vez.

```
(eqv? 1 1)
#t
(eqv? 'a 'a)
#t
(eqv? "" "")
```


Versão Preliminar

```
; depende da implementação de Scheme
(eqv? (lambda (x) x) (lambda (x) x))
; depende da implementação de Scheme
```

O próximo exemplo ilustra a diferença entre `eqv?` e `equal?`:

```
(eqv? '(a b c) (list a b c))
#f
(equal? '(a b c) (list a b c))
#t
```

Quando há muitas formas a escolher, cada uma dependendo de um teste, o uso do `if` será frustrante:

```
(if (> x 0)
    (display "x positivo")
    (if (< x 0)
        (display "x negativo")
        (if (zero? y)
            ( ...
```

Nestas situações é mais conveniente usar a forma especial `cond`, que permite listar vários testes e várias formas a avaliar dependendo de qual teste resultar em valor verdadeiro. Um exemplo do uso de `cond` é um procedimento que lista as soluções reais de uma equação do segundo grau.

Uma equação do segundo grau $ax^2 + bx + c = 0$ pode ter nenhuma, uma ou duas soluções reais, dependendo do valor do seu *discriminante* $b^2 - 4ac$, frequentemente denotado por Δ. O procedimento Scheme a seguir realiza este cálculo:

```
(define discriminante
  (lambda (a b c)
    (- (* b b)
       (* 4 a c))))
```

Quando o discriminante não é negativo as duas soluções são dadas por $(-b \pm \sqrt{\Delta})/(2a)$. Como o cálculo de ambas é idêntico exceto por uma única operação, um único procedimento pode calcular ambas, recebendo como argumento a operação (`+` ou `-`) a ser usada:

Versão Preliminar

```
(define raiz
  (lambda (a b c delta sinal)
    (let ((numerador (- (sinal (sqrt delta))
                        b)))
      (/ numerador (* 2 a)))))
```

Um procedimento Scheme que retorna a lista (possivelmente vazia) de soluções para a equação teria que verificar o discriminante antes de usar o procedimento `raiz` – de outra forma haveria a tentativa de calcular a raiz quadrada de um número negativo, que não é real. O resultado da chamada de procedimento `(raizes-grau-2 a b c)` deve ser:

$$\begin{cases} \texttt{(list (raiz a b c +) (raiz a b c -))} & \text{se } \Delta > 0 \\ \texttt{(list (/ (- b) (* 2 a)))} & \text{se } \Delta = 0 \\ \texttt{(list)} & \text{se } \Delta < 0 \end{cases}$$

A forma especial `cond` escolhe uma dentre várias formas Scheme, e pode ser usada para escolher que forma será avaliada e devolvida como valor do procedimento:

```
(define raizes-grau-2
  (lambda (a b c)
    (let ((delta (discriminante a b c)))
      (cond ((positive? delta)
             (list (raiz a b c delta +)
                   (raiz a b c delta -)))
            ((zero? delta)
             (list (/ (- b) (* 2 a))))
            (else (list))))))
```

A forma geral do `cond` é

Versão Preliminar

```
(cond (teste1 forma1-1
              forma1-2
              ...)
      (teste2 forma2-1
              forma2-2
              ...)
      ...
      (else   forma-else-1
              forma-else-2
              ...))
```

Quando não há `else` em uma forma `cond` e nenhum dos testes é satisfeito, a avaliação do `cond` não retorna valor:

```
(cond (#f 'x))
```

Além de `if` e `cond` há uma outra forma especial que permite avaliar uma dentre diferentes formas Scheme. A forma especial `case` é usada quando se quer comparar um valor com diversos outros.

Um jogo de pôquer poderia representar as cartas internamente como números de 2 a 10 e símbolos `j`, `q`, `k`, `q` e `a`. Para poder pontuar a mão de cada jogador, é necessário transformar estas cartas em números apenas. Usamos a forma `case` para determinar o valor de cada carta:

$$\left\{ \text{valor} = \begin{cases} \text{carta} & \text{se } 2 \leqslant \text{ carta } \texttt{leq10}, \\ 11 & \text{se carta } = \texttt{j}, \\ 12 & \text{se carta } = \texttt{q}, \\ 13 & \text{se carta } = \texttt{k}, \\ 14 & \text{se carta } = \texttt{a}. \end{cases} \right.$$

Este raciocínio "por casos" é traduzido no seguinte procedimento.

Versão Preliminar

```
(define carta->valor
  (lambda (carta)
    (case carta
      ((2 3 4 5 6 7 8 9 10) carta)
      ((j) 11)
      ((q) 12)
      ((k) 13)
      ((a) 14)
      (else (error "Carta não existe!")))))
```

A forma geral de `case` é:

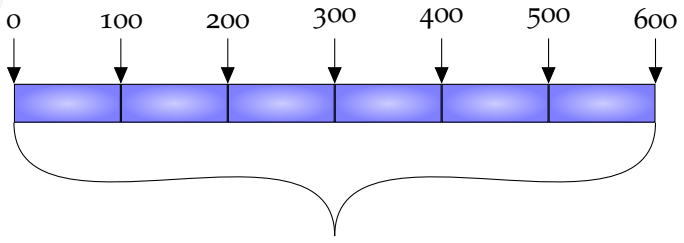
```
(case valor
  (lista-de-valores1 forma1)
  (lista-de-valores2 forma2)
  ...
  (else forma-else))
```

As listas de valores são verificadas na ordem em que aparecem.

Assim como na forma `cond`, o `else` é opcional. Quando não há `else` e nenhum dos casos é satisfeito, a avaliação do `case` não retorna valor.

1.5.1 Gerando eventos com probabilidades dadas

Retomamos agora o exemplo do gerador de números aleatórios, e suponha que queiramos simular um dado em um jogo. Queremos que cada face tenha probabilidade $1/6$ de ser escolhida. Uma maneira simples de fazer isso é escolher um número aleatóreo entre 0 e 599, dividir este intervalo em seis partes e associar cada um destes intervalos menores a uma face do dado. Como podemos presumir que o número será gerado a partir de uma distribuição uniforme, cada face do dado terá probabilidade igual a $100/600 = 1/6$.



Versão Preliminar

O seguinte procedimento Scheme implementa um dado usando esta técnica:

```
(define throw-die
  (lambda ()
    (let ((p (random 600)))
      (cond ((< p 100) 1)
            ((< p 200) 6)
            ((< p 300) 3)
            ((< p 400) 5)
            ((< p 500) 2)
            (else      4)))))
```

Podemos também simular um dado viciado. Se na primeira condição usarmos `(< p 120)` ao invés de `(< p 100)` teremos alocado mais vinte números para a face um (que retiramos da face seis). As probabilidades ficam diferentes:

$$\begin{aligned} p(1) &= \frac{120}{600} = \frac{1}{5} = 0.2 \\ p(6) &= \frac{80}{600} = \frac{2}{15} = 0.1\overline{3} \\ p(3) &= p(5) = p(2) = p(4) = \frac{1}{6} = 0.1\overline{6} \end{aligned}$$

1.6 REPETIÇÕES

Os procedimentos que discutimos até este momento não realizam processos que se repetem: todos executam uma determinada tarefa uma única vez e param. Para definir processos que se repetem, usamos *recursão*: um procedimento realiza diversas tarefas e, em seguida, chama a si mesmo novamente.

Versão Preliminar

```
(define countdown
  (lambda (n)
    (cond ((zero? n)
           (display "It's BREAK TIME!!!")
           (newline))
          (else
           (display n)
           (newline)
           (countdown (- n 1))))))
```

O procedimento acima conta de n até 1, e depois mostra uma mensagem:

```
(countdown 5)
5
4
3
2
1
It's BREAK TIME!!!
```

Ao ser chamado, `countdown` verifica se seu argumento é zero, e se for o caso mostra a mensagem. Se o argumento não for zero, ele é mostrado ao usuário e o procedimento chama a si mesmo, porque após mostrar n ao usuário, só falta contar de $n-1$ a zero e mostrar a mensagem – o que pode ser feito por `(countdown (- n 1))`. A "chamada a si mesmo" feita pelo procedimento tem o nome de "chamada recursiva".

1.6.1 Exemplo: aproximação da razão áurea

A razão áurea, normalmente denotada por φ, é um número irracional que aparece naturalmente em proporções na natureza e é empregado em artes, arquitetura e eventualmente em Computação[10] é a solução para a equação $\frac{a+b}{a} = a/b$, e é igual a $\frac{1+\sqrt{5}}{2}$. Apesar de conhecermos ϕ em função de $\sqrt{5}$, sabemos também que

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$$

10 Por exemplo, em funções de *hashing*, conforme sugerido por Knuth [Knu98b] e mencionado por Berman e Paul [BP05] e Cormen, Leiserson e Rivest [Cor+09] em suas discussões sobre tabelas de *hashing*.

versão preliminar

e portanto podemos aproximar a razão áurea (sem usar qualquer outro número irracional) por frações iteradas:

$\varphi_0 = 1$
$\varphi_1 = 1 + 1/\varphi_0 = 2$
$\varphi_2 = 1 + 1/\varphi_1 = 1.5$
$\varphi_3 = 1 + 1/\varphi_2 = 1.\overline{66}$
$\vdots$

Quanto maior o índice i, mais perto o valor φ_i estará da razão áurea φ. Para obter uma aproximação com garantia de precisão mínima, basta verificar a diferença $\varepsilon = |\varphi_{i+1} - \varphi_i|$. Como esta diferença diminui a cada iteração, ela eventualmente atingirá um valor pequeno o suficiente para ser considerado aceitável.

```
(define calcula-phi
  (lambda (valor tolerancia)
    (let ((proximo (+ 1 (/ 1 valor))))
      (let ((erro (abs (- proximo valor))))
        (if (>= erro tolerancia)
            (calcula-phi proximo tolerancia)
            proximo)))))
```

Uma implementação de Scheme que suporte racionais exatos fará as divisões sem perder a exatidão, se usarmos o valor inicial exato.

```
(calcula-phi 1 0.1)
8/5
(calcula-phi 1 0.00001)
987/610
```

Se usarmos 1.0 ao invés de 1, a mesma implementação fará contas com inexatos, e nos retornará um resultado inexato.

```
(calcula-phi 1.0 0.1)
1.6
(calcula-phi 1.0 0.0001)
1.61805555555556
(calcula-phi 1.0 0.00000001)
1.6180339901756
```


Versão Preliminar

1.7 LISTAS

Uma lista é formada por vários elementos do tipo *par*. Um par em Scheme é uma estrutura com duas partes. As duas partes de um par são chamadas (por motivos históricos[11]) `car` e `cdr`.

Para criar um par usa-se o procedimento `cons` (o "construtor" de listas).

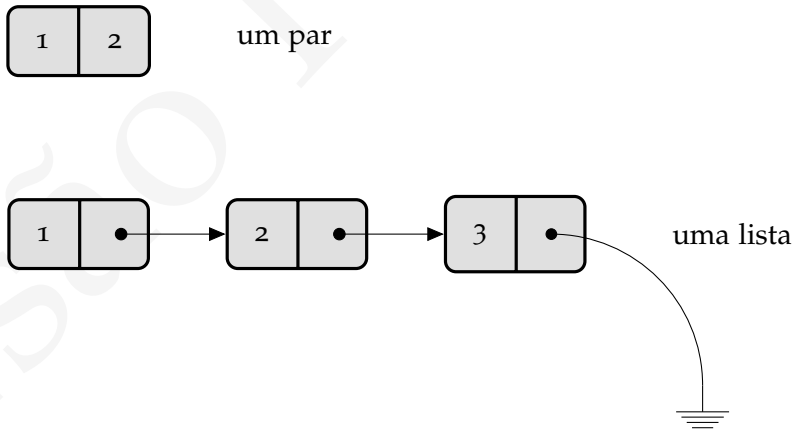
```
(cons 1 2)
(1 . 2)
```

Pedimos ao REPL para executar o procedimento `cons` com argumentos 1 e 2, e ele nos enviou o resultado: o par `(1 . 2)`.

Podemos obter o conteúdo de ambas as partes do par usando os procedimentos `car` e `cdr`:

```
(cons "este é o car" "este é o cdr")
("este é o car " . "este é o cdr")
(car (cons "este é o car" "este é o cdr"))
"este é o car"
(cdr (cons "este é o car" "este é o cdr"))
"este é o cdr"
```

Uma lista em Scheme é uma sequência de elementos armazenados na memória usando pares: cada par tem um elemento no `car`, e seu `cdr` tem uma referência ao próximo par:



11 Originalmente eram abreviações para *contents of address register* e *contents of decrement register*, que eram campos disponíveis nos computador IBM 704 usados na implementação da primeira versão de LISP em 1959 (esta implementação foi feita por Steve Russel). Em Common Lisp, há `first` e `rest` como sinônimos para `car` e `cdr`.

Versão Preliminar

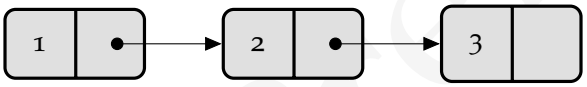
O último par de uma lista tem sempre a lista vazia em seu `cdr` (representado pelo sinal de aterramento no diagrama).

```
(cons 1 '())
(1)
(cons 1 (cons 2 (cons 3 '())))
(1 2 3)
(cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

O último objeto construído é `(1 2 3 . 4)`, que *não* é uma lista em Scheme: `(define x (cons 1 (cons 2 (cons 3 4))))`

```
x
(1 2 3 . 4)
(list? x)
#f
(pair? x)
#t
```

A representação esquemática deste objeto é mostrada no diagrama a seguir.



Há alguns outros procedimentos úteis para trabalhar com strings. Um deles é o `reverse`, que retorna uma cópia da lista na ordem reversa.

```
(reverse '(a b c d e))
(e d c b a)
```

Outros procedimentos importantes são:

- `null?` verifica se um objeto é a lista vazia.
- `list?` verifica se um objeto é uma lista. É importante observar que somente listas que terminam com `'()` são consideradas listas.
- `length` determina o comprimento de uma lista.

A seguir usamos listas e recursão – uma vombinação extremamente comum em linguagens da família Lisp – para construir mais uma parte do jogo de poquer.

Uma implementação de jogo de pôquer em Scheme poderia representar cartas como listas, onde o primeiro elemento da lista é a face da carta (um número ou um símbolo

Versão Preliminar

dentre `j`, `q`, `k` e `a`) e o segundo elemento é seu naipe (um símbolo dentre `paus`, `espadas`, `copas`, `ouros`). A lista `(6 ouros)` representaria um seis de ouros, por exemplo.

Dois procedimentos podem receber uma carta e retornar sua face e seu naipe:

```
(define pega-face
  (lambda (carta)
    (car carta)))

(define pega-naipe
  (lambda (carta)
    (car (cdr carta))))
```

A face de uma carta pode ser um número ou símbolo; para transformá-la em string precisamos verificar seu tipo e usar o procedimento adequado. O procedimento `face->string` transforma uma carta (número ou símbolo) em string.

```
(define face->string
  (lambda (face)
    (cond ((symbol? face)  (symbol->string face))
          ((number? face)  (number->string face)))))
(face->string 'q)
"q"
(face->string 5)
"5"
```

Finalmente, um procedimento `hand->string` recebe uma lista de cartas e retorna uma string com cada carta na forma `face:naipe`:

Versão Preliminar

```
(define hand->string
  (lambda (hand)
    (if (null? hand)
        ""
        (let ((carta (car hand)))
          (string-append
               (face->string   (pega-face carta))  ":"
               (symbol->string (pega-naipe carta)) " "
               (hand->string (cdr hand)))))))

(hand->string '((k ouros)
                (3 espadas)
                (5 ouros)
                (q paus)
                (q copas)))
```

"k:ouros 3:espadas 5:ouros q:paus q:copas"

O procedimento `hand->string` aceita um parâmetro (`hand`) e verifica se o parâmetro é uma lista, depois verifica se a lista passada é vazia. Se for, nada faz porque não há elementos a mostrar. Se a lista não for vazia, mostra-a o primeiro e pula uma linha. Em seguida, o procedimento chama a si mesmo. Nesta segunda chamada, o argumento é `(cdr hand)`.

1.7.1 Recursão linear, iteração linear e recursão na cauda

Procedimentos recursivos podem precisar de memória auxiliar a cada chamada recursiva. Esta seção descreve dois tipos de processo que podem resultar destes procedimentos:

- Processo de recursão linear, em que cada chamada recursiva exige a alocação de mais memória;
- Processo de iteração linear, que precisa de uma quantidade constante de memória, não dependendo do número de chamadas recursivas realizadas.

Esta seção trata destes dois tipos de processo. Começaremos com um exemplo de procedimento que gera um processo recursivo linear. O seguinte procedimento calcula a potência de um número com expoente inteiro:

Versão Preliminar

```
(define power*
  (lambda (x n)
    (if (= n 1)
        x
        (* x (power* x (- n 1))))))
```

O encadeamento de chamadas recursivas feitas por este procedimento para os argumentos 3 e 5 é:

```
(power* (3 5)
(* 3 (power* 3 4))
(* 3 (* 3 (power* 3 3)))
(* 3 (* 3 (* 3 (power* 3 2))))
(* 3 (* 3 (* 3 (* 3 (power* 3 1)))))
(* 3 (* 3 (* 3 (* 3 3))))
(* 3 (* 3 (* 3 9)))
(* 3 (* 3 27))
(* 3  81)
243
```

Evidentemente, quanto maior o expoente, mais memória será necessária para armazenar as operações pendentes de multiplicação. A chamada recursiva a `power*` é usada como argumento para `*`, e somente após a chamada a `*` o valor é retornado:

```
        (* x (power* x (- n 1))))))
```

Após a última chamada recursiva (quando $n = 1$), o procedimento "volta" da recursão multiplicando x por um valor que vai crescendo. Podemos modificar o procedimento para que não seja necessário fazer a multiplicação após a chamada recursiva. usaremos um argumento extra para o procedimento onde guardaremos o valor acumulado das multiplicações:

Versão Preliminar

```
(define power-aux
  (lambda (x n acc)
    (if (= n 0)
        acc
        (power-aux x
                   (- n 1)
                   (* x acc)))))

(define power
  (lambda (x n)
    (power-aux x (- n 1) x)))
```

Esta nova versão do procedimento gera um processo iterativo linear. Não há operações pendentes durante as chamadas recursivas a `power-aux`. Quando chamado com argumentos $3, 4, 3$ as chamadas de procedimento realizadas são:

```
(power-aux 3 4 3)
(power-aux 3 3 9)
(power-aux 3 2 27)
(power-aux 3 1 81)
(power-aux 3 0 243)
243
```

O procedimento `power-aux` precisa de uma quantidade constante de memória, ao contrário do procedimento `power*`.

> Se as chamadas recursivas em um procedimento nunca são usadas como argumentos para outros procedimentos (ou seja, se toda chamada recursiva em um procedimento é a última forma a ser avaliada), ele é chamado de procedimento *recursivo na cauda*.

O padrão da linguagem Scheme obriga toda implementação a otimizar procedimentos recursivos na cauda de forma a não usar memória adicional em cada chamada recursiva[12].

O procedimento `calcula-phi` desenvolvido na Seção 1.6 é recursivo na cauda, e portanto pode ser chamado recursivamente quantas vezes forem necessárias sem que o consumo de memória aumente.

O próximo exemplo mostra a transformação de um procedimento não recursivo na cauda em outro que é, mas sem redução da quantidade de memória necessária.

12 Esta otimização também é feita por compiladores de outras linguagens, mas não é obrigatória para elas

Versão Preliminar

O procedimento detalhado a seguir recebe um objeto, uma lista e uma posição, e retorna uma nova lista com o objeto incluído naquela posição.

```
list-insert 'x (list 'a 'b 'c) 0)
(x a b c)
list-insert 'x (list 'a 'b 'c) 2)
(a b x c)
list-insert 'x (list 'a 'b 'c) 3)
(a b c x)
```

A nova lista a ser retornada por `list-insert` pode ser definida indutivamente da seguinte maneira:

- Se a posição é zero, basta retornar `(cons elt lst)`
- Se a posição não é zero (e portanto o `car` da lista não será alterado), o procedimento retorna uma lista com o mesmo `car` e cujo `cdr` é uma chamada à `list-insert-aux` – desta vez passando `(cdr lst)` e `(- pos 1)`.

A definição indutiva acima é descrita naturalmente em Scheme da seguinte maneira:

```
(define list-insert-aux
  (lambda (elt lst pos)
    (if (= pos 0)
        (cons elt lst)
        (cons (car lst)
              (list-insert-aux elt
                               (cdr lst)
                               (- pos 1))))))
```

O procedimento `list-insert` primeiro verifica se o índice que recebeu está dentro de limites aceitáveis e depois chama `list-insert-aux`:

```
(define list-insert
  (lambda (elt lst pos)
    (if (or (negative? pos)
            (> pos (length lst)))
        (error "list-insert: pos is too far ahead")
        (list-insert-aux elt lst pos))))
```

Esta versão do procedimento não é recursiva na cauda porque há nele uma chamada recursiva como argumento para `cons`:

Versão Preliminar

```
        (cons (car lst)
              (list-insert ...)
```

Este uso de `cons` é necessário para que o procedimento se lembre, em cada chamada recursiva, dos elementos à esquerda da posição atual. Este procedimento pode ser transformado em outro, recursivo na cauda, que chamaremos de `list-insert-tail-rec`. O procedimento `list-insert-tail-rec-aux` receberá um argumento adicional, `left`, e o usará para lembrar-se de que elementos ficaram para a esquerda da posição atual. Como em cada chamada recursiva faremos `left` ser igual a `(cons (car lst) left)`, o argumento `left` será a lista à esquerda *invertida*. Para apresentar o resultado final, podemos então concatenar `(reverse left)` com `(cons elt lst)`:

```
(define list-insert-tail-rec-aux
  (lambda (elt lst pos left)
    (if (= pos 0)
        (append (reverse left)
                (cons elt lst))
        (list-insert-tail-rec-aux elt
                                  (cdr lst)
                                  (- pos 1)
                                  (cons (car lst) left)))))

(define list-insert-tail-rec
  (lambda (elt lst pos)
    (if (or (negative? pos)
            (> pos (length lst)))
        (error "list-insert: pos is too far ahead")
        (list-insert-tail-rec-aux elt lst pos (list)))))
```

Embora o procedimento seja recursivo na cauda, o argumento `left`, cresce a cada chamada recursiva. Além disso, o procedimento ficou menos legível e há uma chamada a `reverse`, que na versão inicial não era necessária. Este exemplo ilustra um fato importante: nem sempre podemos construir procedimentos recursivos que não precisem de memória auxiliar[13].

13 Na verdade é possível criar um procedimento que insere um elemento em uma lista sem o uso de memória auxiliar, mas isto só é possível usando mudança de estado – algo de que só tratamos no Capítulo 3.

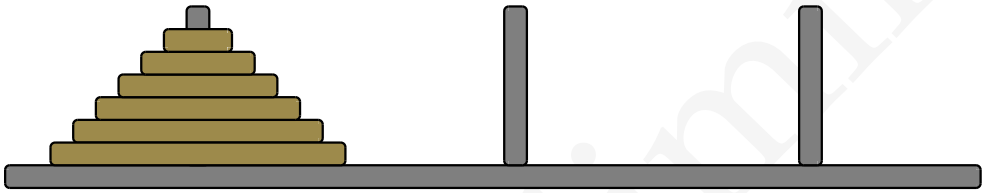
Versão Preliminar

1.7.2 Processo recursivo em árvore

Os procedimentos recursivos que vimos até agora fazem uma única chamada recursiva; trataremos agora de procedimentos que fazem mais de uma chamada a si mesmos. Chamamos o processo gerado por estes procedimentos de *recursão em árvore*.

1.7.2.1 *Exemplo: torres de Hanói*

"Torres de Hanói" é o nome de um quebra-cabeças bastante conhecido. Há três hastes verticais, duas delas vazias e uma com vários discos dispostos do maior para o menor, quando listados de baixo para cima. A figura a seguir mostra o quebra-cabeças com seis discos (normalmente mais discos são usados).



As regras do quebra-cabeças são:

- Somente um disco pode ser movido de cada vez, de uma haste a outra;
- Somente o disco no topo de uma haste pode ser movido, e somente para o topo de outra haste;
- Um disco somente pode ser posto sobre outro maior, nunca sobre um menor.

Daremos às hastes os nomes A, B e C, e vamos supor que queremos mover os discos da haste A para a haste B. Uma solução recursiva para este problema é bastante simples: se conseguirmos mover $n-1$ discos de A para a haste C, usando B como auxiliar restará apenas um disco em A, que podemos mover diretamente para B. Depois, basta movermos $n-1$ discos da haste auxiliar para B (desta vez usando A como auxiliar).

1. $(n-1)$ hastes, $A \to C$ (use B como auxiliar)
2. 1 haste, $A \to B$
3. $(n-1)$ hastes, $C \to B$ (use A como auxiliar)

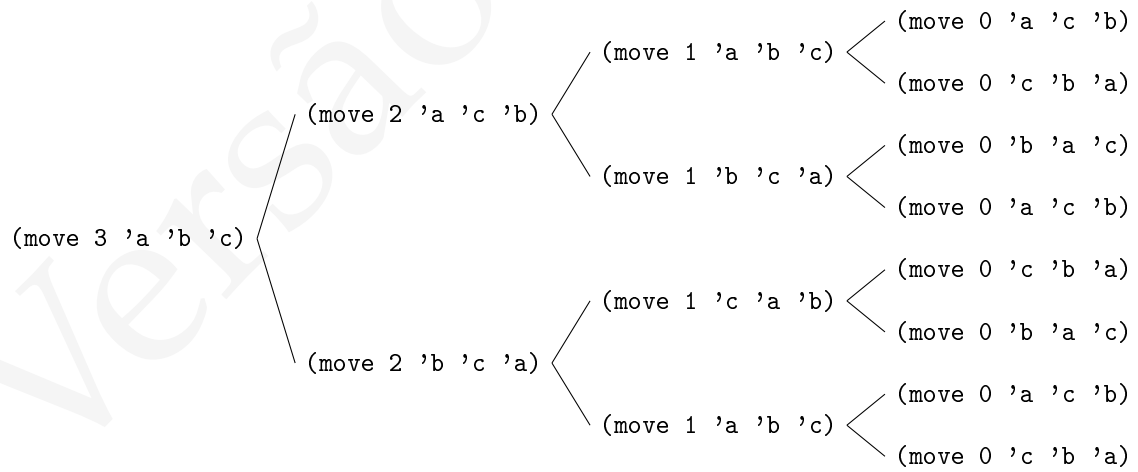
Como caracterizamos a solução para n discos usando soluções para $n-1$ (e sabemos que para zero discos não precisamos fazer nada), temos um algoritmo recursivo para resolver o problema.

Versão Preliminar

Traduziremos este algoritmo para Scheme: o procedimento `move` mostra como transferir n discos de uma haste `from` para outra, `to`, usando uma haste `extra` como auxiliar.

```
(define move
  (lambda (n from to extra)
    (cond ((zero? n) #t)
          (else
            (move (- n 1) from extra to)
            (display from)
            (display " -> ")
            (display to)
            (newline)
            (move (- n 1) extra to from)))))
(move 3 'A 'B 'C)
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
```

Se voltarmos a atenção para a estrutura da computação realizada notaremos que cada a `move` fará necessariamente duas outras chamadas recursivas, cada uma para $n-1$ discos. Podemos representar estas chamadas como uma árvore, como na Figura a seguir.



Versão Preliminar

Usando esta árvore podemos calcular a quantidade de movimentações individuais de disco que nosso algoritmo usará para mover uma pilha de tamanho n. A cada ramificação, um nó da árvore se divide em dois, e os movimentos individuais estão nas folhas da árvore. Temos portanto, 2^n movimentos para n discos.

1.7.3 *named let*

Há situações em que não é conveniente definir um procedimento visível em todo o programa. É possível criar laços locais da mesma forma que variáveis temporárias. Quando o primeiro argumento da forma `let` é uma lista, ela associa valores a variáveis. Quando o primeiro argumento é um símbolo e o segundo é uma lista, um laço é criado.

Para determinar o valor da maior carta de um jogador no jogo de poquer é necessário percorrer a lista de cartas lembrando-se do maior valor já visto; podemos usar *named let* para isto:

```
(define maior-carta
  (lambda (hand)
    (let repete ((cartas hand) (maior 0))
      (if (null? (cartas))
          maior
          (let ((valor (carta->valor (pega-nome (car hand)))))
            (if (> valor maior)
                (repete (cdr cartas) valor)
                (repete (cdr cartas) maior)))))))
```

O procedimento `maior-carta` é recursivo na cauda, embora para verificar a maior de cinco cartas isto seja de pouca importância.

A forma geral do *named* `let` é

```
(let nome ( (var1 val1)
            (var2 val2)
            ... )
  ...
  (nome prox-val1 prox-val2 ...))
```

Para obter um número aleatóreo entre 0 e k, calculamos o número $m = k/(2^3 2)$ e geramos números até conseguir um r menor que m; então podemos usar r/m.

Versão Preliminar

```
(define next-integer-random
  (lambda (x n)
    (let ((m (/ (expt 2 32 n))))
      (let again ((r (next-random x)))
        (if (> r m)
            (again ((r (next-random r))))
            (floor (/ r m)))))))
```

O procedimento `floor`, usado aqui, retorna o maior inteiro menor ou igual a x. Por exemplo,

```
(floor 2)
2
(floor 2.5)
2.0
(floor -2.5)
-3.0
(floor 2.999)
2.0
```

(Falta consertar algo aqui: precisamos do r para passar ao gerador na próxima chamada)

1.7.4 *letrec*

A forma especial `letrec` funciona da mesma maneira que a forma `let`, exceto que as definições dentro de um `letrec` podem fazer referência ao nome sendo definido.

O código a seguir é uma primeira tentativa de criar um procedimento `list-ref`, que retorna o n-ésimo elemento de uma lista.

Versão Preliminar

```
(define list-ref-1
  (lambda (lst n)
    (cond ((not (and (list? lst)
                     (integer? n)))
           (error "list-ref: need a list and an integer"))

          ((negative? n)       (error "n < 0"))
          ((>= n (length lst)) (error "n > max"))
          ((= n 0)             (car lst))

          (else
            (list-ref-1 (cdr lst) (- n 1))))))
```

A verificação de erros é feita a cada chamada recursiva; podemos evitar isto separando a verificação de erros da parte do procedimento que realmente encontra o n-ésimo elemento. Escreveremos um procedimento `list-ref-aux` que faz a busca, e só o chamaremos após ter verificado os parâmetros. Como `list-ref-aux` só faz sentido dentro deste procedimento, podemos defini-lo internamente com `letrec`:

```
(define list-ref
  (lambda (lst n)
    (letrec ((list-ref-aux (lambda (lst n)
                             (if (= n 0)
                                 (car lst))
                             (list-ref-aux (cdr lst) (- n 1)))))
      (cond ((not (and (list? lst)
                       (integer? n)))
             (error "list-ref: need a list and an integer"))

            ((negative? n)             (error "n < 0"))
            ((>= n (- (length lst) 1)) (error "n > max"))
            (else                      (list-ref-aux lst n))))))
```

Como usamos `letrec`, podemos fazer referência às variáveis que estamos definindo (neste caso `list-ref-aux` precisa se referir a si mesmo porque é recursivo).

Implementações de Scheme devem obrigatoriamente oferecer `list-ref`.

É tradicional em programação funcional usar (e construir) procedimentos que operam em listas. Os dois mais conhecidos e de utilidade mais evidente são `map` e `reduce`. O

Versão Preliminar

procedimento `list-map` toma como argumentos uma função e uma lista, e retorna uma nova lista cujos elementos são o resultado da aplicação da função a cada um dos elementos da lista original. Por exemplo,

```
(list-map - '(0 1 2 -3))
```
(0 -1 -2 3)

O procedimento passado para `list-map` deve aceitar exatamente um argumento, de outra forma não haveria como a função ser aplicada.

```
(define list-map
  (lambda (funcao lista)
    (if (null? lista)
        lista
        (cons (funcao (car lista))
              (list-map funcao (cdr lista))))))
```

O padrão define um procedimento `map`, que é mais geral que o que acabamos de construir: o procedimento passado para `map` pode ter qualquer aridade (quantidade de argumentos), e podemos passar diversas listas depois do procedimento:

```
(map expt '(1 3 2)
          '(2 3 4))
```
(1 27 16)

A aridade do procedimento (neste exemplo, 2) deve ser igual ao número de listas. A nova lista conterá em cada posição i o resultado da aplicação do procedimento, tendo como argumentos a lista de i-ésimos elementos das listas. No exemplo acima, o primeiro elemento da lista resultante é `(expt 1 2)`, o segundo `(expt 3 3)` e o terceiro `(expt 2 4)`.

O procedimento `list-reduce` toma um procedimento binário, uma lista, e aplica o procedimento a cada elemento da lista, acumulando o resultado:

```
(list-reduce * -1 '(10 20 30))
```
6000

O primeiro argumento de `list-reduce` é o procedimento a ser usado; o segundo é o valor a ser devolvido caso a lista seja vazia; o terceiro é a lista sobre a qual a operação será aplicada.

```
(list-reduce * -1 '())
```
-1

Versão Preliminar

O exemplo a seguir mostra uma implementação de `list-reduce` usando `letrec` para definir uma função recursiva visível apenas internamente. O procedimento interno `really-reduce` verifica se a lista tem tamanho um; se tiver, retorna a cabeça da lista e em caso contrário aplica o procedimento sobre a cabeça da lista e o resto da computação realizada por `really-reduce` na cauda da lista.

```
(define list-reduce
  (lambda (proc default lista)

    (letrec ((really-reduce
               (lambda (proc lista)
                 (if (= 1 (length lista))
                     (car lista)
                     (proc (car lista)
                           (really-reduce proc
                                          (cdr lista)))))))

      (if (null? lista)
          default
          (really-reduce proc lista)))))
```

(deve entrar uma figura aqui para ilustrar o reduce)

1.7.5 Definições internas

Uma maneira de conseguir o mesmo efeito de um `letrec` é usar uma forma `define` interna:

Versão Preliminar

```
(define list-reduce
  (define really-reduce
    (lambda (proc lista)
      (if (= 1 (length lista))
          (car lista)
          (proc (car lista)
                (really-reduce proc
                               (cdr lista))))))
  (if (null? lista)
      default
      (really-reduce proc lista)))
```

Observe que um `define` interno *não* cria vínculos globais – as variáveis são vinculadas somente dentro do escopo onde o `define` estiver.

```
(define f
  (lambda (a)
    (define twice (lambda (x) (+ x x)))
    (+ (twice a) 1)))
(f 2)
5
g
ERROR: undefined variable: g
```

1.8 FUNÇÕES DE ALTA ORDEM

Em linguagens de programação dizemos que um objeto é *de primeira classe* quando podemos dar-lhe um nome, passá-lo como parâmetro para uma subrotina e usá-lo como valor de retorno. Em todas as linguagens de programação de alto nível números, caracteres, strings e booleanos são objetos de primeira classe. Em linguagens funcionais, funções (procedimentos em Scheme) também são objetos de primeira classe[14]. Nosso `list-reduce` é um exemplo de procedimento de alta ordem, porque um de seus parâmetros é o *procedimento* que deve ser usado.

14 Nas linguagens Lisp os próprios nomes de variáveis são objetos de primeira classe, tornando a manipulação simbólica algo muito simples.

Versão Preliminar

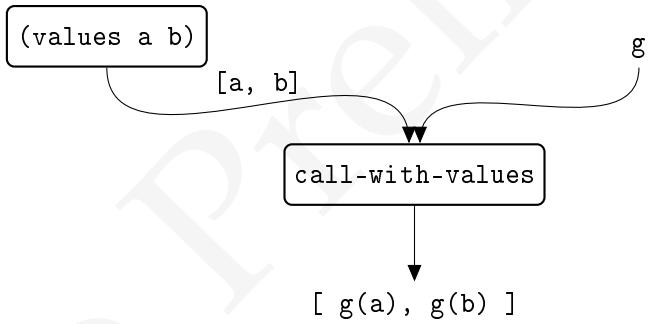
O Capítulo 6 traz uma grande quantidade de exemplos de procedimentos de alta ordem. Nesta seção detalharemos dois exemplos: composição de funções e *Currying*.

Nesta seção usaremos os procedimentos Scheme `apply` e `call-with-values`.

O procedimento `apply` é parte da essência de um interpretador Scheme e será descrito em detalhes no Capítulo 7. Quando chamado com um procedimento `proc` e uma lista como argumentos `args`, `apply` chamará `proc` com a lista de argumentos e retornará o resultado:

```
(+ 3 4 5)
12
(apply + '(3 4 5))
12
```

O procedimento `call-with-values` recebe dois procedimentos como argumentos, chama o primeiro (que não deve receber argumentos), e usa os valores retornados como argumentos para o segundo. O diagrama a seguir ilustra o funcionamento de `call-with-values`. As caixas retangulares são procedimentos sendo executados; a notação `[x, y]` é para pares de valores produzidos por `values`.



Em Scheme há um procedimento para transformar um número complexo de sua representação retangular (a e b, sendo o número $a + bi$) para a representação polar. Reconstruiremos este procedimento, mas retornaremos dois valores – ângulo e magnitude. A fórmula para a conversão é

$$\theta = \operatorname{atan}\left(\frac{y}{x}\right)$$
$$r = \sqrt{x^2 + y^2}$$

Versão Preliminar

```
(define rect->polar
  (lambda (x y)
    (values (atan (/ y x))
            (sqrt (+ (* x x) (* y y))))))
(rect->polar 2 3)
0.982793723247329
3.60555127546399
; 2 values
```

O REPL nos retornou dois valores, 0.982793723247329 e 3.60555127546399. Se tentarmos atribuir estes valores a uma variável, somente o primeiro será usado: `(define z (rect->polar 2 3))`

```
z
0.982793723247329
```

Mais adiante veremos como usar os dois valores.

Para um exemplo mais complexo, podemos querer calcular o mínimo e o máximo de uma lista, retornando-os. Podemos retornar uma lista com dois valores, mas também podemos, em Scheme, *retornar os dois valores*.

```
(define list-max-min
  (lambda (lst)
    (if (null? lst)
        0
        (let loop ((lst-tmp lst)
                   (max (car lst))
                   (min (car lst)))
          (cond ((null? lst)
                 (values max min))
                ((> (car lst) max)
                 (loop (cdr lst) (car lst) min))
                ((< (car lst) min)
                 (loop (cdr lst) max (car lst)))
                (else (loop (cdr lst) max min)))))))
```

O procedimento `list-max-min` tem dois valores de retorno: o maior e o menor elemento.

```
(list-max-min '(3 4 1 9 5))
9
```


Versão Preliminar

```
1
; 2 values
```

Usamos `call-with-values` para pegar estes dois valores, dar a eles temporariamente nomes `l` e `m`, e criar uma lista com os dois.

```
(call-with-values
    (lambda () (list-max-min '(3 4 1 9 5)))
  (lambda (l m)
    (list l m)))
(9 1)
```

Nas próximas subseções tratamos de algumas aplicações de funções de alta ordem.

1.8.1 Número variável de argumentos

Procedimentos Scheme não precisam ter um número fixo de argumentos.

O procedimento `sum-squares` retorna a soma dos quadrados de uma lista de números.

```
(define square
  (lambda (x) (* x x)))

(define sum-squares
  (lambda (lst)
    (if (null? lst)
        0
        (+ (square (car lst))
           (sum-squares (cdr lst))))))
```

Este procedimento funciona com listas de pontos de qualquer tamanho, inclusive a lista vazia (para a qual o valor é zero).

```
(sum-squares '(2 2 1))
9
```

Outro procedimento que poderemos usar é o que dá a norma de um vetor (ou seja, a distância entre este vetor e a origem no plano):

Versão Preliminar

```
(define norm-2d
  (lambda (a b)
    (sqrt (sum-squares (list a b)))))
```

Poderemos precisar também a norma de vetores em três dimensões, e para isso escrevemos um procedimento semelhante ao anterior:

```
(define norm-3d
  (lambda (a b c)
    (sqrt (sum-squares (list a b c)))))
```

Os procedimentos `norm-2d` e `norm-3d` são iguais a não ser pelo argumento `c`. Estamos evidentemente duplicando código! Se tivermos acesso à lista de argumentos do procedimento, `(a b)` para `norm-2d` ou `(a b c)` para `norm-3d`, poderemos simplesmente chamar `(sqrt (sum-squares lista-de-argumentos))`.

Em Scheme, a forma `lambda` deve ter, imediatamente após o símbolo `lambda`, a lista de argumentos. Esta lista pode estar entre parênteses, como em todos os procedimentos que criamos até agora, ou pode ser um símbolo – e neste caso este símbolo será o nome da lista de argumentos:

```
(define norm
  (lambda points
    (sqrt (apply sum-squares points))))
(norm 3 4 10 2)
11.3578166916005
```

Neste exemplo, o símbolo `points` será vinculado à lista de argumentos – e teremos agora um procedimento `norm` que funciona para *qualquer* número de dimensões!

Usando a notação abreviada para definição de procedimentos, podemos ter acesso à lista de argumentos usando `(define (procedimento . argumentos) corpo ...)`:

```
(define (norm . points)
    (sqrt (apply sum-squares points)))
```

1.8.2 Composição

Definimos a seguir o procedimento `o`, que toma duas funções `f` e `g` e retorna a composição $f \circ g$. Para construir esta função, criamos (no `lambda` mais interno) um procedimento

Versão Preliminar

que aplica g. Este procedimento pode retornar muitos valores, que o `call-with-values` coletará e usará como argumentos para `f`.

```
(define o
  (lambda (f g)
    (lambda args
      (call-with-values
          (lambda () (apply g args))
        f))))
```

Definimos o valor de π e usamos a função `square`, que retorna o quadrado de um número (esta última já foi definida na Seção 1.8.1, mas a reproduzimos aqui).

```
(define pi 3.1415926536)

(define square
  (lambda (x) (* x x)))
```

Definimos agora que `squared-sin` é a composição de `square` com seno:

```
(define squared-sin
  (o square sin))
(square (sin (/ pi 4)))
0.5000000000025517
(squared-sin (/ pi 4))
0.5000000000025517
```

1.8.3 Currying

Denotamos por $(X \to Y)$ o conjunto de todas as funções de X em Y (ou seja, o *tipo* "função de X em Y").

Qualquer função $f : A \times B \to C$ de dois argumentos pode ser transformada em outra função $f^c : A \to (B \to C)$ que recebe um único argumento e retorna uma função de um argumento.

Da mesma forma, dada uma função $f : (A_1 \times A_2 \times \cdots \times A_n) \to B$, podemos encontrar uma função $g : A_1 \to (A_2 \to (A_3 \to (\cdots \to B) \cdots))$

Por exemplo, a função que usamos para calcular juros compostos é

$$jc(i, t, v) = v(i + 1)^t.$$

Versão Preliminar

Podemos mudá-la para que aceite apenas a taxa de juros e o período (i, t), e retorne outra função. Esta nova função aceita v e finalmente retorna o valor total:

$$jc_2(i,t) = \lambda(v) \cdot \left[v(i+1)^t\right]$$

Aqui "$\lambda(x,y) \cdot [...]$" significa "função de x e y", da mesma forma que usamos em Scheme.

O procedimento `curry` recebe uma função e retorna a versão *"curried"*.

```
(define (curry fun . args)
  (lambda x
    (apply fun (append args x))))
```

Podemos definir um procedimento que nos retorna a potência de dois usando a técnica de *currying* sobre `expt`:

```
(define power2 (curry expt 2))
(power2 10)
```

```
1024
```

```
(define juros-compostos
  (lambda (i t v)
    (* v (expt (+ 1.0 i) t))))

(define jc2
  (lambda (i t)
    (curry juros-compostos i t)))
(define juros (jc2 0.03 12))
(juros 1000)
```

```
1425.76088684618
```

1.9 CORRETUDE

Embora possamos nos sentir confortáveis ao pensar sobre a corretude de pequenos programas, à medida que o tamanho deles cresce também cresce a necessidade de métodos sistemáticos para nos certificarmos de sua corretude.

Há diferentes métodos para verificar (de maneira limitada) que um programa está correto; abordaremos alguns deles nesta seção.

Versão Preliminar

1.9.1 Testes

(esta seção está incompleta)

Uma maneira simples e prática de verificar programas é escrevendo baterias de teste.

Construiremos um framework para testes unitários em Scheme.

O procedimento `test` recebe o nome de um teste, um procedimento, uma lista de argumentos e um valor esperado de retorno, e devolve uma lista com:

- O nome do teste;
- Um booleano que nos diz se o teste passou ou não;
- O valor retornado pelo procedimento testado;
- O valor esperado.

```
(define test
  (lambda (name proc args expected)
    (let ((result (apply proc args)))
      (list name
            (equal? result expected)
            result
            expected))))
```

A lista retornada por `test` contém toda informação necessária para que possamos construir diferentes relatórios de testes: podemos querer saber somente quantos testes falharam, ou quais falharam, ou ainda, para cada teste que falhou quais foram o resultado e o valor esperado.

Testaremos dois procedimentos: `movement*`, onde cometemos um erro e trocamos `s` por *2*, e `movement`, que está correto.

Versão Preliminar

```
(define movement*
  (lambda (s0 v0 a t)
    (let ((s (+ s0
                (* v0 t)
                (/ (* a t t)
                   2))))
      2)))

(define movement
  (lambda (s0 v0 a t)
    (let ((s (+ s0
                (* v0 t)
                (/ (* a t t)
                   2))))
      s)))

(define tests (list (list "movement*" movement* '(1 1 1.5 2) 6.0)
                    (list "movement" movement '(1 1 1.5 2) 6.0)))
```

Um procedimento `run-tests` aplicará o procedimento `test` a todos os testes da lista.

```
(define run-tests
  (lambda (tests)
    (map (lambda (l)
           (apply test l))
         tests)))
```

Guardamos o resultado de `run-tests` em uma variável `test-data`:

```
(define test-data (run-tests tests))
test-data
((”movement*” #t 6.0 6.0) (”movement” #f 2 6.0))
```

É interessante termos um procedimento que nos forneça o número de testes que passaram e o número de testes que falharam.

Versão Preliminar

```
(define test-stats
  (lambda (test-results)
    (let next ((lst test-results)
               (passed 0)
               (failed 0))
      (cond ((null? lst)
             (list passed failed))
            ((cadar lst)
             (next (cdr lst) (+ passed 1) failed))
            (else
             (next (cdr lst) passed (+ failed 1)))))))
(test-stats test-data)
(1 1)
```

1.9.2 Demonstração de corretude por indução

Os procedimentos que desenvolvemos até agora eram simples o suficiente para que possamos crer em sua corretude. Há algoritmos cuja corretude não é óbvia.

O procedimento `power` visto na seção 1.7.1 não é muito eficiente. O algoritmo que ele usa é:

$$power(x,n) = \begin{cases} 1 & \text{se } n = 0, \\ x\,(power(x,n-1)) & \text{caso contrário.} \end{cases}$$

Quando chamado com argumentos x, n, fará n chamadas recursivas. O seguinte procedimento também calcula a potência de números reais com expoentes inteiros, mas fará $\log_2(n)$ chamadas recursivas:

$$fast\_power(x,n) = \begin{cases} 1 & \text{se } n = 0, \\ x\left(fast\_power(x,\lfloor n/2 \rfloor)^2\right) & \text{se } n \text{ é ímpar,} \\ fast\_power(x,\lfloor n/2 \rfloor)^2 & \text{caso contrário.} \end{cases}$$

Não é imediatamente claro que este procedimento realmente calcula a potência x^n, e menos ainda que fará $\log_2(n)$ chamadas recursivas. É importante, então, conseguirmos demonstrar que o procedimento realmente retornará o valor que queremos. Provaremos aqui a corretude do procedimento `fast_power` (ou seja, que $fast\_power(x,n) = x^n$ para todo $x \in \mathbb{R}$ e para todo $n \in \mathbb{N}$). A prova é por indução em n.

Versão Preliminar

A base de indução é trivialmente verdadeira: para $n = 0$, temos $fast\_power(x, n) = 1$. Nossa hipótese de indução é: $fast\_power(x, k) = x^k$ para $0 \leqslant k \leqslant n - 1$.

Quebraremos o passo de indução em dois casos, um para n par e um para n ímpar, e usaremos nosso modelo de avaliação para desenvolver a prova. Quando n é ímpar,

$$\begin{aligned} fast\_power(x, n) &= x(fast\_power(x, \lfloor n/2 \rfloor))^2 \\ &= xx^{\lfloor n/2 \rfloor}x^{\lfloor n/2 \rfloor} \text{ (pela hip. de indução)} \\ &= xx^{(\lfloor n/2 \rfloor + \lfloor n/2 \rfloor)} \\ &= xx^{n-1} \text{ (porque } n \text{ é ímpar)} \\ &= x^n. \end{aligned}$$

Quando n é par,

$$\begin{aligned} fast\_power(x, n) &= (fast\_power(x, \lfloor n/2 \rfloor))^2 \\ &= x^{\lfloor n/2 \rfloor}x^{\lfloor n/2 \rfloor} \text{ (pela hip. de indução)} \\ &= x^{(\lfloor n/2 \rfloor + \lfloor n/2 \rfloor)} \\ &= x^n \text{ (porque } n \text{ é par),} \end{aligned}$$

e isto conclui a prova.

Para verificar que `fast_power` realmente faz $\log_2(n)$ chamadas recursivas, basta notar que a cada chamada onde o segundo argumento é n, a próxima chamada será feita com $\lfloor \frac{n}{2} \rfloor$; o valor do segundo argumento é dividido por dois a cada chamada recursiva.

1.10 STRINGS

Strings são sequências de caracteres. Em programas Scheme, objetos constantes do tipo string são representados entre aspas duplas, como por exemplo `"uma string"`.

O procedimento `string-ref` é usado para obter o caracter em uma determinada posição de uma string:

```
(define cidade "Atlantis")
cidade
"Atlantis"
(string-ref cidade 3)
#\a
```

O procedimento `string-length` retorna o comprimento de uma string:

Versão Preliminar

```
(string-length ‘‘Abracadabra’’)
11
```

Há dois procedimentos para comparar strings: `string=?`, que verifica se duas strings contém exatamente a mesma sequência de caracteres e `string-ci=?`, que ignora a diferença entre caixa alta e caixa baixa:

```
(define fala-de-hamlet "That skull had a tongue in it")
(string=? fala-de-hamlet "That skull had a tongue in it")
#t
(string=? fala-de-hamlet "That skull had a TONGUE in it")
#f
(string-ci=? fala-de-hamlet "That skull had a TONGUE in it")
#t
```

O procedimento `string-append` concatena as strings que lhe forem passadas como parâmetros:

```
(string-append "A " "terra " "é " "azul!")
”A terra é azul!”
```

O procedimento `string->symbol` retorna um símbolo cuja representação é igual à string passada como argumento, e `symbol->string` recebe um símbolo e retorna sua representação como string:

```
(string->symbol "transforme-me")
transforme-me
(symbol->string ’quero-ser-uma-string)
”quero-ser-uma-string”
```

1.10.1 Exemplo: cifra de César

Há um antigo método de criptografia conhecido como *cifra de César*, que se acredita ter sido usado por Júlio César para transmitir mensagens secretas. A cifra de César consiste em trocar as letras do alfabeto por deslocamento, de maneira a tornar as palavras irreconhecíveis. Se dermos às letras índices iniciando com $a \to 0; b \to 1; \cdots; z \to 25$, uma cifra de César faria a substituição de cada letra com índice i por outra de índice $i + k \mod 26$.

Versão Preliminar

A cifra de César pode ser trivialmente quebrada por força bruta quando se conhece o método usado (há apenas 25 tentativas possíveis). Mesmo quando o método não é conhecido, é possível detectá-lo e decifrar a mensagem através de análise de frequência de letras. Ainda assim cifras de César ainda são usadas em algumas situações.

Implementaremos uma cifra de César como exemplo de uso dos procedimentos Scheme para tratamento de strings.

O procedimento `string-map` recebe um procedimento `proc` que transforma caracteres, uma string, e retorna uma nova string onde os caracteres são o resultado da aplicação de `proc`.

```
(define string-map
  (lambda (proc s)
    (let ((l (string->list s)))
      (let ((new-list (map proc l)))
        (list->string new-list)))))
```

As mensagens que cifraremos serão todas convertidas para caixa alta. Definimos então um procedimento `string-upcase`.

```
(define string-upcase
   (lambda (s)
     (string-map char-upcase s)))
```

Caracteres podem ser transformados em números inteiros usando o procedimento `char->integer`. Para cifrar (ou "traduzir") um caracter, obtemos o índice do caracter `A`, subtraímos do caracter a ser cifrado e somamos um número (módulo 26, para que o resultado seja ainda um caracter alfabético).

Os procedimentos `char->ceasar-idx` e `ceasar-idx->char` traduzem caracteres para índices entre zero e 25.

```
(define char->ceasar-idx
  (lambda (x)
    (let ((a (char->integer #\A)))
      (- (char->integer x) a))))

(define ceasar-idx->char
  (lambda (x)
    (let ((a (char->integer #\A)))
      (let ((b (+ x a)))
        (integer->char b)))))
```

Versão Preliminar

```
(char->ceasar-idx #\B)
1
```

O procedimento `ceasar-translate-char` faz o deslocamento do caracter.

```
(define ceasar-translate-char
  (lambda (c k)
    (ceasar-idx->char (modulo (+ (char->ceasar-idx c)
                                 k)
                              26))))
(ceasar-translate-char #\B 10)
#\L
```

Para decifrar o caracter deslocamos no sentido contrário:

```
(ceasar-translate-char #\L -10)
#\B
```

Para cifrar e decifrar mensagens criamos procedimentos que deslocam caracteres e usamos `string-map` para aplicá-los sobre a mensagem:

```
(define ceasar-encrypt
  (lambda (s)
    (let ((ceasar-enc-10 (lambda (c)
                           (ceasar-translate-char c 10))))
      (string-map ceasar-enc-10
                  (string-upcase s)))))

(define ceasar-decrypt
  (lambda (s)
    (let ((ceasar-dec-10 (lambda (c)
                           (ceasar-translate-char c -10))))
      (string-map ceasar-dec-10
                  (string-upcase s)))))
(ceasar-encrypt "mensagem secreta")
"WOXCKQOWDCOMBODK"
(ceasar-decrypt "WOXCKQOWDCOMBODK")
```

Versão Preliminar

```
"MENSAGEMTSECRETA"
```

Como usamos módulo 26, a tradução só faz sentido para as 26 letras maiúsculas; perdemos o espaço entre as palavras (e o mesmo acontecerá com números e pontuação)! Os procedimentos que traduzem caracteres só devem modificar aqueles que forem alfabéticos. Usaremos o procedimento Scheme `char-alphabetic?` para realizar esta verificação.

```
(define ceasar-translate-char
  (lambda (c k)
    (if (char-alphabetic? c)
        (ceasar-idx->char (modulo (+ (char->ceasar-idx c)
                                     k)
                                  26))
        c)))
(ceasar-encrypt "mensagem secreta")
"WOXCKQOW COMBODK"
(ceasar-decrypt "WOXCKQOW COMBODK")
"MENSAGEM SECRETA"
```

O ROT13 é uma cifra de César que tornou-se popular em fóruns na Internet para evitar que trechos de mensagens sejam lidos por distração (por exemplo, respostas de charadas, descrição de truques para jogos online, partes de filmes e livros). O ROT13 é exatamente igual ao nosso `ceasar-encrypt` se usado com $k = 13$.

1.11 BYTEVECTORS

1.11.1 Exemplo: arquivos WAV

1.12 LISTAS DE ASSOCIAÇÃO

Usaremos uma lista para armazenar um catálogo de filmes. Cada entrada conterá o nome do filme e o local onde ele se encontra (que pode ser uma estante ou o nome da pessoa para quem o emprestamos). Um exemplo de base catálogo neste formato é mostrado a seguir.

Versão Preliminar

```
'(("Run, Lola, run!"           (estante b3))
  ("Fahrenheit 451"            (emprestado "John Doe"))
  ("The day Earth stood still" (estante c4)))
```

Listas como esta são chamadas de *listas de associação*, e sua forma é

```
( (chave valor)
  (chave valor)
  ...
  (chave valor) )
```

Criaremos um procedimento `assoc` procura um item em listas de associação.

```
(define assoc
  (lambda (chave lista)
    (cond ((null? lista)  #f)
          ((equal? (caar lista) chave)  (car lista))
          (else  (assoc chave (cdr lista))))))
```

Quando a chave não é encontrada, o valor de retorno é `#f`:

```
(assoc "O Iluminado" '())
#f
```

Quando a chave é encontrada, `assoc` retorna o par em que ela estava: podemos por exemplo usar `assoc` para descobrir onde está *Os Sete Samurais* de Kurosawa:

```
(assoc "Os Sete Samurais"
       '(("Era uma vez no oeste" (estante b3))
         ("Fahrenheit 451"       (emprestado "John Doe"))
         ("Os Sete Samurais"     (estante c4))))
("Os Sete Samurais" (estante c4))
```

O procedimento `assoc` existe em Scheme, mas sempre retorna `#f` quando um item não está na base de dados. Podemos torná-lo mais flexível, passando como parâmetro o objeto que deve ser retornado quando um item não é encontrado:

Versão Preliminar

```
(define assoc/default
  (lambda (chave lista nao-encontrado)
      (cond ((null? lista)
             nao-encontrado)
            ((equal? (caar lista) chave)
             (car lista))
            (else
             (assoc/default chave
                            (cdr lista)
                            nao-encontrado)))))
```

Quando a chave é encontrada, o comportamento de `assoc/default` é o mesmo de `assoc`:

```
(assoc/default "Os Sete Samurais"
               '(("Era uma vez no oeste" (estante b3))
                 ("Fahrenheit 451"       (emprestado "John Doe"))
                 ("Os Sete Samurais"     (estante c4)))
               'nope)
```

("Os Sete Samurais" (estante c4))

Como não incluímos *La Strada* de Fellini, `assoc/default` nos retornará `nope` quando o procurarmos:

```
(assoc/default "La Strada"
               '(("Era uma vez no oeste" (estante b3))
                 ("Fahrenheit 451"       (emprestado "John Doe"))
                 ("Os Sete Samurais"     (estante c4)))
               'nope)
```

nope

A inclusão na base de dados de filmes pode ser feita com o procedimento `cons`. No entanto, não usaremos `cons` diretamente por dois motivos: primeiro, queremos que o nome do procedimento descreva o que ele faz *no contexto do programa que estamos desenvolvendo* (ele inclui filmes em uma base de dados). Além disso, se algum dia decidirmos trocar a implementação da base de dados e não usarmos mais listas de associação, podemos modificar `inclui-filme` – mas se tivéssemos usado `cons` a mudança não seria tão simples (cada procedimento que inclui itens na base de dados teria que ser modificado).

Versão Preliminar

```
(define inclui-filme cons)
```

Agora acrescentamos *"Dr. Strangelove"* de Stanley Kubrick:

```
(inclui-filme '("Dr. Strangelove" (estante c4))
              '(("Era uma vez no oeste" (estante b3))
                ("Fahrenheit 451"       (emprestado "John Doe"))
                ("Os Sete Samurais"     (estante c4))))
```

```
(("Dr. Strangelove"(estante c4))
("Era uma vez no oeste"(estante b3))
("Fahrenheit 451"(emprestado "John Doe"))
("Os Sete Samurais"(estante c4)))
```

Além de `assoc`, que usa `equal?` para comparar chaves, a linguagem Scheme define `assv`, que usa `eqv?`, e `assq`, que usa `eq?`.

1.13 ABSTRAÇÃO DE DADOS

(esta seção está incompleta)

Para criar um tipo abstrato de dados precisamos de procedimentos para:

- Criar uma objeto deste tipo;
- Verificar se um objeto é deste tipo;
- Verificar se dois objetos deste tipo são iguais;
- Como os objetos deste tipo terão vários componentes, precisamos de procedimentos para acessar cada um deles.

1.13.1 Exemplo: números complexos

O padrão Scheme define o tipo número complexo, mas sua implementação é opcional. Faremos nossa própria implementação de números complexos.

```
(define make-rectangular
  (lambda (a b)
    (list a b)))
```


Versão Preliminar

Para determinar se um objeto é um número complexo criamos o procedimento `complex?`, que verifica se o objeto é uma lista, se tem tamanho 2, e se os dois elementos são números.

```
(define complex?
  (lambda (x)
    (and (list? x)
         (= (length x) 2)
         (number? (car x))
         (number? (cadr x)))))
```

O procedimento `complex=` verifica se dois complexos são iguais.

```
(define complex=
  (lambda (z1 z2)
    (and (= (real-part z1)
            (real-part z2))
         (= (imag-part z1)
            (imag-part z2)))))
```

Usaremos dois procedimentos para extrair as partes real e imaginária do número:

```
(define real-part
  (lambda (c)
    (car c)))

(define imag-part
  (lambda (c)
    (cadr c)))
```

Estes procedimentos são o mínimo que precisamos para criar o tipo de dados abstrato "número complexo".

Agora precisamos redefinir as operações aritméticas. Primeiro guardamos os procedimentos para soma e multiplicação:

```
(define standard+ +)
(define standard* *)
```

Criamos um procedimento que transforma números reais em complexos:

Versão Preliminar

```
(define real->complex
  (lambda (x)
    (if (complex? x)
        x
        (make-rect x 0))))
```

E finalmente definimos um procedimento que soma dois complexos. Usamos um `let` para extrair as partes reais e imaginária de ambos os números e damos nomes a elas de forma que $x = (a + bi)$ e $y = (c + di)$.

```
(define complex+
  (lambda (x y)
    (let ((complex-x (real->complex x))
          (complex-y (real->complex y)))
      (let ((a (real-p complex-x))
            (b (imag-p complex-x))
            (c (real-p complex-y))
            (d (imag-p complex-y)))
        (make-rect (+ a c) (+ b d))))))
```

A operação binária de soma agora deverá verificar se há algum elemento complexo. Se houver, a soma complexa será usada.

```
(define binary+
  (lambda (a b)
    (if (or (complex? a)
            (complex? b))
        (complex+ a b)
        (standard+ a b))))
```

A operação de soma com número arbitrário de argumentos é apenas um `list-reduce` usando a operação binária:

```
(define n-ary+
  (lambda args
    (list-reduce binary+ 0 args)))
(n-ary+ 2 3 (make-rect 1 5))
```

(6 5)

Versão Preliminar

Se quisermos, poderemos modificar o procedimento padrão + para que funcione com complexos:

```
(define + n-ary+)
(+ 7 (make-rect 0 2))
(7 2)
```

Fazemos agora o mesmo para a multiplicação:

```
(define complex*
  (lambda (x y)
    (let ((complex-x (real->complex x))
          (complex-y (real->complex y)))
      (let ((a (real-p complex-x))
            (b (imag-p complex-x))
            (c (real-p complex-y))
            (d (imag-p complex-y)))
        (make-rect (- (* a c) (* b d))
                   (+ (* b c) (* a d)))))))


(define binary*
  (lambda (a b)
    (if (or (complex? a)
            (complex? b))
        (complex* a b)
        (standard* a b))))


(define n-ary*
  (lambda args
    (list-reduce binary* 1 args)))
```

Note que desta vez passamos 1 como valor *default* para `list-reduce`.

O padrão Scheme também define procedimentos para obter a magnitude e o ângulo de um complexo. Nossa implementação é:

```
(define magnitude
  (lambda (args)
    (norm (car args)
          (cadr args))))
```

Versão Preliminar

O ângulo entre um vetor (a, b) no plano complexo e o eixo das abscissas é atan (b/a):

```
(define ang
  (lambda (c)
    (atan (/ (imag-p c)
             (real-p c)))))
```

1.14 FORMATAÇÃO DE CÓDIGO

Nas linguagens do tipo Lisp é comum que o código seja formatado da seguinte maneira:

- Quando um parêntese é aberto sem que seja fechado em uma linha, a próxima linha é indentada à frente (normalmente usam-se dois espaços);
- Parênteses são fechados todos juntos, sem quebra de linha entre eles.

A seguir há um trecho de código formatado conforme a tradição Lisp e outro, que certamente causa estranheza a usuários experientes de Lisp (pessoas acostumadas com programação em C, C++, Java e linguagens semelhantes podem demorar um pouco para se acostumar ao estilo das linguagens da família Lisp).

```
;; Código formatado conforme a prática comum:
(define media-de-tres
  (lambda (a b c)
    (/ (+ a b c)
       3)))


;; Código formatado sem seguir a prática usual:
(define media-de-tres
    (lambda (a b c)
       (/ (+ a b c) 3)
    )
)
```

EXERCÍCIOS

Ex. 1 — Converta as seguintes expressões para a notação prefixa:

Versão Preliminar

a) $a + b + c + d + e + f$

b) $a + b - \frac{1}{(c-b)}$

c) $a + 1 - b - 2 + c + 3$

d) $\frac{(a+b)(c+d)}{e+1}$

e) $\frac{(a+b+c)(d+e+f)}{g+1}$

f) $\frac{2a}{b-c}$

Ex. 2 — Converta as seguintes expressões para a notação infixa:

a) `(+ (* a b c) d e f)`

b) `(+ (- a b) c d)`

c) `(* a b (+ c d) (- e f) g)`

d) `(* 2 a (/ b 4) (+ c 10))`

e) `(/ 1 (+ x y z))`

f) `(* (+ a b c) (* d e (+ f g) (- h i)))`

Ex. 3 — Escreva os procedimentos Scheme a seguir. São todos muito simples, mas úteis para habituar-se com a linguagem.

a) `volume-esfera`, que calcula o volume de uma esfera.

b) `area-circ`, que calcula a área de uma circunferência.

c) `conta-nao-zero`, que recebe uma lista e retorna o número de itens diferentes de zero. *(Diga também o que muda se você usar* `zero?`, `=` *ou* `eq` *para comparar elementos)*

d) `list-even-half`, que recebe uma lista e retorna outra, com a "metade par" da lista (os elementos nas posições pares).

e) `triangle?`, que verifica se três números podem ser as medidas dos lados de um triângulo.

Ex. 4 — Quais das seguintes s-expressões podem ser formas Scheme válidas?

a) `(abracadabra 1 2 3)`

b) `(- 10 2 3)`

c) `(#f)`

d) `(1.0 2.0 3.0`

e) `(a (b (c (d (e (f))))))`

f) `("uma" "lista" "de" "strings")`

Versão Preliminar

g) `"(1 2 3)"`

h) `'(1 2 3)`

i) `(1 2 3)`

j) `(lambda (x) #f)`

Ex. 5 — Explique cuidadosamente o que significam as expressões e quais seus valores:

a) `(lambda (x y) (x y))`

b) `((lambda (x y) (x y)) display display)`

c) `((lambda (x y) (x y)) display 'display)`

Ex. 6 — Explique porque os dois procedimentos a seguir retornam resultados diferentes.

```
(let ((x 1))
  (let ((x 2)
        (y (* x 2)))
    y))


(let ((x 1))
  (let* ((x 2)
         (y (* x 2)))
    y))
```

Ex. 7 — Escreva dois procedimentos que transforme medidas de temperatura entre escalas (de Celsius para Fahrenheit e vice-versa).

Ex. 8 — Escreva um procedimento que calcule a distância entre dois pontos no plano.

Ex. 9 — Construa um procedimento que receba três pontos `A`, `B` e `C` representando os vértices de um triângulo, um quarto ponto `P`, e verifique se `P` está ou não dentro do triângulo.

Ex. 10 — Implemente o procedimento padrão Scheme `member` que recebe um objeto, uma lista e retorna `#f` se o objeto não é um dos elementos da lista. Se o objeto é elemento da lista, `member` deve retornar a sublista começando com `obj`:

```
(member 'x '(a b c))
#f
(member 'x '(a b c x y z)
'(x y z)
```


Versão Preliminar

Ex. 11 — Implemente um programa que calcule a média dos elementos de uma lista; depois, um outro que determine quantos elementos estão acima da média.

Ex. 12 — Implemente procedimentos que implementem operações sobre conjuntos, usando listas para representá-los, como no exemplo a seguir:

```
(make-set)
()
(set-add "elemento"(make-set))
("elemento")
```

Conjuntos não tem elementos repetidos:

```
(let ((a (make-set)))
  (let ((b (set-add 'x a)))
    (let ((c (set-add 'x b)))
      c)))
(x)
```

Implemente também operadores para união e diferença de conjuntos.

Ex. 13 — Uma lista contém strings e outros tipos de dados. Faça um procedimento que seleciona apenas as strings da lista, separando-a em uma outra lista:

```
(seleciona-strings '(o brave new world "that has"
                     such "people" "in" it))
"that has people in"
```

Ex. 14 — Faça um procedimento que receba duas listas ordenadas de números e retorne a intercalação das duas.

```
(intercala '(1 3 4 4 8 9 20)
           '(4 5 12 30)
(1 3 4 4 4 8 9 12 20 30)
```

Ex. 15 — Modifique o intercalador de listas do Exercício 14 para aceitar mais um argumento: um predicado que determina quando um objeto é menor que outro.

Versão Preliminar

```
(intercala '("abacaxi caju goiaba")
           '("banana framboesa uva")
           string<=)
```
("abacaxi" "banana" "caju" "framboesa" "goiaba" "uva")

```
(intercala '(() (a b c) (x x x x x))
           '((1) (1 2) (1 2 3 4 5 6))
           length)
```
(() (1) (1 2) (a b c) (x x x x x) (1 2 3 4 5 6))

Ex. 16 — Faça um procedimento que escreva um número em hexadecimal (base 16). Prove por indução que o procedimento está correto.

Ex. 17 — Faça um procedimento que verifica se uma string é palíndroma.

Ex. 18 — Faça um procedimento Scheme que receba uma string e inverta cada uma de suas palavras. Por exemplo[15],

```
(inverte-palavras "homo sapiens non urinat in ventum")
```
"omoh sneipas non taniru ni mutnev"

Ex. 19 — Faça procedimentos para converter entre a numeração Romana e a numeração usual ocidental moderna.

Ex. 20 — A cifra Atbash consiste em trocar a primeira letra do alfabeto pela última, a segunda pela penúltima, e assim por diante. Implemente-a.

Ex. 21 — Faça um algoritmo que leia duas strings representando fitas de DNA, mas desta vez algumas posições podem conter não apenas A, C, T ou G. Podem conter:

```
R = G A (purina)
Y = T C (pirimidina)
K = G T (ceto)
M = A C (amino)
```

(Representando ambiguidade)
O algoritmo deve informar:

i- Quanto os DNAs seriam similares se, cada vez que houver ambiguidade, considerarmos que as posições não casam;

15 Esta frase está gravada na entrada da praça Max Euwe em Amsterdam.

Versão Preliminar

ii- Quanto os DNAs seriam similares se, cada vez que houver ambiguidade, considerarmos que as posições casam (se for possível).

Por exemplo, 'R' com 'A' contaria como não em (i), mas como sim em (ii). 'R' com 'Y' contaria sempre como não.

Ex. 22 — Escreva uma versão recursiva na cauda do procedimento a seguir:

```
(define fatorial
  (lambda (n)
    (if (< n 2)
        1
        (* n (fatorial (- n 1))))))
```

Ex. 23 — Faça dois procedimentos que calculam π, usando os seguintes métodos:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} \cdots$$
$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

Veja quantas iterações cada método demora para aproximar π com, por exemplo, 10 casas decimais.

Ex. 24 — Implemente o algoritmo Blum-Micali para geração de números aleatóreos, que gera números melhores que o método que mostramos na seção 1.3.5. O algoritmo Blum-Micali é descrito a seguir:
Seja p um primo grande e g uma raiz primitiva[16] modulo p. Seja x_0 uma semente, e

$$x_{i+1} = g^{x_i} \mod p.$$

O i-ésimo bit de saída do algoritmo é 1 se $x_i < \frac{p-1}{2}$, caso contrário é zero.
Você pode usar $p = 5193706330149680375091293062 81$ e $g = 11$.

Ex. 25 — Modifique o procedimento `ceasar-translate-char` para que ele preserve a caixa (alta ou baixa) do caracter.

Ex. 26 — Além dos procedimentos que definimos para números complexos, o padrão Scheme também define o procedimento `make-polar`, que recebe magnitude e ângulo e retorna um número complexo. Implemente `make-polar`.

16 Uma raiz primitiva modulo p é um número g tal que qualquer número coprimo com p é congruente a alguma potência de g módulo p (ou podemos dizer também que g é uma raiz primitiva módulo p se é um gerador do grupo multiplicativo de inteiros modulo p). Note que não é necessário compreender o que é uma raiz primitiva para fazer este exercício, uma vez que damos valores possíveis para p e g no final do enunciado.

Versão Preliminar

Ex. 27 — Uma maneira de gerar todas as $\binom{n}{k}$ combinações de n elementos tomando k de cada vez é usar o algoritmo:

```
VARIE V1 de 0 até n-k-1
  ...
    VARIE V3 de V2+1 até n-1
      VARIE V4 de V3+1 até n
        MOSTRE I1, I2, ...
```

Por exemplo, para $\binom{n}{4}$

```
VARIE V1 de 0 até n-3
  VARIE V2 de V1+1 até n-2
    VARIE V3 de V2+1 até n-1
      VARIE V4 de V3+1 até n
        MOSTRE I1, I2, I3, I4
```

Escreva um procedimento `with-combinations` que aceite n, k e uma função de k argumentos:

```
(with-combinations 3 2
                   (lambda (x1 x2)
                     (print "--> " x1 " " x2)))

--> 0 1
--> 0 2
--> 1 2
```

A função será chamada com cada par de números da combinação.
Este procedimento é definido facilmente se você pensar de maneira indutiva.
Mostre, por indução, que o procedimento realmente chamará a função que recebe como parâmetro com todas as $\binom{n}{k}$ combinações de índices, com os índices variando entre 0 e $n-1$.

Ex. 28 — Refaça o exercício 27 em C e Java.

Ex. 29 — Modifique o procedimento `o` que faz composição de funções (na Seção 1.8) para que aceite um número arbitrário de funções. Por exemplo,

- `(o)` deve retornar `(lambda () (values))`;
- `(o f)` deve retornar `f`;
- `(o f g h)` deve retornar $f \circ g \circ h$.

Ex. 30 — Releia a função `o` que faz composição de funções (na Seção 1.8). `(o o o)` dá como resultado um procedimento:

Versão Preliminar

```
(define ooo (o o o))
ooo
#<procedure>
```

Diga se o procedimento `ooo` faz sentido (se é possível usá-lo). Se `ooo` não fizer sentido, explique exatamente porque; se fizer, diga o que ele faz e mostre um exemplo de uso.

Ex. 31 — O framework para testes unitários da Seção 1.9.1 sempre usa `equal?` para verificar o resultado de testes. Modifique-o para que ele receba como último parâmetro, opcional, um procedimento que verifica igualdade.

Ex. 32 — Modifique o framework para testes unitários da Seção 1.9.1 para que ele funcione com procedimentos que retornam múltiplos valores.

Ex. 33 — Modifique o procedimento `assoc/default` para que ele receba um procedimento a ser usado para comparar as chaves, de forma que possamos usar outros procedimentos além de `equal?`.

Ex. 34 — Tente obter soluções recursivas para diferentes quebra-cabeças, a exemplo do que fizemos na Seção 1.7.2.1 para as Torres de Hanói.

Ex. 35 — Faça um procedimento que receba vários números e retorne o número igual à concatenação dos números dados. Seu programa *não* deve usar strings. Por exemplo,

```
(concatena-numeros 2 35 10 822)
23510822
```

Ex. 36 — Faça um programa para determinar se um número é primo.

Ex. 37 — Faça um programa para determinar a fatoraração de inteiros.

Ex. 38 — Faça um programa para encontrar o *home prime* de um número n. Dado n, $\mathrm{HP}(n)$ é obtido da seguinte forma:

- Obtenha os fatores de n;
- Concatene os dígitos dos fatores, obtendo n';
- Se n' é primo, retorne n'. Senão, retorne $\mathrm{HP}(n')$.

Por exemplo,

Versão Preliminar

$$\begin{aligned} 10 &= 2 \times 5 \\ \rightarrow 25 &= 5 \times 5 \\ \rightarrow 55 &= 5 \times 11 \\ \rightarrow 511 &= 7 \times 73 \\ \rightarrow 773 &\quad \text{é primo} \end{aligned}$$

Ex. 39 — Escreva um programa que leia um ano e mostre o calendário daquele ano (leve em conta anos bissextos!) Por exemplo, para 2011 o calendário seria mostrado da seguinte maneira:

```
      Janeiro
Do Se Te Qa Qi Se Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31

      Fevereiro
Do Se Te Qa Qi Se Sa
       1  2  3  4  5
          .
          .
          .
```

RESPOSTAS

Resp. (Ex. 3) — Evidentemente é possível escrever estes programas de diferentes maneiras; as versões mostradas aqui são apenas uma possibilidade.

a) O volume de uma esfera com raio r é $(4\pi r^3)/3$, portanto:

```
(define volume-esfera
  (lambda (r)
    (/ (* 4 pi r r r) 3)))
```


Versão Preliminar

c)

```
(define conta-nao-zero
  (lambda (lista)
    (cond ((null? lista) 0)
          ((zero? (car lista) (conta-nao-zero (cdr lista))))
          (else (+ 1 conta-nao-zero (cdr lista))))))
```

Se o procedimento = for usado, `conta-nao-zero` só funcionará para listas de números.

d)

```
(define list-even-half
  (lambda (lista)
    (cond ((null? lista) '())
          ((null? (cdr lista)) '())
          (else (cons (cadr lista)
                      (list-even-half (cddr lista)))))))
```

Resp. (Ex. 4) — Podem ser formas válidas: (a), porque o valor da variável `abracadabra` pode ser um procedimento; (b), porque - normalmente é um procedimento; (e), porque o valor da variável a pode ser um procedimento; (g), porque se trata apenas de uma string (um átomo); (h), porque é abreviação de `(quote (1 2 3))`; e (j).

Resp. (Ex. 5) — (a) Uma forma começando com `lambda` é um procedimento. Este procedimento aceita dois parâmetros, x e y. Verificando o corpo do procedimento, pode-se concluir que x deve ser um outro procedimento e y um valor que pode ser passado como argumento para x. (b) A expressão é uma aplicação de procedimento, equivalente a `(display display)`, e o valor do símbolo `display` será mostrado (normalmente este valor será um procedimento). (c) A expressão é equivalente a `(display 'display)`, e *o símbolo* `display` será mostrado (mas não o seu valor).

Resp. (Ex. 9) — Há várias maneiras de resolver este problema. Uma delas é calcular os vetores PA PB e PC, e verificar se os ângulos entre eles (APB, BPC, CPA) somam $2\pi \pm \varepsilon$.

Resp. (Ex. 10) — Uma possível implementação é:

Versão Preliminar

```
(define member
  (lambda (obj lst)
    (cond ((null? lst)              #f)
          ((equal? obj (car lst))   lst)
          (else                     (member obj (cdr lst))))))
```

Resp. (Ex. 11) — Uma versão:

```
(define media
  (lambda (lst)
    (define soma
        (lambda (lst)
            (if (null? lst)
                0
                (+ (car lst) (soma (cdr lst))))))
    (if (null? lst)
        0
        (/ (soma lst) (length lst)))))

(define acima-media
  (lambda (lst)
    (let ((m (media lst)))
      (let loop ((lst-tmp lst)
                 (acima 0))
        (cond ((null? lst-tmp) acima)
              ((> (car lst-tmp) m)
               (loop (cdr lst-tmp) (+ 1 acima)))
              (else
               (loop (cdr lst-tmp) acima)))))))
```

E uma versão minimalista, usando `apply` e `map`.

Versão Preliminar

```
(define media
  (lambda (lst)
    (if (null? lst)
        0
        (/ (apply + lst) (length lst)))))

(define acima-media
  (lambda (lst)
    (let ((m (media lst)))
      (let ((acima (lambda (x) (if (> x m) 1 0))))
        (apply + (map acima lst))))))
```

Resp. (Ex. 13) — Uma possível implementação (recursiva na cauda) é:

```
(define seleciona-strings
  (lambda (uma-lista lista-de-strings)
    (if (null? uma-lista)
        lista-de-strings
        (if (string? (car uma-lista))
            (seleciona-strings (cdr uma-lista)
                               (cons (car uma-lista)
                                     lista-de-strings))
            (seleciona-strings (cdr uma-lista)
                               lista-de-strings)))))
```

Resp. (Ex. 27) — Use um procedimento auxiliar recursivo `combine`, que implementa os loops aninhados. Ele deve receber `(start end n fun args)`, onde `start` e `end` são o valor inicial e final do loop, e `args` é um acumulador com os índices calculados anteriormente.

Versão Preliminar

```
(define combine
  (lambda (start end n fun args)
    ...))

(define with-combinations
  (lambda (n k fun)
    (combine 0 (+ (- n k) 1) n fun '())))
```

Resp. (Ex. 35) — Faça primeiro para dois números, depois para vários. Você pode guardar os números em uma lista, depois calcular a quantidade de dígitos em cada um, e finalmente multiplicar cada número por uma potência de dez. Por exemplo, para a sequência `2 35 10 822`:

números	2	35	10	822
dígitos	1	2	2	3
				$\times 10^{0}$
			$\times 10^{3}$	
		$\times 10^{3+2}$		
	$\times 10^{3+2+2}$			

Versão Preliminar

2 ENTRADA E SAÍDA

Até agora não nos preocupamos com a entrada e saída de dados. Interagimos com o interpretador usando o teclado e recebemos suas respostas diretamente. Neste Capítulo trataremos de entrada e saída de dados em arquivos. Também desenvolveremos parte de uma biblioteca para geração de imagens vetoriais.

2.1 ARQUIVOS E PORTAS

Em Scheme a entrada e saída de dados se dá através de *portas*. Uma porta é um dispositivo abstrato onde podemos escrever ou ler.

O procedimento `open-input-file` abre um arquivo para leitura e retorna uma porta de entrada; já o procedimento `open-output-file` abre um arquivo para saída e retorna uma porta de saída. Quando o arquivo não puder ser aberto, uma condição de erro será levantada.

Podemos verificar se um objeto é uma porta porta (e se é de entrada ou de saída) com os procedimentos `port?`, `input-port?` e `output-port?`.

```
(open-input-file "um-arquivo.txt")
#<input-port>
```

```
(let ((x (open-output-file "whatever.txt")))
  (output-port? x))
#t
```

Após terminar a leitura ou gravação de dados em um arquivo, fechamos a porta de entrada (ou saída) com `close-input-port` ou `close-output-port`.

Todos os procedimentos que realizam entrada e saída recebem um último argumento opcional que determina que porta será usada. Quando este argumento não está presente, a operação é feita usando duas portas padrão: "entrada corrente" e "saída corrente". Estas duas portas podem ser obtidas pelos procedimentos `current-input-port` e `current-output-port`.

Versão Preliminar

O procedimento `display` envia um objeto Scheme para uma porta, de maneira que pareça legível para humanos, mas *não* de forma que possa ser lido novamente pelo ambiente Scheme. Por exemplo, strings são mostradas sem aspas (e portanto da mesma forma que símbolos).

```
(let ((out (open-output-file "saida.txt")))
  (display "Seu número da sorte é: " out)
  (display (next-random 322) out)
  (newline out)
  (display "Tenha um bom dia!" out)
  (close-output-port out))
```

O procedimento `newline` envia para uma porta uma quebra de linha. O trecho acima gravará, no arquivo `saida.txt`:

```
Seu número da sorte é: 2041087718
Tenha um bom dia!
```

Para escrever apenas um caracter em uma porta de saída, usamos `write-char`.

O procedimento `read-char` lê um caracter de uma porta de entrada. Já `peek-char` retorna o próximo caracter, mas não o consome.

O procedimento `eof-object?` é usado para verificar se um objeto lido de uma porta representa o fim de um arquivo.

Os procedimentos `read` e `write` são usados para ler e escrever formas Scheme em sua representação externa. Uma forma escrita por `write` pode ser lida por `read`: strings são escritas com aspas, caracteres são escritos com `\#`, etc. Para ilustrar a diferença entre `display` e `write`, podemos simplesmente reescrever o trecho anterior usando `write`:

```
(let ((out (open-output-file "saida.txt")))
  (write "Seu número da sorte é: " out)
  (write (next-random 321) out)
  (write #\newline out)
  (write "Tenha um bom dia!" out)
  (close-output-port out))
```

O conteúdo do arquivo `saida.txt` após a execução deste programa será:

```
"Seu número da sorte é: "2041087718#\newline"Tenha um bom dia!"
```

que é a sequência de objetos Scheme que escrevemos, de forma que possam ser lidos novamente pelo interpretador (uma string, um número, um caracter `#\newline` e outra string).

Versão Preliminar

Usamos `write`, por exemplo, para criar um procedimento `save-db` que grava em um arquivo nossa base de dados de filmes.

```
(define save-db
  (lambda (db file-name)
    (let ((out (open-output-file file-name)))
      (write db out)
      (close-output-port out))))
```

Gravaremos nossa base de dados em um arquivo:

```
(save-db '(("Dr. Strangelove" (estante c4))
           ("Era uma vez no oeste" (estante b3))
           ("Fahrenheit 451" (emprestado "John Doe"))
           ("Os Sete Samurais" (estante c4)))
         "movies.dat")
```

O conteúdo de `movies.dat` será:

```
(("Dr. Strangelove" (estante c4))
 ("Era uma vez no oeste" (estante b3))
 ("Fahrenheit 451" (emprestado "John Doe"))
 ("Os Sete Samurais" (estante c4)))
```

O procedimento `load-db` recupera a base de dados de um arquivo, mas é um pouco diferente de `save-db`:

```
(define load-db
  (lambda (file-name)
    (let ((in (open-input-file file-name)))
      (let ((db (read in)))
        (close-input-port in)
        db))))
```

Precisamos de um `let` a mais, porque se a última forma fosse `(close-input-port in)` o valor de retorno do procedimento seria o valor que ela retorna. Por isso guardamos o valor lido por `read` em uma variável temporária `db` e a retornamos depois.

Além de `read` e `write`, os outros procedimentos para entrada e saída aceitam um último argumento opcional que indica a porta a ser usada. O exemplo do número da sorte poderia ser escrito da seguinte maneira:

Versão Preliminar

```
(let ((out (open-output-file "saida.txt")))
  (display "Seu número da sorte é: " out)
  (display (next-random 111) out)
  (newline out)
  (display "Tenha um bom dia!" out)))
```

Usamos três `displays` e um `newline`, e em todos indicamos a porta de saída `out`. Pode ser conveniente fixar uma porta uma única vez, para que seja usada em diversas operações de saída (ou de entrada). `With-input-from-file` e `with-output-to-file` executam um procedimento (sem argumentos) modificando a entrada e saída padrão.

```
(with-output-to-file "saida.txt"
  (lambda ()
    (display "Seu número da sorte é: ")
    (display (next-random 111))
    (newline)
    (display "Tenha um bom dia!")))
```

O código abaixo lerá o conteúdo do arquivo `saida.txt` e o mostrará na porta de saída corrente.

```
(with-input-from-file "saida.txt"
  (lambda ()
    (let loop ((c (read-char)))
      (if (not (eof-object? c))
          (begin (display c)
                 (loop (read-char)))))))
```

É muitas vezes inconveniente usar diversas chamadas a `display` e `newline` para mostrar muitos objetos. Para isso poderíamos criar então dois procedimentos, `print` e `print-line`, que recebem uma lista de argumentos e usa `display` para mostrar cada um.

Versão Preliminar

```
(define print-list
  (lambda (lst)
    (if (not (null? lst))
        (begin (display (car lst))
               (print-list (cdr lst))))))

(define print
  (lambda objetos
    (print-list objetos)))

(define print-line
  (lambda objetos
    (print-list objetos)
    (newline)))
```

Estes procedimentos permitem escrever linhas de uma só vez:

```
(define film-status caadr)
(define film-title  car)
(define film-place  cadadr)

(define where-is
  (lambda (title db)
    (let ((x (assoc title db)))
      (cond ((not x)
             (print-line "Não encontrei o filme " title))
            ((eqv? 'emprestado (film-status x))
             (print-line "O filme " title
                         " está emprestado para "
                         (film-place x)))
            (else
             (print-line "O filme " title
                         " está na estante "
                         (film-place x)))))))
(where-is "Era uma vez no oeste" d)
O filme Era uma vez no oeste está na estante b3
(where-is "Fahrenheit 451" d)
```

Versão Preliminar

```
O filme Fahrenheit 451 está emprestado para John Doe
```

2.1.1 Verificando e removendo arquivos

(esta seção está incompleta)

Ao usar `open-input-file`, podemos verificar se o arquivo existe com o procedimento `file-exists?`. O procedimento `delete-file` remove um arquivo, dado seu nome.

```
(if (file-exists? "arquivo")
    (delete-file "arquivo"))
```

2.1.2 Portas de strings

R^{7}RS, SRFI-6

(esta seção está incompleta)

Além de ler e escrever de arquivos, é possível criar portas para ler e escrever em strings, como se fossem arquivos. O procedimento `open-input-string` recebe um único parâmetro (uma string) e retorna uma porta de entrada a partir dessa string:

```
(define in-string "100 200")

(let ((in (open-input-string in-string)))
  (let ((x (read in)))
    (let ((y (read in)))
      (display (+ x y))))
  (close-input-port in))
```

```
300
```

Podemos abrir uma porta de saída para string com `open-output-string`. Cada vez que quisermos a string com os dados já gravados podemos obtê-la com o procedimento `get-output-string`.

Versão Preliminar

```
(let ((out (open-output-string)))
  (display "-----" out)
  (let ((str (get-output-string out)))
    (display str)
    (do ((i 0 (+ 1 i)))
        ((= i 5))
      (display i out)
      (display "  " out)
      (display (* i (expt -1 i)) out)
      (newline out))
    (let ((str (get-output-string out)))
      (display str)))
  (close-output-port out))
```

```
0 0
1 -1
2 2
3 -3
4 4
```

Portas de strings podem ser fechadas com `close-input-port` e `close-output-port`, mas `get-output-string` continuará retornando o conteúdo gravado, mesmo após a porta ter sido fechada.

No exemplo a seguir os procedimentos `show-one-option` e `show-options` enviam dados para uma porta de saída.

```
(define show-one-option
  (lambda (pair out)
    (display (car pair) out)
    (display "  --  " out)
    (display (list-ref pair 1) out)
    (newline out)))
```

Versão Preliminar

```
(define show-options
  (lambda (option-alist out)
    (for-each (lambda (option)
                (show-one-option option out))
              option-alist)))
```

Se definirmos agora uma lista de associação com pares de opção e texto a ser mostrado,

```
(define options '( (-h "show help")
                   (-v "verbose")
                   (-n "no action, just simulate") ))
```

Podemos enviar para `(current-output-port)` ou para uma string, como mostra o seguinte trecho.

```
(let ((o-port (open-output-string)))
  (show-options options o-port)
  (close-output-port o-port)
  (get-output-string o-port))
```

2.2 UM GERADOR DE XML

Nosso programa se restringirá a gerar XML da seguinte forma:

```
<tag atributo1="um atributo"
     atributo2="outro atributo">

   texto <tag2 ...> ... </tag2> mais texto

</tag>
```

Uma característica importante de XML é sua estrutura: as tags são definidas recursivamente, de modo semelhante às listas em Lisp. Será interessante então se a representação da estrutura XML for parecida com S-expressões Lisp. O pedaço de XML acima será representado da seguinte maneira:

Versão Preliminar

```
(tag atributo1 "um atributo"
     atributo2 "outro atributo"
     "texto"
     (tag2 ...)
     "mais texto")
```

Esta representação não é ambígua, e pode ser lida da seguinte maneira:

- A cabeça de uma lista é sempre um símbolo que dá nome a uma tag;
- Logo depois do nome da tag, pode haver um símbolo. Se houver, é o nome de um atributo – e seu valor (uma string) vem em seguida. Enquanto houver sequências de símbolo/string, os lemos como pares atributo/valor;
- Após a sequência de atributos, há a lista de elementos internos da tag, que podem ser strings ou listas:
  - Quando um elemento interno for uma string, ele representa texto dentro da tag;
  - Quando um elemento interno for uma lista, ele representa uma outra tag dentro do texto.

O gerador de XML que construiremos neste Capítulo receberá uma tag XML representada como S-expressão e escreverá sua tradução para XML na porta saída atual. Desta forma o gerador não será referencialmente transparente no sentido estrito. A alternativa seria fazê-lo produzir uma string XML e depois escrever a string em uma porta, mas para isto usaríamos uma grande quantidade de chamadas a `string-append`, que aloca uma nova string cada vez que é usado. Para documentos grandes (ou para grande quantidade de documentos pequenos) o consumo de memória e tempo de processamento se tornariam um problema.

Há outras maneiras de S-expressões em XML de que trataremos no Capítulo 8 (por exemplo, ao invés de receber a S-expressão como parâmetro, poderíamos avaliá-la, e o efeito colateral seria a saída da tradução da S-expressão na porta de saída).

Começamos definido um procedimento `xml-write-attribute` que recebe uma lista contendo dois elementos – um nome e um valor – e mostra a representação do atributo XML.

Versão Preliminar

```
(define xml-write-attribute
  (lambda (name-value)
    (let ((name  (car name-value))
          (value (cadr name-value)))
      (display " ")
      (display name)
      (display "=\"")
      (display value)
      (display "\""))))
(xml-write-attribute '(href "http://nowhere"))
href="http://nowhere"
```

O procedimento `xml-write-open-tag` recebe o nome de uma tag, sua lista de atributos, e mostra a abertura da tag.

```
(define xml-write-open-tag
  (lambda (name attr)
    (display "<")
    (display name)
    (for-each xml-write-attribute attr)
    (display ">")
    (newline)))
(xml-write-open-tag 'div '((font "helvetica") (color "blue")))
<div font="helvetica" color="blue">
```

O procedimento `xml-write-close-tag` fecha uma tag.

```
(define xml-write-close-tag
  (lambda (name)
    (display "</")
    (display name)
    (display ">")
    (newline)))
(xml-write-close-tag 'body)
</body>
```

Versão Preliminar

Para obter a lista de atributos e o ponto da lista onde inicia o conteúdo interno da tag, usaremos o procedimento `get-attr-list/start-data`. Este procedimento retorna uma lista cujos elementos são a lista de atributos (no formato de lista de associação) e a sublista de `lst` onde começa a parte interna da tag (ou seja, logo depois da parte onde estão os atributos). Criamos um único procedimento para estas duas tarefas porque a a segunda parte (a sublista) é exatamente o que nos restará depois de termos coletado os atributos – e se tivéssemos dois procedimentos para isto teríamos que percorrer duas vezes os atributos.

```
(define get-attr-list/start-data
  (lambda (lst)
    (let next ((lst  lst)
               (attr '()))
      (if (or (null? lst)
              (not (symbol? (car lst))))
          (list (reverse attr) lst)
          (next (cddr lst)
                (cons (list (car lst)
                            (cadr lst))
                      attr))))))

(get-attr-list/start-data '(font "helvetica"
                            color "blue"
                            "This goes inside"))
```

(((font "helvetica") (color "blue")) ("This goes inside"))

O procedimento `xml-write-data` traduzirá a parte interna de tags XML. Este procedimento recebe uma lista (a sublista daquela que descreve a estrutura XML, mas iniciando na parte que descreve o interior da tag).

Versão Preliminar

```
(define xml-write-data
  (lambda (lst)
    (if (not (null? lst))
        (let ((element (car lst)))
          (if (string? element)
              (display element)
              (xml-write-tag element))
          (xml-write-data (cdr lst))))))
(xml-write-data '("This goes inside" " and this goes too"))
This goes inside and this goes too
```

Agora nos falta um procedimento que receba uma estrutura XML (uma tag) e a traduza.

```
(define xml-write-tag
  (lambda (tag-object)
    (let ((tag-name (car tag-object))
          (attr/start-data (get-attr-list/start-data
                                (cdr tag-object))))

      (let ((attr       (car  attr/start-data))
            (start-data (cadr attr/start-data)))

        (xml-write-open-tag tag-name attr)
        (xml-write-data start-data))
      (xml-write-close-tag tag-name))))
```

Testaremos nosso gerador de XML com uma variante simples de HTML:

```
(xml-write-tag '(html (head (title "My HTML page"))
                      (body (h1 color "blue"
                                "My H1 heading")
                            "My text!")))
<html>
<head>
<title>
My HTML page</title>
</head>
```


Versão Preliminar

```
<body>
<h1 color="blue">
My H1 heading</h1>
My text!</body>
</html>
```

2.3 GRÁFICOS VETORIAIS

(esta seção está incompleta)

O formato SVG para gráficos vetoriais é XML. Construiremos uma biblioteca para construção de gráficos vetoriais, e os gravaremos usando nosso gerador de XML.

Primeiro faremos um procedimento que aceita como argumentos atributos e valores, e os devolva em uma lista, com os valores transformados em string (porque é desta forma que atributos XML são escritos).

O procedimento auxiliar `number->string/maybe` transforma números em strings, mas mantém os outros valores intactos.

```
(define number->string/maybe
  (lambda (x)
    (if (number? x)
        (number->string x)
        x)))
(number->string/maybe 10)
"10"
(number->string/maybe 'x)
x
```

```
(define make-attr-list
  (lambda args
    (map number->string/maybe args)))
(make-attr-list 'x 1 'y 2 'z 3)
(x "1"y "2"z "3")
```

Versão Preliminar

Definiremos procedimentos para criar linhas e triângulos em SVG, representando-os como S-expressões. O procedimento `make-svg-line` aceita dois pontos, duas cores (a do interior e do traçado) e um número, que dá a largura do traçado.

```
(define make-svg-line
  (lambda (x1 y1 x2 y2 stroke fill stroke-width)
    (cons 'line
          (make-attr-list 'x1 x1
                          'y1 y1
                          'x2 x2
                          'y2 y2
                          'fill fill
                          'stroke stroke
                          'stroke-width stroke-width))))
(define line (make-svg-line 2 3 10 20 "bluered"5))
line
(line x1 "2" y1 "3"
x2 "10" y2 "20"
fill "red"
stroke "blue"
stroke-width "5")
(xml-write-tag line)
<line x1="2" y1="3"
x2="10" y2="20"
fill="red"
stroke="blue"
stroke-width="5">
</line>
```

O formato SVG não define triângulo como forma básica, por isso o construiremos como um polígono. Ao descrever um polígono em SVG, precisamos incluir um atributo `points`, que contém uma lista de pontos na forma `"x1,y1 x2,y2, ...`. Usaremos um procedimento que recebe uma lista de números representando pontos e retorna uma string com a lista de pontos.

Versão Preliminar

```
(define list-of-points
  (lambda args
    (let next-point ((points (map number->string args))
                     (result ""))
      (if (null? points)
          result
          (next-point (cddr points)
                      (string-append result
                                     (car points)  ","
                                     (cadr points) " "))))))
(list-of-points 23 25 10 20 50 60)
"23,25 10,20 50,60 "
```

Finalmente, o procedimento `make-svg-triangle` recebe três pontos, cores para borda e interior, a largura da borda e retorna a representação do triângulo.

```
(define make-svg-triangle
  (lambda (x1 y1
       x2 y2
       x3 y3
       stroke fill stroke-width)
    (cons 'polygon
          (make-attr-list 'fill fill
                          'stroke stroke
                          'stroke-width stroke-width
                          'points (list-of-points x1 y1
                                                  x2 y2
                                                  x3 y3)))))

(make-svg-triangle 10   20
                   100  200
                   1000 2000
                   "red" "blue" 4)
(polygon fill "blue"
stroke "red"
stroke-width "4"
```

Versão Preliminar

```
points ”10,20 100,200 1000,2000 ”)
```

Uma imagem SVG é uma tag XML `svg`, com atributos `version` e `xmlns`, contendo as tags que descrevem figuras em seu interior.

```
(define make-svg-image
  (lambda (objects)
    (append (list 'svg
                  'version 1.1
                  'xmlns "http://www.w3.org/2000/svg")
            objects)))

(define image
  (make-svg-image (list (make-svg-triangle 0   0
                                           200 200
                                           0   200
                                           "blue"
                                           "rgb(255,0,0)"
                                           3)
                        (make-svg-line 0   100
                                       100 100
                                       "green"
                                       "green"
                                       5))))
(xml-write-tag image)
<svg version=”1.1” xmlns=”http://www.w3.org/2000/svg”>
<polygon fill="rgb(255,0,0)”
stroke=”blue” stroke-width=”3” points=”0,0 200,200 0,200 ”>
</polygon>
<line x1=”0” y1=”100” x2=”100” y2=”100”
fill=”green” stroke=”green” stroke-width=”5”>
</line>
</svg>
```

Para gerar um arquivo XML válido precisamos incluir um preâmbulo. Não entraremos nos detalhes desta parte do XML; simplesmente o incluiremos como uma string.

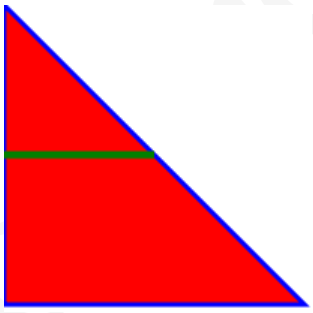
Versão Preliminar

```
(define svg-preamble
  "<?xml version=\"1.0\" standalone=\"no\"?>
<!DOCTYPE svg PUBLIC
         \"-//W3C//DTD SVG 1.1//EN\"
         \"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd\">")
```

O procedimento `image->xml` transforma uma descrição de imagem em S-expressões em XML e a grava em um arquivo.

```
(define image->xml
  (lambda (img file)
    (with-output-to-file file
      (lambda ()
        (display svg-preamble)
        (newline)
        (xml-write-tag img)))))
(image->xml image "whoo.svg")
```

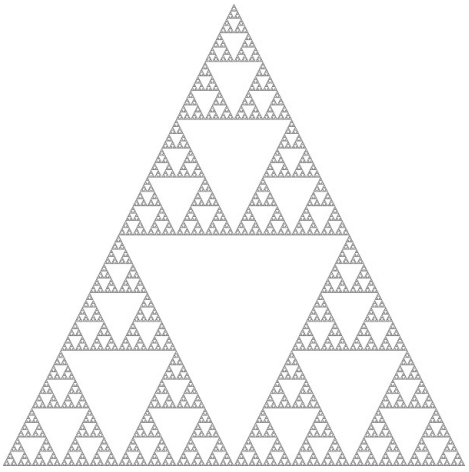
O arquivo `whoo.svg` conterá a seguinte imagem:



2.3.1 Exemplo: triângulo de Sierpinski

O fractal conhecido como triângulo de Sierpinski é um objeto auto-similar, ilustrado na figura a seguir.

Versão preliminar



O triângulo de Sierpinski pode ser gerado da seguinte maneira: primeiro desenhe um triângulo ABC em preto. Depois, execute os passos a seguir:

- Marque os pontos médios $[AB]$, $[BC]$, $[AC]$ de cada lado do triângulo;
- Desenhe um triângulo branco usando estes pontos médios como vértices;
- Agora a figura conterá três triângulos pretos, $[AC]C[BC]$, $A[AC][AB]$ e $[AB][BC][B]$. Repita esta operação para cada um deles.

É evidente que não poderemos executar este processo indefinidamente; nosso gerador de triângulos de Sierpinski aceitará como parâmetro o número de iterações (quando o número for zero um único triângulo preto será desenhado).

Versão Preliminar

```
(define make-sierpinski
  (lambda (x1 y1 x2 y2 x3 y3 fill-out fill-in n)

    (define make-sierpinski-aux
      (lambda (x1 y1 x2 y2 x3 y3 fill n acc)
        (if (positive? n)
            (let ((x12 (/ (+ x1 x2) 2.0))
                  (y12 (/ (+ y1 y2) 2.0))
                  (x13 (/ (+ x1 x3) 2.0))
                  (y13 (/ (+ y1 y3) 2.0))
                  (x23 (/ (+ x2 x3) 2.0))
                  (y23 (/ (+ y2 y3) 2.0)))
              (append acc
                      (list
                        (make-svg-triangle x12 y12 x23 y23 x13 y13
                                           fill fill 0))
                      (make-sierpinski-aux x1 y1 x12 y12 x13 y13
                                           fill (- n 1) '())
                      (make-sierpinski-aux x2 y2 x12 y12 x23 y23
                                           fill (- n 1) '())
                      (make-sierpinski-aux x3 y3 x13 y13 x23 y23
                                           fill (- n 1) '())))
            acc)))

    (make-svg-image
     (append (list (make-svg-triangle x1 y1 x2 y2 x3 y3
                                      fill-out fill-out 0))
             (make-sierpinski-aux x1 y1 x2 y2 x3 y3
                                  fill-in n '())))))
```

O procedimento `make-sierpinski-aux` calcula os pontos médios dos três lados do triângulo, desenha um triângulo com vértices nestes três pontos e em seguida chama a si mesmo recursivamente para desenhar os outros três triângulos.

A figura no início desta seção foi gerada usando o código a seguir.

Versão Preliminar

```
(let ((s (make-sierpinski 0 600 300 0 600 600
                          "black" "white"
                          10)))
  (images->xml s "sierpinski.svg"))
```

O tempo usado por este programa é exponencial (proporcional a 3^n), assim como o tamanho do arquivo gerado.

EXERCÍCIOS

Ex. 40 — Faça um procedimento que copie um arquivo, mantendo o conteúdo intacto *exceto* por números, que devem ser multiplicados por -1. Suponha que `entrada.txt` contenha:

```
1984 é de George Orwell, e 42 é a resposta para a pergunta
fundamental sobre a vida, o universo e tudo mais.
```

Após `(muda-numeros "entrada.txt" "saida.txt")` O arquivo `saida.txt` deve conter:

```
-1984 é de George Orwell, e -42 é a resposta para a pergunta
fundemental sobre a vida, o universo e tudo mais.
```

Ex. 41 — Faça um procedimento que intercale dois arquivos. O procedimento deve aceitar três nomes de arquivo: duas entradas e uma saída. Os dois arquivos de entrada devem conter números e devem estar ordenados. O arquivo de saída deve conter os números de ambas as entradas, em ordem. Você não deve trazer todos os números para a memória de uma vez.

Ex. 42 — Modifique o intercalador do Exercício 41 para que funcione com qualquer tipo de informação (não apenas números). O procedimento precisará de mais um argumento, que é o procedimento para comparar dois elementos.

Ex. 43 — Modifique o intercalador do Exercício 42 para eliminar itens duplicados.

Ex. 44 — Faça um programa que leia uma frase (ou texto) e diga a média do tamanho das palavras do texto.

Ex. 45 — Faça um programa que leia cédulas de um arquivo, cada uma no seguinte formato:

```
eleitor: nnnnn
voto: vvvvv
```

Versão preliminar

E compute o vencedor da eleição. Se uma cédula tiver valor inválido nos campos `nnnnn` ou `vvvvv`, ela deve ser contada como nulo.

Ex. 46 — Modifique o Exercício anterior para usar algum método que satisfaça o critério de Condorcet na eleição.

Ex. 47 — A meia-vida biológica de uma substância é o tempo necessário para que sua concentração no sangue diminua pela metade. Por exemplo, o Salbutamol[1] tem meia-vida de 1.6h – o que significa que se em um momento t a concentração de Salbutamol no sangue é x, no momento $t + 1.6h$ será $x/2$.
Faça um programa Scheme que pergunte ao usuário:

- a meia-vida biológica de uma substância,
- a concentração inicial,
- um intervalo de tempo para a simulação,

e depois simule a evolução da concentração pelo período de tempo informado.
Por exemplo, a meia-vida biológica do Clorambucil[2] é 1.5h. Se informarmos ao programa uma concentração inicial de "100", a meia-vida de 1.5 e um intervalo de tempo de 12h, ele deverá mostrar os valores de concentração como mostrado abaixo:

```
Digite a concentração inicial, a meia-vida e o tempo
 para a simulação:
100
1.5
12

simulação:
tempo -- concentração
0  --  100.0
1.5  --  50.0
3.0  --  25.0
4.5  --  12.5
6.0  --  6.25
7.5  --  3.125
9.0  --  1.5625
10.5  --  0.78125
12.0  --  0.390625
```

Tente isolar procedimentos de entrada e saída em poucas partes do programa.

1 O Salbutamol é usado para o alívio de broncoespasmos

2 Clorambucil é uma droga usada em quimioterapia para tratamento de leucemia

Versão Preliminar

Ex. 48 — Modifique o exercício 47 para que ele gere uma string com o relatório.

Ex. 49 — Usando o gerador de XML que construímos neste Capítulo, implemente o procedimento `xml->string`, que recebe nossa representação de XML como S-expressões e devolve uma string. Não duplique código: faça pequenas alterações nos procedimentos que já existem.

Ex. 50 — Crie um formato, usando S-expressões, para armazenar dados de uma planilha de cálculo. Inicialmente, pense em armazenar em cada célula somente números ou texto. Escreva um programa Scheme que exponha uma API que permita construir a planilha, alterá-la e gravar no formato CSV.

Ex. 51 — Modifique o formato de planilha de cálculo do Exercício 50 para que seja possível armazenar também em cada célula S-expressões que fazem referência a outras células e onde seja possível usar procedimentos matemáticos de Scheme.

Ex. 52 — Use a API construída nos exercícios 50 e 51 para fazer uma planilha de cálculo em modo texto (a cada modificação do usuário, a planilha é mostrada novamente).

Ex. 53 — Mude o gerador de XML para que ele escreva tags vazias no formato abreviado. Por exemplo, ao invés de `<br></br>` ele deve escrever `<br />`.

Ex. 54 — Se você conhece gramáticas livres de contexto, construa um parser para XML, compatível com o gerador que criamos.

Ex. 55 — Faça um programa que leia um arquivo com texto e conte a frequência das palavras. Depois, o programa deve gerar um arquivo SVG com cada palavra do texto (sem repetição), onde o tamanho da fonte é proporcional à frequência da palavra. Depois tente implementar as seguintes modificações:

- Use cores aleatóreas;
- Use cores em um gradiente, sendo que a cor de cada palavra depende também da sua frequência;
- Faça as palavras serem dispostas em posições aleatóreas na imagem;
- Faça as palavras serem dispostas em posições aleatóreas, mas sem que uma fique sobre a outra.

Ex. 56 — Torne o gerador de triângulos de Sierpinski mais flexível: ele deve gerar triângulos não equiláteros (pense em que argumentos você deve passar ao procedimento).

Ex. 57 — Faça o gerador de triângulos de Sierpinski usar cores para os diferentes triângulos: as cores podem ser determinadas de diferentes maneiras. Por exemplo:

Versão Preliminar

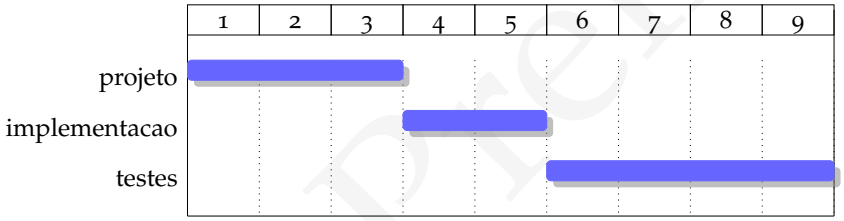
- A cor pode depender da iteração corrente
- A cor de cada triângulo pode depender das cores das três arestas que ele toca (o primeiro triângulo deveria ter arestas de cores diferentes).

Ex. 58 — Quando geramos um triângulo de Sierpinski com largura e altura definidas para a imagem e tentamos iterar indefinidamente, haverá uma iteração em que a imagem não mudará, porque sua resolução é limitada. Modifique o gerador de triângulos de Sierpinski para que ele não pergunte mais quantas iterações queremos, e itere apenas o número de vezes que for possível dado o tamanho da imagem.

Ex. 59 — Elabore um conjunto de procedimentos para criar gráficos de Gantt.
Por exemplo, veja a descrição a seguir.

```
(gantt (horizon 9)
       (task projeto 1 3)
       (task implemetacao 4 2)
       (task testes 6 4))
```

Essa descrição poderia se usada para gerar o seguinte gráfico de Gantt:



RESPOSTAS

Resp. (Ex. 58) — Verifique a cada iteração o tamanho dos lados do triângulo. Se um deles for menor que dois, pare.

Versão Preliminar

Versão Preliminar

3 ESTADO, AMBIENTE, ESCOPO E FECHOS

É vantajoso, tanto quanto possível, manter a propriedade de transparência referencial em programas. Há motivos, no entanto, para quebrar esta regra. Um deles é a construção de procedimentos mais eficientes através da modificação dos valores de variáveis (de que trataremos neste Capítulo). Esta é a visão de variáveis na programação imperativa, onde o conceito fundamental não é o de *função*, e sim o de *instrução:* um programa imperativo não é um conjunto de funções que transformam valores, e sim um conjunto de instruções que modificam dados em posições da memória.

3.1 MODIFICANDO O ESTADO DE VARIÁVEIS

Conceitualmente, temos até agora duas funções relacionadas a variáveis, seus nomes e valores: o *ambiente* associa nomes a lugares, e a *memória* associa lugares a valores. A forma especial `define` extende o ambiente global, incluindo nele mais um vínculo entre nome e lugar; já a forma especial `set!` é usada em Scheme para modificar o conteúdo de uma variável.

```
(define a 10)
a
10
(/ 1 a)
0.1
(set! a 0)
a
0
(/ 1 a)
ERROR: division by zero
```

Podemos trocar o valor armazenado em uma variável por outro de tipo diferente:

```
(string-append a "D'immenso")
ERROR in string-concatenate: invalid type, expected string: 10
(set! a "M'illumino")
```


Versão Preliminar

```
a
"M'illumino"
(string-append a "D'immenso")
"M'illumino D'immenso"
```

3.2 QUADROS E AMBIENTES

O modelo de avaliação que usamos no Capítulo 1 não contempla a possibilidade de modificarmos o valor de uma variável: símbolos eram usados para dar nomes a valores, e o vínculo entre nome e valor nunca mudava. Por isso pudemos ter certeza de que procedimentos sempre produzirão os mesmos resultados se chamados com os mesmos argumentos.

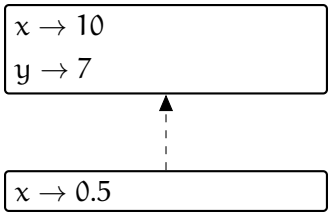
Trocaremos este modelo por outro, onde trataremos variáveis não como nomes associados a valores, mas como *lugares* na memória do computador onde podemos armazenar valores.

> Um *quadro* representa a associação entre nomes e locais. Há um quadro chamado *global*, que sempre é visível por todas as formas; a forma `lambda` cria novos quadros, contendo ligações para os nomes de seus parâmetros.

```
(define x 10)
(define y 7)

((lambda (x)
   (* x y))
 0.5)
3.5
```

A figura a seguir ilustra a situação deste exemplo: há no ambiente global dois vínculos, `x`$\rightarrow$ 10 e `y`$\rightarrow$ 7. Ao ser avaliada, a forma `lambda` cria um novo quadro com mais um vínculo `x`$\rightarrow$ 0.5. Dentro desta forma, para avaliar os argumentos de `*`, o interpretador buscará os nomes primeiro no quadro local, e depois no quadro anterior (que neste caso é o quadro global). Assim, os valores usados para `x` e `y` serão 0.5 e 7.

Versão Preliminar



As formas especiais `let`, `let*` e `letrec` são "açúcar sintático" para `lambda`, por isso também criam quadros com as variáveis que introduzem.

Sabendo os comprimentos dos lados de um triângulo, podemos calcular sua área usando a fórmula de Heron:

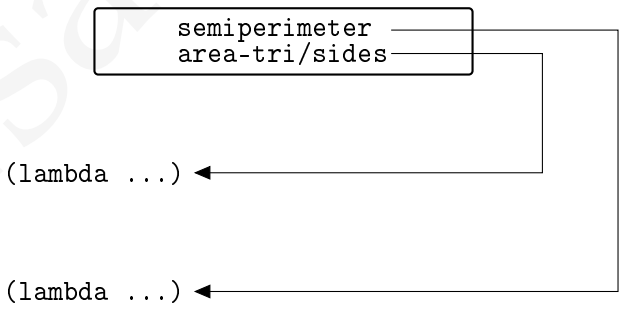
$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

onde s é o *semiperímetro*, $(a+b+c)/2$.

Os dois procedimentos Scheme a seguir implementam a fórmula de Heron.

```
(define semi-perimeter
  (lambda (s1 s2 s3)
    (/ (+ s1 s2 s3) 2)))


(define area-tri/sides
  (lambda (a b c)
    (let ((s (semi-perimeter a b c)))
      (sqrt (* s (- s a)
                 (- s b)
                 (- s c))))))
```

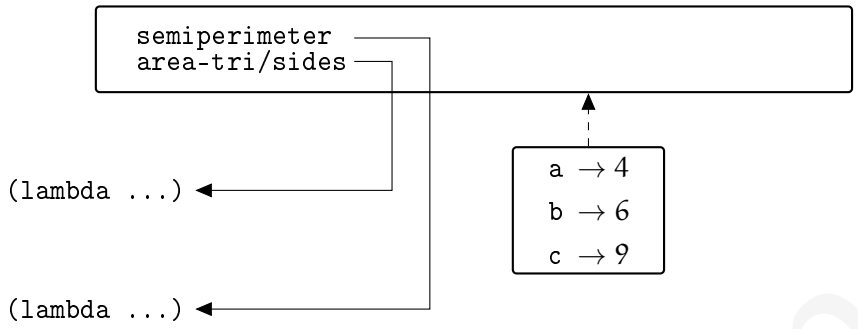
Ao definirmos estes dois procedimentos no REPL, adicionamos seus dois nomes ao ambiente global, como mostra a figura a seguir:



As setas partindo dos nomes somente mostram que os valores nestes locais são ponteiros para procedimentos; trataremos destes detalhes na seção 3.6.

Versão Preliminar

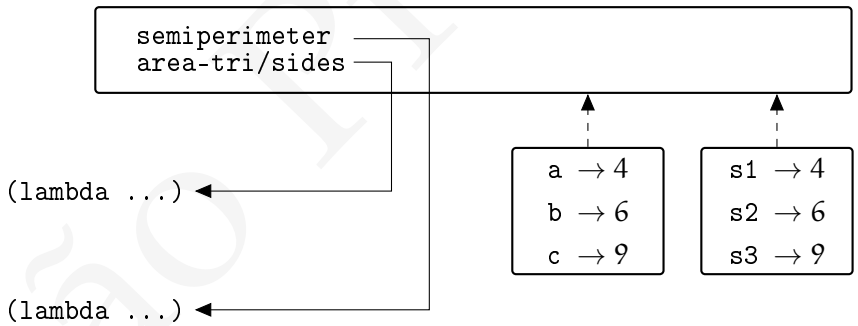
Quando aplicamos o procedimento `area-tri/sides` a três argumentos (por exemplo, 4, 6 e 9), um novo quadro é criado com os vínculos para os argumentos `a`, `b` e `c`:



A linha pontilhada mostra o caminho que o interpretador seguirá caso não encontre um nome neste quadro. Como o procedimento `area-tri/sides` foi definido no REPL, o quadro imediatamente acima dele será o quadro global.

Este novo quadro e o ambiente global formam o *ambiente* onde a aplicação do procedimento será avaliada: durante a avaliação, todos os nomes serão procurados nestes dois quadros.

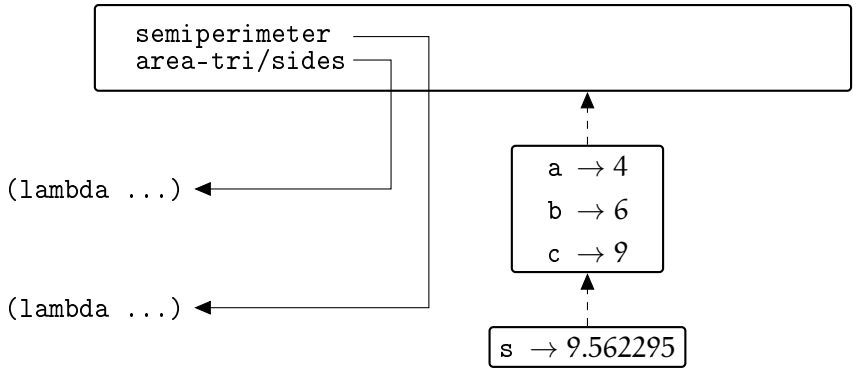
Ao encontrar a forma `(let ((s ...)))`, o interpretador precisará criar ainda outro quadro com um local para `s`. Para determinar o valor a ser armazenado no local denotado por `s`, o interpretador precisa invocar `semi-perimeter`, e outro quadro é criado:



Este quadro não está ligado aos anteriores, porque quando `semi-perimeter` foi definido, o ambiente em vigor era apenas o do quadro global. *O ambiente onde a forma* `semi-perimeter` *será avaliada é formado apenas por este novo quadro e o quadro global!* As variáveis `a`, `b`, `c` e `s` não são visíveis neste procedimento.

Quando o resultado de `semi-perimeter` é devolvido, o quadro onde ele foi avaliado é removido e o valor armazenado no local da variável `s`:

Versão Preliminar



Estes três quadros juntos formam o ambiente local da forma `let`. Quando as formas dentro do `let` foram avaliadas, os nomes serão procurados em cada quadro, na ordem induzida pelas setas pontilhadas – *que é exatamente a ordem em que as variáveis locais e argumentos aparecem no texto do programa!*

Após o cálculo da área, o interpretador nos devolverá o valor 9.56229574945264, e todos os quadros serão destruídos, exceto global.

Este novo modelo nos servirá neste Capítulo. Inicialmente não usaremos ainda procedimentos que tenham variáveis livres: *todas* as variáveis usadas em cada procedimento devem ser globais, passadas como parâmetro ou definidas dentro do procedimento (esta regra é a que se aplica também a programas C). É importante distinguir entre três tipos de ambiente:

- *Local*: contém vinculações para nomes declarados no bloco de código sendo avaliado;
- *Não-local*: contém vinculações para nomes declarados fora do bloco de código sendo avaliado;
- *Global*: contém vinculações para nomes que podem ser acessados em qualquer parte do programa. Estes nomes são declarados fora de qualquer bloco.

3.2.1 Escopo estático

O escopo de uma vinculação é a parte do programa em que ela é válida.

Usando *escopo estático*, a vinculação de um nome no ambiente é determinada pelo seguinte algoritmo:

i) Se o nome foi declarado no bloco sendo avaliado, aquela vinculação será usada. Caso contrário,

Versão Preliminar

ii) Se o nome não foi declarado no bloco em avaliado, ele deve ser buscado nos blocos que o envolvem, do imediatamente envolvente até o mais distante. Se todos os blocos envolventes tiverem sido verificados e a declaração não encontrada,

iii) Se o nome está no ambiente global, aquela vinculação será usada, caso contrário não há vinculação para aquele nome no ambiente.

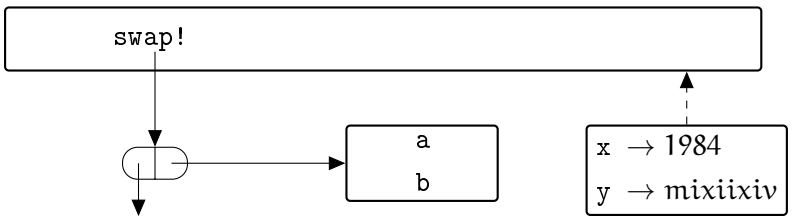
Pode-se informalmente dizer que o trecho de código onde um nome é visível é o bloco onde foi declarado e todos os blocos aninhados dentro dele, e por este motivo muitas vezes usa-se "escopo léxico" como sinônimo de "escopo estático".

3.2.2 Passagem de parâmetros por referência

Quando um procedimento é aplicado em Scheme ele recebe os valores resultantes da avaliação de seus argumentos, que são *copiados* em seu ambiente local. Não há como um procedimento modificar o valor de uma variável que não faça parte de seu ambiente, e o procedimento a seguir não funciona:

```
(define swap!
  (lambda (a b)
    (let ((tmp a))
      (set! a b)
      (set! b tmp))))

(let ((x 1984)
      (y 'mcmlxxxiv))
  (swap! x y)
  (display x))
1984
```

O diagrama mostrando os ambientes global, de `swap!` e do `let` mostra claramente em `swap!` as referências aos nomes `a` e `b` são resolvidas no quadro local. Na verdade não haveria como `swap!` ter acesso aos vínculos e modificar o conteúdo de `x` e `y`.



Versão Preliminar

Podemos no entanto modificar qualquer parte de uma estrutura: listas e vetores não tem seus elementos copiados quando passados como parâmetro; o ambiente Scheme passará uma referência à estrutura. É possível então construirmos um mecanismo de passagem de parâmetros por referência de maneira muito simples. No exemplo abaixo, o procedimento `box` constrói uma "caixa" com um elemento dentro; `unbox` retira o elemento da caixa, e `setbox!` modifica o conteúdo da caixa.

```
(define box list)
(define unbox car)
(define setbox! set-car!)
```

Poderíamos ter usado `cons` ao invés de `list`, para garantir que `box` somente aceitará um argumento:

```
(define box
  (lambda (x)
    (cons x '())))
```

Se passarmos para `swap!` duas listas, poderemos usar `set-car!` para modificá-las.

```
(define swap!
  (lambda (a b)
    (let ((tmp (unbox a)))
      (setbox! a (unbox b))
      (setbox! b tmp))))
(define x (box 'valor-de-x))
(define y (box 'valor-de-y))
(swap! x y)
(unbox x)
valor-de-y
(unbox y)
valor-de-x
```

3.2.3 Cuidados com o ambiente global

Poderíamos modificar nosso gerador de números aleatóreos para não precisarmos mais passar o valor anterior sempre que quisermos obter um número:

Versão Preliminar

```
(define aleat 112)

(define seed-random!
  (lambda (s)
    (set! aleat s)))

(define next-random!
  (lambda ()
    (let ((aleat-novo (linear-congruencial aleat
                                           1103515245
                                           12345
                                           (expt 2 32))))
      (set! aleat aleat-novo)
      aleat-novo)))
```

Infelizmente, esta solução depende da criação de uma variável `aleat` no ambiente global para armazenar o valor do último número gerado.

Se armazenarmos este código em um arquivo `random.scm`, um programador poderá mais tarde carregá-lo e usar nosso código. Se ele tiver uma variável `aleat` em seu programa, ela será modificada cada vez que um número aleatóreo for gerado (possivelmente dando ao programador algumas horas de dor de cabeça).

Além deste problema poderíamos encontrar outro: se usarmos nosso gerador em um programa com mais de uma thread, elas poderão tentar usar o gerador ao mesmo tempo, podendo deixá-lo em estado inconsistente, ou gerando o mesmo número para todas. Uma implementação melhor do gerador de números aleatóreos será apresentada na Seção 3.6.

O ambiente global é útil, no entanto, quando testamos pequenos trechos de programas Scheme. No resto deste texto restringiremos nosso uso de variáveis globais a pequenos exemplos e testes.

3.3 LISTAS

Além de lugares denotados diretamente por nomes, podemos também modificar valores dentro de estruturas.

Os procedimentos `set-car!` e `set-cdr!` modificam o `car` e o `cdr` de um par.

```
(define x (cons 'one 'two))
x
```


Versão Preliminar

```
(one . two)
(set-car! x 1)
x
(1 . two)
(set-cdr! x '())
x
(1)
```

A lista vazia, `'()`, é uma constante e não é possível modificar seus `car` e `cdr`.

Usando `set-car!` implementamos `map!`, uma versão de `map` que modifica a lista passada como argumento ao invés de criar uma nova lista.

```
(define map!
  (lambda (f lst)
    (if (not (null? lst))
        (begin (set-car! lst (f (car lst)))
               (map! f (cdr lst))))))
```

Procedimentos como `map!` que alteram estruturas de dados são muitas vezes chamados de *destrutivos*, e sua implementação é normalmente muito diferente de seus semelhantes não-destrutivos.

3.3.1 Modificações no primeiro elemento de uma lista

Nesta Seção modificaremos dois procedimentos puros que removem um elemento de uma lista. Embora os procedimentos não puros sejam mais eficientes no uso de tempo e memória, perceberemos que há uma série de dificuldades em sua elaboração, e o resultado final são dois procedimentos bem mais complexos do que suas contrapartes puras.

O procedimento `remove-first` recebe uma lista e um predicar `compare?`, e devolve uma nova lista onde a primeira ocorrência de algum item que satisfaça `compare?` é removida.

```
(define remove-first
  (lambda (lst compare?)
    (cond ((null? lst)           lst)
          ((compare? (car lst)) (cdr lst))
          (else
            (cons (car lst) (remove-first (cdr lst)
                                          compare?))))))
```


Versão Preliminar

```
(remove-first '(1 2 #\a 3) char?)
(1 2 3)
```

O procedimento `extract-first` é muito parecido com `remove-first`, mas retorna uma lista com dois elementos: o primeiro será `#f` quando nenhum item tiver satisfeito `compare?` ou uma lista contendo o elemento extraído; o segundo elemento será a nova lista, sem o elemento.

```
(define extract-first
  (lambda (lst compare?)
    (cond ((null? lst)
           (list '#f lst))
          ((compare? (car lst))
           (list (list (car lst))
                 (cdr lst)))
          (else
           (let ((res (extract-first (cdr lst) compare?)))
             (list (car res)
                   (cons (car lst)
                         (cadr res))))))))
(remove-first '(1 2 #\a 3 #\b) char?)
(1 2 3 #\b)
(extract-first '(1 2 #\a 3 #\b) char?)
((#\a) (1 2 3 #\b))
(extract-first '(1 2 #\a 3) port?)
(#f (1 2 #\a 3 #\b))
```

Fazer modificações destrutivas em uma lista é um pouco mais difícil do que pode parecer inicialmente, porque há dois casos que exigem atenção:

- Se a lista é vazia, não podemos usar nela o procedimento `set-car!`. Isto não é um problema para procedimentos que removem elementos, mas certamente é um problema quando queremos incluir elementos;
- Se a lista é unitária – por exemplo `(um-simbolo-solitario)` – e queremos remover seu único elemento, não podemos fazê-lo. Não há como um procedimento `remove-first!` transformar `(um-simbolo-solitario)` na lista vazia usando apenas `set-car!` e `set-cdr!`, porque eles apenas modificam o conteúdo de um par *que já existe*. Seria necessário fazer `(set! lst '())`. No entanto, como já discutimos

Versão Preliminar

na Seção 3.2.2 o ambiente que `remove-first!` pode modificar é local, e ele apenas mudará o valor de seu parâmetro.

Para tratar adequadamente destes casos precisamos passar a lista por referência para os procedimentos que a modificam.

O procedimento `extract-first!` que faremos retorna `#f` quando o elemento não é encontrado ou uma lista em caso contrário. A lista contém o elemento removido da lista. Não precisamos retornar também a lista, uma vez que a lista original é modificada e o chamador já tem acesso a ela.

O argumento `boxed-list` de `extract-first!` é a lista, passada por referência. Se a lista é unitária e seu único elemento satisfaz `pred?`, a caixa `boxed-list` é modificada e passa a conter a lista vazia. Se a lista tem mais elementos, mas o primeiro satisfaz o predicado, o valor do segundo é copiado sobre o primeiro, e o segundo é removido.

```
(define extract-first!
  (lambda (boxed-list pred?)
    (let ((lst (unbox boxed-list)))

      (define extract-first-aux! ...)

      (cond ((and (= (length lst) 1)
                  (pred? (car lst)))
             (setbox! boxed-list '())
             (list (car lst)))

            ((pred? (car lst))
             (let ((x (list (car lst))))
               (set-car! lst (cadr lst))
               (set-cdr! lst (cddr lst))
               x))

            (else
             (extract-first-aux! lst))))))
```

Em outros casos, `extract-first-aux!` é usado. Os membros da lista são comparados, iniciando com o segundo; quando um deles satisfaz `pred?` o anterior é removido (para isto mantemos uma referência para o anterior e verificamos o `cadr`).

Versão Preliminar

```
(define extract-first-aux!
  (lambda (lst)
    (cond ((< (length lst) 2)
           #f)
          ((pred? (cadr lst))
           (let ((x (list (cadr lst))))
             (set-cdr! lst (cddr lst))
             x))
          (else
           (extract-first-aux! (cdr lst))))))
```

Testamos agora os três casos. Primeiro, extraímos um elemento do meio de uma lista:

```
(let ((a (box (list 1 2 #\a 3 #\b))))
  (let ((x (extract-first! a char?)))
    (print x)
    (print (unbox a))))
```

```
(a)
(1 2 3 b)
```

Verificamos também que extrair do início da lista funciona:

```
(let ((a (box (list #\a 1 2 3 #\b))))
  (let ((x (extract-first! a char?)))
    (print x)
    (print (unbox a))))
```

```
(a)
(1 2 3 b)
```

E finalmente, `extract-first!` extrai corretamente o único elemento de uma lista.

```
(let ((a (box (list #\a))))
  (let ((x (extract-first! a char?)))
    (print x)
    (print (unbox a))))
```

```
()
(1 2 3 b)
```

Versão Preliminar

3.3.2 Listas circulares

É possível construir uma lista em que o `cdr` de um dos elementos é igual ao primeiro par da lista:

```
(define make-circular!
  (lambda (lista)
    (let loop ((l lista))
      (cond ((null? l)         lista)
            ((null? (cdr l))   (set-cdr! l lista)
                               lista)
            (else              (loop (cdr l)))))))
```

O procedimento `make-circular!` procura pelo último elemento da lista modifica seu `cdr` para que aponte para o primeiro elemento.

```
(define a (make-circular! (list 1 2)))
(pair? a)
#t
(list? a)
#f
(car a)
1
(cadr a)
2
(caddr a)
1
```

O construtor de listas `list`, quando chamado duas vezes com os mesmos argumentos, retorna listas diferentes:

```
(eqv? (list 1) (list 1))
#f
```

No entanto, o `car` e `caddr` de uma lista circular de dois elementos ficam no mesmo local na memória:

```
(define x (list 1))
(define y (list 2))
(define circular-2 (make-circular! (list x y)))
(eqv? (car circular-2) (caddr circular-2))
```

Versão Preliminar

```
#t
```

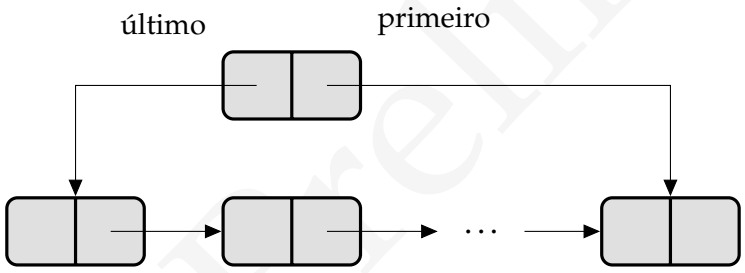
3.3.3 Filas

Como outro exemplo de mutação em listas, implementaremos uma fila, representada internamente como lista.

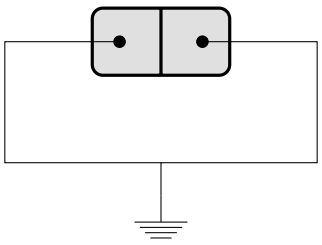
Ao implementar uma fila, precisaremos de referências para o primeiro *e o último* elemento. O `car` da fila apontará para o *último* elemento, e o `cdr` para o *primeiro*. Inicialmente ambos são a lista vazia.

Trocamos início com final: o início da fila será o final da lista, de forma que objetos serão incluídos no final da lista (após o último) e retirados do início. Para evitar confusão, usaremos o termo "lista interna" para a lista dentro da fila.

Trocamos início com final porque se usássemos a ordem natural de lista em Scheme, incluiríamos no começo, mas a remoção do último seria difícil.



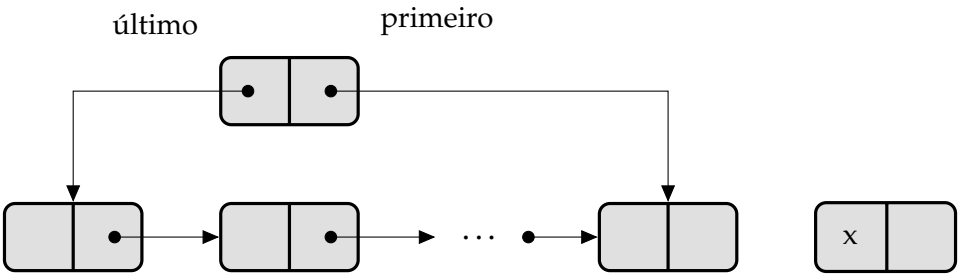
Uma fila vazia tem tanto `car` como `cons` apontando para a lista vazia.



```
(define make-q
 (lambda ()
   (cons '() '())))
```

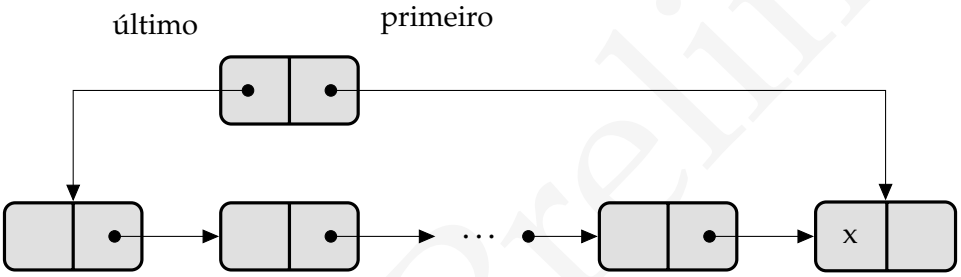
Para enfileirar um elemento, criamos uma nova lista com `(list e)`, que será incluída no final da lista interna já existente.

Versão Preliminar



Se a lista interna estava vazia, precisamos usar `(set-car! q new-last)`, para incluir a nova lista na posição da fila. Não podemos usar `set-car!` ou `set-cdr!` diretamente na lista vazia. Quando já há algum elemento na lista interna, modificaremos o `cdr` do último (que apontava para a lista vazia) para que aponte para a nova lista que criamos. Em seguida, modificamos a informação sobre o último. Se não modificássemos a lista interna e criássemos uma nova, teríamos que percorrê-la até o final e alocar outra lista.

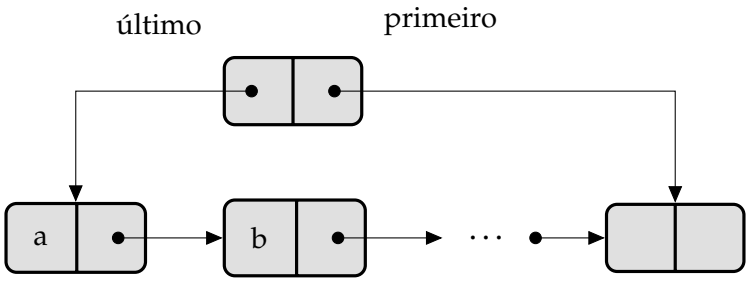
Após a mudança no `car` da fila:

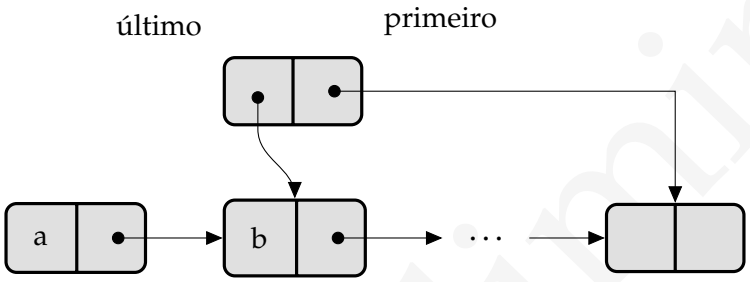



```
(define enqueue!
  (lambda (e q)
    (let ((queue (car q))
          (first  (cdr q)))
      (let ((new-first (list e)))
        (if (null? queue)
            (begin (set-car! q new-first)
                   (set-cdr! q new-first))
            (begin (set-cdr! first new-first)
                   (set-cdr! q new-first)))))))
```

Para desenfileirar, extraímos a informação do primeiro da lista interna de dados e depois avançamos a referência para o primeiro.

Versão Preliminar





Se após desenfileirar a lista interna ficar vazia precisamos mudar também o ponteiro para o último.

```
(define dequeue!
  (lambda (q)
    (let ((queue (car q)))
      (if (null? queue)
          (error "trying to dequeue from empty queue")
          (let ((res (car queue)))
            (set-car! q (cdr queue))
            (if (null? (car q))
                (set-cdr! q '()))
            res)))))
```

Como nossa fila é implementada como uma lista, o procedimento para verificar se a fila é vazia é trivial:

```
(define empty-q?
  (lambda (q)
    (null? (car q))))
```

Pode ser interessante termos um procedimento para encontrar um elemento no meio da fila.

Versão Preliminar

O procedimento `find-in-queue` usa o procedimento `member`, e retorna a sublista que inicia com o primeiro elemento para o qual `cmp?` retorna `#t`.

```
(define find-in-queue
  (lambda (q cmp?)
    (member (car q) cmp?)))
```

R^{7}RS

O procedimento `member` até o padrão R^{7}RS não aceitava o terceiro argumento `cmp?`.

```
(define q (make-q))

(begin
  (enqueue! 1 q)
  (enqueue! 2 q)
  (enqueue! #f q)
  (enqueue! 3.5 q)
  (enqueue! 4 q))
```

Como exemplo, procuramos por algum booleano: `(find-in-queue q boolean?)`

```
(#f 3.5 4 5.1 6)
```

Se procurarmos por um caracter, o resultado não será uma lista, mas o booleano `#f`:

```
(find-in-queue q char?)
#f
```

Se quisermos poder buscar um elemento na fila e extraí-lo (fugindo assim da disciplina de fila) podemos usar o procedimento `extract-first!` que desenvolvemos na Seção 3.3.1.

```
(define queue-extract!
  (lambda (q cmp?)
    (let ((the-list (box (car q))))
      (let ((x (extract-first! the-list cmp?)))
        (if (null? (unbox the-list))
            (begin (set-car! q '())
                   (set-cdr! q '())))
        x))))
```

Podemos notar que há em `queue-extract!` uma complexidade oriunda ainda de nosso uso de mutação da lista: temos que passar a lista por referência para `extract-first!`, e modificar `car` e `cdr` de `q` quando a lista volta vazia.

Versão Preliminar

3.3.4 Listas de associação

(FIXME: falta uma explicação detalhada para os procedimentos a seguir!)

Agora que temos métodos para modificar listas, podemos também querer modificar listas de associação, transformando-as em bases de dados. Criaremos três procedimentos para criar, modificar e consultar listas de associação.

Como não podemos modificar a lista vazia, usaremos uma lista contendo a lista vazia na criação de uma nova lista de associação:

```
(define make-alist
  (lambda () (list '())))
```

Para modificar uma lista, criamos o procedimento`alist-set!`.

- Se não há elementos (ou seja, se o `car` da lista é vazio), simplezmente mudamos o `car` para o par que estamos inserindo.
- Se a chave que queremos modificar já se encontra na lista (verificamos isso com `assoc`), usamos `set-cdr!` para modificar a lista, inserindo o novo elemento no início
- Se a chave não se encontra na lista, não fazemos nada

```
(define alist-set!
  (lambda (alist key value)
    (if (null? (car alist))
        (set-car! alist (cons key value))
        (let ((found (assoc key alist)))
          (if found
              (set-cdr! found value)
              (let ((old-car (car alist))
                    (old-cdr (cdr alist)))
                (set-car! alist (cons key value))
                (set-cdr! alist (cons old-car old-cdr))))))))
```

O procedimento `assoc` não funcionará na lista `(())`; precisamos criar um procedimento `alist-find`:

Versão Preliminar

```
(define alist-find
  (lambda (key alist)
    (if (null? (car alist))
        #f
        (assoc key alist))))
```

3.3.5 Árvores e grafos

(esta seção está incompleta)

Nesta Seção usaremos listas para construir estruturas não lineares (árvores e grafos). Estas estruturas serão construídas de forma que possamos percorrê-las andando por suas arestas, mas não permitirá acesso imediato a um nó a partir de seu nome: para encontrar um nó e determinar seu conteúdo e seus filhos ou vizinhos, será necessário primeiro "caminhar" pelo grafo até encontrá-lo.

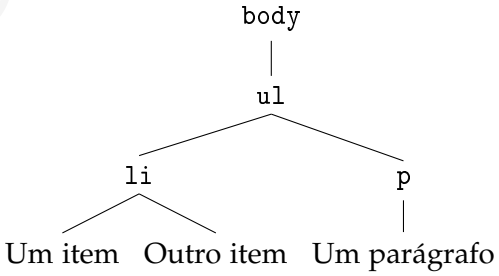
3.3.5.1 *Árvores*

Um nó de uma árvore pode ser representado como uma lista: primeiro elemento éo conteúdo do nó, e os outros são os filhos deste nó.

Considere o seguinte trecho de HTML.

```
<body>
  <ul> <li> Um item </li>
       <li> Outro item </li>
  </ul>
  <p> Um parágrafo </p>
</body>
```

A árvore abaixo representa este trecho.



Versão Preliminar

O nó `<p>`, por exemplo, seria representado como `(p ("Um parágrafo"))`. A árvore completa é mostrada abaixo.

```
(body
  (ul
    (li
      ("Um item")
      ("Outro item")))
  (p
    ("Um parágrafo")))
```

Esta árvore representa o trecho de código HTML a seguir.

```
<body>
    <ul>
        <li>Um item</li>
        <li>Outro item</li>
    </ul>
    <p>Um parágrafo</p>
</body>
```

Os procedimentos a seguir constituem uma interface para representação de árvores binárias desta maneira: são todos muito simples, apenas oferecendo uma barreira de abstração para `car`, `cadr` e `cddr`.

```
(define tree-make-node
  (lambda (data)
    (list data '() '())))

(define tree-data  car)
(define tree-left  cadr)
(define tree-right caddr)

(define tree-set-data! set-car!)
```

Para modificar o filho esquerdo, basta mudar o `car` do `cdr` do nó.

```
(define tree-set-left!
  (lambda (node child)
    (set-car! (cdr node) child)))
```

Versão Preliminar

Já o filho direito deve ser incluído como lista, para que o nó continue sendo também uma lista.

```
(define tree-set-right!
  (lambda (node child)
    (set-cdr! (cdr node) (list child))))
```

Os procedimentos para remover filhos precisam apenas modificá-los para que sejam iguais à lista vazia.

```
(define tree-delete-left!
  (lambda (node)
    (tree-set-left! node '())))

(define tree-delete-right!
  (lambda (node)
    (tree-set-right! node '())))
```

Podemos ainda modificar estes procedimentos para que funcionem com qualquer número de filhos por nó.

```
(define tree-make-node list)

(define tree-child-ref
  (lambda (node index)
    (list-ref node (+ 1 index))))

(define tree-set-child!
  (lambda (node index child)
    (set-car! (list-ref node (+ 1 index) child))))
```

3.3.5.2 *Grafos*

A representação que demos para árvores funciona com ciclos, e portanto também nos permite construir grafos[1].

1 Não incluímos aqui grafos desconexos.

Versão Preliminar

3.4 STRINGS

O procedimento `string-set!` pode ser usado para modificar uma string em uma determinada posição:

```
(define cidade "Atlantis")
(string-set! cidade 6 #\a)
(string-set! cidade 7 #\.)
cidade
"Atlanta."
```

```
(define string-map
  (lambda (proc . strings)
    (let ((len (string-length (car strings))))
      (let ((str-new (make-string len)))
        (do ((i 0 (+ i 1)))
            ((= i len) str-new)
          (string-set! str-new i
                       (apply proc (map (lambda (x)
                                          (string-ref x i))
                                        strings))))))))
(string-map char-upcase "Atlantis")
"ATLANTIS"
```

O procedimento `substring` retorna um pedaço da string recebida como argumento:

```
(substring "paulatinamente" 3 9)
"latina"
```

3.5 VETORES

Vetores são estruturas que mapeiam índices numéricos em objetos. Os objetos em um vetor podem ser de tipos diferentes.

Vetores constantes são representados em código Scheme da mesma forma que listas, exceto que o símbolo # é usado antes dos parênteses de abertura:

```
(quote (uma lista))
(uma lista)
```


Versão Preliminar

```
(quote #(um vetor))
#(um vetor)
```

O procedimento `make-vector` cria vetores com tamanho fixo:

```
(define v (make-vector 5))
```

Podemos passar um terceiro elemento para `make-vector`, que será usado como um valor inicial para todos os elementos do vetor.

Os procedimentos `vector-ref` e `vector-set!` são usados para ler e escrever em posições específicas de um vetor. O acesso a cada posição leva tempo constante (ao contrário do acesso ao n-ésimo elemento de uma lista, que normalmente leva tempo proporcional ao tamanho da lista).

```
(vector-set! v 0 10)
v
#(10 ? ? ? ?)
(vector-ref v 0)
10
```

Há um procedimento `vector->list` que transforma vetores em listas, e outro, `list->vector`, que faz a operação oposta.

```
(vector->list v)
(10 ? ? ? ?)

(list->vector '("abc"#\d #\e 10)
#("abc" #\d #\e 10)
```

3.5.1 Iteração com *do*

Embora iterar sobre vetores seja muito simples com *named let*, é hábito comum o uso de outra forma especial para fazê-lo.

A forma especial `do` aceita duas listas de argumentos seguidas de formas Scheme. A primeira lista descreve as variáveis do loop, suas formas iniciais e as funções que as modificam. A segunda lista descreve o teste de parada e o valor de retorno.

A forma geral do `do` é

Versão Preliminar

```
(do ( (var1 inicio1 modifica1)
      (var2 inicio2 modifica2)
      ... )
    (teste resultado)
  forma1
  forma2
  ... )
```

O exemplo a seguir inicializa `i` com 1, `x` com a lista vazia, e segue iterando até que `i` seja igual a dez. Após cada iteração, `i` passa a ser `(+ i 1)` e `x` passa a ser `(cons (* i 2) x)`.

```
(do ((i 1   (+ i 1))
     (x '() (cons (* i 2) x)))
    ((= i 10) 'final)
  (display i)
  (display ": ")
  (display x)
  (newline))
```

```
1: ()
2: (2)
3: (4 2)
4: (6 4 2)
5: (8 6 4 2)
6: (10 8 6 4 2)
7: (12 10 8 6 4 2)
8: (14 12 10 8 6 4 2)
9: (16 14 12 10 8 6 4 2)
final
```

O exemplo a seguir é um procedimento que mostra somente os elementos não zero de um vetor.

Versão Preliminar

```
(define mostra-nao-zeros
  (lambda (vet)
    (display "[ ")
    (do ((i 0 (+ i 1)))
        ((= i (vector-length vet)))
      (let ((e (vector-ref vet i)))
        (cond ((not (zero? e))
               (display "(")
               (display i)
               (display " -> ")
               (display (vector-ref vet i))
               (display ") "))))))
    (display " ]")))

(let ((v (make-vector 5 2)))
  (vector-set! v 2 0)
  (vector-set! v 3 1)
  (vector-set! v 4 0)
  (mostra-nao-zeros v))
```

```
[ (0 -> 2) (1 -> 2) (3 -> 1) ]
```

Definiremos um procedimento `vector-swap!` que troca dois elementos de um vetor, dados seus índices.

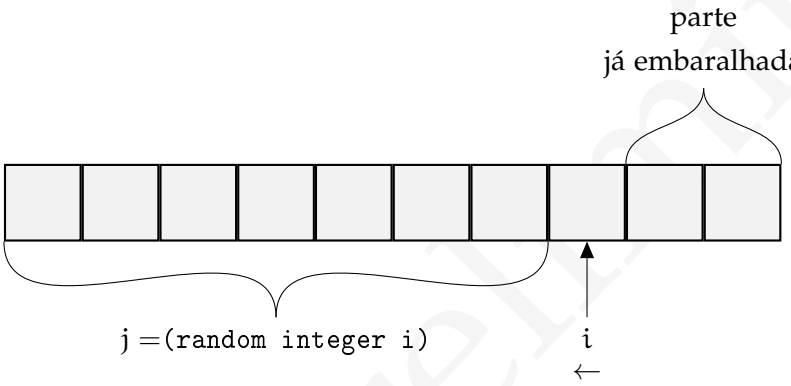
```
(define vector-swap!
  (lambda (vec i j)
    (let ((tmp (vector-ref vec i)))
      (vector-set! vec i (vector-ref vec j))
      (vector-set! vec j tmp))))
```

O procedimento `vector-shuffle!` modifica um vetor, deixando seus elementos em ordem aleatórea:

Versão Preliminar

```
(define vector-shuffle!
  (lambda (vec)
    (do ((i (- (vector-length vec) 1) (- i 1)))
        ((= i 0))
      (let ((j (random-integer i)))
        (vector-swap! vec i j)))))
```

O método usado em `vector-shuffle!` é simples: partimos do final do vetor e escolhemos algum elemento à esquerda. Trocamos o elemento atual com o escolhido; depois movemos o índice para a esquerda e recomeçamos.



Podemos exemplificar o uso de `vector-shuffle` criando um vetor de vinte elementos ordenados e aplicando o procedimento sobre ele.

```
(define v (make-vector 20))
;; após este DO, o vetor v será #(0 1 2 ... 17 18 19)
(do ((i 0 (+ i 1)))
      ((= i 20))
   (vector-set! v i i))
(vector-shuffle! v)
v
#(14 17 7 11 19 16 18 6 15 12 3 8 2 9 10 4 1 0 5 13)
```

O procedimento `vector-map` é análogo ao `map` para listas.

Versão Preliminar

```
(define vector-map
  (lambda (proc . vecs)
    (let ((len (vector-length (car vecs))))
      (let ((vec-new (make-vector len)))
        (do ((i 0 (+ i 1)))
            ((= i len) vec-new)
          (vector-set! vec-new i
                       (apply proc (map (lambda (x)
                                          (vector-ref x i))
                                        vecs))))))))
```

Uma variante `vector-map!` modifica o primeiro dos vetores, escrevendo nele o resultado da operação:

```
(define vector-map!
  (lambda (proc . vecs)
    (let ((len (vector-length (car vecs))))
      (let ((vec-new (car vecs)))
        (do ((i 0 (+ i 1)))
            ((= i len) vec-new)
          (vector-set! vec-new i
                       (apply proc (map (lambda (x)
                                          (vector-ref x i))
                                        vecs))))))))
```

O procedimento `vector-fold` aplica um procedimento a todos os elementos de um vetor, dois a dois:

```
(define vector-fold
  (lambda (proc init vec)
    (do ((i 0 (+ i 1))
         (res init (proc res (vector-ref vec i))))
        ((= i (vector-length vec)) res))))
```

O corpo do `do` em `vector-fold` é vazio: conseguimos escrever o que queríamos usando apenas as partes de inicialização e atualização das variáveis e o teste.

Versão Preliminar

3.5.2 Mais um gerador de números aleatóreos

(esta seção está incompleta)

O gerador de números aleatóreos apresentado como exemplo no Capítulo 1 apresenta diversos problemas. O algoritmo Blum-Micali, apresentado no Exercício 24 produz números de melhor qualidade, mas é muito lento.

Como exemplo de uso de vetores construiremos um gerador melhor para números aleatóreos usando o método da multiplicação com *carry* – que produz números de qualidade e é bastante rápido.

Dados uma base b (preferencialmente potência de 2), um multiplicador a e $r+1$ sementes (r resíduos de b $x_0, x_1, \cdots, x_{r-1}$, e um *carry* inicial $c_{r-1} < a$, o n-ésimo número é calculado da seguinte forma:

$$\begin{aligned} x_n &= (ax_{n-r} + c_{n-1}) \bmod b \\ c_n &= \left\lfloor \frac{ax_{n-r} + c_{n-1}}{b} \right\rfloor, \; n \geqslant r \end{aligned}$$

A saída do gerador é $x_r, x_{r+1}, \cdots$

Em nossa implementação manteremos os valores a, b, c e o índice i do próximo x_i a ser usado em um vetor junto com os valores dos x_i. Isto tornará mais conveniente o uso do gerador (precisaremos passar apenas um argumento ao gerar um novo número).

O procedimento `make-mwc` constrói um vetor que representa um gerador do tipo multiplicador com *carry*, incluindo ali seu estado. Neste exemplo o gerador sempre é criado com $a = 809430660$, $b = 2^{32}$ e $c = 362436$, mas estes três números poderiam ter sido passados como parâmetros (assim como as sementes, que são geradas usando um outro PRNG "`random`".

Versão Preliminar

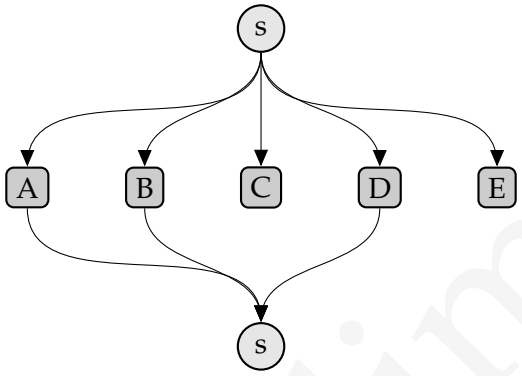
```
(define mwc
  (lambda (mwc-w/state)
    (let ((i (vector-ref mwc-w/state 0))
          (a (vector-ref mwc-w/state 1))
          (b (vector-ref mwc-w/state 2))
          (c (vector-ref mwc-w/state 3)))
      (let ((value (+ (* a (vector-ref mwc-w/state i)) c)))
        (vector-set! mwc-w/state i (modulo value b))
        (vector-set! mwc-w/state 3 (quotient value b))
        (vector-set! mwc-w/state 0
                     (if (= i (- (vector-length mwc-w/state) 1))
                         4
                         (+ i 1)))
        (vector-ref mwc-w/state i)))))


(define make-mwc
  (lambda ()
    (let ((a 809430660)
          (b (expt 2 32))
          (c 362436)
          (vec (make-vector 260)))
      (vector-set! vec 0 4) ;; start at 4
      (vector-set! vec 1 a)
      (vector-set! vec 2 b)
      (vector-set! vec 3 c)
      (do ((i 4 (+ 1 i)))
          ((= i 259))
        (vector-set! vec i (random (- (expt 2 32) 1))))
      vec)))
```

3.5.3 Exemplo: o esquema de compartilhamento de segredos de Shamir

Nosso próximo exemplo do uso de vetores é um pequeno programa para compartilhar segredos. Temos um número secreto que queremos esconder, mas gostaríamos que ele fosse revelado quando algumas pessoas de confiança decidissem fazê-lo. Distribuímos então "chaves" a estas n pessoas, e quando uma parte delas (um terço, metade, ou a

Versão Preliminar

quantidade que decidirmos) combinar as chaves, o número será revelado. O problema é que queremos que *qualquer* grupo de tamanho suficiente possa revelar o segredo. A figura a seguir exemplifica uma situação em que um segredo s é dividido entre cinco pessoas (A, B, C, D e E): cada uma recebe uma *partilha* do segredo. Quaisquer tres delas, podem juntar suas partilhas revelar o segredo – mas com menos de tres partilhas não é possível obter o segredo. No exemplo da figura, A, B e D reuniram suas partilhas e revelaram o segredo.



O esquema de compartilhamento de segredos que implementaremos foi desenvolvido por Adi Shamir em 1979, por isso o chamaremos de *SSSS (Shamir's Secret Sharing Scheme)*. A ideia chave é que com dois pontos conseguimos representar uma única reta; com três pontos, uma parábola; com quatro, um polinômio de grau 3 e, de maneira geral, podemos representar unicamente um polinômio de grau k usando $k+1$ pontos. Além disso, com um ponto a menos não há como adivinhar ou aproximar o polinômio de maneira eficiente: há infinitas retas passando pelo ponto $(2,3)$, e o mesmo acontece com polinômios de grau maior. Por exemplo, com os pontos$\{(2,3),(3,j),(4,6)\}$ temos, para $j=4$,

$$\frac{x^2}{2}-\frac{3x}{2}+4;$$

para $j=0$,

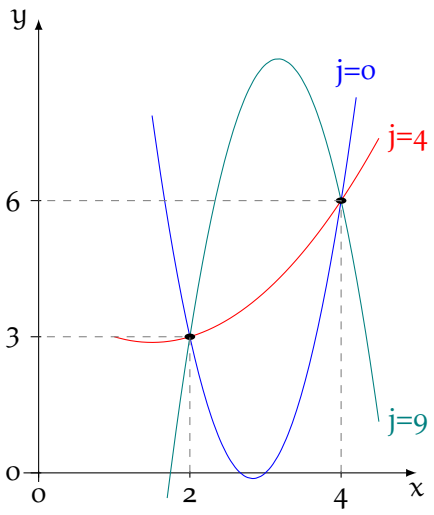
$$\frac{9x^2}{2}-\frac{51x}{2}+36;$$

para $j=9$,

$$\frac{-9x^2}{2}+\frac{57x}{2}-36.$$

Estas parábolas passando por $(2,3)$ e $(4,6)$ são ilustradas na figura a seguir:

Versão Preliminar



Se quisermos então compartilhar um segredo entre vinte pessoas, determinando que quaisquer 5 delas podem juntas revelar o segredo, simplesmente criamos um polinômio $a(x)$ de grau 4 cujo termo constante a_0 é o segredo. Damos um ponto de a para cada pessoa, e com cinco pontos conseguimos determinar o polinômio (e também o segredo).

Por razões que fogem ao objetivo deste texto, o SSSS usa aritmética modular – na verdade, aritmética módulo p, onde p é um número primo grande. Usaremos $p = 98322681213245072070809537 7479$.

Para compartilhar um segredo entre w partes com limiar igual a t:

1. Escolhemos aleatoreamente $t-1$ números menores que p, que chamaremos de $a_1, \cdots, a_{t-1}$. Temos agora um polinômio:

$$a(x) = a_0 + \sum_{j=1}^{t-1} a_j x^j \mod p$$

   onde a_0 é o segredo.

2. Para cada parte $1 \leqslant i \leqslant w$, calculamos $a(i)$ e entregamos a essa parte o par $(i, a(i))$.

Usaremos o procedimento `eval-poly` para obter o valor de uma função (dada por um polinômio) em um ponto:

Versão Preliminar

```
(define eval-poly
  (lambda (pol x)
    (let loop ((i 0))
      (if (= i (vector-length pol))
          0
          (+ (* (vector-ref pol i)
                (expt x i))
             (loop (+ i 1)))))))
```

Para combinar um segredo e distribuir chaves a n partes, sendo k delas suficientes para revelar o segredo, usamos o procedimento `ssss-combine-number`. Note que o segredo deve ser um número.

```
(define large-prime 98322681213245072070809537747 9)

(define ssss-split-integer
  (lambda (secret t n)
    (let ((coefs (make-vector (- t 1))))
      (vector-set! coefs 0 secret)
      (do ((i 1 (+ i 1)))
          ((= i (- t 1)))
        (vector-set! coefs i (random-integer large-prime)))
      (let ((shares (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((= i n))
          (vector-set! shares i
                       (cons (+ 1 i)
                             (modulo (eval-poly coefs (+ 1 i))
                                     large-prime))))
        shares))))
```

Para obter o segredo a partir de k chaves, usaremos o polinômio interpolador de Lagrange: dados os t pares $(x_i, a(x_i))$, o valor do polinômio a no ponto x é dado por

$$\begin{aligned} l(x) &= \sum_{j=1}^{n} l_j(x) \mod p \\ l_j(x) &= y_j \prod_{k=1;k\neq j}^{t} \frac{(x - x_k)}{(x_j - x_k)} \mod p. \end{aligned}$$

Versão Preliminar

O segredo é o termo constante a_0 do polinômio, por isso basta obtermos o valor do polinômio no ponto zero, $l(0)$. Isso simplifica a computação do segredo:

$$\begin{aligned} l(0) &= \sum_{j=1}^{n} y_j l_j(x) \mod p \\ l_j(x) &= \prod_{k=1;k\neq j}^{t} \frac{x_k}{(x_k - x_j)} \mod p. \end{aligned}$$

O procedimento `lagrange-aux` calcula $l_j(x)$:

```
(define lagrange-aux
  (lambda (keys j)
    (let ((n (vector-length keys))
          (prod 1)
          (xj (car (vector-ref keys j))))
      (do ((k 0 (+ k 1)))
          ((= k n) prod)
        (if (not (= j k))
            (let ((xk (car (vector-ref keys k))))
              (set! prod (* prod
                            (/ xk (- xk xj))))))))))
```

A recuperação do segredo é feita pelo procedimento `ssss-restore-integer`, que calcula $l(0)$.

```
(define ssss-combine-integer
  (lambda (keys)
    (let ((n (vector-length keys))
          (sum 0))
      (do ((j 0 (+ j 1)))
          ((= j n) (modulo sum large-prime))
        (let ((yj (cdr (vector-ref keys j))))
          (set! sum (+ sum (* yj (lagrange-aux keys j)))))))))
```

Agora testaremos o sistema de compartilhamento de segredos criando um segredo compartilhado por cinco pessoas; queremos que quaisquer três delas possam recuperar o segredo:

```
(ssss-split-integer 1221 3 5)
#((1 . 821975868960993690647660253447)
(2 . 660724925789536660587225128194)
```

Versão Preliminar

```
(3 . 499473982618079630526790002941)
(4 . 338223039446622600466354877688)
(5 . 176972096275165570405919752435))
```

Usamos pares `(x y)` dentro do vetor porque o número x, que varia de 1 a 5, é parte da chave compartilhada de cada participante. Usamos `ssss-restore-integer` para recuperar o segredo a partir dos fragmentos 2, 4 e 5:

```
(ssss-combine-integer '#((2 . 660724925789536660587225128194)
                         (4 . 338223039446622600466354877688)
                         (5 . 176972096275165570405919752435)))
1221
```

Podemos fazer o mesmo com 1, 2 e 3:

```
(ssss-combine-integer '#((1 . 821975868960993690647660253447)
                         (2 . 660724925789536660587225128194)
                         (3 . 499473982618079630526790002941)))
1221
```

3.6 FECHOS

Nesta seção retiraremos a restrição a procedimentos com variáveis livres que havíamos imposto no início do Capítulo. Uma característica importante de Scheme e de outras linguagens com suporte a procedimentos de primeira classe é a possibilidade de, ao passar um procedimento como parâmetro (ou retorná-lo), enviar junto com ele seu ambiente. O procedimento abaixo ilustra como isso pode ser feito.

```
(define retorna-procedimento
  (lambda ()
    (let ((uma-variavel 1000))
      (lambda () uma-variavel))))
```

Como Scheme implementa escopo estático, `uma-variavel` sempre será visível dentro dos blocos de código internos ao `let` que a definiu – inclusive o `(lambda () uma-variavel)`. Quando este procedimento é retornado por `retorna-procedimento`, ele continua podendo acessar `uma-variável`:

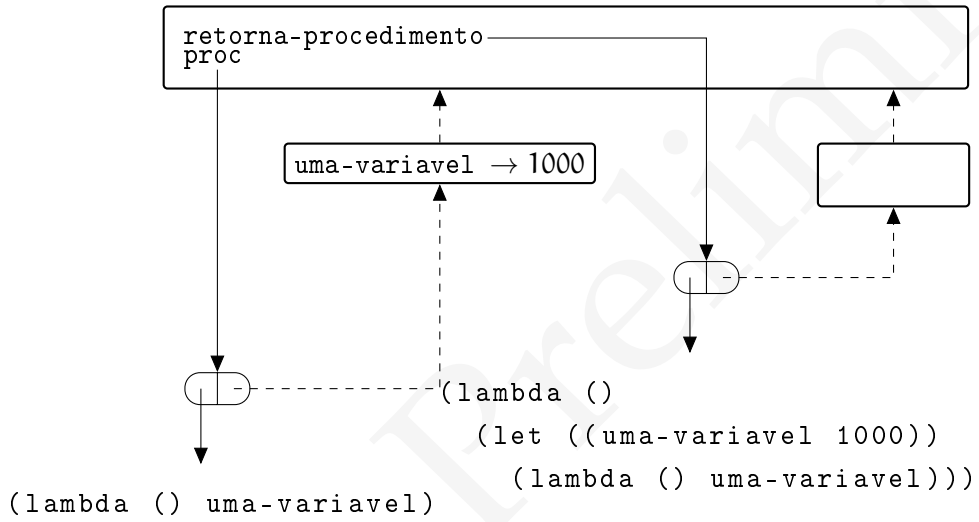
Versão Preliminar

```
(define proc (retorna-procedimento))
(proc)
1000
```

O nome dado ao procedimento `proc` (que leva junto seu ambiente léxico contendo a vinculação de `uma-variável`) é *fecho*[2]. Há programadores que chamam fechos de *let over lambda* ("let sobre lambda"), lembrando a maneira como são implementados[3].

Um *fecho* é composto de um procedimento e de seu ambiente léxico.

O diagrama a seguir mostra os quadros e ambientes após a avaliação de `(define retorna-procedimento ...)` e `(define proc ...)`.



Os retângulos são quadros; os procedimentos são representados pelos objetos com cantos arredondados, tendo no lado esquerdo a definição do procedimento e no lado direito uma referência ao ambiente que deve ser usado quando o procedimento for aplicado. As linhas contínuas definem vínculos de variáveis e as linhas tracejadas mostram a hierarquia de quadros.

O procedimento `faz-contador` retorna três fechos, que usam uma variável local de `faz-contador`:

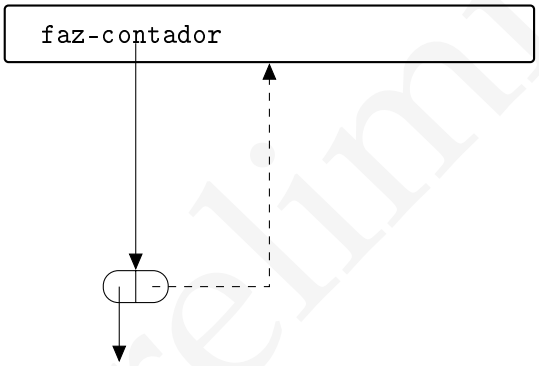
2 *Closure* em Inglês.

3 Let Over Lambda é também o título de um livro sobre técnicas avançadas de programação em Common Lisp[Hoyo8].

Versão Preliminar

```
(define faz-contador
  (lambda ()
    (let ((valor 0))
      (list (lambda ()
              valor)
            (lambda ()
              (set! valor (+ valor 1)))
            (lambda ()
              (set! valor (- valor 1)))))))
```

O diagrama a seguir ilustra o ambiente global após a definição de `faz-contador`.



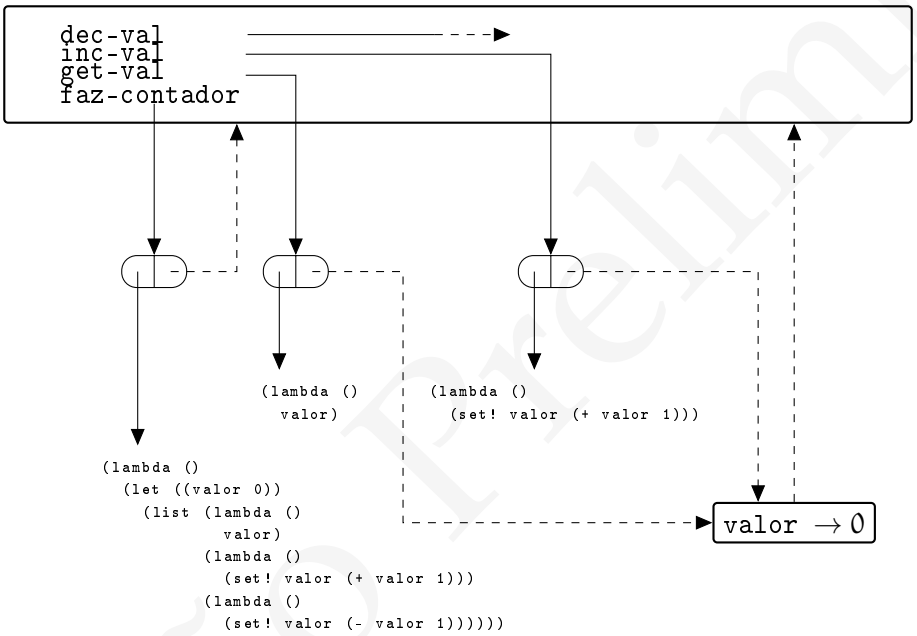

```
(lambda ()
  (let ((valor 0))
    (list (lambda ()
            valor)
          (lambda ()
            (set! valor (+ valor 1)))
          (lambda ()
            (set! valor (- valor 1))))))
```

Cada vez que `faz-contador` for chamado, uma nova instância da variável `valor` será criada. Os três procedimentos definidos dentro de `faz-contador` podem acessá-la; como estes três são retornados, eles podem ser usados *fora* de `faz-contador` para usar a variável`valor`:

Versão Preliminar

```
(define get-val #f)
(define inc-val #f)
(define dec-val #f)

(let ((procs (faz-contador)))
  (set! get-val (list-ref procs 0))
  (set! inc-val (list-ref procs 1))
  (set! dec-val (list-ref procs 2)))
```

A próxima figura mostra o ambiente logo após a execução da forma `let` acima (o valor de `dec-val` foi omitido por falta de espaço).



Os procedimentos `get-val`, `inc-val` e `dec-val` são visíveis no ambiente global (e portanto a partir do REPL), e todos fazem referência à variável `valor` no quadro que foi criado quando chamamos `faz-contador`.

```
(get-val)
0
(inc-val)
(get-val)
1
(dec-val)
(dec-val)
```

Versão Preliminar

```
(get-val)
-1
```

Outra chamada a `faz-contador` retorna um novo contador, com uma variável `valor` em um novo quadro.

```
(define g #f)
(define i #f)
(define d #f)

(let ((procs (faz-contador)))
  (set! g (list-ref procs 0))
  (set! i (list-ref procs 1))
  (set! d (list-ref procs 2)))
(g)
0
(i)
(i)
(get-val) ;; é diferente do anterior
-1
(g)
2
```

3.6.1 Um novo gerador de números aleatóreos

O gerador de números aleatóreos que desenvolvemos na Seção 1.3.5 é ligeiramente inconveniente: para usá-lo, temos que manter em uma variável o valor do último número aleatóreo gerado e repassá-lo ao gerador cada vez que precisarmos de um novo número. Podemos nos livrar deste problema construindo o gerador como um fecho. O procedimento `get-linear-congruential` recebe uma semente e retorna um gerador de números aleatóreos que pode ser chamado sem argumentos:

Versão Preliminar

```
(define get-linear-congruential
  (lambda (seed)
    (let ((next seed))
      (lambda ()
        (let ((n (linear-congruencial next
                                      1103515245
                                      12345
                                      (expt 2 32))))
          (set! next n)
          next)))))

(define next-random! (get-linear-congruential 1111))
```

Ao dar nome ao gerador retornado por `get-linear-congruential` usamos o sinal `!` porque de fato, cada vez que for aplicado o procedimento `next-random!` modificará o valor de uma variável.

3.6.2 Caixas e passagem por referência com fechos

Ao invés de listas para construir caixas e implementar passagem de parâmetros por referência podemos usar procedimentos anônimos para guardar valores. O exemplo a seguir é dado por Daniel Friedman e Mathias Felleisen em "The Seasoned Schemer"[FF95].

Uma caixa é um procedimento que recebe um valor `x` e retorna um fecho `s`. Este fecho recebe um procedimento `s` e o aplica com dois argumentos. Este procedimento deve ser um *seletor*: quando quisermos o valor de `x`, passaremos no lugar de `s` um procedimento que retorna seu primeiro argumento. Quando quisermos mudar o valor de `x`, passamos um procedimento que seleciona seu segundo argumento e o aplica.

```
(define box
  (lambda (x)
    (lambda (s)
      (s x   ;; primeiro argumento: retorna x
        (lambda (new)  ;; segundo: modifica x
          (set! x new))))))
```

Para modificar o valor na caixa, passamos a ela um procedimento que *seleciona seu segundo argumento* e o aplica com o argumento `new`:

Versão Preliminar

```
(define setbox!
  (lambda (box new)
    (box (lambda (x set)
           (set new)))))
```

Para obter o valor na caixa, passamos a ela um procedimento que seleciona seu *primeiro* argumento e o retorna.

```
(define unbox
  (lambda (box)
    (box (lambda (x set) x))))
(define x (box "A Tonga da Mironga"))
(define y (box "do Kabuletê"))
(swap! x y)
(unbox x)
"do Kabuletê"
(unbox y)
"A Tonga da Mironga"
```

3.6.3 Um micro sistema de objetos

No paradigma de programação orientada a objetos, a ideia central é modelar o mundo como objetos e organizá-los em classes. Cada objeto de uma mesma classe tem os mesmos atributos (variáveis locais, que somente são visíveis dentro do objeto) e objetos interagem trocando mensagens. Cada objeto tem uma lista de mensagens que pode responder – para cada mensagem ele implementa um *método*.

As variáveis que fechos Scheme "carregam" são semelhantes aos atributos de objetos. Para implementar um sistema de objetos simples em Scheme com fechos são necessários dois procedimentos:

- Um para criar um objeto de uma classe;
- Um para enviar uma mensagem a um objeto.

O exemplo a seguir mostra como um objeto *host* pode ser criado. O procedimento que cria o objeto é específico (não há ainda um `define-class`), mas ilustra o funcionamento do mecanismo.

Versão Preliminar

O procedimento `make-host` cria variáveis hostname e ip, depois retorna métodos para acessá-las.

```
(define make-host
  (lambda () ;; quando make-host for chamada,
    (let ((hostname "")  ;; Crie duas variáveis novas neste
          (ip ""))       ;; ambiente

      (lambda (msg)  ;; e retorne uma função de 1 argumento
                     ;; com acesso às variáveis
        (case msg
          ((get-hostname) (lambda () hostname))
          ((get-ip)       (lambda () ip))
          ((set-hostname) (lambda (new-name)
                            (set! hostname new-name)))
          ((set-ip)       (lambda (new-ip)
                            (set! ip new-ip)))))))))
```

A função devolvida por make-host é um fecho.

```
(define msg
  (lambda (obj . args)
    ((obj msg args))))
```

Damos então dar nomes aos métodos:

```
(define get-hostname
  (lambda (host)
    (msg host 'get-hostname)))

(define set-hostname
  (lambda (host name)
    (msg host 'get-hostname name)))
```

Este sistema de objetos pode suportar herança. Por exemplo, "server" pode herdar os atributos e métodos de host:

Versão Preliminar

```
(define make-server
  (lambda ()
    (let ((listen '(80 22))
          (my-host (make-host)))
      (lambda (msg)
        (case msg
          ((get-ports)
           (lambda () listen))
          ((add-service)
           (lambda (port)
             (set! listen (cons port listen))))
          ((remove-service)
           (lambda (port)
             (set! listen (delete port listen))))
          (else (my-host msg)))))))

(define s (make-server))

((s 'get-hostname))
((s 'set-hostname) "x")
((s 'get-hostname))
((s 'get-ports))
((s 'remove-service) 22)
((s 'add-service) 443)
```

Como o objetivo desta discussão era apenas o de ilustrar o mecanismo pelo qual fechos podem emular um sistema de objetos, não foram desenvolvidas facilidades como uma macro[4] `define-class`, que a partir de um nome de classe e lista de atributos geraria um procedimento para criar objetos da classe e outros, para acessar os atributos e métodos.

A possibilidade de implementar fechos está intimamente ligada à disciplina de escopo léxico, mas nem toda linguagem com escopo estático suporta fechos. Em C, por exemplo, uma função pode retornar um ponteiro para outra função – mas a função retornada não leva consigo o ambiente em que foi definida, porque em C não há como retornar um ponteiro para função que não tenha sido definida no nível base.

4 Uma macro é uma forma especial definida pelo usuário.

Versão Preliminar

3.6.4 Exemplo: gerenciador de *workflow*

3.7 ESCOPO DINÂMICO

Usando escopo dinâmico, a vinculação válida para um nome é a mais recentemente criada durante a execução do programa. Por exemplo,

```
(define proporcoes
  (lambda (valores)
     (map (lambda (v) (/ v total ))
          valores)))
```

Em Scheme a variável `total` teria que ser global, de outra forma não estará no ambiente de `proporcoes` (que inclui os nomes visíveis de acordo com regras de escopo léxico):

```
(let ((total 10)) (proporcoes '(2 4)))
Error: unbounded variable total
```

Em uma linguagem com escopo dinâmico o código acima produziria a lista `(1/5 2/5)`.

Há diferença entre escopo estático e dinâmico apenas quando da determinação de nomes não globais e não locais.

O uso de escopo dinâmico é usado normalmente para evitar a passagem de muitos parâmetros para procedimentos. Por exemplo,

```
(let ((x (calcula)))
  (let ((base 16))
    (display x)
    (newline))
  (display x))
```

Poderia ter o efeito de mostrar `x` em hexadecimal e depois usando a base que estava sendo usada antes. É importante notar que o uso de variáveis globais para emular escopo dinâmico, além de criar a possibilidade de conflitos de nomes no ambiente global, pode tornar o programa pouco legível:

Versão Preliminar

```
(let ((x (calcula)))
  (let ((old-base base))
    (set! base 16)
    (display x)
    (newline)
    (set! base old-base)
    (display x)))
```

Cada procedimento teria que se lembrar de modificar os valores de base, usando uma variável local (para entender porque `old-base` não pode ser global, basta imaginar o que aconteceria se `calcula` também mudasse `base` usando a mesma global `old-base`).

Uma solução seria determinar que o procedimento `display` aceitasse um parâmetro adicional, `base`:

```
(let ((x (calcula)))
  (display x 16)
  (newline)
  (display x 10))
```

No entanto, se há muitas variáveis a serem passadas dessa forma, o programa pode ficar ilegível. Por isso em algumas situações a primeira opção (o uso de escopo dinâmico) é usada.

Embora em Scheme o escopo seja estático[5], emular escopo dinâmico não é difícil.

O padrão R^{7}RS de Scheme descreve uma maneira de criar variáveis com escopo dinâmico[6], usando um procedimento `make-parameter`, semelhante à forma especial `define`, e uma forma especial `parameterize`, semelhante ao `let`.

O procedimento `make-parameter` cria variáveis, chamadas de objetos-parâmetro[7], que podem ter seu valor modificado usando disciplina de escopo léxico. O valor retornado por `make-parameter` é um procedimento que aceita um ou nenhum argumento.

```
(define base (make-parameter 10))
```

Quando `base` é chamado sem argumentos, retorna o valor do objeto-parâmetro. Quando é chamado com um argumento, o valor é alterado temporariamente e retornado:

```
(base)
10
```

5 Na verdade, com as primeiras implementações de Scheme Guy Steele e Gerald Sussman conseguiram mudar a opinião então comum, de que o escopo dinâmico era mais simples de implementar.

6 Esta maneira de implementar escopo dinâmico já era descrita na SRFI-39, e foi incorporada no padrão R^{7}RS.

7 *Parameter objects* em Inglês.

Versão Preliminar

```
(base 2)
2
(base)
10
```

A forma especial `parameterize` é semelhante ao `let`, mas as variáveis vinculadas devem ser objetos-parâmetro. Os valores destes objetos são modificados apenas durante a execução do trecho definido pelo escopo de `parameterize`, inclusive em precedimentos chamados a partir dali.

```
(define show
  (lambda (x)
    (display "x = ")
    (display (number->string x (base)))
    (newline)))
```

Como o parâmetro `base` é 10 por default, este valor será usado quando aplicamos `show`.

```
(show 20)
x = 20
```

No entanto, ao mudarmos `base` para 2, alteramos o comportamento de qualquer procedimento usado dentro do escopo do `parameterize`:

```
(parameterize ((base 2))
  (show 20))
x = 10100
```

Ao criar um objeto-parâmetro com `make-parameter` podemos determinar também um procedimento a ser usado sempre que o valor do objeto for mudado. Esse procedimento é passado como segundo argumento a `make-parameter`. Por exemplo, como o procedimento `number->string` só aceita bases inteiras entre 2 e 16, podemos verificar sempre se o valor passado para `parameterize` está correto. O procedimento `valid-base` verifica se uma base é válida.

```
(define valid-base
  (lambda (b)
    (if (and (integer? b) (<= 2 b 16))
    b
    (error "base inválida"))))
```

Versão Preliminar

Agora podemos usar `make-parameter` passando `valid-base` como segundo argumento:

```
(define base (make-parameter 10 valid-base))
```

Quando tentamos usar uma base inválida, `valid-base` produzirá um erro:

```
(parameterize ((base 0))
  (show 10))
Error: base inválida
```

Os primeiros sistemas Lisp usavam escopo dinâmico, mas Scheme e Common Lisp adotaram o escopo estático por default porque o escopo dinâmico traz aos programas diversos problemas, desde dificuldade de legibilidade até para a verificação de tipos (que não pode mais ser feita em tempo de compilação, ou antes da interpretação). Há, no entanto, aplicações de escopo dinâmico – por exemplo, a implementação de sistemas para suporte a programação orientada a aspectos[Cono3].

EXERCÍCIOS

Ex. 60 — Escreva versões destrutivas dos procedimentos pedidos no Exercício 12: `set-add!`, `set-union!` e `set-difference!` (os procedimentos devem alterar seus primeiros elementos). Explique porque não é possível usar exatamente a mesma representação de conjuntos usada naquele exercício.

Ex. 61 — Escreva um procedimento `remove!` que receba um elemento, uma lista e remova o elemento da lista, modificando-a.

Ex. 62 — Escreva um procedimento `append!` que receba várias listas e as concatene, alterando o último `cdr` de cada uma para que aponte para a próxima lista.

Ex. 63 — Mostre que `make-circular!` poderia ser trivialmente construído usando o procedimento `append!` do Exercício 62.

Ex. 64 — Escreva um procedimento `list-set!` que receba uma lista, um elemento, uma posição e troque o elemento naquela posição, modificando a lista.

Ex. 65 — Teste a implementação do esquema de compartilhamento de segredos de Shamir (desenvolvido na Seção 3.5.3), que usa um limiar t de n pessoas. Use o procedimento `with-combinations` que pede o exercício `ex-combinacoes` nos testes. Até que valor de t e n você consegue testar?

Versão Preliminar

Ex. 66 — A implementação do esquema de compartilhamento de segredos de Shamir na Seção 3.5.3 é um pouco inconveniente. Modifique-o, construindo alguma abstração que aceite cadeias de caracteres como segredos.

Ex. 67 — Mostre que o problema descrito na Seção 3.3.1 também surge ao implementar listas ligadas na linguagem C, quando usamos a definição para nó de lista e o protótipo de função para remover um elemento mostrados a seguir. C

```
// NULL representa a lista vazia.

struct list_node {
    void *data;
    struct list_node* next;
};

typedef struct list_node * list_node_ptr;

// Não funcionará para lista com um só elemento:
void* remove (list_node_ptr list, ...);
```

Mostre como resolver o problema.

Ex. 68 — Implemente `box`, `unbox` e `set-box!` usando vetores.

Ex. 69 — Escreva um procedimento `string-map!` que modifique cada caracter de uma string aplicando um procedimento, da mesma forma que `map!` para listas.

Ex. 70 — Faça um programa que encontre palavras em uma matriz de letras (as palavras podem estar em linhas, colunas ou diagonais). Tente não duplicar código.

Ex. 71 — Faça um programa que leia a descrição de um tabuleiro de xadrez (um caracter por peça; minúsculas para brancas, maiúsculas para pretas) e determine se um dos reis está em cheque.

Ex. 72 — Faça um procedimento `faz-ciclo` que receba uma lista de procedimentos `procs` e devolva um procedimento que devolve um dos procedimentos na lista `procs` (eles devem ser devolvidos em ordem, e quando chegar ao último voltar ao primeiro):

Versão Preliminar

```
(let ((ciclo (faz-ciclo + - * /)))
  (let ((a 5)
        (b 2))
    (let loop ((i 0))
      (if (< i 4)
          (let ((op (ciclo)))
            (display (op a b))
            (newline))))))
```

```
7
3
10
2.5
```

Ex. 73 — Escreva `vector-shuffle` – uma versão não destrutiva do procedimento `vector-shuffle!` descrito neste Capítulo. Seu procedimento deve criar um novo vetor, com os mesmos elementos do vetor dado, mas distribuídos uniformemente. Um requisito adicional interessante: se o gerador de números aleatóreos for alimentado com a mesma semente antes de `vector-shuffle` e de `vector-shuffle!`, o resultado deve ser idêntico:

```
v
#(0 1 2 3 4 5 6 7 8 9)
(random-seed x)
(vector-shuffle v)
#(8 4 5 9 2 6 1 0 3 7)
(random-seed x)
vector-shuffle! v
v
#(8 4 5 9 2 6 1 0 3 7)
```

(`random-seed` alimentará o gerador usado nos dois procedimentos de embaralhamento.)

Ex. 74 — Se você já cursou Linguagens Formais e Automata, diga (não precisa implementar) como poderia implementar um autômato finito determinístico que, em cada estado, chama uma função.

Ex. 75 — Desenhe diagramas mostrando os ambientes após a instanciação de dois fechos `host`, descritos neste Capítulo.

Versão Preliminar

Ex. 76 — Escreva o procedimento `juros-compostos-mem`, cujo funcionamento é descrito na Seção 1.3.3.1 (página 15).

Ex. 77 — Faça uma função que implemente um *closure* sobre as seguintes variáveis:

- Saldo: um valor numérico;
- Itens: uma lista de itens e quantidades. Por exemplo, `((banana 10) (notebooks 2) (saco-batata 35))`.

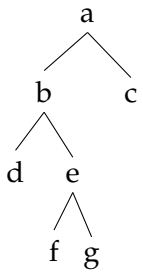
O criador do fecho deve aceitar valores iniciais para estas variáveis.
Este fecho pode representar um agente negociador. As funções que devem ser retornadas, e que usam as variáveis do fecho, são:

- Uma para comprar um item (se já existe na lista, some a quantidade; senão, adicione à lista);
- Uma para vender um item;
- Uma para verificar a lista de itens;
- Uma para verificar o saldo.

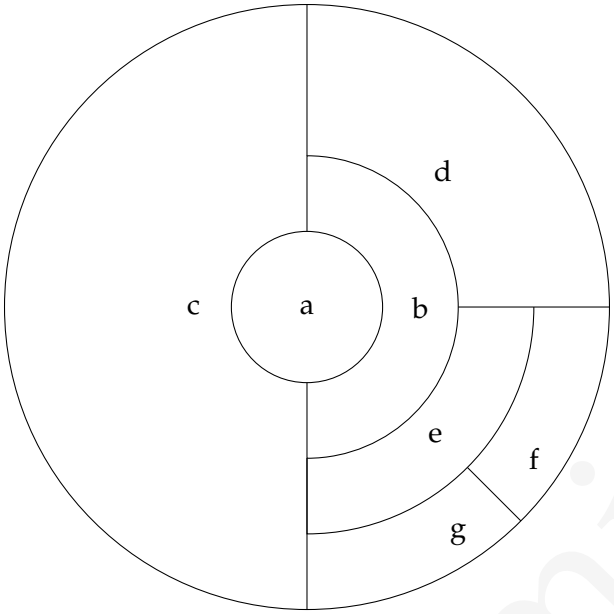
Ex. 78 — Faça um programa que instancie dois ou três fechos do exercício anterior, inicialize cada um diferentemente e depois faça cada um comprar ou vender aleatoreamente para o outro. Depois de `k` rodadas, mostre o estado de cada um. Há o problema de determinar o preço de cada item. use uma tabela global, e faça os agentes comprarem e venderem pelo preço global $\pm\delta$, onde `delta` é um número aleatóreo entre -10% e +10% do preço.

Ex. 79 — Mostre como implementar herança múltipla no sistema de objetos descrito neste Capítulo.

Ex. 80 — Crie um programa que leia uma árvore representada como lista Scheme e produza uma representação dela como circunferência, como no exemplo a seguir.



Versão Preliminar



RESPOSTAS

Resp. (Ex. 67) — Use ponteiro para ponteiro ao passar a lista para as funções.

Resp. (Ex. 68) — A implementação com vetores é muito simples:

```
(define box
  (lambda (x)
    (make-vector 1 x)))

(define unbox
  (lambda (b)
    (vector-ref b 0)))

(define set-box!
  (lambda (b x)
    (vector-set! b 0 x)))
```

Versão Preliminar

Resp. (Ex. 69) — Há várias maneiras de implementar `string-map!`. As soluções apresentadas aqui usam um índice `i` e avaliam `(string-set! s i (f (string-ref s i)))`:

```
(define string-map!
  (lambda (f s)
    (letrec ((string-map-aux!
              (lambda (f s i)
                (string-set! s i (f (string-ref s i)))
                (if (positive? i)
                    (string-map-aux! f s (- i 1))))))
    (string-map-aux! f s (- (string-length s) 1)))))

(define string-map!
  (lambda (f s)
    (let next ((i 0))
      (if (< i (string-length s))
          (begin
            (string-set! s i (f (string-ref s i)))
            (next (+ i 1)))))))

(define string-map!
  (lambda (f s)
    (do ((i 0 (+ i 1)))
        ((= i (string-length s)))
      (string-set! s i (f (string-ref s i))))))
```

Versão Preliminar

Versão Preliminar

4 BIBLIOTECAS MODULARES

Nos programas dos Capítulos anteriores usamos declarações (`import ...`) para termos acesso a alguns procedimentos. A versão R^{7}RS de Scheme define que os procedimentos padrão sejam organizados em módulos. Desta forma o programador pode carregar apenas os módulos que interessam ao seu programa (isto é particularmente importante, por exemplo, em sistemas embarcados). A divisão em módulos também permite ao implementador do ambiente Scheme escolher os procedimentos que incluirá em seu sistema.

Assim como os procedimentos padrão, qualquer programa escrito em Scheme pode ser dividido em módulos. Construir um programa inteiro de uma só vez é difícil porque há aspectos demais do programa interagindo, e o programador precisa trabalhar com informação demais a respeito das diferentes partes e funcionalidades do programa.

Separando um programa em diferentes módulos podemos também reusar estes módulos. Suponha que tenhamos escrito um programa que produz gráficos de barra mostrando a ocupação de discos rígidos, e como parte dele tenhamos desenvolvido a biblioteca para geração de gráficos vetoriais do Capítulo 2. Meses depois precisamos construir um programa que cria capas de livros em formato SVG. Se o primeiro programa tiver sido construído como um só bloco, o código que gera SVG provavelmente terá referências à geração de gráficos de barra e nos tomará algum tempo separar e reorganizar este código para que possamos reusá-lo. Se tivermos construído um "módulo SVG", poderemos simplesmente usá-lo no novo programa:

```
(import svg)

;; O código do gerador de capas de livros vai aqui, e pode
;; usar a biblioteca SVG já feita
```

Da mesma forma, algum tempo depois podemos precisar de um gerador de XML para alguma outra aplicação; se tivermos construído um módulo "XML" poderemos simplesmente usá-lo.

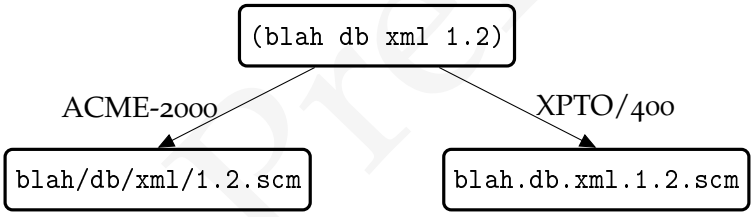
Versão preliminar

4.1 CONSTRUINDO MÓDULOS

Uma biblioteca deve ter um nome e conter diversas *declarações*.

```
(define-library (simbolo1 simbolo2 ...)
  ;; aqui ficam declarações que dizem quais outras
  ;; bibliotecas são usadas por esta, e também
  ;; quais procedimentos e variáveis desta biblioteca
  ;; podem ser vistos externamente.
  (begin
    ;; a implementação do módlo, com suas variáveis
    ;; e procedimentos, fica aqui. ))
```

O nome da biblioteca deve necessariamente ser uma lista de símbolos que identifique a biblioteca unicamente. Por exemplo, `(blah database xml 1.2)` poderia identificar uma biblioteca da empresa Blah (a versão 1.2 de uma biblioteca do grupo "database", que manipula bases de dados em XML). A Figura a seguir ilustra como essa biblioteca seria procurada em dois sistemas hipotéticos, ali chamados de "ACME-2000" (que parece usar diretórios no estilo Unix) e "XPTO/400".



Não há restrição ao uso e organização destas listas, exceto que os identificadores `scheme` e `srfi` não podem ser usados na primeira posição da lista em bibliotecas de usuários – por exemplo, não podemos usar o nome `(scheme physics)` para uma biblioteca de Física, mas podemos usar `(blah physics)`, onde o nome "`blah`" identifica a pessoa ou empresa que desenvolveu a biblioteca. Se tivermos duas bibliotecas, uma de Física para jogos 3D e outra para cálculo numérico usado em laboratório de Física, podemos definir as bibliotecas `(blah game physics)` e `(blah lab physics)`.

Uma implementação de Scheme pode não permitir a definição de bibliotecas diretamente a partir do REPL – pode ser necessário criar cada biblioteca em um arquivo, que deve ficar onde o sitema possa encontrá-lo.

Cada implementação de Scheme pode escolher como procurar uma biblioteca a partir de seu nome. Por exemplo, uma implementação pode interpretar `(blah database xml 1.2)` como o arquivo `1.2.scm` no diretório `blah/database/xml`, e outra pode interpretar como o arquivo `database_xml_1.2` no diretório `blah`.

Versão preliminar

O exemplo aseguir mostra uma biblioteca extremamente simples, que define o valor de π, vinculando-o ao símbolo `pi`:

```
(define-library (blah valor-de-pi)
  (import (scheme base))
  (export pi)
  (begin
    (define pi 3.14)))
```

Tivemos que explicitamente importar a biblioteca `(scheme base)` porque ela só é carregada automaticamente quando iniciamos o REPL, mas não está disponível para bibliotecas a não ser que declaremos explicitamente que a queremos.

Podemos testar a biblioteca carregando-a a partir do REPL:

```
pi
Error: unbound variable: pi
(import (blah valor-de-pi))
(pi)
3.14
```

O módulo `valor-de-pi` tem uma única declaração antes do `begin`: a linha "`(export pi)`" determina que o símbolo `pi`, interno ao módulo, seja visível por quaisquer programas ou módulos que façam `(import (blah valor-de-pi))`. Uma forma da declaração `export` é

```
(export nome1 nome2 ...)
```

O `export` indica que os símbolos `nome1`, `nome2` etc devem ser *exportados* pelo módulo, e portanto programas que usem este módulo terão acesso a estes símbolos e aos vínculos que foram criados pelo módulo.

Versão Preliminar

```
(define-library (blah my-math)
  (export pi e phi fib)

  (begin
    ;; O programa que usa esta biblioteca faz cálculos
    ;; usando pi, e e phi, por isso os definiremos e
    ;; exportaremos:
    (define pi  3.141592653589793)
    (define e   2.718281828459045)
    (define phi 1.618033988749895)

    ;; O programa também precisa calcular números de
    ;; Fibonacci, então exportamos este procedimento:
    (define fib
      (lambda (n)
        (/ (- (expt phi n)
              (expt (- 1 phi) n))
           (sqrt 5))))

    ;; fac não será exportada
    (define fac
      (lambda (n)
        (if (< n 2)
            1
            (* n (fac (- n 1)))))))))
```

A biblioteca `(blah my-math)` exporta os símbolos `pi`, `e`, `phi` e `fib`, mas *não* exporta `fac`, que não poderá ser usado:

```
(import (blah my-math))
pi
3.141592653589793
(/ 1 e)
0.367879441171442
```

Versão Preliminar

```
(do ((i 1 (+ i 1)))
          ((= i 11))
            (print i " " (fib i)))
1 1.0
2 1.0
3 2.0
4 3.0
5 5.0
6 8.0
7 13.0
8 21.0
9 34.0
10 55.0
(fac 5)
Error: unbound variable: fac
```

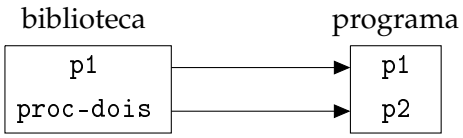
4.2 EXPORTANDO NOMES

Cada biblioteca pode exportar alguns de seus símbolos, vinculados a variáveis ou procedimentos. Há duas maneiras de exportar símbolos: uma é como no exemplo anterior, simplesmente declarando que o símbolo é exportado. A outra é renomeando o símbolo para que ele tenha outro nome fora da biblioteca:

```
(define-library (blah uma-biblioteca)
  (export p1
          (rename proc-dois p2))
  (import (scheme base)
          (scheme write))
  (begin
    (define p1
      (lambda () (display "Um")))
    (define proc-dois
      (lambda () (display "Dois"))))
```

Versão Preliminar

A próxima Figura ilustra como os procedimentos são vistos na biblioteca e no programa que os usa.



Podemos verificar na prática que `proc-dois` é renomeado quando importamos `uma-biblioteca`.

```
(import uma-biblioteca)
(p1)
Um
(p2)
Dois
(proc-dois)
ERROR: undefined variable: proc-dois
```

4.2.1 Exemplo: biblioteca de números pseudoaleatóreos

Construiremos uma biblioteca para geração de números pseudoaleatóreos usando os procedimentos que desenvolvemos na Seção 1.3.5.

Versão Preliminar

```
(define-library (blah random-numbers)
  (export (rename make-lc make-random))
  (import (scheme base))

  (begin
    (define linear-congruencial
      (lambda (x a b m)
        (modulo (+ (* a x) b) m)))

    (define make-lc
      (let ((state 0))
        (lambda (x)
          (set! state x)
          (list
            (lambda ()
              (set! state (linear-congruencial state
                                               1103515245
                                               12345
                                               (expt 2 32)))
              state)

            (lambda ()
              (set! state (/ (next-int-lc state)
                             (- (expt 2 32) 1)))
              state)))))))
```

Testamos agora o gerador:

```
(import (blah random-numbers))
(define r (make-random 20))
(define ri (car r))
(define rr (cdr r))
(ri)
595480765
(ri)
143664818
(ri)
```

Versão Preliminar

2966873603

4.3 IMPORTANDO NOMES

Até agora importamos bibliotecas inteiras, simplesmente usando `(import (blah biblioteca))`. Nesta Seção trataremos de como importar partes de bibliotecas, possivelmente mudando os nomes importados.

Uma declaração de importação é da seguinte forma:

`(import` *`declaracao-de-import`*`)`

Cada *declaracao-de-import* pode ser uma dentre:

i) Um nome de biblioteca. Todos os nomes exportados pela biblioteca serão importados da maneira como a biblioteca define.

ii) `(only` *declaracao-de-import* `nome ...)` – apenas uma lista de nomes será importada.

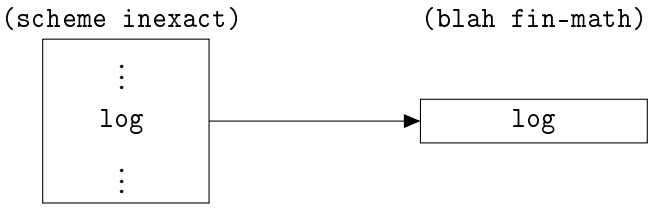
iii) `(except` *declaracao-de-import* `nome ...)` – apenas uma lista de nomes *deixará de ser* importada.

iv) `(prefix` *declaracao-de-import* `prefixo)` – os nomes serão importados, mas com um prefixo adicional.

v) `(rename` *declaracao-de-import1* `(nome nome2) ...)` – cada `nome` será renomeado para `nome2`.

Por exemplo, se uma biblioteca para Matemática Financeira usa, da biblioteca `(scheme inexact)`, apenas o procedimento `log`, ela pode importá-lo isoladamente:

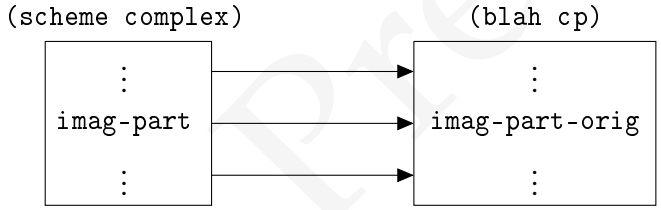
```
(define-library (blah fin-math)
  (import (scheme base)
          (only (scheme inexact) log))
   ...)
```



Versão Preliminar

No próximo exemplo uma variável é usada para guardar quantas vezes `imag-part` é chamado resultando em número diferente de zero.

```
(define-library (blah cp)
  (import (scheme base)
          (scheme inexact)
          (except (scheme complex) imag-part)
          (rename (scheme complex) (imag-part imag-part-orig)))
  (export get-imag-count imag-part real-part angle magnitude
          make-polar make-rectangular)
  (begin
    (define imag-count 0)
    (define (get-imag-count) imag-count)
    (define (imag-part x)
      (let ((y (imag-part-orig x)))
        (if (not (zero? y))
            (set! imag-count (+ 1 imag-count)))
        y))))
```



4.4 INCLUINDO ARQUIVOS

x

Uma biblioteca pode incluir diretamente conteúdo de algum arquivo, bastando que seja incluída em sua definição uma declaração de inclusão, que pode ser de uma das seguintes formas:

- `(include` *arquivo1 arquivo2* `...)` – os arquivos *arquivo1*, *arquivo2*, ... serão incluídos e processados como se fizessem parte do texto da biblioteca.
- `(include-ci` *arquivo1 arquivo2* `...)` – semelhante a `include`, mas não haverá diferenciação entre maiúsculas de minúsculas.

Versão Preliminar

Como exemplo, a biblioteca que definimos para números aleatóreos poderia ser especificada da seguinte forma:

```
(define-library (blah random-numbers)
  (export (rename make-lc make-random))
  (import (scheme base))
  (include "linear-congruential.scm"))
```

Assim podemos manter os procedimentos em um arquivo, `linear-congruential.scm`, sem ter qualquer preocupação com os possíveis conflitos entre os nomes dos procedimentos da biblioteca e nomes usados no resto do programa. Podemos usar os procedimentos como biblioteca simplesmente incluindo-os em um arquivo com uma declaração `define-library` semelhante a esta última.

4.5 EXPANSÃO CONDICIONAL

Muitas vezes uma biblioteca deve ser implementada de maneira diferente, dependendo do sistema (sistema operacional, hardware, implementação de Scheme etc) em que será usada. É possível identificar, durante a leitura do código fonte, diversas características do sistema, e então expandir o código da biblioteca de acordo com estas características. Para isso usamos a forma especial `cond-expand`. Por exemplo, `(cond-expand full-unicode)` será transformado em `$t` se o sistema Scheme sendo usado tem suporte complero a Unicode, mas será transformado em `#f` caso contrário. Essa transformação se dá antes de o programa ser interpretado ou compilado.

O `cond-expand` é sempre da seguinte forma.

`(cond-expand` *`clausula-cond-expand`* `...)`

Uma cláusula cond-expand é da forma

`(`*`requisito declaracao`*`)`

Um *requisito* pode ser:

- *feature*: a *feature* deve estar presente.3
- `(library` *biblioteca*`)`: significa que a biblioteca deve estar presente.
- `(and` *requisito* `...)`: significa que todos os requisitos devem ser satisfeitos.
- `(or` *requisito* `...)`: significa que algum dos requisitos deve ser satisfeito.

Versão Preliminar

- `(not` *requisito*`)`: significa que o requisito não pode ser satisfeito.

A última cláusula pode ser `(else` *declaracao-de-biblioteca* `)`.

Por exemplo, supondo que `complex` é uma característica, podemos primeiro verificar se a implementação de Scheme suporta números complexos antes de usá-los.

```
(define-library (blah norm)
  (import (scheme base)
          (scheme inexact))
  (cond-expand ((not complex) (import (blah cpx))))
  (cond-expand (complex (import (scheme complex))))
  (export norma)
  (begin
    (define (norm x)
        (sqrt (+ (* (real-part x)
                    (real-part x))
                 (* (imag-part x)
                    (imag-part x)))))))
```

Uma lista completa de *features* definidas no padrão R^{7}RS pode ser encontrada na Seção B.4.

EXERCÍCIOS

Ex. 81 — Crie módulos para as bibliotecas desenvolvidas nos Capítulos 1, 2 3 (cifra de César, banco de dados de filmes, gerador de XML, gerador de gráficos SVG e esquema de compartilhamento de segredos de Shamir). Mostre em exemplos como importar procedimentos destes módulos, possivelmente mudando seus nomes.

Versão Preliminar

Versão Preliminar

5 VETORES, MATRIZES E NÚMEROS

Este Capítulo aborda computação numérica com matrizes em Scheme.

5.1 MATRIZES

Scheme padrão não oferece qualquer suporte para a representação de matrizes, porque elas podem ser muito facilmente implementadas usando vetores: uma matriz $m \times n$ pode ser implementada como um vetor de tamanho mn, onde as linhas são dispostas consecutivamente.

Ao criar a matriz, alocamos um vetor de tamanho $mn + 1$. A última posição do vetor será usada para guardarmos o número de colunas:

```
(define make-matrix
  (lambda (m n)
    (let ((res (make-vector (+ 1 (* m n)) 0)))
      (vector-set! res (* m n)  n)
      res)))
```

Esta forma de representação é chamada de *row-major*.

A matriz

$$\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}$$

é representada na ordem *row-major* como `#(A B C D E F)`.

Se usássemos a ordem *column-major* listaríamos as colunas, uma de cada vez, e a mesma matriz seria representada internamente como `#(A D B E C F)`.

Usaremos *row-major* em nossa implementação. O número de elementos na matriz é o tamanho do vetor decrementado de um, e o número de linhas é o número de elementos dividido pelo número de colunas.

Versão Preliminar

```
(define (matrix-size x)
  (- (vector-length x) 1))

(define (matrix-cols x)
  (vector-ref x (- (vector-length x) 1)))

(define (matrix-rows x)
  (/ (matrix-size x) (matrix-cols x)))
```

O elemento na posição (i, j) da matriz é o elemento na posição $iC + j$, onde C é o número de colunas. Criamos então os procedimentos `matrix-ref` e `matrix-set!` para ler e modificar elementos da matriz.

```
(define (matrix-ref mat i j)
  (vector-ref mat (+ j (* i (matrix-cols mat)))))

(define (matrix-set! mat i j x)
  (vector-set! mat (+ j (* i (matrix-cols mat))) x))
```

Usaremos também um procedimento `read-matrix` que recebe uma porta e lê dela uma matriz. Os dois primeiros números lidos são os números de linhas e colunas; os outros são os elementos da matriz, linha a linha.

```
(define read-matrix
  (lambda (port)
    (let ((n (read port)))
      (let ((m (read port)))
        (let ((mat (make-matrix m n)))
          (do ((i 0 (+ 1 i)))
              ((= i m) mat)
            (do ((j 0 (+ 1 j)))
                ((= j n))
              (matrix-set! mat i j (read port)))))))))
```

Também precisaremos de um procedimento que mostre uma matriz em uma porta de saída.

Versão Preliminar

```
(define write-matrix
  (lambda (mat out)
    (let ((m (matrix-rows mat))
          (n (matrix-cols mat)))
      (write n out)
      (display #\space out)
      (write m out)
      (newline out)
      (do ((i 0 (+ 1 i)))
          ((= i m))
        (do ((j 0 (+ 1 j)))
            ((= j n))
          (write (matrix-ref mat i j) out)
          (display #\space out))
        (newline out)))))
```

Assim como fizemos com listas, strings e vetores, criaremos procedimentos `matrix-map` e `matrix-map!` para aplicar um procedimento em cada elemento de uma sequência de matrizes. A implementação destes procedimentos é semelhante à de seus análogos para vetores.

```
(define matrix-map
  (lambda (proc . mats)
    (let ((cols (matrix-cols (car mats)))
          (rows (matrix-rows (car mats))))
      (let ((mat-new (make-matrix rows cols)))
        (do ((i 0 (+ i 1)))
            ((= i rows) mat-new)
          (do ((j 0 (+ j 1)))
              ((= j cols))
            (matrix-set! mat-new i j
                         (apply proc (map (lambda (x)
                                            (matrix-ref x i j))
                                          mats)))))))))
```


Versão Preliminar

```
(define matrix-map!
  (lambda (proc . mats)
    (let ((cols (matrix-cols (car mats)))
          (rows (matrix-rows (car mats))))
      (let ((mat-new (car mats)))
        (do ((i 0 (+ i 1)))
            ((= i rows) mat-new)
          (do ((j 0 (+ j 1)))
              ((= j cols))
            (matrix-set! mat-new i j
                         (apply proc (map (lambda (x)
                                            (matrix-ref x i j))
                                          mats)))))))))
```

5.2 OPERAÇÕES COM VETORES E MATRIZES

Usando as variantes de `map` e `fold` para vetores podemos facilmente escrever procedimentos para realizar operações sobre vetores. Soma e multiplicação por escalar são uma aplicação trivial de `vector-map` (ou `vector-map!`).

```
(define vector+scalar
  (lambda (k v)
    (vector-map (lambda (x) (+ x k)) v)))


(define vector*scalar
    (lambda (k v)
    (vector-map (lambda (x) (* x k)) v)))


(define vector+scalar!
  (lambda (k v)
    (vector-map! (lambda (x) (+ x k)) v)))
```

A soma de vetores é igualmente simples.

Versão Preliminar

```
(define vector+vector
  (lambda (v1 v2)
    (vector-map + v1 v2)))


(define vector+vector!
  (lambda (v1 v2)
    (vector-map! + args)))
```

O produto interno de dois vetores requer um `vector-map` e um `vector-fold`.

```
(define vector*vector
  (lambda (v1 v2)
    (vector-fold + 0 (vector-map * v1 v2))))
```

A soma de matrizes com escalares ou com outras matrizes é semelhante às operações análogas para vetores.

```
(define matrix+scalar
  (lambda (k m)
    (matrix-map (lambda (x) (+ x k) m))))


(define matrix+matrix
  (lambda (a b)
    (matrix-map + a b)))
```

O procedimento a seguir realiza multiplicação de matrizes.

Versão Preliminar

```
(define matrix*matrix
  (lambda (a b)
    (let ((m (matrix-rows a))
          (o (matrix-cols a))
          (n (matrix-cols b)))
      (let ((c (make-matrix m n)))
        (do ((i 0 (+ 1 i)))
            ((= i m) c)
          (do ((j 0 (+ 1 j)))
              ((= j n))
            (set! val 0)
            (do ((k 0 (+ 1 k)))
                ((= k o))
              (set! val (+ val (* (matrix-ref a i k)
                                  (matrix-ref b k j))))
              (matrix-set! c i j val)))))))))
```

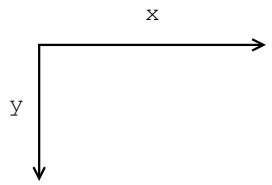
5.3 CRIANDO IMAGENS GRÁFICAS

(esta seção está incompleta)

Usaremos nossa implementação de matriz para armazenar imagens como matrizes de pixels.

O formato para armazenar as imagens será o Netpbm, descrito no Apêndice A. O procedimento `write-pbm` recebe uma imagem e uma porta de saída, e grava a imagem no formato Netpbm.

No formato PBM a origem fica no canto superior esquerdo da imagem e a coordenada (x, y) representa a posição x pontos à direita e y pontos abaixo da origem:



Preto e branco são representados respectivamente por um e zero, então devemos dar nomes a estas cores para que nosso programa fique compreensível.

Versão Preliminar

```
(define black 1)
(define white 0)
```

Uma imagem no formato PBM inicia com uma linha onde há apenas os caracteres `P1`, outra linha com o número de colunas e linhas da imagem, e em seguida os números representando os pixels, uma linha por vez.

```
(define write-pbm-1
  (lambda (m out)
    (display "P1" out)
    (newline out)
    (write-matrix m out)))
```

O procedimento `make-image` será idêntico a `make-matrix`.

```
(define make-image make-matrix)
(define image-rows matrix-rows)
(define image-cols matrix-cols)
```

O ponto $(0,0)$ da imagem representa o canto superior esquerdo. `(image-set! img i j c)` Modificará a cor do pixel (i,j) para `c`, sendo que `i` cresce da origem para baixo, e `j` cresce para a direita.

```
(define image-set! matrix-set!)
(define image-ref  matrix-ref)
```

Criaremos uma imagem 200×200 quadriculada; os quadrados terão 10 pixels de lado.

Para um pixel (x,y), os números $\lfloor\frac{x}{10}\rfloor$ e $\lfloor\frac{y}{10}\rfloor$ descrevem os valores de x e y correspondentes ao canto superior esquerdo do quadrado 10×10 onde o ponto se encontra. Plotaremos cada ponto somente quando as paridades destes dois números forem diferentes. Usaremos para isto um ou exclusivo, que não existe em Scheme padrão mas cuja implementação é extremamente simples:

```
(define xor
  (lambda (a b)
    (not (eqv? a b))))
```

O predicado `even-odd-square?` determina se um ponto está em um quadrado que deve ser plotado.

Versão Preliminar

```
(define even-odd-square?
  (lambda (i j)
    (define f
      (lambda (x)
        (even? (quotient x 10))))
    (xor (f i) (f j))))
```

Criaremos um procedimento `for-each-pixel!` que recebe como argumentos uma imagem e um procedimento que determina se um ponto deve ser plotado, e modifica os pixels da imagem dependendo do resultado da aplicação do procedimento sobre cada pixel.

```
(define for-each-pixel!
  (lambda (img pred)
    (let ((h (image-rows img))
          (w (image-cols img)))
      (do ((i 0 (+ i 1)))
          ((= i h))
        (do ((j 0 (+ j 1)))
            ((= j w))
          (if (pred i j)
              (image-set! img i j white)
              (image-set! img i j black)))))))
```

Finalmente, `draw-image` recebe a imagem e um nome de arquivo e grava a imagem naquele arquivo:

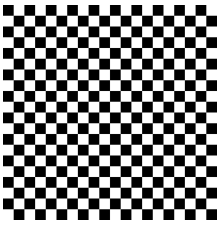
```
(define draw-image
  (lambda (img name)
    (let ((out (open-output-file name)))
      (write-pbm-1 img out)
      (close-output-port out))))
```

Agora podemos gerar nossa imagem quadriculada:

```
(let ((img (make-image 200 200)))
  (for-each-pixel! img even-odd-square?)
  (draw-image img  "quadriculado.pbm"))
```

O arquivo `quadriculado.pbm` conterá a seguinte imagem:

Versão Preliminar



5.3.1 Plotando funções

Para plotar funções usaremos o resultado da aplicação da função, *que pode ser inexato*, para determinar coordenadas de pixels da imagem. Será necessário então criar funções que transformam números inexatos em inteiros.

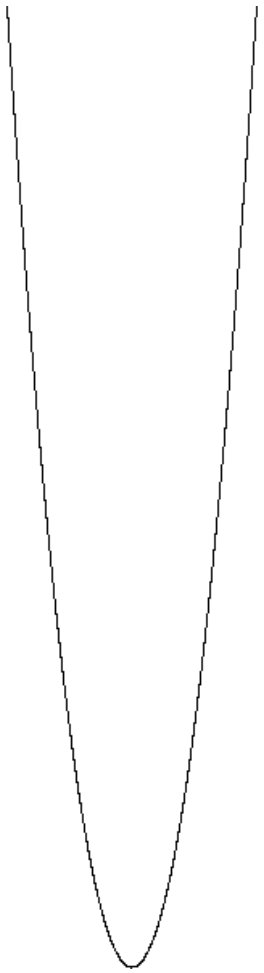
```
(define int-floor
  (lambda (x)
    (inexact->exact (floor x))))

(define int-ceiling
  (lambda (x)
    (inexact->exact (ceiling x))))


(define plot
  (lambda (fn from to y-min y-max step)
    (let ((x-size (int-ceiling (- to from)))
          (y-size (int-floor   (- y-max y-min))))

      (let ((img (make-image y-size x-size))) ;; trocamos x e y
        (for-each-pixel! img (lambda (x y) #f))
        (do ((x from (+ x step)))
            ((>= x to))
          (let ((y (fn x)))
            (if (and (< y y-max)
                     (> y y-min))
                (set-point! img
                            (int-floor (- y-size (- y y-min)))
                            (int-floor (- x from))
                            white))))
        img))))
```


Versão Preliminar

Testamos agora nossa função `plot`:

```
(draw-image (plot (lambda (z) (/ (* z z) 10))
                  -90 +90
                  -1 +600
                  0.05)
            "function.pbm")
```

O arquivo `function.pbm` conterá a imagem do gráfico da função $(x^2)/10$.



5.3.2 Rotação

Desenvolvemos nesta Seção um procedimento que realiza a rotação de uma imagem. O leitor encontrará boas introduções à Computação Gráfica nos livros de Luiz Velho [VG04] e de Foley, van Dam, Feiner e Hughes [Fol+95].

Versão preliminar

Para obermos uma rotação de θ radianos de uma imagem no sentido *horário*, tomamos cada ponto da imagem como um vetor coluna e calculamos sua nova posição multiplicando-o por uma *matriz de rotação*:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\text{sen}\,\theta \\ \text{sen}\,\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Podemos usar um procedimento que multiplique as matrizes, mas teríamos que tomar cada ponto (x, y) da imagem, inseri-lo em uma matriz 2×1, multiplicar por uma matriz quadrada, obtendo outra matriz 2×1 e extrair os novos pontos. É mais fácil obtermos formas fechadas para x' e y':

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\text{sen}\,\theta + y\cos\theta$$

Esta operação rotacionará a imagem para fora da região onde ela estava antes. Usaremos o procedimento a seguir para determinar se um pixel está dentro da imagem ou não.

```
(define pixel-inside
  (lambda (x y img)
    (and (>= x 0)
         (>= y 0)
         (< x (image-rows img))
         (< y (image-cols img)))))
```

O procedimento `rotate` operacionaliza exatamente a idéia que apresentamos.

```
(define rotate
  (lambda (img theta)
    (let ((img2 (make-image (image-rows img)
                            (image-cols im)))
          (ct (cos theta))
          (st (sin theta)))
      (for-each-pixel img
                      (lambda (x y v)
                        (let ((xx (int-floor (- (* x ct) (* y st))))
                              (yy (int-floor (+ (* x st) (* y ct)))))
                          (if (pixel-inside xx yy img2)
                              (set-point! img2 xx yy v)))))
      img2)))
```

Versão Preliminar

Testamos o procedimento:

```
(set! q (let ((img (make-image 50 600)))
          (for-each-pixel! img even-odd-square?)
          (draw-image img  "quadriculado.pbm")
          img))

(set! q2 (rotate q (/ pi 8)))

(draw-image q2 "rq.pbm")
```

5.3.3 Exemplo: conjuntos de Julia

Nesta Seção elaboraremos um procedimento que constrói uma família de fractais conhecidos como *conjuntos de Julia*. Richard Crownover [Cro95] oferece uma introdução básica a fractais e sistemas caóticos.

Para uma função f e um valor x_0, dizemos que a função iterada $f^{(n)}(x_0)$ é o valor $f(f(\cdots f(x_0)\cdots))$, onde há n aplicações de f. O procedimento a seguir calcula uma função iterada em um ponto (`stop?` é um predicado que, dado um valor, determina se o processo deve continuar ou não).

```
(define function-iterate
  (lambda (f x n stop?)
    (if (or (zero? n)
            (stop? x))
        x
        (function-iterate f (f x) (- n 1) stop?))))
```

Testaremos o iterador de funções: primeiro definimos uma função `/2`, que retorna a metade de um número; depois, pedimos dez iterações de `/2` para o número um, mas somente até o valor ser menor que 0.2:

```
(define (/2 x) (/ x 2))
(function-iterate /2 1 10 (lambda (x) (< x 0.2)))
1/8
```


Versão Preliminar

O procedimento iterou a função até 0.25, mas parou em $1/8 = 0.125$, que é menor que 0.2. Se usarmos `(lambda (x) #f)` para o parâmetro `stop?`, o procedimento somente terminará após as dez iterações:

```
(function-iterate /2 1 10 (lambda (x) #f))
```

```
1/1024
```

O conjunto de Julia de uma função f é a borda do conjunto de pontos z no plano complexo para os quais

$$\lim_{n\to\infty} |f^{(n)}(z)| = \infty$$

É comum denotar o conjunto de Julia de uma função por $J(f)$.

Trataremos aqui apenas de funções complexas quadráticas da forma $f(z) = z^2 + c$, onde c é uma constante. A norma de um número complexo é a distância entre sua representação no plano complexo e a origem: $|a + bi| = \sqrt{a^2 + b^2}$. Para as funções quadráticas com as quais lidaremos, se $|c| < 2$, $z \in \mathbb{C}$ e existe algum n tal que $|f^{(n)}(z)| \geqslant 2$, então $\lim_{n\to\infty} |f^{(n)}(z)| = \infty$. Podemos plotar o conjunto de Julia de uma função da seguinte maneira: Para cada ponto z no plano, calculamos os valores de $f(n)$ para algum n e verificamos se a norma torna-se maior ou igual a 2.

Construiremos um programa que, a partir de uma função quadrática complexa, desenha o conjunto de Julia *preenchido* para aquela função. O conjunto de Julia preenchido é o conjunto dos pontos para os quais a função iterada da forma que descrevemos não diverge (tomamos esta decisão porque este desenho parecerá mais interessante do que somente a borda do conjunto de Julia).

Precisaremos de um procedimento para calcular a norma de números complexos:

```
(define complex-norm
  (lambda (x)
    (let ((a (real-part x))
          (b (imag-part x)))
       (sqrt (+ (* a a)
                (* b b))))))
```

Nosso procedimento `in-julia` fará no máximo 150 iterações:

Versão Preliminar

```
(define in-julia?
  (lambda (f x max)
    (let ((v (function-iterate f x 150
                               (lambda (z)
                                 (> (complex-norm z)
                                    max)))))
      (< (complex-norm v) max))))
```

A primeira função para a qual plotaremos o conjunto de Julia é $f(z) = z^2 - 0.2 + 0.5\varphi i$.

```
(define phi (/ (+ 1 (sqrt 5)) 2))

(define complex-quad
  (lambda (x)
    (+ (* phi 0+0.5i)
       (- 0.2)
       (* x x))))
```

O procedimento `quadratic-in-julia?` aceita as duas coordenadas de um ponto, as dimensões da janela onde queremos plotar e determina se o ponto deve ser plotado ou não.

- Para plotarmos o conjunto nos quatro quadrantes, são feitas translações nos dois eixos, de maneira que o centro da janela seja passado para a função como $(0, 0)$;
- Multiplicamos os valores de x e y por um terço da largura da imagem, de forma que possamos visualizar a parte "interessante" da imagem;
- `in-julia?` é chamada, com o complexo formado pelo ponto (x, y) e com o limite de 2.

Versão Preliminar

```
(define quadractic-in-julia?
  (lambda (x y x-size y-size)
    (let ((half-x (/ x-size 2))
          (half-y (/ y-size 2))
          (scale-factor (/ 3 x-size)))
      (let ((tr-x (- x half-x))
            (tr-y (- y half-y)))
        (let ((sc-x (* tr-x scale-factor))
              (sc-y (* tr-y scale-factor)))
          (in-julia?  complex-quad
                      (make-rectangular sc-x sc-y)
                      2))))))
```

Agora só nos falta `make-julia`, que efetivamente constrói a imagem passando a função `quadratic-in-julia?` para `for-each-pixel!`

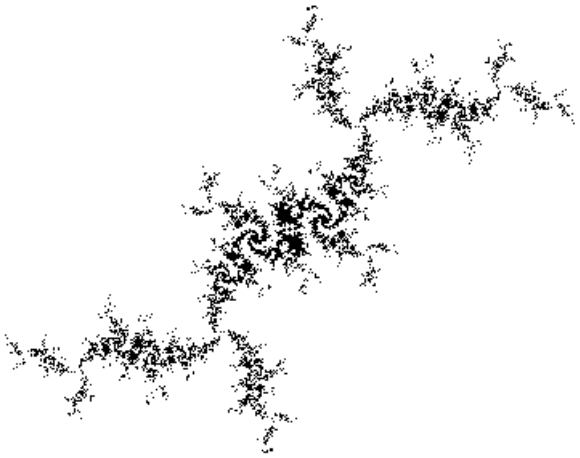
```
(define make-julia
  (lambda (x-size y-size fun)
    (let ((img (make-image x-size y-size)))
      (for-each-pixel! img
                       (lambda (x y)
                         (fun x y
                              x-size
                              y-size)))
      img)))
```

Geramos a imagem do conjunto de Julia para $f(z) = z^2 - 0.2 + 0.5\varphi i$:

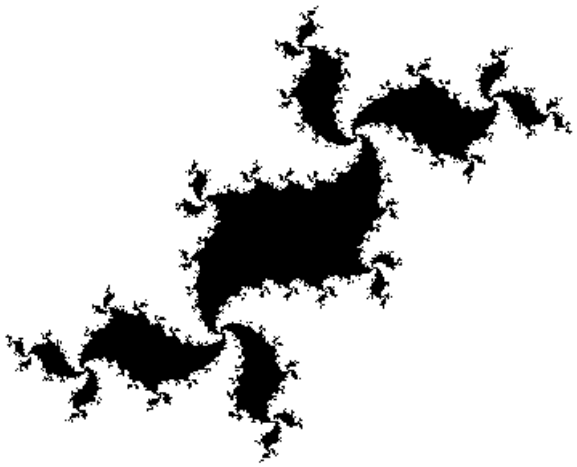
```
(let ((julia-set (make-julia 300
                             400
                             quadractic-in-julia?)))
  (draw-image julia-set
              "julia-set.pbm"))
```

O conteúdo do arquivo `julia-set.pbm` será a figura abaixo:

Versão Preliminar



Para $z^2 - 0.2 + 0.75\mathrm{i}$, a imagem gerada é



Outras funções que geram conjuntos de Julia interessantes são $\frac{\pi}{2}z^2 - 0.2 + 0.5\mathrm{i}$, $z^2 + -0.835 - 0.2321\mathrm{i}$, $z^2 - 0.11 + 0.65569999\mathrm{i}$ e $z^2 + (\varphi - 2) + (\varphi - 1)\mathrm{i}$.

EXERCÍCIOS

Ex. 82 — Faça uma função que aceite como argumento um vetor de números inteiros e retorne um outro vetor, com os mesmos elementos, mas em ordem diferente: todos os ímpares à esquerda e todos os pares à direita.

Ex. 83 — Escreva um programa que leia um arquivo com texto e mostre um histograma da frequência de cada palavra no texto.

Versão preliminar

Ex. 84 — Faça um programa que multiplica matrizes. Se você conhece o algoritmo de Strassen, implemente tanto o algoritmo ingênuo como o de Strassen.

Ex. 85 — Refaça os métodos de criação e acesso a matrizes usando ordem *column-major*.

Ex. 86 — Neste Capítulo descrevemos uma forma de implementação de matrizes usando um único vetor para representar uma matriz. Mostre como generalizar este esquema de representação para um número arbitrário de dimensões e implemente o esquema para 3 e 4 dimensões.

Ex. 87 — Implemente um esquema alternativo para representar matrizes: ao invés de usar um único vetor para cada matriz, uma matriz pode ser representada como um vetor de vetores.

Ex. 88 — Quando muitos elementos de uma matriz são iguais a zero, dizemos que a matriz é *esparsa*. Construa um esquema de representação de matrizes represente matrizes esparsas de maneira econômica, sem representar os zeros.

Ex. 89 — Escreva um programa que mostre um dos calendários: Islâmico (lunar), Judaico (lunissolar) ou Chinês (lunissolar). Use matrizes para representar o calendário.

Ex. 90 — Implemente um procedimento que calcule o determinante de uma matriz. Inicialmente, use expansão por cofatores. Depois tente implementar um algoritmo mais rápido [BH74].

Ex. 91 — Implemente uma biblioteca simples de Álgebra Linear para $\mathbb{R}^n$ e $\mathbb{C}^n$.

Ex. 92 — Reimplemente a biblioteca do Exercício 91 para que use qualquer corpo.

Ex. 93 — Implemente procedimentos para realizar escala e cisalhamento de imagens. Para escala com fatores M_x e M_y, a matriz de transformação é

$$\begin{pmatrix} M_x & 0 \\ 0 & M_y \end{pmatrix}$$

Para cisalhamento paralelo ao eixo x, com mapeando x em $x + cy$, a matriz é

$$\begin{pmatrix} 1 & c \\ 0 & 1 \end{pmatrix}$$

Para cisalhamento paralelo ao eixo y,

$$\begin{pmatrix} 1 & 0 \\ c & 1 \end{pmatrix}.$$

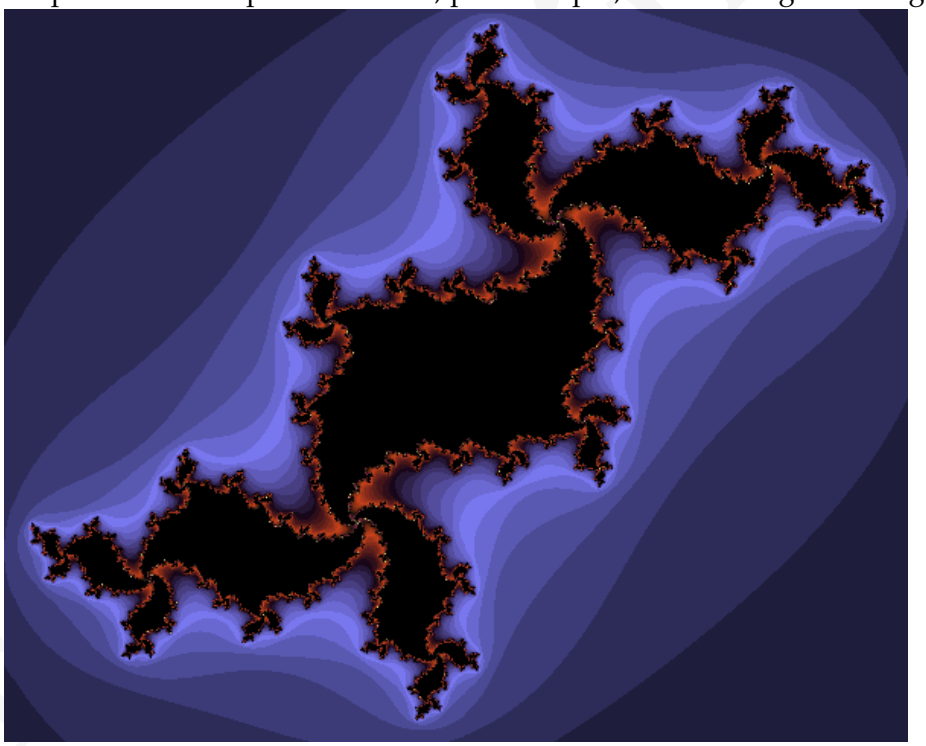
Versão Preliminar

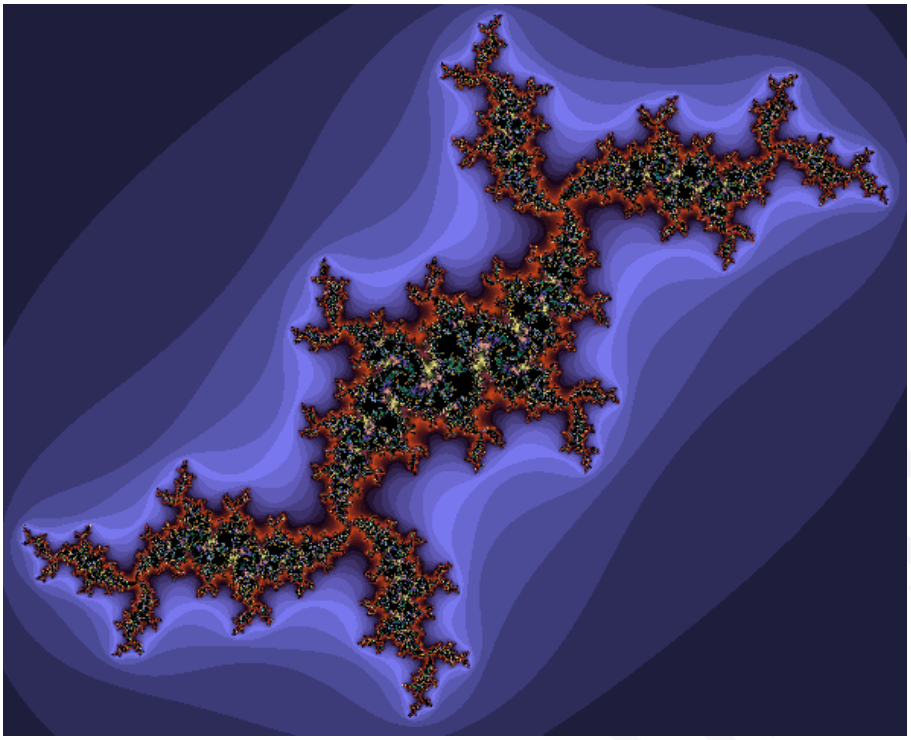
Ex. 94 — O procedimento de escala do Exercício 93 não é satisfatório: ele produz o mesmo número de pixels, mas em um espaço maior, tornando a imagem "menos densa". Experimente com métodos para resolver este problema.

Ex. 95 — Plote parte do conjunto de Julia para $f(z) = c\,\mathrm{sen}\,(z)$, com $c = 0.9 + 0.5i$. Você terá que mudar a escala da imagem e decidir o valor a partir do qual pode dizer que a sequência diverge.

Ex. 96 — As imagens de conjunto de Julia que geramos estão invertidas, porque nosso eixo dos complexos (indexado pela linha na imagem) cresce para baixo. Conserte este pequeno problema e mude também o procedimento `make-julia` para aceitar um parâmetro opcional que determina se os eixos devem ser também plotados.

Ex. 97 — Modifique o plotador de conjuntos de Julia para que ele plote pontos diferentes com cores diferentes, dependendo do quão rápido cada ponto diverge (use o número de iterações realizadas antes do valor da função passar de 2). Os conjuntos de Julia que geramos em preto e branco poderiam ficar, por exemplo, como nas figuras a seguir.





Versão Preliminar

Versão Preliminar

6 LISTAS E SEQUENCIAS

Este Capítulo trata principalmente de procedimentos que operam em sequências (listas, strings e vetores), retornando novas sequências.

6.1 LISTAS

Os procedimentos `list-map` e `list-reduce`, descritos no Capítulo 1, são os exemplos mais simples de procedimentos para listas.

Um procedimento que nos será útil mais tarde é o `iota`, descrito na SRFI-1, que constrói listas com sequências de números. O `iota` recebe obrigatoriamente um argumento `count` que determina o tamanho da lista a ser criada. Opcionalmente, dois outros argumentos podem ser usados: o segundo é `start`, que dá o valor inicial a ser usado (o que ficará no `car` da lista); o segundo é o `step`, que determina a distância entre dois valores (o "passo" dado de um valor a outro).

Uma versão muito simples de `iota` é esta:

```
(define iota-rec
  (lambda (count start step)
    (if (< count 1)
        '()
         (cons start (iota (- count 1) (+ start step) step)))))
```

Esta versão, no entanto, tem um pequeno inconveniente: a necessidade de sempre passar os parâmetros `start` e `step`.

A próxima versão de iota é melhor, embora um pouco mais longa: como queremos dois parâmetros opcionais, o primeiro `let` é um pouco extenso.

Versão Preliminar

```
(define iota
  (lambda (count . rest)
    (let ((start (if (not (null? rest))
                     (car rest)
                     0))
          (step (if (and (not (null? rest))
                         (not (null? (cdr rest))))
                    (cadr rest)
                    1)))
      (if (< count 1)
          '()
          (cons start (iota (- count 1) (+ start step) step))))))
(iota 5)
(0 1 2 3 4)
(iota 5 1)
(1 2 3 4 5)
(iota 5 1 0.25)
(1.0 1.25 1.5 1.75 2.0)
(iota 5 1 -0.5)
(1.0 0.5 0.0 -0.5 -1.0)
```

Para verificar se uma mão de pôquer tem todas as cartas de um mesmo naipe será útil o procedimento `unary-every`, que aceita um predicado e uma lista, e retorna `#t` quando o predicado retornar `#t` para todos os elementos da lista:

```
(unary-every positive? '(0 1 2))
#f
(unary-every symbol? '(a b c))
#t
```

A implementação de `unary-every` é bastante simples:

Versão Preliminar

```
(define unary-every
  (lambda (pred l)
    (let ev ((lista l))
      (or (not (pair? lista))
          (and (pred (car lista))
               (ev (cdr lista)))))))
```

O procedimento `flush?` construido usando `unary-every` é bastante simples:

```
(define flush?
  (lambda (naipes)
    (unary-every? (lambda (x)
                    (eqv? (x) (car naipes)))
                  naipes)))
```

O procedimento `straight?` pode ser construído de diferentes maneiras. Uma delas é verificar, para cada elemento da lista de valores, se ele é igual ao próximo subtraído de um:

```
(define straight?
  (lambda (valores)
    (if (< (length valores) 2)
        #t
        (and (eqv? (+ 1 (car valores))
                   (cadr valores))
             (straight? (cdr valores))))))
```

Outra forma de verificar uma sequência usa dois procedimentos, `take` e `drop`. O primeiro toma os n primeiros elementos de uma lista, e o segundo retorna a lista sem os n primeiros elementos:

```
(take '(a b c) 2)
(a b)
(drop '(a b c) 1)
(b c)
```

O próximo exemplo mostrar uma possível implementação destes procedimentos.

Versão Preliminar

```
(define take
  (lambda (lista n)
    (if (< n 1)
        '()
         (cons (car lista)
               (take (cdr lista) (- n 1))))))

(define drop
  (lambda (lista n)
     (if (< n 1)
         lista
         (drop (cdr lista) (- n 1)))))


(define sub1
  (lambda (n) (- n 1)))

(define straight-2?
  (lambda (valores)
    (let ((tam (- (length valores) 1)))
      (let ((valores-1 (take valores tam))
            (valores-2 (cdr valores)))
        (equal? valores-1 (map sub1 valores-2))))))
```

A lista `valores-1` é igual à lista `valores`, sem o último elemento. A lista `valores-2` é igual ao `cdr` da lista `valores`. Basta emparelhar ambas e verificá-las paralelamente e em ordem para verificar se é uma sequência:

```
cartas:    8  9 10  j q
valores-1: 8  9 10 11
valores-2: 9 10 11 12
```

Em cada posição, o valor do elemento de `valores-2` é igual ao valor do respectivo elemento de `valores` somado de um.

Esta abordagem é mais limpa e elegante que a anterior, mas é menos eficiente: a lista é percorrida uma vez na avaliação da forma `(take valores tam)` e outra na avaliação de `(equal? valores1 (map ...))`. No entanto, como as listas usadas para representar mãos de jogo de pôquer são muito pequenas, esta diferença em eficiência não será notada na prática.

Versão Preliminar

O procedimento que verifica um *royal flush* é bastante simples porque usa os outros já desenvolvidos:

```
(define royal?
  (lambda (valores naipes)
    (and (= (car valores) 10)
         (straight? valores)
         (flush? naipes))))
```

Podemos precisar filtrar uma lista, selecionando apenas alguns elementos.

```
(define filter
  (lambda (pred? lst)
    (cond ((null? lst) lst)
          ((pred? (car lst))
           (cons (car lst)
                 (filter pred? (cdr lst))))
          (else (filter pred? (cdr lst))))))
```

É possível codificar várias listas relacionadas em uma só, de maneira que a lista resultante tenha elementos de cada uma das listas originais alternando-se.

```
(zip '("Julius" "Winston" "Napoleao")
     '(cesar    primeiro-ministro imperador)
     '(roma     inglaterra franca))
'( "Julius" cesar roma
"Winston" primeiro-ministro inglaterra
"Napoleao" imperador franca)
```

A implementação de `zip` precisa separar os `cars` e `cdrs` de todas as listas e adicionar os `cars` gradualmente, até que as listas tenham se esgotado:

```
(define zip
  (lambda listas
    (if (any null? listas)
        '()
        (let ((cars (map car listas))
              (cdrs (map cdr listas)))
          (append cars (apply zip cdrs))))))
```


Versão Preliminar

O procedimento `zip` sabe quantas listas são passadas como argumentos; um procedimento `unzip` não tem como adivinhar o número de listas, e portanto precisa dele como argumento.

```
(define unzip
  (lambda (n lista)
    (let loop ((result (make-list n (list)))
               (lst lista))
      (if (< (length lst) n)
          (map! reverse! result)
          (let ((to-unzip (take lst n)))
            (loop (map! cons to-unzip result)
                  (drop lst n)))))))
```

Uma mão de pôquer pode ser representada como uma lista de cartas, e cada carta pode por sua vez ser representada por uma lista onde o primeiro elemento determina o valor e o segundo o naipe da carta. Por exemplo,

```
'((4 copas)
  (9 paus)
  (k ouros)
  (k paus)
  (k copas))
```

O procedimento `unzip2` é parecido com `unzip`, mas aceita listas de pares:

```
(unzip2 '((4 copas)
          (9 paus)
          (k ouros)
          (k paus)
          (k copas))
(4 9 k k k)
(copas paus ouros paus copas)
```

Um procedimento para pontuar mãos de pôquer pode separar os valores dos naipes das cartas, já que algumas das verificações dependem somente dos valores ou somente das cartas. Ordenar os valores também pode ser útil para identificar sequências.

Versão Preliminar

```
(define pontua
  (lambda (hand)
    (let-values (((cartas naipes) (unzip2 hand)))
      (let ((valores (sort (map carta->valor
                                cartas)
                           <)))
        (cond ((royal? valores naipes)
               4000)
              ((and (straight? valores)
                    (flush? naipes))
               3000)
              ((straight? valores)
               2000)
              ((flush? naipes)
               1000)
              (else (apply max valores)))))))
```

6.1.1 Permutações

Construiremos agora dois procedimentos: o primeiro é o procedimento `permutations` que receba uma lista com `n` elementos e liste todas as permutações da lista.

```
(permutations '(1 2 3))
((1 2 3) (2 1 3) (2 3 1)
(1 3 2) (3 1 2) (3 2 1))
```

O segundo procedimento é `with-permutations`, que recebe uma lista com `n` elementos e um procedimento de `n` argumentos, e chame este procedimento com todas as permutações da lista.

```
(with-permutations
  '(1 2 3)
  (lambda args
    (display "argumentos: ")
    (display args)
    (newline)))
```

Versão Preliminar

```
argumentos: (1 2 3)
argumentos: (2 1 3)
argumentos: (2 3 1)
argumentos: (1 3 2)
argumentos: (3 1 2)
argumentos: (3 2 1)
```

Primeiro construiremos `permutations`. Nosso algoritmo é recursivo: se uma lista é vazia, retornamos `(())`. Se a lista não é vazia, separamos `car` e `cdr`; depois computamos recursivamente todas as permutações do `cdr`, e em cada uma delas incluímos o `car`, em todas as posições possíveis.

Começamos pelo procedimento que lista todas as maneiras de inserir um elemento `x` em uma lista. Se a lista é vazia, só há uma maneira, e o procedimento retorna `((x))`. Se a lista não é vazia, o procedimento produz todas as maneiras de incluir `x` *no* `cdr`, e as põe em uma lista `cdr-with-x`. Nestas listas, falta o `car`. O procedimento então usa `map` para incluir o `car` em cada uma delas.

```
(define insert*
  (lambda (x lst)
    (if (null? lst)
        (list (list x))
        (let ((cdr-with-x (insert* x (cdr lst))))
          (cons (cons x list)
                (map (lambda (lst) (cons (car list) lst))
                     cdr-with-x))))))
(insert* 'x '(a b c))
((x a b c) (a x b c) (a b x c) (a b c x))
```

Agora podemos construir o procedimento `permutations`.

```
(define permutations
  (lambda (lst)
    (if (null? lst)
        '(())
        (apply append (map (lambda (perm)
                             (insertions (car list) perm))
                           (permutations (cdr list)))))))
```


Versão Preliminar

```
(permutations '(a b c d))
((a b c d) (b a c d) (b c a d)
(b c d a) (a c b d) (c a b d)
(c b a d) (c b d a) (a c d b)
(c a d b) (c d a b) (c d b a)
(a b d c) (b a d c) (b d a c)
(b d c a) (a d b c) (d a b c)
(d b a c) (d b c a) (a d c b)
(d a c b) (d c a b) (d c b a))
```

Finalmente, o procedimento `with-permutations` toma uma lista de argumentos e um procedimento, e aplica o procedimento a todas as permutações da lista.

```
(define with-permutations
  (lambda (lst proc)
    (for-each (lambda (permutation)
                (apply proc permutation))
              (permutations lst))))


(with-permutations '(a b c)
                   (lambda (x y z)
                     (print x '-> y '-> z)))
```

```
a->b->c
b->a->c
b->c->a
a->c->b
c->a->b
c->b->a
```

Outras formas de gerar permutações são descritas no livro de Knuth [Knu05].

EXERCÍCIOS

Ex. 98 — Escreva os procedimentos:

a) `unary-any`

b) `vector-iota`

Versão Preliminar

c)`vector-every`, `vector-any`

d)`vector-shuffe`

e)`unzip2`, usado no procedimento de pontuação para pôquer

f)`delete-duplicates`

Ex. 99 — A seguinte versão do predicado `flush?` usa o procedimento `delete-duplicates` do exercício anterior. Diga se esta versão é mais ou menos eficiente que a dada no texto, que usa `every`.

```
(define flush?
  (lambda (naipes)
    (equal? (delete-duplicates naipes)
            (take naipes 1))))
```

Ex. 100 — Porque não faz sentido construir um procedimento `vector-zip!`?

Ex. 101 — Escreva o procedimento `for-each-sublist`, que funciona de maneira semelhante a `for-each`, mas ao invés de aplicar um procedimento a cada *elemento* da lista, aplica o procedimento a cada *sublista*. Os exemplos a seguir mostram a diferença entre `for-each` e `for-each-sublist`.

```
(for-each (lambda (elt)
            (display elt)
            (newline))
          '(ou o Brasil acaba com a saúva...))
```

```
ou
o
Brasil
acaba
com
a
saúva...
```

```
(for-each-sublist (lambda (sub)
                    (display sub)
                    (newline))
                  '(ou o Brasil acaba com a saúva...))
```


Versão Preliminar

```
(ou o Brasil acaba com a saúva...)
(o Brasil acaba com a saúva...)
(Brasil acaba com a saúva...)
(acaba com a saúva...)
(com a saúva...)
(a saúva...)
(saúva...)
```

Ex. 102 — Termine o jogo de pôquer. Faça seu jogo gradualmente, implementando diferentes características de cada vez: primeiro, implemente procedimentos para verificar se uma mão tem uma quadra, terno, *full house*, dois pares ou um par, um por vez. Note que *royal flush* > *straight flush* > *quadra* > *full house* > *flush* > *sequencia* > *terno* > *dois pares* > *par*, e há várias regras para desempate que você pode implementar aos poucos.

Ex. 103 — Modifique o procedimento `permutations` para que ele não lista elementos repetidos.

Ex. 104 — Modifique o procedimento `with-permutations` para que ele retorne os valores produzidos para cada aplicação do procedimento.

```
(with-permutations '(1 2 3)
  (lambda (x y z)
    (- x y z)))
(-4 -2 -2 -4 0 0)
```

Diga como o programador poderá saber qual valor corresponde a qual permutação dos argumentos (e mostre que sua solução sempre funciona).

Resp. (Ex. 92) — Os procedimentos da biblioteca devem aceitar, além dos operandos, as operações de soma e multiplicação do corpo.

Resp. (Ex. 101) — A implementação é bastante simples.

Versão Preliminar

```
(define for-each-sublist
  (lambda (proc lst)
    (if (not (null? lst))
        (begin (proc lst)
               (for-each-sublist proc (cdr lst))))))
```

Resp. (Ex. 103) — Crie um procedimento `remove-dups` que elimina elementos repetidos em uma lista, e aplique-o antes de aplicar `permutations`.

Resp. (Ex. 104) — Mostre que tanto os resultados de `permutations` e os de `with-permutations` podem ser combinados com `zip` criando pares corretos de permutação/valor-tesultado.

Versão Preliminar

Parte II.

Conceitos Avançados

Versão Preliminar

Versão Preliminar

7 EVAL

O Capítulo 1 mencionou brevemente o *Read-Eval-Print Loop*; `read` e `print` são procedimentos simples para entrada e saída de dados, descritos no Capítulo 2, mas `eval` está no coração de um interpretador Scheme.

Em Scheme, o procedimento `eval` aceita um trecho de código, uma especificação de ambiente, e então avalia o código usando aquele ambiente:

```
(eval codigo ambiente)
```

O ambiente passado para `eval` pode ser obtido de três diferentes procedimentos. O primeiro deles, `(scheme-report-environment versao)`, devolve um ambiente "limpo", com apenas as vinculações dos símbolos definidos pelo padrão Scheme determinado por `versao` (por exemplo 5 para R^{5}RS). Já `null-environment` devolve um ambiente com apenas os nomes das formas especiais do padrão Scheme[1]:

```
(eval 'if (null-environment))
#<primitive-builtin-macro! if>
(eval '+ (null-environment))
Unbound variable: +
```

O terceiro procedimento, `(interaction-environment)`, devolve o ambiente usado no momento em que foi chamado. O próximo exemplo ilustra a diferença entre os procedimentos `scheme-report-environment` e `interaction-environment`.

```
(define + -)
```

No ambiente atual a conta não funciona:

```
(+ 4 3)
1
(eval '(+ 4 3) (interaction-environment))
1
```

Se a implementação de Scheme mantém separados o ambiente do REPL e o ambiente Scheme padrão (nem todas o fazem), podemos usar o ambiente padrão para avaliar a mesma forma `(+ 4 3)`:

```
(eval '(+ 4 3) (scheme-report-environment 5))
```

1 Nem toda implementação de Scheme retornará alguma informação a respeito de formas especiais; algumas delas, diante de uma tentativa de avaliar o símbolo `if`, simplesmente devolverão uma mensagem de erro.

Versão Preliminar

```
7
```

`null-environment` só contém formas especiais:

```
(eval '(+ 4 3) (null-environment)) ;; => ERRO!
(eval '(if #f 'a 'b) (null-environment))
'b
```

O mais interessante:

```
(eval '+ (scheme-report-environment 5))
<#procedure C_plus>
(define + (eval '+ (scheme-report-environment 5)))
(+ 4 3)
7
```

Uma possível implementação de REPL é mostrada a seguir:

```
(define my-repl
  (lambda ()
    (print "> ")
    (print (eval (read) (interaction-environment)))
    (newline)
    (my-repl)))
```

Fica claro então o motivo do nome "read-eval-print" – que é a ordem em que são avaliados os procedimentos `(print (eval (read) ...))`.

Se `my-repl` for usado em um interpretador Scheme, só haverá uma maneira de sair dele: enviando a forma `(exit)`. No entanto, quando isso for feito, o hospedeiro Scheme avaliará a forma e terminará sua execução também. Para evitar que isso ocorra, pode-se capturar a forma `(exit)` (isso evidentemente não impede que o usuário force o hospedeiro a terminar, por exemplo enviando a forma `(if #t (exit))`).

```
(define my-repl
  (lambda ()
    (print "> ")
    (let ((form (read)))
    (if (not (equal? form '(exit)))
        (begin (print (eval form
                            (interaction-environment)))
               (my-repl))))))
```


Versão Preliminar

7.1 PROCEDIMENTOS QUE MODIFICAM O AMBIENTE GLOBAL

(esta seção está incompleta)

Declarações internas em Scheme não definem variáveis globais:

```
(define def-a
  (lambda ()
    (define a 50)
    a))
(def-a)
50
```

O `define` interno criou um vínculo para `a` no ambiente interno ao `lambda`, mas este ambiente não é acessível no nível global.

```
a
ERROR: undefined variable: a
```

No entanto, é possível criar definições globais a partir de procedimentos:

```
(define global-environment (interaction-environment))

(define def-a
  (lambda ()
    (eval '(define a 50) global-environment)
    a))
(def-a)
50
a
50
```

Este último exemplo funciona porque a forma `(define a 50)` foi avaliada no ambiente `global-environment`, uma variável que contém o ambiente corrente do REPL.

Versão Preliminar

7.2 PROGRAMAÇÃO GENÉTICA

Um dos poucos casos onde o procedimento `eval` é a melhor maneira de resolver um problema é a implementação de programação genética.

Programação genética [deJo6; PLM08; ES03; LP10; Koz92] é um método *programação automática*: através de programação genética é possível obter programas de computador sem a necessidade de desenvolver os algoritmos usados nestes programas.

O método de programação genética é inspirado na Teoria da Evolução natural de Darwin: cada programa é visto como um indivíduo em uma população (a população inicial é um conjunto de programas gerado aleatoreamente). Um sistema de Programação Genética então simula ideias da Teoria da Evolução Natural modificando esses programas: dois programas podem se reproduzir sexuadamente através de uma operação de *cross-over* e também pode haver mutação, que modifica alguma característica de um indivíduo. Os indivíduos obtidos através destas operações constituem uma nova *geração*. Assim como na Evolução Natural, indivíduos que se adequam melhor ao meio tem probabilidade maior de reproduzir e portanto de propagar seu DNA para as próximas gerações. Em Programação Genética, a adequação de um programa é verificada através de sua execução: deve haver alguma maneira de medir a "qualidade" de um programa e compará-lo com outros.

O algoritmo básico de Programação Genética é:

1) Gere aleatoreamente uma população inicial de formas Scheme usando apenas os operadores definidos e números aleatóreos pequenos (entre um e quinze, por exemplo);

2) Execute cada programa da população e meça o valor de cada um deles usando uma função de adequação (*fitness*);

3) Crie uma nova população:
   a) Copie os melhores programas;
   b) Faça mutação em alguns dos programas;
   c) Crie novos programas por *cross-over*.

A seguir detalharemos um exemplo: temos um conjunto de pontos (x, y) e queremos encontrar uma função que quando aplicada aos valores de x aproxime os valores de y com o menor erro possível. Se nos restringíssemos a funções lineares, teríamos um problema de *regressão linear*. No entanto, não imporemos qualquer restrição à estrutura da função.

Versão Preliminar

Em nosso exemplo Um indivíduo na população de programas será uma forma Scheme como por exemplo `(+ (* 4 (sin x)) (/ 5 x))`. A função de fitness só precisa avaliar a expressão – para isto ela deve ser transformada em uma expressão λ:

```
(define sexp->lambda
  (lambda (var exp)
    (list  'lambda `(,var) exp)))
```

A lista `'(+ (* 4 (sin x)) (/ 5 x))` representaria, então, a função $4\sin(x) + 5/x$, e o procedimento `sexp->lambda` a transformará em uma lista representando uma função anônima Scheme:

```
(sexp->lambda 'x '(+ (* 4 (sin x)) (/ 5 x)))
(lambda (x) (+ (* 4 (sin x)) (/ 5 x)))
```

Esta função anônima pode ser passada para `eval`

```
(define expr (sexp->lambda 'x '(+ (* 4 (sin x)) (/ 5 x))))
(eval (list expr 1))
8.36588393923159
((eval expr) 1)
8.36588393923159
```

O procedimento `sum-squares` calcula a soma dos quadrados de uma lista:

```
(define sum-squares
  (lambda (l)
    (let ((square (lambda (x) (* x x))))
      (reduce + 0
              (map square l)))))
```

O procedimento `get-fitness` retorna um procedimento para calcular a adequação de um indivíduo. Recebe uma lista de pontos e outra de valores. Devolve um procedimento que se lembra dos valores e pontos originais e sabe calcular o erro quadrático do indivíduo: calcula cada valor aproximado com `(map funcao valores)` e depois cria uma lista com a diferença entre os valores corretos e os valores dados pela nova função, com `map - valores valores-bons`. Em seguida devolve a soma dos quadrados da lista `diferenca`.

Versão Preliminar

```
(define get-fitness
  (lambda (pts val)
    (let ((valores-bons val)
          (pontos pts))
      (lambda (funcao)
        (let ((valores-aprox (map funcao pontos)))
          (let ((diferenca (map - valores-aprox
                                valores-bons)))
            (sum-squares diferenca)))))))

(define funcao-desconhecida
  (lambda (a)
    (+ 1 (* a a))))
(define pontos '(1 4 9 10))
(define valores (map funcao-desconhecida pontos))
valores
(2 17 82 101)
```

Dados pontos e valores, fit dirá quão próximo chegamos Os valores serão negativos (quando ainda há diferença) ou zero (quando conseguimos valores exatos para todos os pontos).

```
(define fit (get-fitness pontos valores))
```

Em programação genética, um indivíduo é um trecho de programa. O indivíduo abaixo parece bom:

```
(define individuo '(+ 2 (* x x)))
```

O valor da função de fitness deste indivíduo é -4:

```
(fit (eval (sexp->lambda 'x individuo) (interaction-environment)))
-4
```

O próximo indivíduo é ainda melhor:

```
(define individuo-melhor '(+ 0.5 (* x x)))
(fit (eval (sexp->lambda 'x individuo-melhor) (interaction-environment)))
-1
```

Versão Preliminar

O fitness do indivíduo perfeito é zero:

```
(define individuo-perfeito '(+ 1 (* x x)))
(fit (eval (sexp->lambda 'x individuo-perfeito) (interaction-environment)))
0
```

O operador de mutação pode trocar números ou operadores nas formas Scheme; o de *cross-over* deve trocar sub-árvores (ou sub-expressões) de dois indivíduos.

Usaremos `list-reduce` e `list-filter`, além de `list-count`, que conta a quantidade de átomos em uma lista, inclusive em suas sublistas:

```
(define list-count
  (lambda (lst)
    (+ (length lst)
       (list-reduce + 0
                    (map list-count
                         (list-filter list? lst))))))
```

O operador de cross-over certamente precisará de um procedimento que troca os `cdr`s de duas listas:

```
(define swap-cdr!
  (lambda (a b)
    (let ((ptr-a (cdr a))
          (ptr-b (cdr b)))
      (set-cdr! a ptr-b)
      (set-cdr! b ptr-a))))
```

O procedimento `deep-for-each` aplica um procedimento em todos os átomos de uma lista, recursivamente.

```
(define deep-for-each
  (lambda (proc lst)
    (define do-it
      (lambda (e)
        (if (list? e)
            (deep-for-each proc e)
            (proc e))))
    (for-each do-it lst)))
```

Precisaremos de um procedimento `count-nonempty-sublists` para contabilizar a quantidade de sublistas não vazias em uma lista.

Versão Preliminar

```
(define count-nonempty-sublists
  (lambda (lst)
    (if (null? lst)
        0
        (let ((rest (count-nonempty-sublists (cdr lst)))
              (head (+ 1 (if (list? (car lst))
                             (count-nonempty-sublists (car lst))
                             0))))
          (+ head rest)))))
```

Os operadores de mutação e cross-over escolherão um ponto aleatóreo na forma para fazer modificações. O procedimento `nth-nonempty-sublist` encontra este ponto, que é a n-ésima sublista não vazia.

```
(define nth-nonempty-sublist
  (lambda (the-list n)
    (let loop ((lst the-list)
               (n 0)
               (wanted n))
      (cond ((null? lst)   n)
            ((> n wanted)  'OVER)
            ((= n wanted)  lst)
            (else          (let ((head (if (list? (car lst))
                                           (loop (car lst) (+ n 1) wanted
                                           (+ n 1))))
                             (cond ((symbol? head)  'OVER)
                                   ((list? head)    head)
                                   (else
                                     (loop (cdr lst) head wanted)))))))))
```

Podemos testar `nth-nonempty-sublist`:

Versão Preliminar

```
(do ((i 0 (+ 1 i)))
    ((= i 12))
  (display "===========\n")
  (newline)
  (display i)
  (display "th: ")
  (display "-- ")
  (display (nth-nonempty-sublist '(1 (2) (3 (4 5) 6) 7) i))
  (newline))
```

O operador de cross-over seleciona um ponto aleatóreo em cada uma das duas listas e troca os `cdr`s nestes dois pontos, retornando duas novas listas.

```
(define cross-over!
  (lambda (A B)
    (let ((size-A (count-nonempty-sublists A))
          (size-B (count-nonempty-sublists B)))
      (let ((idx-A (random-integer size-A))
            (idx-B (random-integer size-B)))
        (let ((ptr-A (nth-nonempty-sublist A idx-A))
              (ptr-B (nth-nonempty-sublist B idx-B)))
          (swap-cdr! ptr-A ptr-B))))))
```

Os operadores são armazenados em listas e classificados por aridade.

```
(define ops-by-arity
  '((1 '(inv sqrt log exp abs sin cos tan asin acos atan))
    (2 '(expt))
    (n '(+ - * /))))
```

(esta seção está incompleta)

7.3 SCHEME EM SCHEME

Dois procedimentos formam o coração de um interpretador Scheme: `eval` e `apply`; o primeiro avalia formas, e o segundo aplica procedimentos. Nesta seção construiremos em Scheme um interpretador Scheme minimalista, incluindo `eval`, `apply` e um REPL. Nosso

Versão Preliminar

interpretador não será um ambiente Scheme completo, mas ilustrará claramente como formas são avaliadas.

O interpretador reconhecerá `def` ao invés de `define` para criar vínculos.

7.3.1 Construção do interpretador

Começaremos com procedimentos para representar o ambiente; depois, procedimentos internos do interpretador para criar e modificar vínculos no ambiente com `def` e `set!`. Em seguida, implementamos procedimentos que tratam as formas `if` e `lambda`. Finalmente implementaremos `eval` e `apply`.

7.3.1.1 *Ambiente*

Em nosso interpretador o ambiente será representado como uma lista de associações. O procedimento `find-in-env` encontra uma variável em um ambiente, sinalizando um erro quando a variável não for encontrada.

```
(define find-in-env
  (lambda (var env)
    (let ((binding (assoc var env)))
      (if binding
          (list-ref binding 1)
          (error "Variável não vinculada" var)))))
```

O procedimento `extend-env` extende um ambiente com novos vínculos: recebe uma lista de variáveis, uma lista de valores, um ambiente e retorna o novo ambiente, onde as novas variáveis são vinculadas aos novos valores. Usaremos `extend-env` quando um procedimento for aplicado (seus parâmetros, definidos em uma forma `lambda`, serão variáveis no ambiente extendido).

Versão Preliminar

```
(define extend-env
  (lambda (vars values env)
    (cond ((or (null? env)
               (null? vars))
           env)
          (else
           (cons (list (car vars) (car values))
                 (extend-env (cdr vars)
                             (cdr values)
                             env))))))
```

Os novos vínculos são incluídos com `cons`, e por isso obedecem disciplina de pilha. É importante também que

- Quando um ambiente é extendido por um `lambda` (ou `let`), as novas variáveis devem ser visíveis *apenas dentro deste ambiente*, e não no ambiente antigo, que foi extendido – o que de fato ocorre, porque referências anteriores só existiam aos vínculos mais à frente na lista;
- Vínculos novos devem ter prioridade sobre os antigos (que já existiam antes do ambiente ser extendido). Isto evidentemente também acontecerá, já que `find-in-env` pesquisará os vínculos da cabeça até a cauda da lista.

Por exemplo, o ambiente do primeiro `let` no código a seguir contém um vínculo `x` $\rightarrow$ 10; já o ambiente do segundo `let` é extendido com `x` $\rightarrow$ 50 e `y` $\rightarrow$ 20:

```
(let ((x 10))
  (display x)
  (let ((x 50)
        (y 20))
    (+ x y)))
```

Supondo que o primeiro ambiente tinha apenas o vínculo para `x`, ele seria representado por `((x 10))`. Ao avaliar o `display`, o interpretador fará:

```
(find-in-env 'x '((x 10)))
10
```

Já ao avaliar o `+` o interpretador usará uma extensão do ambiente anterior.

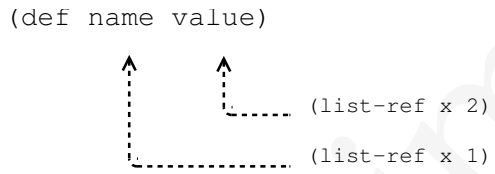
Versão Preliminar

```
(extend-env '(x y) '(50 20) '((x 10)))
((x 50) (y 20) (x 10))

(find-in-env 'x '((x 50) (y 20) (x 10)))
50
```

7.3.1.2 *def e set!*

Quando a S-expressão é uma definição de uma nova variável, o nome e valor são adicionados ao ambiente. Em nosso Scheme usaremos `def` ao invés de `define`.

Usaremos `def-name` e `def-val` para o nome e valor de uma nova definição.




```
(define def-name (lambda (x) (list-ref x 1)))
(define def-val  (lambda (x) (list-ref x 2)))
```

Assim, `do-def!` recebe uma expressão da forma `(def a b)`, um ambiente, e cria um vínculo para `a` neste ambiente, armazenando no local de `a` o valor da expressão `b`, que é avaliada neste momento.

Versão Preliminar

```
(define do-def!
  (lambda (exp env)
    (let ((new-binding (cons (def-name exp)
                             (list (eval# (def-val exp) env)))))

      (if (null? (car env))
          (set-car! env new-binding)
          (begin
            ;; O cdr do ambiente passa a ser o atual car
            ;; seguido do cdr anterior. O efeito é de duplicar
            ;; a cabeça da lista
            (set-cdr! env
                      (cons (car env)
                            (cdr env)))
            ;; O car do ambiente passa a ser a nova vinculação
            (set-car! env new-binding))))))
```

Quando se trata de um `set!`, o valor já armazenado no ambiente é modificado.

```
(define do-set!
  (lambda (exp env)
    (let ((name (list-ref exp 1)))
      (let ((binding (assoc name env)))
        (if binding
            (set-cdr! binding
                      (list (eval# (def-val exp) env)))
            (error "do-set! -- variável inexistente!"))))))
```

7.3.1.3 *if*

Quando `eval` recebe uma expressão condicional (com `if` na cabeça da lista), avaliará o teste e decidirá entre avaliar a "forma então", logo após o teste, ou a "forma senão", logo depois.

Versão Preliminar

```
(define if-test (lambda (x) (list-ref x 1)))
(define if-then (lambda (x) (list-ref x 2)))
(define if-else (lambda (x) (list-ref x 3)))

(define do-if
  (lambda (exp env)
    (cond ((eval# (if-test exp) env)
           (eval# (if-then exp) env))
          (else
           (eval# (if-else exp) env)))))
```

7.3.1.4 *lambda*

Uma S-epressão que comece com `lambda` é um procedimento anônimo, então `eval` construirá um objeto que o representa, contendo os parâmetros formais, o corpo e o ambiente. Assim, uma forma `(lambda (...) ...)` será transformada em uma lista `(proc (lambda (...) ...) ambiente)`. É importante que o procedimento se lembre do ambiente onde foi definido para que possamos implementar escopo léxico.

```
(define make-proc
  (lambda (exp env)
    (list 'proc exp env)))
```

7.3.1.5 *eval*

O procedimento `eval` aceita como parâmetros uma S-expressão e um ambiente, e seu valor de retorno é definido em casos: se a S-expressão é "auto-avaliante" (ou seja, seu valor é ela mesma), ela é retornada sem modificações.

```
(define auto-eval?
  (lambda (exp)
    (or (boolean? exp)
        (number? exp)
        (string? exp)
        (primitive-proc? exp)
        (and (pair? exp)
             (is-a? exp 'proc)))))
```


Versão Preliminar

Um símbolo será sempre interpretado como nome de uma variável, por isso podemos definir um predicado `var?` que idêntico a `symbol?`:

```
(define var? symbol?)
```

Qualquer lista não vazia pode ser uma aplicação de procedimento, e o procedimento `is-a?` verifica se uma lista inicia com um certo símbolo.

```
(define is-a?
  (lambda (exp sym)
    (eqv? (car exp) sym)))

(define proc? pair?)
```

Por exemplo, para a forma `f = (if ...)`, temos que `(is-a? f 'if)` é verdadeiro.

Para as formas dentro de `lambda` ou `begin` é necessário um mecanismo para avaliação de várias formas em sequência. Para isso implementamos `eval-seq`.

```
(define eval-seq
  (lambda (exp env)
    (cond ((null? exp)
           (error "eval-seq: null sequence"))
          ((null? (cdr exp))
           (eval# (car exp) env))
          (else
           (eval# (car exp) env)
           (eval-seq (cdr exp) env)))))
```

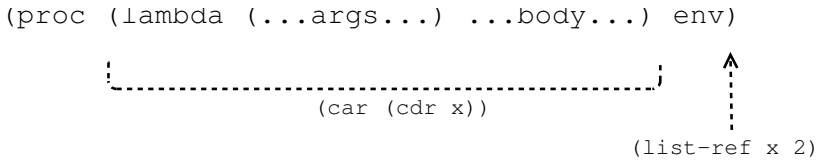
O procedimento `eval#` é mostrado a seguir.

Versão Preliminar

```
(define eval#
  (lambda (exp env)
    (cond ((auto-eval? exp)     exp)

          ((var? exp)           (find-in-env exp env))

          ((is-a? exp 'def)     (do-def! exp env))

          ((is-a? exp 'set!)    (do-set! exp env))

          ((is-a? exp 'quote)   (cadr exp))

          ((is-a?  exp 'if)     (do-if exp env))

          ((is-a? exp 'lambda) (make-proc exp env))

          ((is-a? exp 'begin)   (eval-seq (cdr exp) env))

          ((proc? exp)          (apply# (eval# (car exp) env)
                                        (cdr exp)
                                        env))
          (else
            (error "Eval não sabe o que fazer com " exp)))))
```

7.3.1.6 *apply*

Quando uma S-expressão não é qualquer das formas especiais (`def`, `set!`, `quote`, `if`, `lambda`, `begin`, e é uma lista, nosso interpretador presumirá que se trata de uma aplicação de procedimento. Ele avaliará a cabeça da lista e chamará `apply#`, que aceita três argumentos: o primeiro deve ser um procedimento, o segundo uma lista de argumentos e o terceiro é o ambiente a ser usado na avaliação.



Versão Preliminar

```
(define lambda-args+body (lambda (x) (cdr (car (cdr x)))))

(define lambda-args (lambda (x) (car (lambda-args+body x))))
(define lambda-body (lambda (x) (cdr (lambda-args+body x))))
(define lambda-env  (lambda (x) (list-ref x 2)))


(define apply#
  (lambda (proc args env)
    (cond ((primitive-proc? proc)
           (apply proc (map (lambda (a)
                              (eval# a env))
                            args)))
          (else
           (eval-seq (lambda-body proc)
                     (extend-env (lambda-args proc)
                                 (map (lambda (a)
                                        (eval# a env))
                                      args)
                                 (lambda-env proc)))))))
```

Se um procedimento é primitivo, `apply` simplesmente usa o `apply` da implementação subjacente de Scheme, passando a ele o procedimento e os argumentos, já avaliados.

Se o procedimento não é primitivo, então ele é representado como um objeto com três partes: argumentos formais, corpo e ambiente (este é o ambiente em vigor quando o procedimento foi definido). Para avaliar um objeto deste tipo nosso interpretador:

- Avalia os argumentos formais no ambiente *atual* (`env`);
- Extende o ambiente *do procedimento* com os vínculos avaliados para os parâmetros formais;
- Chama `eval-seq` para avaliar todas as formas no corpo do procedimento com este ambiente extendido.

7.3.1.7 *Procedimentos auxiliares*

Precisaremos do `eval` nativo de Scheme para inicializar o ambiente de nosso interpretador.

Versão Preliminar

```
(define scheme-eval
  (lambda (exp)
    (eval exp (interaction-environment))))
```

Guardamos em `primitive-names` uma lista de nomes que usaremos do ambiente nativo Scheme.

```
(define primitive-names
  '(= < > string=? string-append + - * /
    list car cdr display newline list cons))

(define primitive-name?
  (lambda (p)
    (member p primitive-names)))
```

O predicado `primitive-proc?`, usado em `apply#`, simplesmente verifica se um procedimento é um daqueles que incluímos na lista de procedimentos nativos:

```
(define primitive-proc?
  (lambda (p)
    (member p (map scheme-eval primitive-names))))
```

7.3.2 Usando o interpretador

Para testar nosso interpretador, criamos um ambiente vazio:

```
(define ne (list '()))
```

Incluimos no ambiente os vínculos para diversos procedimentos padrão de Scheme:

```
(for-each  (lambda (name)
             (eval# (list 'def name (scheme-eval name))
                    ne))
           primitive-names)
```

Agora usamos nossa forma especial `def` para criar alguns vínculos.

Versão Preliminar

```
(begin
  (eval# '(def a 5) ne)
  (eval# '(def b 15) ne)
  (eval# '(def c 25) ne)
  (eval# '(def d 35) ne))
```

Verificamos que os vínculos realmente foram criados:

```
(eval# 'b ne)
15
(eval# 'c ne)
25
(eval# '+ ne)
<#procedure _plus>

(eval# '(lambda (a) (* 2 a)) ne)
(proc (lambda (a) (* 2 a))
((d 35) (c 25) (b 15) (a 111)
(car #<procedure (car arg)>) ... ))
```

Agora verificaremos se o ambiente local de um `lambda` se sobrepõe sobre o global como esperaríamos de uma implementação Scheme:

```
(eval# '((lambda (a) (* 2 a)) 500) ne)
1000
(eval# '((lambda (c) (* 2 a)) 500) ne)
222


(eval# '(def inc-show
             ((lambda ()
                ((lambda (counter)
                   (list (lambda () (set! counter (+ 1 counter)))
                         (lambda () counter)))
                 0))))
      ne)
```

Guardamos os dois fechos nas variáveis `show` e `inc`,

```
(eval# '(def show (car (cdr inc-show))) ne)
(eval# '(def inc (car inc-show)) ne)
```

Versão Preliminar

e finalmente verificamos que nosso interpretador de fato implementa fechos corretamente:

```
(eval# '(show) ne)
0
(eval# '(inc) ne)
(eval# '(show) ne)
1
```

Finalmente, construiremos um REPL!

```
(define o-repl
  (lambda (in out)
    (let ((env (list '())))
      (for-each  (lambda (name)
             (eval# (list 'def name (scheme-eval name))
                    env))
           primitive-names)
      (eval# '(def *prompt* "toylisp:: ") env)
      (let loop ()
        (display (eval# '*prompt* env) out)
        (let ((forma (read in)))
          (display (eval# forma env) out)
          (newline)
          (loop))))))
```

Iniciamos o REPL de nosso `toylisp` assim:

```
(o-repl (current-input-port)
        (current-output-port))
```

7.3.3 Discussão

Um interpretador de uma linguagem escrito nela mesma é chamado de *meta-interpretador circular*[2].

2 A ideia de meta-interpretador já estava presente no trabalho de Alan Turing, que descreveu uma "Máquina de Turing Universal", capaz de ler a descrição de outra Máquina de Turing e simular seu funcionamento[Tur37; Tur38]. O primeiro interpretador implementado de fato em computadores e amplamente usado foi o do LISP de John McCarthy.

Versão Preliminar

É evidente que um meta-interpretador circular não será normalmente usado para a prática comum de programação, e pode por isso parecer inútil. No entanto, meta-interpretadores são muito valiosos em algumas situações: uma delas é a prototipagem de variantes da linguagem. Por exemplo, se quisermos implementar uma versão de Scheme com tipagem forte, ou com avaliação preguiçosa por default, podemos criar um meta-interpretador ao invés de construir um núcleo Scheme em C.

Outra situação onde meta-interpretadores são importantes é na discussão das características de linguagens de programação. Como exemplos disso há o livro introdutório de programação de Abelson e Sussman[AS96], que descreve meta-interpretadores circulares e criam variantes deles a fim de ilustrar o funcionamento de diferentes características de linguagens; o livro de Friedman e Wand[FW08] sobre linguagens de programação, que aos poucos constrói diferentes linguagens de programação, todas com alguma similaridade com Scheme; e finalmente o artigo de 1978 de Guy Steele e Gerald Sussman[SS78], que usa meta-interpretadores Scheme na discussão de alguns aspectos de linguagens, dentre eles a diferença entre escopo estático e dinâmico.

7.4 AMBIENTE DE PRIMEIRA CLASSE

7.5 QUANDO USAR EVAL

Da mesma forma que macros e continuações, o uso de `eval` é normalmente desencorajado.

O primeiro (e mais importante) motivo para evitar `eval` está relacionado a segurança de aplicações: quando dados entrados pelo usuário são passados para `eval`, o programa torna-se vulnerável a ataques (o usuário pode passar qualquer forma Scheme para ser interpretada, e não apenas aquelas imaginadas pelo programador). Este problema pode ser minimizado se filtrarmos as formas passadas para `eval`, mas isto não é simples. No entanto, quando não se trata de um servidor mas sim de uma aplicação isolada que deve ser usada individualmente por usuários, este problema torna-se menos importante; se o código passado para `eval` é gerado pelo próprio programa (como em Programação Genética), o problema deixa de existir.

O código passado para `eval` não é compilado, ainda que o resto do programa seja, e isto pode tornar o programa mais lento que o necessário[3];

Finalmente, o uso indiscriminado de `eval` pode tornar um programa ilegível e dificultar sua manutenção.

3 Podemos imaginar uma implementação Scheme que compile código sob demanda, mas a compilação de cada forma passada para `eval` também demandaria tempo.

Versão Preliminar

EXERCÍCIOS

Ex. 105 — Muitas linguagens (por exemplo Python e PHP) tem uma função `eval`, que toma uma string e a interpreta como um programa. Descreva o que teria que ser modificado no sistema de programação genética deste capítulo se o `eval` de Scheme somente aceitasse strings.

Ex. 106 — A função de fitness que usamos no sistema de programação genética calcula a soma dos quadrados dos erros entre os valores desejados para y e os valores obtidos pelo indivíduo (a forma Scheme). Suponha que queiramos aproximar $f(x) = \cos(x)$. Uma forma Scheme `(+ (cos x) 500)` terá um valor de fitness muito baixo, quando na verdade está apenas transladada. Pense em uma maneira de valorizar mais este indivíduo, justifique sua ideia e a implemente. Comente os resultados: a nova função de fitness melhorou o desempenho do sistema?

Ex. 107 — Modifique o interpretador meta-circular deste capítulo para incluir avaliação preguiçosa por default.

Ex. 108 — Modifique o interpretador meta-circular descrito neste Capítulo para incluir a forma especial `cond`.

Ex. 109 — Escreva um interpretador de Scheme em C. Você pode começar com o padrão R^4RS, que é bastante simples, e aos poucos adicionar características de R^5RS e R^7RS.

RESPOSTAS

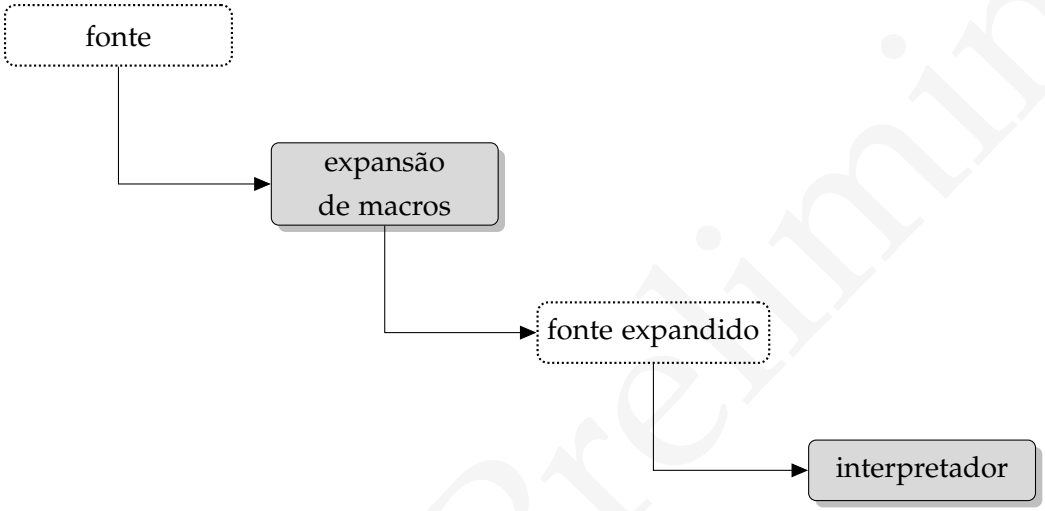
Resp. (Ex. 106) — Podemos calcular as distâncias entre os pontos alvo e os pontos calculados pelo programa, $\vec{d} = |y_i - \hat{y}_i|$. A variância do vetor $\vec{d}$ é uma maneira de medir quanto as curvas são semelhantes, a não ser por translação vertical: $\sigma^2(\vec{d}) = \frac{1}{n-1}\sum_i (\overline{d} - d_i)^2$. A função de fitness pode usar uma média ponderada entre o erro quadrático e e a adequação c da curva: por exemplo, $0.8c + 0.2e$.

Versão Preliminar

8 MACROS

Macros são regras para transformar programas antes que eles sejam interpretados ou compilados. Estas transformações se dão em pequenos trechos e código.

A próxima Figura mostra um diagrama do processo de expansão de macros.

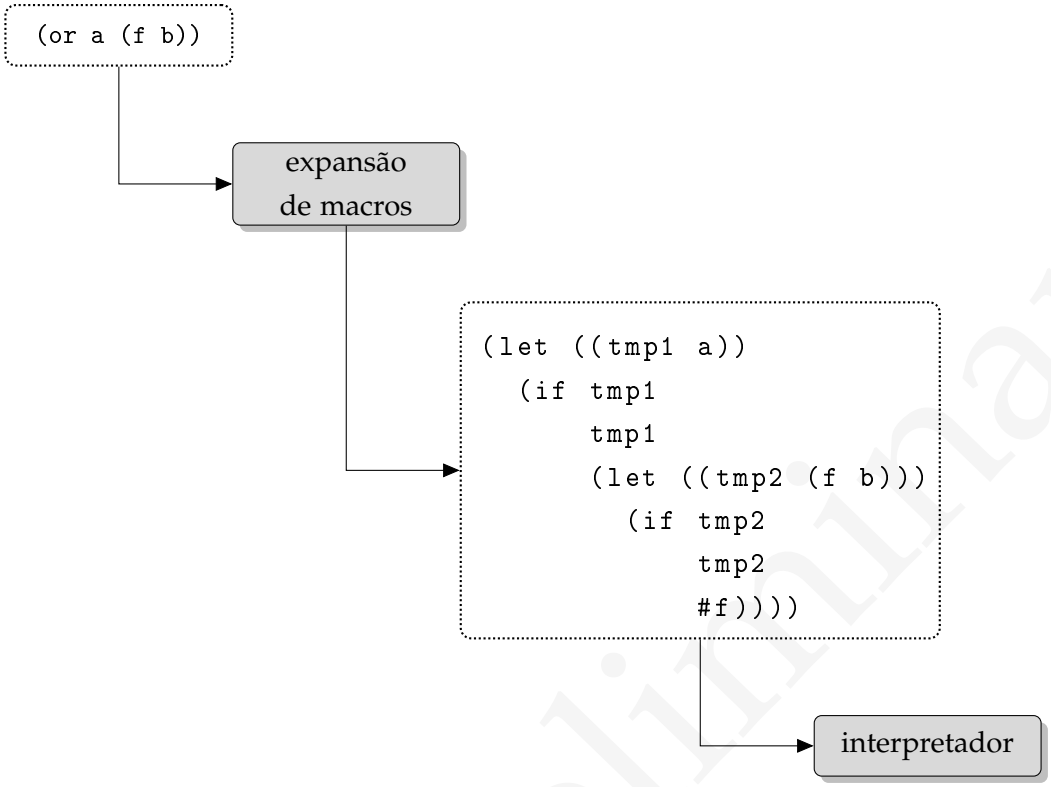


Formas especiais são macros pré-definidas na implementação Scheme. Por exemplo a forma especial `or` é transformada em uma série de `if`s encadeados. A forma `(or a (f b))` é expandida para

```
(let ((tmp1 a))
  (if tmp1
      tmp1
      (let ((tmp2 (f b)))
        (if tmp2
            tmp2
            #f))))
```

Esta expansão avalia os argumentos do `or` no máximo uma vez, da esquerda para a direita, e deixa de avaliar quando um dos argumentos for diferente de `#f` – isso é exatamente o que determina o padrão de Scheme. A Figura a seguir ilustra este exemplo de expansão de macro.

Versão Preliminar



As transformações feitas durante a expansão de macros podem ser arbitrariamente complexas, e são feitas usando linguagens de programação (isso contrasta com a expansão de macros em C, onde só o que se pode fazer é substituir variáveis em *templates*).

Há vários diferentes sistemas de macro para Scheme; este capítulo aborda primeiro um sistema mais antigo e que não chegou a ser padronizado, chamado `defmacro` ou `define-macro`, e depois disso o sistema padrão do R^{5}RS e alguns outros. Embora o padrão R^{5}RS defina apenas a mini-linguagem `syntax-rules` para construção de transformadores de sintaxe, a grande maioria das implementações de Scheme oferece também algum outro mecanismo. É mais natural que uma discussão sobre macros e higiene inicie com macros não higiênicas, para que fique claro o problema que as macros higiênicas tentam resolver.

Este Capítulo inicia com uma breve discussão da forma especial *quasiquote*, usada na construção de listas e especialmente útil em macros. Depois há a descrição de três sistemas de macros: o não-higiênico `define-macro`, o sistema higiênico `syntax-rules` e o sistema com renomeação explícita, onde a higiene é controlada.

Versão Preliminar

8.1 QUASIQUOTE

Descrevemos no Capítulo 1 a forma especial `quote`, que impede a avaliação de seu argumento. Assim, `(quote (+ 10 5))` resulta em uma lista, e não no número 15. Ao escrever macros não-higiênicas usaremos outras formas especiais que tem efeito parecido com o de `quote`: nossas macros são procedimentos que geram formas Scheme (que na grande maioria das vezes são listas).

Se quisermos construir a lista `(set! stack (cons elt stack))` mantendo todos os símbolos fixos exceto `stack` e `elt`, que devem ser parâmetros, podemos escrever

```
(lambda (stack elt)
  (list 'set! stack (list 'cons elt stack)))
```

mas esta intercalação de `list` com símbolos, alguns com `quote` e outros sem, pode se tornar confusa.

A forma especial `quasiquote` funciona como `quote`, impedindo a avaliação de seu argumento, mas com uma exceção: a forma especial `unquote`, quando usada dentro de `quote`, determina que seu argumento seja avaliado. Podemos entender `quasiquote` e `unquote` como duas chaves: uma que desliga a avaliação (`quasiquote`) e outra que a liga novamente (`unquote`). O exemplo a seguir ilustra o funcionamento de `quasiquote`.

```
(quasiquote ((unquote (+ 1 2))
             (+ 3 4)))
```

```
(3 (+ 3 4))
```

Assim como `(quote forma)` é normalmente abreviado por `'forma`, também `(quasiquote forma)` é abreviado por `` `forma `` e `(unquote forma)` é abreviado como `,forma`. O exemplo anterior pode ser reescrito com as abreviações:

```
`(,(+ 1 2) (+ 3 4))
```

```
(3 (+ 3 4))
```

Nosso primeiro exemplo fica muito mais claro usando quasiquote: podemos pensar como se tivéssemos um *template* de lista, e inserimos valores nas posições onde há vírgulas.

```
(lambda (stack elt)
  `(set! ,stack (cons ,elt ,stack)))
```

Versão Preliminar

8.1.1 Unquote-splicing

Ao construir listas constantes, podemos querer fazer algo semelhante a `cons`: dada uma lista constante `(f1 f2 f3)`, queremos a lista `(f0 f1 f2 f3)`. Não conseguiremos fazê-lo com `quasiquote`:

```
(let ((a '(f1 f2 f3)))
  `(f0 ,a))
(f0 (f1 f2 f3))
```

Nosso problema é que o valor de `a` é uma lista, e o `quasiquote` usado na segunda linha usou este valor para o segundo elemento de outra lista. Se quisermos "espalhar" os elementos da lista `a` dentro de outra lista, temos que citá-la com `unquote-splicing`:

```
(let ((a '(f1 f2 f3)))
  `(f0 (unquote-splicing a)))
(f0 f1 f2 f3)
```

A abreviação para `(unquote-splicing forma)` é `,@forma`.

```
(let ((a '(f1 f2 f3)))
  `(f0 ,@a))
(f0 f1 f2 f3)
```

8.2 TRANSFORMADORES DE SINTAXE

Nos sistemas modernos de macros para Scheme, as macros são definidas da seguinte maneira:

```
(define-syntax nome-da-macro
  <syntax transformer>)
```

Uma macro tem um nome e seu corpo contém um transformador de sintaxe *(syntax transformer)*. As próximas Seções apresentam em detalhes dois transformadores de sintaxe, `er-macro-transformer` e `syntax-rules`.

Versão Preliminar

8.3 R^{5}RS E SYNTAX-RULES

O padrão R^{5}RS definiu uma linguagem específica para a escrita de macros.

O transformador de sintaxe `syntax-rules` funciona usando casamento de padrões. A definição de uma macro com `syntax-rules` é como segue.

```
(define-syntax <nome>
  (syntax-rules (<palavras-chave>)
    ((<padrao_1>) <template_1>)
    ...
    ((<padrao_n>) <template_n>)))
```

Durante a fase de expansão de macros, quando uma forma for encontrada iniciando com o nome de uma macro, o interpretador tentará casar a forma com um dos padrões listados. O primeiro padrão a casar será expandido de acordo com seu template.

Começamos com um exemplo bastante simples, e talvez artificial, que trará uma compreensão melhor da natureza das macros e do transformador `syntax-rules`. Suponha que queiramos atribuir o mesmo valor a duas variáveis. Normalmente faríamos

```
(begin
  (set! a val)
  (set! b val))
```

Se este trecho de código se repete muitas vezes em um programa, seria interessante abstraí-lo de forma que pudéssemos simplesmente usar

```
(set-both! a b val)
```

No entanto, não podemos fazê-lo com um procedimento.

```
(define proc-set-both!
  (lambda (a b val)
    (set! a val)
    (set! b val)))
```

O procedimento `proc-set-both!`, ao ser usado, avaliará os argumentos `a` e `b`, e quando a forma `(set! a val)` for avaliada, o sistema Scheme dirá que a variável `a` não está vinculada:

```
(define u 1)
(define v 2)
(proc-set-both! u v 20)
```

Versão Preliminar

```
u
1
v
2
```

O que aconteceu foi:

- Ao chamarmos `proc-set-both!`, seus argumentos – os elementos da lista `(u v 20)` – deveriam ser avaliados antes de serem passados ao procedimento. Então o procedimento é chamado com os argumentos `(1 2 20)`.
- Dentro do procedimento, as referências a `a` e `b` são para variáveis locais que não tinham vínculo. Elas são ambas modificadas e passam a ter o valor 20.
- Ao retornar do procedimento, as variáveis `a` e `b`, *que eram locais*, são abandonadas.

Gostaríamos então de escrever procedimentos de forma que seus argumentos não fossem automaticamente avaliados – e para isso usaremos macros.

```
(define-syntax set-both!
  (syntax-rules ()
    ((set-both! a b value) ;; <= este é o padrão que deve
                           ;; casar com a forma
     ;; abaixo está o template de código que
     ;; substituirá a forma:
     (begin
       (set! a value)
       (set! b value)))))
(define u 1)
(define v 2)
(set-both! u v 20)
u
20
v
20
```

Passamos a forma `(set-both! u v 20)` para o sistema Scheme, e a forma foi expandida *antes* de ser avaliada; a forma expandida, que foi passada para o interpretador, é

Versão Preliminar

```
 (begin
       (set! u 20)
       (set! v 20)
```

Prosseguimos agora com mais exemplos de macros usando `syntax-rules`, usando muitos de seus diferentes recursos.

Não há em Scheme uma forma especial que repita a execução de um trecho de programa um número determinado de vezes. Uma macro poderia fazê-lo. Construiremos uma macro `vezes`: o código que deverá ser gerado a partir de "`(vezes n body ...)`" é:

```
(let loop ((i 0))
  (if (< i n)
      (begin
        body ...
        (loop (+ 1 i)))
      #f))
```

Queremos mostrar tres vezes as onomatopeias *POW!* e *BONK!*, típicas de um antigo seriado de televisão.

```
(vezes 3 (display "POW! ")
         (display "BONK! "))
```

Esta forma será expandida para:

```
(let loop ((i 0))
  (if (< i n)
      (begin
        (display "POW! ")
        (display "BONK! ")
        (loop (+ 1 i)))
      #f))
```

A macro `vezes` é definida a seguir.

Versão Preliminar

```
(define-syntax vezes
  (syntax-rules ()
    ((vezes n body ...) ;; <= padrao
      ;; template:
      (let loop ((i 0))
        (if (< i n)
            (begin
              body ...
              (loop (+ 1 i)))
            #t)))))
(vezes 3 (display "POW! ") (display "BONK! "))
POW! BONK! POW! BONK! POW! BONK! #t
```

A elipse ("...") usada na definição da macro casa com a cauda de uma lista.

> *As variáveis usadas dentro de* `syntax-rules` *não tem qualquer relação com as variáveis do programa Scheme.*

A macro acima usa uma variável "i" dentro do `syntax-rules`, mas o código gerado não contém uma variável "i":

```
(vezes 3 (display i)(newline))
Error: unbound variable: i
(let ((i 2)) (vezes 3 (display i) (newline)))
2
2
2
#t
```

8.3.1 Palavras-chave

A macro `vezes` define a expansão para formas que começam com o identificador `vezes`, como se ele fosse um procedimento. Em algumas situações pode ser necessário incluir palavras-chave em outras posições da S-expressão. Um exemplo simples é a implementação de uma macro `for`.

Versão Preliminar

```
(FOR x IN lista DO: body)
(FOR x FROM 1 TO n DO: body)
```

Uma primeira tentativa de construir a macro `for` é:

```
(define-syntax for
  (syntax-rules ()

    ((for x in lista do: body ...)
      ;; template para percorrer lista:
      (let loop ((l lista))
        (if (not (null? l))
            (let ((x (car l)))
              body ...
              (loop (cdr l)))
            #f)))

    ((for x from start to end do: body ...)
      ;; template para contagem:
      (do ((x start (+ 1 x)))
          ((> x end))
        body ...))))
```

Aparentemente, a macro funciona:

```
(for e in '(a b c) do: (display e) (newline))
a
b
c
#f
```

No entanto, ela parece funcionar também quando não deveria:

```
(for e blah '(a b c) blah (display e) (newline))
a
b
c
#f
```

Versão Preliminar

O padrão `(for x in lista do body ...)` contém cinco identificadores após o `for`, e cada um dos cinco casou com um dos argumentos passados para a macro:

```
x         → e
in        → blah
lista     → '(a b c)
do:       → blah
body ...  → (display e) (newline)
```

Este problema não aparenta ser tão sério, mas a segunda forma do `for` não funciona:

```
(for i from 10 to 15 do (display i) (newline))
Error: (car) bad argument type: 10
```

O interpretador nos diz que tentou aplicar `car` no argumento "10". Isso significa que a regra aplicada foi a *primeira* (para listas) e não a segunda (para contagem).

De fato, o casador de padrões percorre cada padrão, do primeiro ao último, e para de percorrê-los quando encontra um que case com os argumentos da macro. Neste caso, os argumentos casam com o primeiro padrão:

```
x         → i
in        → from
lista     → 10
do:       → to
body ...  → 15 do (display i) (newline)
```

O problema parece ser simples: a macro não sabe que certos identificadores deveriam ser usados para diferenciar um padrão de outro. Para informarmos a macro disto basta incluirmos a lista de literais como primeiro argumento para `syntax-rules`:

Versão Preliminar

```
(define-syntax for
  (syntax-rules (in do: from to)  ;; <== aqui!

    ((for x in lista do: body ...)
      ;; template para percorrer lista:
      (let loop ((l lista))
        (if (not (null? l))
            (let ((x (car l)))
              body ...
              (loop (cdr l)))
            #f)))

    ((for x from start to end do: body ...)
      ;; template para contagem:
      (do ((x start (+ 1 x)))
          ((> x end))
        body ...))))
```

Com esta modificação a macro já não aceita mais qualquer identificador nas posições de `in`, `do`, `from` e `to`:

```
(for x blah '(a b c) blah (display x) (newline))
Error: during expansion of (for ...) - no rule matches form: (for x blah
(quote (a b c)) blah (display x) (newline))
```

Além disso, passa a funcionar para os dois casos:

```
(for i from 10 to 15 do: (display i) (newline))
10
11
12
13
14
15

(for x in '(a b c) do: (display x) (newline))
a
b
c
```

Versão Preliminar

```
#f
```

8.3.2 Higiene

Macros criadas com `syntax-rules` são chamadas de *higiênicas* porque os símbolos usados nas especificações de macros com `syntax-rules` não tem relação com os símbolos usados como nomes para variáveis do programa Scheme. *Uma macro* `syntax-rules` *não pode inserir vínculos no programa!*

```
(define-syntax with-epsilon
    (syntax-rules ()
      ((_ expr)
        (let ((epsilon 0.000001)) expr))))
(with-epsilon (> 0.5 epsilon))
ERROR: undefined variable: epsilon
```

8.3.3 Número variável de parâmetros

As elipses usadas nas macros `vezes` e `for` permitem usar listas de tamanho arbitrário como parâmetros para macros.

Na maioria das implementações de Lisp (inclusive Scheme), `and` e `or` são implementados como macros.

```
(define-syntax meu-and
  (syntax-rules ()
    ((meu-and) #t)
    ((meu-and e1 e2 ...)
     (if e1
         (meu-and e2 ...)
         #f))))
(meu-and 1 2)
#t
(meu-and #f 4)
#f
(meu-and)
```


Versão Preliminar

```
#t
```

8.3.4 Erros de sintaxe

R^{7}RS

A forma `syntax-error` pode ser usada para indicar que uma macro foi usada de maneira incorreta.

A macro `vezes` que construímos não funciona quando não informamos o número de repetições:

```
(vezes)
ERROR: car: not a pair: #<opcode cons>
```

A mensagem de erro não é muito útil. Podemos torná-la mais informativa usando `syntax-error`.

```
(define-syntax vezes
  (syntax-rules ()
    ((vezes n body ...) ;; <= padrao
      ;; template:
      (let loop ((i 0))
        (if (< i n)
            (begin
              body ...
              (loop (+ 1 i)))
            #t)))
    ((vezes)
     (syntax-error "vezes: número de vezes não informado"))))
(vezes)
ERROR: vezes: número de vezes não informado
```

8.3.5 Depurando macros

Muitas implementações de Scheme oferecem algum procedimento que permite verificar como uma forma será expandida.

Versão Preliminar

```
(define-syntax unless
  (syntax-rules ()
    ((_ test then)
     (if (not test)
         then))))

(expand '(unless (= 1 2)
           (print 'x)))
(IF (not198 (= 1 2)) (print (quote x)))
```

Quando o casamento dos padrões na macro causa confusão, podemos criar uma macro que mostre os vínculos de cada variável do padrão.

8.3.6 A linguagem completa de *syntax-rules*

(Esta seção está incompleta)

A forma especial `syntax-rules` gera um *transformador de sintaxe* – um procedimento que recebe uma S-expressão e devolve outra. Sua forma geral é

```
(syntax-rules (<literal_1> <literal_2> ...)
  ( ( <padrao_1> )
    ( <template_1> ) )
  ( ( <padrao_2> )
    ( <template_2> ) )
  ... )
```

A lista `<literal_1> <literal_2> ...` pode ser vazia.

Os padrões usados em `syntax-rules` podem ser listas, vetores, identificadores ou constantes.

- `...` é uma *elipse*. Na discussão a seguir ela será representada por "`ELIPSE`";
- Identificadores aparecendo na lista de palavras-chave são *literais*;
- Outros identificadores são *variáveis de padrão*.

Ao decidir se uma entrada `X` casa com um padrão `P` as seguintes possibilidades são verificadas, *nesta ordem*:

Versão Preliminar

1. P é uma variável – neste caso o casamento é de P com X;
2. P é um identificador literal e X é um identificador literal com o mesmo vínculo;
3. P é da forma `(P1 ... Pn)` e X é uma lista de elementos. Os elementos da lista casam com `P1` a `Pn`;
4. P é da forma `(P1 ... Pn . Px)` e X é uma lista (própria ou imprópria) de n ou mais elementos. Os n primeiros casam com P1 o n-ésimo `cdr` casam com `Px`;
5. P é da forma `(P1 ... Pk Pe ELIPSE Pm+1 ... Pn)` e X é uma lista própria de n elementos. Os k primeiros elementos devem casar com `P1 ... Pk`. Os outros $m - k$ elementos casam com `Pe`; os outros $n - m$ elementos casam com `Pm+1` até `Pn`;
6. P é da forma `(P1 ... Pk Pe ELIPSE Pm+1 ... Pn . Px)`, e X é uma lista *imprópria* com n elementos. Os k primeiros elementos de X casam com `P1 ... Pk`. Os outros $m - k$ elementos casam com `Pe`; os outros $n - m$ elementos casam com `Pm+1` até `Pn`; o último `cdr` casa com `Px`;
7. P é da forma `#(P1 ... Pn)` e X é um vetor cujos n elementos casam com P1...Pn.
8. P é da forma `#(P1 ... Pk Pe ELIPSE Pm+1 ... Pn)`, e X é um vetor com n *ou mais* elementos. Os primeiros k elementos do vetor devem casar com `P1 ... Pk`. Os próximos $m - k$ elementos devem casar com `Pe`. Os outros $n - m$ elementos devem casar com `Pm+1` e `Pn`;
9. P é um dado diferente de lista, vetor ou símbolo e X é `equal?` a P.

Quando uma entrada casa com um padrão, ela é substituída pelo template correspondente, já com as trocas identificadas no casamento de padrões. Neste template, os literais permanecem inalterados, mas as variáveis são renomeadas (de forma transparente ao programador) para não gerar conflito com nomes já usados em outras partes do código.

8.3.7 Exemplo: estruturas de controle

É comum em Scheme e em outros lisps a implementação de estruturas de controle como macros. Esta seção apresenta três macros simples para controle: `when`, `unless` e `while`.

A macro `when` é útil quando queremos avaliar várias expressões dependendo do resultado de um teste, como neste exemplo:

Versão Preliminar

```
(if (> n 0)
  (begin
    ...
    ...))
```

Podemos usar o `if` com um só braço, mas precisamos do `begin` – e isto é ligeiramente inconveniente. O `cond` também não parece natural:

```
(cond ((> n 0)
       ...
       ...))
```

Criaremos então uma forma `when`, que resolverá o problema:

```
(when (> n 0)
  ...
  ...)


(define-syntax when
  (syntax-rules ()
    ((_ test body ...)
     (if test
         (begin body ...)))))
```

A macro `unless` é semelhante a `when`, a não ser pela negação do teste.

```
(define-syntax unless
  (syntax-rules ()
    ((_ test body ...)
     (if (not test)
         (begin body ...)))))
```

A macro `while` é também muito simples:

Versão Preliminar

```
(define-syntax while
  (syntax-rules ()
    ((_ test body ...)
     (let loop ()
       (if test
         (begin body ...
                (loop)))))))
```

8.3.8 Exemplo: *framework* para testes unitários

8.3.9 Sintaxe local

Definições de macro (válidas para todo o programa) são feitas com `define-syntax`. Definições locais de macro podem ser feitas com `let-syntax` e `letrec-syntax`, que funcionam exatamente como `define-syntax`, exceto por definirem macros visíveis apenas no escopo onde estiverem.

No próximo exemplo, a macro `vezes` é introduzida apenas para uso em um procedimento (e não poderá ser usada fora dele):

```
(define linha
  (lambda (n)
    (let-syntax ((vezes
                  (syntax-rules ()
                    ((vezes condicao body ...)
                     (let loop ((i 0))
                       (if (< i n)
                           (begin
                             body ...
                             (loop (+ 1 i)))
                           #f))))))
      (vezes n
        (write-char #\-))
      (newline))))
```

O exemplo a seguir encontra-se no documento que define o padrão R^{5}RS:

Versão Preliminar

```
(letrec-syntax
    ((my-or (syntax-rules ()
              ((my-or) #f)
              ((my-or e) e)
              ((my-or e1 e2 ...)
               (let ((temp e1))
                 (if temp
                     temp
                     (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))
```

A macro definida com `letrec-syntax` é definida recursivamente, e passará por várias expansões até que o código gerado não tenha mais macros a serem expandidas.

A lista de variáveis no `let` do último exemplo pode parecer um pouco estranha. Os identificadores `let` e `if` são redefinidos – e Scheme permite fazê-lo!

```
(let ((let odd?))
  (display let))

(let ((if even?))
  (display if))

(let ((if even?))
  (display (if #f 1 2)))
```

Versão Preliminar

8.3.10 Armadilhas de *syntax-rules*

O transformador de macros `syntax-rules` é útil e fácil de usar quando se quer escrever macros simples. Ao desenvolver macros que envolvem mais engenhosidade surgem diversos pequenos problemas e armadilhas que devem ser conhecidos pelo programador.

8.3.10.1 *Símbolos livres no corpo da macro*

Se no corpo de uma macro há referência a um símbolo que não estava no padrão que casou, a vinculação dele será buscada no ambiente em vigor quando a macro foi *definida* (antes de ser expandida em qualquer lugar):

```
(define-syntax soma
  (syntax-rules ()
    ((_ a b)
     (+ a b c))))
```

A macro foi definida fora de qualquer procedimento ou bloco, portanto o `c` se refere ao vínculo global do símbolo `c`. Uma primeira tentativa de uso da macro falhará porque `c` não está definido:

```
(soma 3 4)
ERROR: unbound variable: c
```

Usar `let` cria um novo ambiente com um novo vínculo para `c`, mas este não é o vínculo global para `c`, portanto não resolverá o problema:

```
(let ((c 10)) (soma 3 4))
ERROR: unbound variable: c
```

Se um vínculo global para `c` for definido, a macro funcionará:

```
(define c 10)
(soma 3 4)
17
```

8.3.10.2 *Não há sobreposição para variáveis de padrão*

O problema descrito a seguir foi descrito por Oleg Kiselyov.

Como em Scheme o escopo das variáveis é léxico, o `x` na função `inc` é diferente do `x` em `funcao`. Diz-se que o símbolo com definição interna sobrepõe (*shadows*) o outro.

Versão Preliminar

```
(define funcao
  (lambda (x)
    (let ((inc (lambda (x) (+ 1 x))))
      (inc (* 2 x)))))
(funcao 3)
7
```

O procedimento acima é equivalente a:

```
(define funcao
  (lambda (x)
    (let ((inc (lambda (y) (+ 1 y))))
      (inc (* 2 x)))))
```

onde trocamos `x` por `y` no procedimento interno.

Em macros definidas com `syntax-rules` símbolos em macros internas *não* tem prioridade sobre os mais externos:

```
(define-syntax mac
  (syntax-rules ()
    ((_ x)
     (let-syntax
         ((inc
           (syntax-rules ()
             ((_ x) (+ 1 x)))))
       (inc (* 2 x))))))
```

Uma tentativa de usar `(mac 3)` resultará em erro, porque o `x` do primeiro padrão (de `mac`) é o mesmo do segundo (o de `inc`):

```
(macroexpand '(mac 3))


(let-syntax ((inc (syntax-rules ()
                    ((_ 3) (+ 1 3)))))
  (inc (* 2 3)))
```

O padrão de `inc` já foi escrito com o valor casado com `x` no padrão de `mac`.

A solução para este problema é sempre usar símbolos diferentes em macros aninhadas:

Versão Preliminar

```
(define-syntax mac
  (syntax-rules ()
    ((_ x)
     (let-syntax
         ((inc
           (syntax-rules ()
             ((_ y) (+ 1 y)))))
       (inc (* 2 x))))))
```

8.4 MACROS COM RENOMEAÇÃO EXPLÍCITA

Mencionamos neste Capítulo que uma definição de macro é normalmente da forma

```
(define-syntax nome-da-macro
  <syntax transformer>)
```

e apresentamos o transformador de sintaxe `syntax-rules`.

Há outro tipo de transformador de sintaxe, que chamamos de *transformador com renomeação explícita*, ou `er-transformer`.

Há duas diferenças essencias entre `syntax-rules` e `er-transformer`:

- Usando `syntax-rules` especificamos a transformação de uma forma em outra usando uma linguagem casamento de padrões. Ao usar `er-transformer`, é necessário escrever um procedimento Scheme que receba a forma original e a transforme em outra. Dizemos que `er-transformer` é um transformador *"de baixo nível"* e que `syntax-rules` é *"de alto nível"*.
- Enquanto `syntax-rules` é higiênico, `er-transformer` não é. A não ser que peçamos o contrário, os símbolos usados quando transformamos formas tem os mesmos vínculos que no resto do programa.

Um transformador `er-transformer` é sempre da seguinte forma:

```
(er-macro-transformer
  (lambda (x ren cmp)
    ;; código que lê a forma x e devolve uma forma
    ;; transformada, usando os procedimentos ren e cmp.))
```

Versão Preliminar

- `ren` é o nome de um procedimento que renomeia um símbolo, devolvendo outro símbolo que não capturará qualquer variável;
- `cmp` é o nome de um procedimento que compara dois símbolos usados na macro.

A macro `vezes`, que havíamos construído com `syntax-rules`, pode ser definida também com `er-macro-transformer`.

```
(define-syntax vezes
  (er-macro-transformer
    (lambda (x r c)
      (let ((n (cadr x))
            (body (cddr x))
            (rloop (r 'loop))
            (ri (r 'i))
            (r< (r '<)))
        `(,(r 'let) ,rloop ((,ri 0))
          (,(r 'if) (,r< ,ri ,n)
            (,(r 'begin)
              ,@body
              (,rloop (+ 1 ,ri)))
            #t))))))
```

8.4.1 Macros anafóricas

Em Lisp, dizemos que uma macro é *anafórica* quando ela nos permite usar uma expressão como "`isto`" ou "`it`" para nos referirmos a uma expressão (uma forma) já avaliada[1].

1 *Anáfora* é um termo que tem dois significados. Pode se referir tanto a uma figura de linguagem onde uma palavra ou grupo de palavras se repete muitas vezes em frases ou versos, como no exemplo a seguir:

> *Morena de Angola que leva o chocalho amarrado na canela*
> *Será que ela mexe o chocalho ou o chocalho é que mexe com ela*
> *Será que ela tá na cozinha guisando a galinha à cabidela*
> *Será que esqueceu da galinha e ficou batucando na panela*

A repetição de *"Será que"* é chamada de anáfora.
Pode também denominar, de acordo com a nomenclatura da Linguística, uma partícula como "ele", quando ela faz referência a alguma expressão anterior:

> *"Arnesto nos convidou pra um samba, ele mora no Brás"*

Neste exemplo, *"ele"* é uma anáfora.

Versão Preliminar

Suponha que queiramos escrever um procedimento anônimo (porque só o usaremos uma vez) e recursivo; poderíamos usar `(lambda ...)` para isso, mas não teríamos como fazê-lo recursivo porque ele não tem um nome que possamos chamar recursivamente:

```
(lambda (n)
        (if (< n 2)
            1
            (* n (EU-MESMO (- n 1)))))
```

Neste exemplo tentamos escrever o procedimento fatorial[2] recursivo usando uma expressão `lambda`, mas não pudemos faze-lo porque não temos um nome para incluir na posição `EU-MESMO`.

Podemos usar `letrec`,mas teríamos que inventar um nome para o procedimento. Seria interessante se pudéssemos usar alguma palavra como `self` ou `this` nesta situação. Para isso a forma especial `lambda` teria que *introduzir um vínculo para o símbolo* `self`. A forma `lambda` é parte do núcleo da linguagem Scheme, e não queremos modificá-la. Podemos no entanto criar uma nova forma, `alambda` (*anaphoric lambda*), que vincula o símbolo `self` ao procedimento anônimo que estamos criando.

```
(define-syntax alambda
  (er-macro-transformer
   (lambda (form ren cmp)
     (let ((args (cadr form))
           (body (caddr form)))
       `(letrec ((self (lambda ,args ,body)))
          self)))))
```

O trecho a seguir:

```
(alambda args
  body)
```

será expandido para

```
(letrec ((self (lambda args
                 body)))
  self)
```

A definição de fatorial com `alambda` é exatamente como havíamos tentado, trocando `EU-MESMO` por `self`:

2 Na prática a função fatorial normalmente é calculada como logaritmo da função Gama.

Versão Preliminar

```
(define f
  (alambda (n)
           (if (< n 2)
               1
               (* n (self (- n 1))))))
(f 5)
120
```

Podemos escrever diversas outras macros anafóricas. Um outro exemplo é o `aif`.

Em Scheme, qualquer valor diferente de `#f` pode ser usado como verdadeiro em um condicional:

```
(if 0 'ok 'nope)
ok
(if "uma string"'ok 'nope)
ok
(not #x)
#f
(not -1)
#f
```

Dizemos que Scheme tem suporte a *booleanos generalizados*.

A forma `if` avalia um teste condicional, mas não nos dá acesso a ele:

```
(if (func x y)
    (display (func x y))) ; <= tivemos que repetir (func x y)
```

Da mesmo forma que a macro `alambda` vincula `self` ao procedimento sendo definido, a macro `aif` vincula `it` ao resultado do teste do `if`.

```
(define-syntax aif
  (er-macro-transformer
   (lambda (x r c)
     (let ((xtest (cadr x))
           (xthen (caddr x))
           (xelse (cadddr x)))
       `(,(r 'let) ((it ,xtest))
         (,(r 'if) it ,xthen ,xelse))))))
```

Criamos um procedimento `func` para testar nossa macro.

Versão Preliminar

```
(define func
  (lambda a b)
    (if (= a b)
        #f
        (- a b)))
```

Agora podemos usar `it` no corpo do `if` para nos referir ao resultado do teste.

```
(aif (func 2 3)
     (display it)
     (display 'nope))
```

```
-1
```

```
(aif (func 10 10)
     (display it)
     (display 'nope))
```

```
nope
```

Macros como `aif` e semelhantes (`acond`, `awhile`, `awhen`, por exemplo) não são muito usadas em Scheme, porque não é comum o uso de booleanos generalizados em programas Scheme. Estes são comuns, no entanto, em programas Common Lisp (onde também estas macros são mais frequentemente usadas).

8.5 PROBLEMAS COMUNS A TODOS OS SISTEMAS DE MACRO

Há sutilezas e pequenas armadilhas inerentes ao desenvolvimento de macros, higiênicas ou não. Esta seção trata de alguns desses tópicos.

8.5.1 Número de avaliações

A macro a seguir é uma implementação de `or`:

Versão Preliminar

```
(define-syntax or*
  (syntax-rules ()
    ((_) #f)
    ((_ test)
     (if test
         test
         #f))
    ((_ test1 test2 ...)
     (if test1
         test1
         (or* test2 ...)))))
(or* (begin (display 5)(newline) 10) #f)
5
5
10
```

O 5 foi mostrado duas vezes porque foi *avaliado* duas vezes:

```
(expand '(or* (begin (display 5) (newline) 10) #f)
(if (begin (display 5) (newline) 10) (begin (display 5) (newline) 10) (or*
#f))
```

É comum que se queira avaliar os argumentos de macros uma única vez. No exemplo acima o argumento da macro apenas mostra o número 5 – o que não parece um problema grave. No entanto, os efeitos colaterais do argumento da macro poderiam ser mais importantes: uma variável pode ser incrementada duas vezes ao invés de uma, ou uma mensagem pode ser enviada pela rede mais de uma vez, por exemplo.

O problema pode ser resolvido avaliando cada argumento uma única vez em uma variável temporária:

Versão Preliminar

```
(define-syntax or*
  (syntax-rules ()
    ((_) #f)
    ((_ test)
     (let ((result test))
       (if result
           result
           #f)))
    ((_ test1 test2 ...)
     (let ((result test1))
       (if result
           result
           (or* test2 ...))))))
(or* (begin (display 5) (newline) 10) #f)
5
10

(expand '(or* (begin (display 5) (newline) 10) #f))
(let ((result108 (begin (display 5) (newline) 10))) (if result108 result108
(or* #f)))
```

8.5.2 Tipos de variáveis e seus valores

(esta seção está incompleta)

Durante a expansão de macros, não necessariamente temos informação a respeito dos tipos de variáveis ou de seus valores.

Embora macros possam expandir a si mesmas recursivamente, há cuidados que devemos tomar quando usarmos este tipo de mecanismo.

8.6 QUANDO USAR MACROS

Macros são uma forma de modificar S-expressões antes de sua interpretação, e são úteis quando uma função não poderia ser usada porque não se quer avaliar as expressões

Versão Preliminar

passadas como parâmetros (como por exemplo na implementação de `my-and` mostrada neste capítulo).

Não se recomenda o uso de macros para ganhar eficiência; qualquer lentidão em um programa deve ser identificada com ferramentas adequadas (*profilers*), e uma vez identificado o problema, as soluções tentadas devem ser primeiro a melhoria do algoritmo usado e a limpeza de código, e em seguida a declaração de alguns procedimentos como *inline*[3] (como fazê-lo depende da implementação de Scheme). Uma das coisas que se pode fazer com funções e não com macros é usá-las como parâmetros. Suponha, por exemplo, que uma função `plus1` seja usada como forma curta de `(+ 1 x)`, e que se queira diminuir o tempo usado para chamá-la:

```
(define plus1
  (lambda (x) (+ 1 x)))
```

A função `plus1` é entidade de primeira classe, e pode ser passada como argumento para procedimentos:

```
(map plus1 '(1 2 3))
'(2 3 4)
(apply plus1 '(3))
4
```

Para tentar reduzir o tempo usado em chamadas de função, uma solução ingênua é implementar + como macro:

```
(define-syntax plus1
  (syntax-rules ()
    ((_ arg1)
     (+ 1 arg1))))
(macroexpand '(plus1 10))
(+ 1 10)
```

No entanto. macros não são entidades de primeira classe:

```
(map plus1 '(1 2 3))
ERROR: unbound variable plus1
(apply plus1 '(1 2 3))
ERROR: unbound variable plus1
```

3 O mesmo vale para a linguagem C!

Versão Preliminar

8.7 ABSTRAÇÃO DE DADOS COM MACROS

No Capítulo 1 construímos abstrações de diferentes objetos através da elaboração de procedimentos que operam sobre eles. Internamente, agregamos as partes de cada objeto em listas. Este Capítulo mostra algumas limitações da representação interna com listas e discute uma abstração sintática para representar objetos.

Uma maneira supostamente natural de representar partículas em Scheme é usando listas.

```
(define faz-particula
  (lambda (pos massa vel acel)
    (list 'particula pos massa vel acel)))
```

O primeiro elemento da lista é um símbolo que determina o tipo de dado "partícula". O procedimento `particula?` pode então verificar se um objeto Scheme é uma partícula:

```
(define particula?
  (lambda (p)
    (and (list? p)
         (not (null? p))
         (eqv? 'particula (car p)))))
```

Precisamos também de acessores para cada campo do objeto partícula. Criamos então uma barreira de abstração para isolar os procedimentos para listas:

```
(define pos   (lambda (x) (list-ref x 1)))
(define massa (lambda (x) (list-ref x 2)))
(define vel   (lambda (x) (list-ref x 3)))
(define acel  (lambda (x) (list-ref x 4)))

(define set-pos!
  (lambda (p v) (set-car! (list-tail p 1) v)))
(define set-massa!
  (lambda (p v) (set-car! (list-tail p 2) v)))
(define set-vel!
  (lambda (p v) (set-car! (list-tail p 3) v)))
(define set-acel!
  (lambda (p v) (set-car! (list-tail p 4) v)))
```

Queremos também um procedimento que mostra uma partícula em formato facilmente legível:

Versão Preliminar

```
(define particula->string
  (lambda (p)
    (define position->string
      (lambda (xy)
        (string-append "(x:" (number->string (car xy))
                       " y:" (number->string (cadr xy))
                       ")")))
    (string-append "p="  (position->string (pos p))
                   " m=" (number->string (massa p))
                   " v=" (number->string (vel p))
                   " a=" (number->string (acel p)))))
```

Agora podemos criar e manipular objetos do tipo `particula`:

```
(faz-particula '(2 3)
               3.0
               4.0
               1.5)
```

```
(particula (2 3) 3.0 4.0 1.5)
```

```
(let ((part (faz-particula '(2 3)
                           3.0
                           4.0
                           1.5)))
  (display (particula->string part))
  (newline)
  (set-pos! part '(4 5))
  (display (particula->string part))
  (newline))
```

```
p=(x:2 y:3) m=3.0 v=4.0 a=1.5
p=(x:4 y:5) m=3.0 v=4.0 a=1.5
```

O esquema de representação que desenvolvemos é insatisfatório:

- Temos que definir getters e setters manualmente, e de forma inconveniente para cada estrutura;

Versão Preliminar

- Temos que lidar com detalhes que não nos deveriam interessar, como o primeiro elemento da lista, que a identifica como partícula;
- O esquema é ineficiente para tipos com muitos campos, já que a busca em listas é feita linearmente[4].

Usar hashtables poderia tornar os acessores mais rápidos:

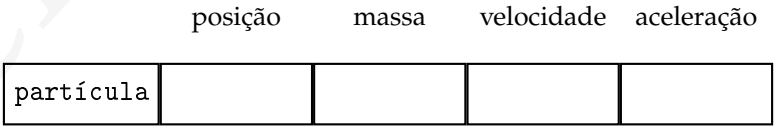
```
(define faz-particula
  (lambda ()
    (let ((p make-hash-table))
      (hash-table-set! 'sou-particula #t)
      p)))

(define particula?
  (lambda (p)
    (hash-table-exists? p 'sou-particula)))

(define pos
  (lambda (p)
    (hash-table-ref p 'pos)))
```

No entanto, as hashtables podem consumir mais memória que listas ou vetores feitos "sob medida", e ainda temos que especificar todos os setters e getters manualmente, assim como o predicado `particula?` e o procedimento `faz-particula`.

O ideal é construir um mecanismo que nos permita abstrair qualquer estrutura onde há diversos elementos referenciados por procedimentos.

Queremos armazenar os campos da estrutura em um vetor, para que o acesso seja rápido, e também queremos poder apenas enumerar as partes da estrutura e talvez os nomes dos procedimentos para acessar estas partes, mas certamente não queremos ter que pensar na implementação destes procedimentos.

Armazenaremos a estrutura inteira em um vetor, e usaremos a primeira posição para armazenar o nome do tipo da estrutura:



4 Implementações de Scheme podem, no entanto, armazenar alguns dos primeiros elementos de listas de forma otimizada – o que não invalida nosso argumento.

Versão Preliminar

Usaremos um procedimento auxiliar `cria`, que recebe um nome, uma lista de campos, e instancia um objeto com estes campos, já incluindo o nome da estrutura na posição zero do vetor:

```
(define cria
  (lambda (nome campos)
    (let ((v (make-vector (+ 1 (length campos)))))
      (vector-set! v 0 nome)
      v)))
```

Um procedimento `is-of-type?` verifica se um objeto é de um dado tipo. Idealmente também verificaríamos se é um vetor, e se ele tem pelo menos um elemento, mas por simplicidade isso foi omitido:

```
(define is-of-type?
  (lambda (obj type)
    (eqv? type (vector-ref obj 0))))
```

Usaremos duas macros para criar novos tipos. A primeira, `define-procs-campo`, usa como argumentos:

- O índice do campo no vetor;
- Uma lista com os nomes de getters e setters. Cada par getter/setter é uma lista – por exemplo, `((get-nome set-nome!) (get-valor set-valor!))`

Quando a lista for vazia, a macro expandirá para "`(begin)`". Quando houver itens na lista, a expansão será

```
(begin
  (define get (lambda (struc) (vector-ref struc i)))
  (define set (lambda (struc a) (vector-set! struc i a)))
  (define-procs-campo (+ 1 i) x))
```

Onde `get` e `set` são variáveis de macro. A macro usa a si mesma recursivamente, incrementando o valor do índice do vetor.

Versão Preliminar

```
(define-syntax define-procs-campo
  (syntax-rules ()
    ((_ i ((get set) . x))
     (begin
       (define get (lambda (str) (vector-ref str i)))
       (define set (lambda (str a) (vector-set! str i a)))
       (define-procs-campo (+ 1 i) x)))
    ((_ a b)
     (begin))))
```

A segunda macro que definiremos é `define-structure`, que recebe:

- Um nome de tipo;
- Um nome para o procedimento que criará objetos deste tipo;
- Um nome para o predicado que verificará se objetos são deste tipo;
- Uma lista de getters e setters, no formato que usamos para a macro `define-procs-campo`

```
(define-syntax define-structure
  (syntax-rules ()
    ((_ nome maker pred? . campos)
     (begin
       (define pred?
         (lambda (obj) (is-of-type? obj (quote nome))))
       (define maker
         (lambda () (cria (quote nome)
                          (quote campos))))
       (define-procs-campo 1 campos)))))
```

Agora podemos definir tipos de dados arbitrários. Definiremos então o tipo `ponto`:

```
(define-structure
  ponto
  make-pt
  ponto?
  (get-x set-x!)
  (get-y set-y!))
```

A expansão do `define-strucutre` para `ponto` é mostrada a seguir:

Versão Preliminar

```
(begin
  (define
    ponto?
    (lambda (obj) (is-of-type? obj (quote ponto))))
  (define
    make-pt
    (lambda ()
      (cria (quote ponto)
            (quote ((get-x set-x!) (get-y set-y!))))))
  (begin
    (define get-x  (lambda (struc)   (vector-ref struc 1)))
    (define set-x! (lambda (struc a) (vector-set! struc 1 a)))
    (begin
      (define get-y  (lambda (struc)   (vector-ref
struc (+ 1 1))))
      (define set-y! (lambda (struc a) (vector-set! struc (+ 1 1) a)))
      (begin))))
```

8.8 EXEMPLO: TRACE

(esta seção está incompleta)

Implementaremos um método para acompanhar as chamadas e retornos de um procedimento.

O procedimento `with-trace` recebe dois argumentos: o primeiro é o *nome* de um procedimento, e o segundo é o procedimento; retorna outro procedimento, que realiza o *trace*, mostrando o nome do procedimento sendo chamado e seus argumentos (antes da chamada) e o valor retornado (após a chamada).

Versão Preliminar

```
(define with-trace
  (lambda (proc proc-original)
    (lambda args
      (display "chamando ")
      (display (cons proc args))
      (newline)
      (let ((result (apply proc-original args)))
        (display " => ")
        (display (cons proc args))
        (display " = ")
        (display result)
        (newline)
        result))))
```

Apesar de podermos usar `member` para encontrar um elemento em uma lista, definiremos aqui o procedimento `find`, que retorna `#t` ou `#f`.

```
(define find
  (lambda (x lst)
    (cond ((null? lst)
           #f)
          ((eq? x (car lst))
           #t)
          (else
           (find x (cdr lst))))))
```

Testamos nossa implementação de `find`:

```
(find 'a '(x y z a b c))
#t
```

Agora, para acompanharmos as chamadas a `find`, precisamos usar o procedimento retornado por `with-trace`:

```
(set! find (with-trace 'find find))
(find 'a '(x y z a b c))
chamando (find a (y z a b c))
chamando (find a (z a b c))
chamando (find a (a b c))
=> (find a (a b c)) = #t
```

Versão Preliminar

```
=> (find a (z a b c)) = #t
=> (find a (y z a b c)) = #t
=> (find a (x y z a b c)) = #t
#t
```

O `(set! ...)` que usamos não é conveniente; como não podemos encapsulá-lo em um procedimento, construiremos uma macro:

```
(define-syntax trace
  (syntax-rules ()
    ((_ fun)
     (set! fun (with-trace 'fun fun)))))
```

Agora podemos acompanhar as chamadas de um procedimento usando a macro `find`.

```
(trace find)
(find 'a '(x y z a b c))
```

8.9 ANTIGAS MACROS NÃO HIGIÊNICAS: DEFINE-MACRO

Diversas variantes do mecanismo `define-macro` eram usados informalmente por muitas implementações Scheme antes de R^{5}RS. Como não havia padronização, sua sintaxe pode variar (mas não muito) nas implementações que a suportam (e muitas implementações deixaram de suportá-l-a). Uma das variantes de `define-macro` tem a seguinte forma:

```
(define-macro MACRO-NAME
  (lambda MACRO-ARGS
    MACRO-BODY ...))
```

A transformação de código feita por `define-macro` é exatamente aquela que seu procedimento interno fizer. Os argumentos `MACRO-ARGS` são os que a macro usará. Um primeiro exemplo é uma macro que apenas mostra sua lista de argumentos, mas *antes do programa ser interpretado*:

Versão Preliminar

```
(define-macro mostra
  (lambda args
    (display args)
    (newline)
    #t))
(mostra 1 'simbolo '(lista))
(1 'simbolo '(lista))
#t
```

Os argumentos da macro não foram avaliados, e foram mostrados exatamente como haviam sido passados, *em tempo de expansão de macro*:

```
(mostra 1 simbolo (lista))
(1 simbolo (lista))
#t
```

A macro `gera-mostra` não mostra os argumentos em tempo de expansão de macro, mas gera código que o faz:

```
(define-macro gera-mostra
  (lambda args
    (list 'begin
          (list 'for-each 'display (list 'quote args))
          '(newline)
          #t)))
(gera-mostra 1 'simbolo '(lista))
1'simbolo'(lista)
#t
```

O procedimento `macroexpand` recebe uma lista e expande a macro na posição do `car`):

```
(macroexpand '(gera-mostra 1 'simbolo '(lista)))
(begin (for-each display '(1 'simbolo '(lista))) (newline) #t)
```

Como a macro não avalia seus argumentos, é possível passar a ela como parâmetros símbolos sem vinculação:

Versão Preliminar

```
(gera-mostra 1 simbolo (lista))
1simbolo(lista)

(macroexpand '(gera-mostra 1 simbolo (lista)))
(begin (for-each display '(1 simbolo (lista))) (newline) #t)
```

Uma maneira simples de escrever macros com `define-macro` é partir da expansão, depois escrever um procedimento que gere a expansão, e só então trocar `define` por `define-macro`.

Por exemplo, a macro `gera-mostra` poderia ter sido desenvolvida assim:

```
(lambda args
  (begin
    (for-each display args)
    (newline)
    #t)
```

Um procedimento que gera o código acima é:

```
(define gera-mostra
  (lambda args
    (list 'begin
          (list 'for-each 'display (list 'quote args))
          '(newline)
          #t)))
(gera 'arg1 'arg2)
(begin (for-each display '(arg1 arg2)) (newline) #t)
```

Finalmente, trocando `define` por `define-macro` a macro é obtida:

```
(define-macro gera-mostra
  (lambda args
    (list 'begin
    (list 'for-each 'display (list 'quote args))
    '(newline)
    #t)))
```

A macro abaixo ilustra novamente, e de maneira ainda mais clara, a diferença entre código executado durante o tempo de expansão da macro e durante a avaliação:

Versão Preliminar

```
(define-macro m
  (lambda (x)
    (display "Mensagem mostrada durante a expansão da macro")
    (newline)
    `(begin (display ,x)
            (newline))))
```

Dentro de um laço, a macro é expandida uma única vez, mas o código expandido é usado várias vezes:

```
(do ((i 0 (+ 1 i))) ((= i 4)) (m 'teste))
Mensagem mostrada durante a expansão da macro
teste
teste
teste
teste
```

Quando `macroexpand` é usado, a mensagem é mostrada (porque a macro está sendo expandida), mas a lista resultante da expansão da macro não contem o `display` que mostra a mensagem:

```
(macroexpand '(m 'teste))
Mensagem mostrada durante a expansão da macro
(begin (display 'teste) (newline))
```

Há um problema importante com o uso de `define-macro`, descrito na próxima subseção.

8.9.1 Captura de variáveis

A macro a seguir é uma tentativa de implementar o comando `for`, que poderia ser usado como `(for i 0 10 (display i))`:

Versão Preliminar

```
(define-macro
  (lambda (dirty-for i i1 i2 . body)
    `(let ((start ,i1) (stop ,i2))
       (let loop ((,i start))
         (unless (>= ,i stop) ,@body (loop (+ 1 ,i)))))))

(let ((start 42))
  (dirty-for i 1 3 (display start) (newline)))
1
1
```

O número 42 não foi impresso, como se poderia esperar. A expansão da macro mostra o problema:

```
(macroexpand (dirty-for i 1 3 (display start) (newline)))

(let ((start 1) (stop 3))
  (let loop ((i start))
    (unless (>= i stop) (display start) (newline)
      (loop (+ 1 i)))))
```

O símbolo `start`, usado na macro `dirty-for`, é também usado fora da macro; embora provavelmente não tenha sido a intenção do programador, as duas variáveis tem o mesmo vínculo.

Isso pode ser remediado mudando o nome de `start` dentro da macro – por exemplo, `ssttaarrtt`, que dificilmente seria usado fora da macro. No entanto, não há garantia de que o símbolo não será usado fora da macro (na verdade certamente seria quando dois `dirty-for` forem aninhados), e nomes como este tornam o código mais difícil de ler.

A solução para este problema é um mecanismo para gerar novos símbolos, com a garantia de que nunca serão iguais a símbolos escolhidos pelo programador. Este mecanismo é implementado no procedimento `gensym`, que podemos usar para gerar estes símbolos:

```
(define simbolo (gensym))
simbolo
G7
(eqv? simbolo 'G7)
#f
```

Versão preliminar

O símbolo retornado por `gensym` neste exemplo é representado como `G7`, mas verificamos também que o símbolo "`G7`" definido pelo usuário não é equivalente ao gerado por `gensym`.

Os símbolos gerados por `gensym` tem uma *representação externa* que pode lembrar símbolos escolhidos pelo programador, mas o sistema Scheme se lembrará que estes símbolos não podem ser `eqv?` a quaisquer outros incluídos no texto do programa: um símbolo gerado por `gensym` só é igual a si mesmo.

Assim, ao invés de incluir diretamente os símbolos `start`, `stop` e `loop`, podemos escrever a macro usando para eles símbolos únicos gerados por `gensym`, mas usando nomes compreensíveis como apelidos:

```
(define-macro better-for
  (lambda (i i1 i2 . body)
    ;; Este let não é expandido: é código interno da macro
    (let ((start (gensym))
          (stop  (gensym))
          (loop  (gensym)))

     ;; A lista gerada abaixo será devolvida ao expandir a macro;
     ;; Como usamos quasiquote e start/stop/loop estão incluídos
     ;; SEM quote (com vírgula antes), o símbolo incluído na expansão
     ;; da macro é o gerado por gensym.

     `(let ((,start ,i1)
            (,stop ,i2))
        (let ,loop ((,i ,start))
          (if (< ,i ,stop)
              (begin ,@body
                     (,loop (+ 1 ,i)))))))))

(let ((start 42))
   (better-for i 1 3 (display start) (newline)))
42
42
```

A expansão da macro mostra como os símbolos foram trocados:

```
(macroexpand '(better-for i 1 3 (display start) (newline)))
```

Versão Preliminar

```
(let ((G11 1)
      (G12 3))
  (let G13 ((i G11))
    (if (< i G12)
      (begin (display start)
             (newline)
             (G13 (+ 1 i))))))
```

Como durante a *escrita* da macro o programador pode usar variáveis com nomes razoáveis (como `start`, `stop` e `loop`), os nomes "estranhos" na expansão não são um problema.

Se ainda assim for interessante usar nomes que façam sentido mesmo após a expansão da macro (para facilitar a depuração do programa, por exemplo), basta passar ao procedimento `gensym` uma string, e ela será usada como base para o nome do símbolo:

```
(gensym "start")
start14
```

Embora a macro `better-for` não capture variáveis definidas pelo programador no contexto onde a macro é usada, em situações fora do comum ainda é possível que a macro se comporte de maneira diferente do esperado:

```
(let ((if 'x))
   (less-dirty-for i 1 3
                   (display i)))
```

O programa acima muda a vinculação do símbolo `if`, usado na macro, e a após a expansão de `less-dirty-for` será:

```
(let ((if 'x))
   (let ((G11 1)
         (G12 3))
     (let G13 ((i G11))
       (if (< i G12)
         (begin (display start)
                (newline)
                (G13 (+ 1 i)))))))
```


Versão Preliminar

Neste código, `if` não é mais forma especial, e sim uma variável cujo valor é o símbolo `x`. Isto significa que `(if ...)` passa a ser uma aplicação de procedimento. Interpretadores Scheme podem seguir dois caminhos aqui:

- Avaliar a variável `if`, verificar que não é procedimento e sinalizar um erro, ou
- Avaliar os parâmetros do que considera ser uma chamada de procedimento. Neste caso, tentará avaliar a forma `(begin ...)`, que chama `loop` recursivamente – e o programa entrará em um laço infinito.

8.9.2 Mais sobre macros não-higiênicas

É comum o nome de "não-higiênico" a sistemas de macro que permitem captura de símbolos da mesma forma que `define-macro`.

Há dois livros somente sobre técnicas de uso deste tipo de macro (de baixo nível e não higiênica): o de Paul Graham [Gra93] e o de Doug Hoyte [Hoy08]. A linguagem usada em ambos os livros é Common Lisp, cujo sistema de macros funciona da mesma forma que `define-macro`, e o conteúdo dos livros (em particular o de Paul Graham) é aplicável a programas Scheme usando `define-macro`, com pequenas modificações.

EXERCÍCIOS

Ex. 110 — Escreva uma macro similar à macro `set-both!` descrita neste Capítulo, mas que aceite várias variáveis como argumento: `(set-all! x y z 10)`.

Ex. 111 — Reescreva o jogo de pôquer descrito nos Capítulos anteriores usando `define-structure`.

Ex. 112 — Escreva uma macro `xnor`, que é semelhante a `xor` exceto que o primeiro argumento é o número de formas que devem ser verdadeiras.

```
(xnor 2 #f #f 'a 'b #f)  ==> #t
(xnor 3 #f #f 'a 'b #f)  ==> #f
(xnor 1 #f 1 #f)  ==> #t
```

Ex. 113 — Faça uma macro que permita escrever Lisp em notação posfixa. Um exemplo de uso:
`((y x +) 2 *)` deve ser expandido para `(* 2 (+ x y))`

Ex. 114 — Imagine uma função que mude o valor de uma variável para `#f`:

Versão Preliminar

```
(define faz-falso!
  (lambda (x)
    (set! x #f)))
```

Porque ela não funciona? Como conseguir que `(faz-falso! variavel)` funcione de forma correta?

Ex. 115 — Usando `define-syntax`, escreva a macro `(when condicao corpo)`, onde o corpo pode ser composto de várias formas.

Ex. 116 — Escreva uma macro que troque os valores de duas variáveis, e diga porque ela não pode ser um procedimento.

Ex. 117 — Modifique a macro `while` descrita neste Capítulo para que ela permita ao usuário determinar o valor de retorno do `while`.

Ex. 118 — `syntax-error`, usado para sinalizar erro de sintaxe, poderia ser implementado pelo usuário como uma macro?

Ex. 119 — Na Seção 8.4 há um exemplo de macro `with-epsilon` usando `syntax-rules`. A macro não funciona porque `syntax-rules` é higiênico. Escreva uma macro `with-epsilon` que permita usar diferentes valores e nomes para ε. Discuta a utilidade da macro.

Ex. 120 — Usando `er-macro-transformer`, reescreva as macros definidas neste Capítulo com `syntax-rules`. Depois, faça o contrário (reescreva as macros que foram desenvolvidas com `er-macro-transformer`, mas usando `syntax-rules`). Discuta quais macros foram mais fáceis de reescrever.

Ex. 121 — Mostre que a linguagem do sistema de macros `syntax-rules` é Turing-equivalente. Em seguida discuta as consequências disso.

Ex. 122 — Tente construir um sistema de módulos. Comece com algo muito simples, e aos poucos adicione recursos.

RESPOSTAS

Resp. (Ex. 113) — Com renomeação explícita a tarefa é relativamente simples: receba a expressão, reverta as listas que não são literais (ou seja, que não estão dentro de `quote`) *e suas sublistas* e retorne o resultado como a nova forma.

Versão Preliminar

Resp. (Ex. 114) — Não funciona porque o `x` passado para o procedimento `set!` é local ao procedimento (ele foi *copiado* do argumento real). Para que funcione, deve ser refeito como macro:

```
(define-syntax faz-falso!
(syntax-rules ()
  ((_ x)
   (set! x #f))))
```

Resp. (Ex. 119) —

```
(define-syntax with-epsilon
  (syntax-rules ()
    ((_ e-name e-value expr)
      (let ((e-name e-value)) expr))))
```

Usar diferentes nomes para ε pode ser interessante se estamos incluíndo em nosso programa grandes partes de programas já elaborados que fazem referência a ε com diferentes nomes (`epsilon`, `tiny`, `negligible`, etc). No entanto, o mesmo efeito pode ser conseguido sem macros.

Versão Preliminar

Versão Preliminar

9 CASAMENTO DE PADRÕES

O trecho de código a seguir mostra a definição da função fatorial em Scheme:

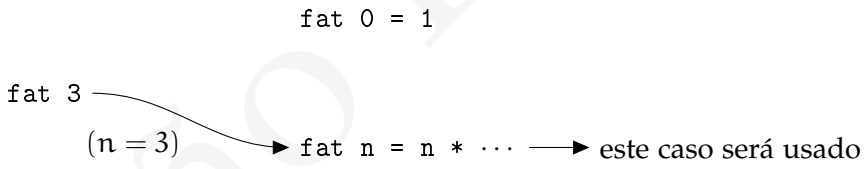
```
(define fat
  (lambda (n)
    (if (= n 0)
        1
        (* n (fat (- n 1))))))
```

Compare com a definição de fatorial a seguir:

```
fat 0 = 1
fat n = n * fat (n - 1)
```

Este segundo trecho de código é exatamente como se define a função fatorial em Haskell. É muito mais limpo, e não foi necessário usar explicitamente um `if`.

Quando um programador usa a função fatorial definida em Haskell, pedindo por exemplo o valor de `fat 0` ou de `fat 3`, o programa Haskell comparará o argumento (zero ou tres) com o padrão na definição da função. O primeiro padrão a casar com o argumento determina qual caso será usado.



Dizemos que Haskell suporta *chamada de procedimentos direcionada por padrões*, ou *casamento de padrões*. O compilador saberá que, quando a função `fat` for usada com o argumento zero, seu resultado deve ser um, e nos outros casos deve ser `n * fat (n-1)` – e transformará internamente aquela definição em outra mais explícita, parecida com a versão Scheme.

Em Scheme podemos usar casamento de padrões nas formas `lambda`.

Versão Preliminar

9.1 USANDO MACROS PARA CASAMENTO DE PADRÕES

É possível criar macros que permitem escrever código usando casamento de padrões. Criaremos uma macro `mlambda` (abreviação de *matching lambda*) que nos permitirá, por exemplo, escrever a função fatorial como segue:

```
(define fat
 (mlambda
    (0    ->  1)
    (?n   ->  (* n (fat (- n 1))))))
```

A macro `mlambda` transformará seus argumentos em uma forma `cond`, expandindo para

```
(lambda (n)
   (cond ((equal? (n 0))  1)
         (else   (* n (fat (- n 1))))))
```

As limitações de nosso casador de padrões serão:

- Os nomes dos argumentos nos padrões são precedidos por `?`, para que o casador de padrões saiba a diferença entre um símbolo (constante) e um nome de argumento. Assim, no padrão `(a ?b ?c d)` há dois símbolos constantes (`a` e `d`) e duas variáveis (`?b` e `?c`).
- A aridade do procedimento deve ser fixa.
- O último padrão deve conter todos os argumentos, para simplificar a implementação.
- As posições e nomes dos argumentos devem ser iguais em todos os padrões, exceto quando repetimos nomes para indicar que os argumentos devem ser iguais: `(a a 0 -> (...))` significa que neste caso o primeiro e segundo argumentos são iguais.
- Ao comparar argumentos formais com argumentos reais durante a chamada do procedimento, será sempre usado o predicado `equal?`.

Precisaremos de um predicado que determina se um objeto é um nome de argumento formal. Como determinamos que um argumento formal deve ser um símbolo que comece com o caracter `?`, basta usar `symbol-string` e verificar o primeiro caracter da cadeia.

Versão Preliminar

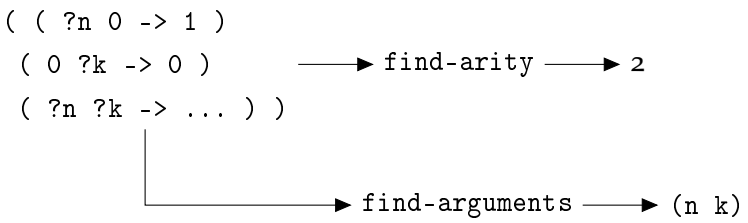
```
(define matching-symbol?
  (lambda (s)
    (and (symbol? s)
         (eq? (string-ref (symbol->string s) 0)
              #\?))))
(matching-symbol? 'abc)
#f
(matching-symbol? '?xz)
#t
(matching-symbol? 2)
#f
(matching-symbol? "a")
#f
```

Como teremos de usar sinais de interrogação nos argumentos formais, usaremos também um procedimento que os remove. O procedimento `strip-question-mark` a seguir realiza isso: primeiro, verifica se o argumento é um nome de variável para casamento de padrào (ou seja, se é realmente um s'imbolo precedido de `?`). Se não for, um erro ocorre. Se o argumento for apropriado, usa `symbol->string` para convertê-lo em string, depois retira o primeiro caracter, e finalmente usa `string->symbol` para transformá-lo novamente em símbolo.

```
(define strip-question-mark
  (lambda (s)
    (if (not (matching-symbol? s))
        (error "Cannot strip ?")
        (let ((str (symbol->string s)))
          (let ((str-new (string-copy str 1 (string-length str))))
            (string->symbol str-new))))))
(strip-question-mark '?abc)
abc
```

Para expandir a macro, precisamos determinar os nomes dos argumentos e a aridade do procedimento. Criaremos dois procedimentos, `find-arity` e `find-arguments` para isso.

Versão Preliminar



O procedimento `find-arity` determina a aridade da forma – basta observar a posição em que o símbolo `->` aparece. (O procedimento `list-index`, usado aqui, é definido na SRFI-1 SRFI-1 – `(list-index pred? lista)` retorna o índice do primeiro elemento da lista para o qual `pred?` retorna `#t`).

```
(define find-arity
  (lambda (form)
    (list-index (lambda (x) (eq? x '->))
                form)))
```

O procedimento `find-arguments` determina quais são os argumentos. Como determinamos que o último padrão deve conter os nomes de todos os argumentos, basta tomá-lo e remover os sinais de interrogação (o procedimento toma a última linha, determina a aridade e aplica `strip-question-mark` em todos os elementos da lista).

```
(define find-arguments
  (lambda (form)
    (let ((last-line (last form)))
      (let ((arity (find-arity last-line)))
        (map strip-question-mark (take last-line arity))))))
```

O exemplo a seguir ilustra o funcionamento destes procedimentos.

```
(find-arguments '((1  2  3  4   -> #f)
                  (?a ?b 0  "x" -> "blah")
                  (?a ?b ?c ?d   -> (display 'ok)
                                    (+ a b c d))))
```

```
(a b c d)
```

Agora processamos cada uma das linhas. Elas estão todas no formato *padrão* `->` *formas*. Transformaremos o padrão em cada linha em um teste, para termos então uma cláusula `cond`. Por exemplo, o trecho

Versão Preliminar

```
(?x 0 -> (display "y=0, usando x-1")
         (- x 1))
```

será transformado na forma

```
((equal? x 0)
 (display "y=0, usando x-1")
 (- x 1))
```

que pode ser usada como uma cláusula `cond`.

Para realizar esta transformação, em cada linha tomamos uma a uma as posições do padrão e:

- Se o elemento é um argumento formal e está em sua posição correta, nada faça;
- Se o elemento `e1` é um argumento e está *fora* de sua posição, onde na verdade deveria estar o argumento `e2`, inclua o teste `(equal? e1 e2)`. Por exemplo, se os argumentos formais forem `(a b c)` e os argumentos reais (usados na chamada do procedimento) forem `(a a 0)`, o programador quer dizer que os dois primeiros parâmetros devem ser iguais e o terceiro deve ser zero. Incluímos então o teste `(equal? a b)`.
- Se o elemento não é argumento formal, inclua no teste o código que compara este elemento com o argumento desta posição

O procedimento `process-arg` fará exatamente isso. Seus parâmetros são `x`, o argumento a ser processado, `pos`, a posição do argumento, e `args`, a lista de argumentos formais.

```
(define process-arg
  (lambda (x pos args)
    (let ((formal-arg-in-pos (list-ref args pos)))
      (cond ((and (matching-symbol? x)
                  (not (eq? (strip-question-mark x)
                            formal-arg-in-pos)))
             `(equal? ,formal-arg-in-pos
                      ,(strip-question-mark x)))
            ((not (matching-symbol? x))
             `(equal? ,formal-arg-in-pos
                      ,x))
            (else #t)))))
```

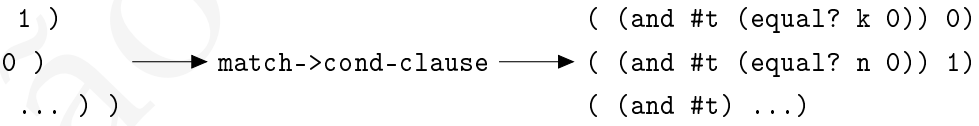

Versão Preliminar

```
(process-arg '?b 1 '(a b c))
()
(process-arg '?b 0 '(a b c))
(equal? a b)
(process-arg ''oh'' 2 '(a b c))
(equal? c "oh")
```

O procedimento `process-arg` processa apenas um argumento. A seguir construímos o procedimento `pattern->test`, que processa um padrão inteiro, resultando em um teste.

```
(define pattern->test
  (lambda (pattern args)
    (cons 'and
          (list-tabulate (length args)
                         (lambda (i)
                           (process-arg (list-ref pattern i)
                                        i
                                        args))))))
(pattern->test '(?a 3 ?b) '(a b c))
(and #t (equal? b 3) (equal? c b))
```

Há um `#t` extra dentro do `and`. Não precisamos nos preocupar com isso, porque não faz diferença: `(and #t x)` é o mesmo que `x`. Além disso, um bom compilador eliminará esse `#t` adicional.




```
(define match->cond-clause
  (lambda (match-line args arity)
    (let ((pat (take match-line arity))
          (then (drop match-line (+ 1 arity))))
      (cons (pattern->test pat args)
            then))))
```


Versão Preliminar

```
(match->cond-clause '(?a 3 ?b -> (one) (two)) '(a b c) 3)
((and (#t (equal? b 3) (equal? c b))) (one) (two))
```

O procedimento `mlambda-aux` é um protótipo da macro que queremos; ele transforma formas `mlambda` no código expandido de uma forma `lambda`.

```
(define mlambda-aux
  (lambda (in)
    (let ((args (find-arguments in))
          (arity (find-arity (car in))))
    (cons 'cond
          (map (lambda (line) (match->cond-clause line args arity))
               in)))))


(mlambda-aux '((0  -> 1)
               (?n -> (* n (fat (- n 1))))))
(cond
((and (equal? n 0)) 1)
((and #t) (* n (fat (- n 1)))))
```

Escrevemos o procedimento `mlambda-aux` para poder testar a expansão da macro antes de torná-la de fato uma macro. Só nos falta agora transformar `mlambda-aux` em macro.

```
(define-syntax mlambda
  (er-macro-transformer
   (lambda (in ren cmp)
     (let ((args (find-arguments (cdr in)))
           (arity (find-arity (cadr in))))
       `(lambda ,args (,(ren 'cond)
                       ,@(map (lambda (line)
                                (match->cond-clause line args arity))
                              (cdr in))))))))
```

Implementamos agora a função fatorial usando casamento de padrões.

```
(define fat (mlambda (0  -> 1)
                     (?n -> (* n (fat (- n 1))))))
```

Versão Preliminar

```
(fat 0)
1
(fat 5)
120
```

Uma função que torna o casamento de padrões mais interessante é a função de Ackermann, que recebe dois argumentos inteiros e retorna um inteiro[1].

$$A = \begin{cases} n+1 & \text{se } m = 0, \\ A(m-1,1) & \text{se } m > 0 \text{ e } n = 0, \\ A(m-1,A(m,n-1)) & \text{se } m > 0 \text{ e } n > 0. \end{cases}$$

Como a função é definida recursivamente usando vários casos, ela pode ser implementada de maneira natural com um `cond`:

```
(define ack
  (lambda (m n)
    (cond ((= 0 m) (+ n 1))
          ((= 0 n) (ack (- m 1) 1))
          (else    (ack (- m 1) (ack m (- n 1)))))))
```

No entanto, o código fica mais limpo e aproxima-se mais da definição da função quando usamos casamento de padrões:

```
(define ack (mlambda (0  ?n -> (+ 1 n))
                     (?m  0 -> (ack (- m 1) 1))
                     (?m ?n -> (ack (- m 1) (ack m (- n 1))))))
(ack 2 2)
7
```

Construímos também a função que determina o coeficientes binomiais $\binom{n}{k}$, em Haskell e em seguida em Scheme. Usaremos a seguinte fórmula:

$$\binom{n}{k} = \begin{cases} 1 & \text{se } k = 0 \\ 0 & \text{se } n = 0 \\ \left\lfloor \frac{n\binom{n-1}{k-1}}{k} \right\rfloor & \text{em outros casos.} \end{cases}$$

1 A função de Ackermann é um exemplo muito simples de função total computável que não é recursiva primitiva.

Versão Preliminar

Esta definição é mais eficiente do que a tradução direta da definição usando fatoriais.

Em Haskell, o código é sintaticamente bastante próximo à definição:

```
choose n 0 = 1
choose 0 k = 0
choose n k = choose (n-1) (k-1) * n `div` k
choose 5 3
10
```

No entanto, a leitura da última linha da definição da função pode gerar confusão, se não estiver claro que os argumentos na chamada de `choose` são `(n-1)` e `(k-1)`, e que o resto da linha se refere ao que fazer com o resultado da chamada recursiva a `choose`: multiplicar por `n` e dividir por `k`.

Em Scheme, a definição é um pouco diferente.

```
(define choose
  (mlambda
    (?n  0 -> 1)
    ( 0 ?k -> 0)
    (?n ?k -> (quotient (* n (choose (- n 1) (- k 1)))
                        k))))
(choose 5 3)
10
```

9.2 UNIFICAÇÃO

Há uma idéia relacionada a casamento de padrões que serve como fundamento para sistemas de inferência de tipos e sistemas de programação em Lógica – trata-se da *unificação de termos*.

Um sistema de casamento de padrões realiza o mapeamento de uma lista de variáveis em valores, mas sempre em uma direção. Se tivermos um padrão `(?a 2 ?b 4)` e argumentos `(1 2 3 4)`, poderíamos simbolizar o casamento do padrão com a lista de argumentos como

`(?a 2 ?b 4)` ~ `(1 2 3 4)`.

Versão Preliminar

Um casador de padrões poderá determinar valores para as variáveis do *lado esquerdo* (o padrão): `?a` $\rightarrow$ `1`, e `?b` $\rightarrow$ `3`. Ele não determinará valores para o lado direito, onde há a lista `(1 2 3 4)`.

O processo de unificação mapeia variáveis de *ambos* os lados. Por exemplo, se tivermos duas listas, `(?a 2)` e `(1 ?b)`, a unificação das duas listas nos informará que devemos substituir `?a` por `1` e `?b` por `2` para que as listas fiquem iguais.

Dados dois padrões (que serão para nós duas formas Scheme), queremos uma *substituição* que, quando aplicada nas duas formas, as tornará *textualmente idênticas*. Uma substituição determina como trocar as variáveis por outros valores. Por exemplo, as formas já dadas como exemplo

```
(?a 2 ?b 4) ~ (1 2 3 ?c).
```

podem ser tornadas iguais através do unificador

`?a` $\rightarrow$ `1`

`?b` $\rightarrow$ `3`

`?c` $\rightarrow$ `4`

Para as listas `(?a 2 (f ?x) 3)` e `((g 5) ?x ?y ?z)`, a unificação resultará em

`?a` $\rightarrow$ `(g 5)`

`?x` $\rightarrow$ `2`

`?y` $\rightarrow$ `(f 2)`

`?z` $\rightarrow$ `3`

Esta substituição tornará as duas formas iguais, mesmo havendo variáveis em ambas (e mesmo havendo uma variável, `?x`, presente nas duas formas).

Construiremos um procedimento de unificação em Scheme[2].

Representaremos as substituições como listas de associação. A substituição

$$\{A \rightarrow 10, B \rightarrow x, C \rightarrow f(z)\}$$

será descrita como

```
((?a 10) (?b x) (?c (f z)))
```

Denotaremos variáveis por símbolos começando com uma interogação, por isso definimos `variable?` sendo a mesma coisa que `matching-symbol?`. Para verificar se uma variável tem vínculo em uma substituição usamos `bound?`, que é somente um novo nome

2 Muito fortemente baseado no elegante código de Peter Norvig [Nor91; Nor92].

Versão Preliminar

para `assq`. Para obter o valor de uma variável em uma substituição, usamos a composição de `car` e `assq`. O procedimento `bind` cria um novo vínculo em uma substituição.

```
(define variable? matching-symbol?)
(define bound? assq)
(define (value v subst) (cdr (assq v subst)))
(define (bind var val sub) (cons (cons var val) sub))
```

O procedimento `occurs?` recebe uma variável `var`, um termo `x`, uma substituição, e verifica se `var` ocorre em `x`. Quado `var` é igual a `x`, o procedimento retorna `#t`. Se não for este o caso, verifica se `x` tem vínculo na substituição. Se tiver, verifica recursivamente se `var` ocorre no *valor* de `x` na substituição. Se `x` não tem vínculo na substituição, e se é par (inclusive lista), chama recursivamente a si mesmo nos dois lados do par (ou na cabeça e cauda da lista, se o par for uma lista).

```
(define occurs?
  (lambda (var x sub)
    (cond ((eq? var x) #t)
          ((bound? x sub)
           (occurs? var (value x sub) sub))
          ((pair? x) (or (occurs? var (car x) sub)
                         (occurs? var (cdr x) sub)))
          (else #f))))
```

O procedimento `uni-var` unifica uma variável `var` com algum termo `val`. Se forem iguais, a substituição não é alterada; se `var` ou `val` tiverem vínculo, o valor é obtido e `unify` é chamado. Se `var` ocorre em `val`, o procedimento falha. Em outros casos, a substituição é extendida para conter o novo vínculo de `var`.

```
(define uni-var
  (lambda (var val sub)
    (cond ((eq? var val) sub)
          ((bound? var sub)
           (unify (value var sub) val sub))
          ((and (variable? val) (bound? val sub))
           (unify var (value val sub) sub))
          ((occurs? var val sub) #f)
          (else (bind var val sub)))))
```

`Unify` recebe dois termos e possivelmente uma substituição `s`.

Versão Preliminar

```
(define unify
  (lambda (x y . s)
    (let ((sub (if (null? s) '() (car s))))
      (cond ((equal? x y) sub)
            ((eq? sub #f) #f)
            ((variable? x) (uni-var x y sub))
            ((variable? y) (uni-var y x sub))
            ((not (or (pair? x) (pair? y))) #f)
            (else (unify (cdr x) (cdr y)
                         (unify (car x) (car y) sub)))))))
```

Os exemplos a seguir ilustram o uso do procedimento `unify`.

```
(unify '(f ?c (f z)) '(f (g ?x y) ?d))
((?c (g ?x y)) (?d (f z)))
(unify '?c 'y)
((?c y))
(unify '?c '?y)
((?c ?y))
(unify '(f ?a) '(f (f b)))
((?a (f b)))
(unify '(g ?x (z (f b))) '(g (f b) (z ?x)))
((?x (f b)))
(unify '(g ?x (z (f b))) '(g (f a) (z ?x)))
#f
(unify '?x '(f (g ?x)))
#f
```

O procedimento `unity` apenas retorna uma lista representando uma substituição. Podemos querer modificar as variáveis refletindo a substituição encontrada. A seguir está listado procedimento `uni-set!`, que recebe os mesmos argumentos que `unify`, mas retorna um template que pode ser usado em uma macro. Para cada substituição – por exemplo `(?x 5)` – o procedimento incluirá um `set!` – no exemplo dado, `(set! ?x 5)`.

Versão Preliminar

```
(define uni-set!
  (lambda (t1 t2)
    (cons 'begin
          (let ((u (uni t1 t2)))
            (map (lambda (p)
                   (list 'set!
                         (strip-question-mark (car p))
                         (cadr p)))
                 u)))))
```

O procedimento `uni-set!` pode ser usado para construir uma macro que produza todas as chamadas `set!` necessárias para que os dois termos passem a ter os mesmos valores.

```
(uni-set! '(f ?a) '(f (f b)))
(begin (set! a (f b)))

(uni-set! '(g ?x (z (f b))) '(g (f b) (z ?x)))
(begin (set! x (f b)))
```

No entanto, `uni-set!` não sabe lidar com variáveis que não serão substituídas: no exemplo a seguir, `?x` é deixado sem modificação no padrão.

```
(uni-set! '(f ?c (f z)) '(f (g ?x y) ?d))
(begin (set! d (f z)) (set! c (g ?x y)))
```

EXERCÍCIOS

Ex. 123 — Escreva as seguintes funções usando a forma `mlambda`.

a) Mínimo múltiplo comum

b) n-ésimo número de Fibonacci

Ex. 124 — Escreva uma macro para casamento de padrões que permita escrever definições de procedimentos sem usar `lambda` e usando menos parênteses, como no exemplo da função de Ackermann a seguir.

Versão Preliminar

```
(mdef ack
   0 n -> (+ n 1)
   m 0 -> (ack (- m 1) 1)
   m n -> (ack (- m 1) (ack m (- n 1))))
```

Ex. 125 — A macro `mlambda` desenvolvida neste Capítulo usa somente `equal?` para comparar parametros. Construa uma nova versão da macro que permita indicar o predicado a ser usado em cada parâmetro.

Ex. 126 — Modifique o casador de padrões para que faça também casamento de listas e vetores.

Ex. 127 — Quando o algoritmo de unificação dado neste Capítulo falha, retorna o booleano `#f`. Modifique-o para que retorne o motivo pelo qual a unificação não foi possível. (Se o valor retornado for uma lista de associações, é o unificador; se não for, é o motivo pelo qual a unificação é impossível).

Ex. 128 — Crie *benchmarks* para o algoritmo de unificação. Tente otimizá-lo e compare suas soluções com a versão apresentada neste Capítulo.

Ex. 129 — Crie um procedimento `apply-sub`, que aplica uma substituição em uma forma Scheme. Queremos, por exemplo, aplicar a substituição $\{D \to de, F \to efe, U \to um\}$ sobre a expressão

$$g(A\ x\ D\ e\ F),$$

que representamos em Scheme como `(g ?a x ?d e ?f)`.
O resultado seria como mostramos a seguir. `(apply-sub '((?d de) (?f efe) (?u um)) '(g ?a x ?d e ?f))`

```
(g ?a x de e efe)
```

Ex. 130 — Se não quisermos usar o símbolo `?` para marcar variáveis, ainda poderíamos implementar o algoritmo de unificação? Queremos que seja possível unificar os seguintes termos:

```
(unify (a  b 1)
       ('x 2 1))
```

```
((a x) (b 2))
```


Versão Preliminar

Ex. 131 — O algoritmo de unificação pode também ser escrito como a seguir (o código usa uma pilha p e constrói uma substituição θ).

```
unify(T1, T2):
    s ← ∅
    empilhe T1 = T2 em p
    falha ← falso
    enquanto p não vazia e falha ≠ verdadeiro:

        desempilhe X = Y

        se X évariável não ocorrendo em Y:
            substitua X por Y na pilha e em θ
            adicione X = Y a θ

        senão se X e Y são variáveis ou constantes iguais:
            continue

        senão se X = f(X1,...,Xn) e Y = f(X1,...,Xn):
            empilhe Xi = Yi para i = 1,...,n em p

        senão:
            falha ← verdadeiro

    se falha retorne falso
    senão retorne θ
```

Implemente este algoritmo em Scheme e comente sobre como ele difere daquele dado neste Capítulo.

Ex. 132 — Escreva a macro `uni-set!`, como sugerido no final do Capítulo, usando `er-macro-transformer`. Tente fazer a macro gerar um erro se houver variáveis no padrão que não serão substituídas (ou ignorá-las e não incluir seus vínculos). Você poderia escrever esta macro facilmente com `syntax-rules`?

RESPOSTAS

Resp. (Ex. 123) — Item (b):

Versão Preliminar

```
(define fib
  (mlambda (0  -> 0)
           (1  -> 1)
           (?n -> (+ (fib (- n 1))
                     (fib (- n 2))))))
```

Resp. (Ex. 129) — A versão a seguir mapeia seu procedimento interno `substitute-one`, que toma um elemento da expressão e tenta aplicar a substituição. `Substitute-one` verifica se seu argumento é uma lista. Se for, chama `apply-sub` recursivamente; se não for, toma o elemento, procura-o na substituição (com assoc) e devolve o novo valor.

```
(define apply-sub
  (lambda (sub exp)
    (let ((substitute-one
           (lambda (x)
             (if (and (list? x)
                      (not (null? x)))
                 (apply-sub sub x)
                 (let ((new-x (assoc x sub)))
                   (if new-x
                       (cadr new-x)
                       x))))))
      (map substitute-one exp))))
```

Resp. (Ex. 130) — Sim, mas `unify` deverá ser uma macro para que seja possível fazer `(unify a 2)`, por exemplo. Note que é prefeitamente possível fazer `(unify a 'b)`, porque trata-se da abreviação de `(unify a (quote b))` – os únicos símbolos isolados que a macro unificadora verá são as variáveis, já que `(quote x)` é (sintaticamente – e é apenas o que interessa a uma macro) uma lista.

Versão Preliminar

10 CONTINUAÇÕES

Sendo minimalista, Scheme não traz no núcleo da linguagem mecanismos para suporte a multitarefa, tratamento de exceções, *backtracking* e corotinas. Ao invés disso, Scheme suporta uma única primitiva que permite construir todos esses mecanismos. Essa primitiva é a *continuação*.

10.1 DEFININDO CONTINUAÇÕES

Informalmente, uma continuação é semelhante a uma fotografia do mundo no estado em que ele está num determinado momento. A seguinte história é inspirada em uma tentativa de um programador Scheme de explicar continuações a um programador Common Lisp.

> Um *hacker* está em seu porão escuro, trabalhando freneticamente na versão 20.9342-2 de seu editor de textos, quando percebe algo terrível: o nível de cafeína em seu sangue está muito baixo, e ele pretendia trabalhar ininterruptamente nas próximas 48 horas. Ele se lembra então que pouco dinheiro restou depois da compra de seu novo monitor de 42 polegadas – ele poderá comprar apenas um copo de café.
>
> Conhecendo continuações, o hacker não se abala – ele deixa um adesivo amarelo em seu monitor dizendo o que iria fazer em seguida, sai para comprar café e em pouco tempo está de volta ao seu porão. Ele ainda *não* toma seu café; logo antes de levar o copo à boca, ele chama o procedimento Scheme `call-with-current-continuation`, que devolve uma fotografia do ambiente ao seu redor naquele momento. Ele então deixa esta foto em uma caixa perto da porta.
>
> Depois de tomar seu café e sentir-se imensamente aliviado, o hacker volta ao computador e passa a fazer o que quer que estava escrito no adesivo amarelo colado no monitor.
>
> Algumas horas depois, ele sente que precisa de mais cafeína (e desta vez não tem um centavo sequer!) O hacker deixa novamente no monitor um adesivo com a tarefa que iria fazer em seguida e vai até a porta. Ele pega a foto que havia deixado na caixa e aplica como se fosse um procedimento Scheme, e...

Versão Preliminar

> No instante seguinte o hacker está novamente com seu copo de café na mão! Ele toma novamente o café e segue para seu computador. Chegando lá, ele passa novamente a fazer o que estava indicado no adesivo no monitor – e que desta vez é diferente do adesivo amarelo usado na primeira vez que foi tomar café.
>
> O hacker continua programando e usando as "fotos" (continuações) até finalmente terminar a versão 20.9342-2 do editor de textos, que traz cinco novas *features* e quarenta e sete novos bugs que não existiam na versão 20.9342-1.

Esta pequena história deve ajudar o leitor a compreender o conceito de continuação, que é semelhante à "foto" obtida pelo protagonista. Para conseguir a continuação, o personagem precisou usar um procedimento Scheme cujo nome é `call-with-current-continuation`, muitas vezes abreviado `call/cc`.

Descreveremos continuações usando dois outros conceitos: o de *contexto* de uma subexpressão e o de *procedimento de escape*.

10.1.1 Contextos

Em uma S-expressão S, o contexto de uma subexpressão s é obtido da seguinte maneira:

- Troca-se s por □;
- Construímos um procedimento que aceite □ como parâmetro e cujo corpo seja exatamente a expressão obtida no passo anterior.

Por exemplo, na expressão `(* 2 (log (+ x y)))`, o contexto da subexpressão `(+ x y)` é:

- Primeiro passo: `(* 2 (log □))`
- Segundo passo: `(lambda (□) (* 2 (log □))`

Ou seja, o contexto da subexpressão `(+ x y)` é um procedimento que tomaria seu resultado e *continuaria* a computação.

O contexto de uma computação só é de interesse quando o programa está em execução. Assim, faz sentido modificar ligeiramente a definição de contexto dada acima:

- Primeiro passo: Troca-se s por □;
- Segundo passo: avalia-se a expressão com □ até que a computação não possa mais continuar;

Versão Preliminar

- Terceiro passo: Um procedimento é construído que aceita □ como parâmetro e cujo corpo é exatamente a expressão obtida no passo anterior.

Como exemplo calcularemos o contexto de `(+ 2 x)` na expressão

```
(if (> 10 20)
    (+ a (/ b c))
    (* (+ 2 x) 5))
```

Primeiro, trocamos `(+ 2 x)` por □:

```
(if (> 10 20)
    (+ a (/ b c))
    (* □ 5))
```

Depois, avaliamos a expressão até não podermos mais continuar:

```
(* □ 5)
```

Finalmente, construímos o procedimento:

```
(lambda (□)
  (* □ 5)
```

e este é o contexto de `(+ 2 x)` naquela expressão. O segundo passo nos permitiu selecionar o segundo braço do `if`, eliminando-o.

10.1.2 Procedimentos de escape

Um *procedimento de escape* é um procedimento especial. Quando, dentro de uma computação, um procedimento de escape é chamado, o resultado de toda a computação passa a ser igual ao resultado do procedimento de escape: a expressão onde o procedimento de escape estava inserido é ignorada.

Por exemplo, a expressão `(* (+ 10 (* 5 20)) 0.5)` é normalmente avaliada na seguinte ordem:

```
(* (+ 10 (* 5 20)) 0.5) =
(* (+ 10 100) 0.5) =
(* 110 0.5) =
55
```

Supondo a existência de um procedimento `escape` que transforma outros procedimentos em procedimentos de escape, a expressão `(* ((escape +) 10 (* 5 20)) 0.5) =` é

Versão Preliminar

avaliada de outra forma:

```
(* ((escape +) 10 (* 5 20)) 0.5) =
(* ((escape +) 10 100) 0.5) =
(+ 10 100) =            ← aqui está o "escape"
110
```

O procedimento `(escape +)` eliminou a computação que esperava para acontecer (o `(* ···)`).

10.1.3 Continuações

O procedimento Scheme `call-with-current-continuation` (normalmente abreviado como `call/cc`) é usado da seguinte forma:

```
(call/cc
  (lambda (k)
    ...
    (k ...)))
```

O único argumento de `call/cc` é uma função de um argumento (chamada de *recebedor*). `call/cc` fará o seguinte:

- Determinará o contexto atual (é importante lembrar-se de que o contexto é um procedimento);
- Chamará o procedimento `escape` no contexto atual, gerando um procedimento de escape (chamado de *continuação*);
- A variável k, no contexto do recebedor, será a continuação.

Dentro do corpo da função `(lambda (k) ...)`, há uma chamada à continuação k. Quando a chamada acontecer, a computação até ali será ignorada e a continuação usará o contexto anterior (de quando `call/cc` foi chamado).

10.1.4 Exemplos

O trecho de código a seguir soma um com "algo": `(+ 1 □)`.

Versão Preliminar

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
```

O procedimento call/cc tem um único argumento: `(lambda (k) ...)`; o que será passado em `k` para este procedimento é exatamente o "estado atual" da computação, `(+ 1 □)`.

Ao encontrar o lambda, o interpretador terá:

```
(lambda (k)
  (+ 2 (k 3)))
```

Mas o valor de `k` é uma *continuação* – um histórico de computação, neste caso `(+ 1 □)`.

Ao chegar na forma `(k 3)`, o valor de `k` é chamado como procedimento, e ele descartará a computação atual, trazendo a antiga de volta. O contexto atual passa a ser `(+ 1 □)`. A computação será `(+ 1 3)`, e o resultado é 4!

O programa *escapou* da computação ("`+ 2`") para continuar outra.

```
(+ 1 (call/cc
       (lambda (k)
         (+ 2 (k 3)))))
```

`(k 3)` escapa para este ponto

Podemos também armazenar uma continuação em uma variável e usá-la quantas vezes quisermos:

```
(define r #f)

(+ 1 (call/cc
       (lambda (k)
         (set! r k)
         (+ 2 (k 3)))))
```

`r` tem o valor do contexto (da continuação) anterior:

```
(r 5)
6
(+ 3 (r 5))
6
```

Não importa onde `r` seja invocado, ele sempre continuará a computação anterior.

Versão Preliminar

Uma continuação obtida com `call/cc` não precisa necessariamente ser usada.

```
(call/cc
  (lambda (k)
    #t))
```

Este trecho de código sempre retornará `#t`, e o procedimento `k` nunca será usado.

10.2 UM EXEMPLO SIMPLES: ESCAPANDO DE LAÇOS

Pode ser vantajoso abandonar um laço antes da última iteração em algumas situações. O procedimento a seguir calcula o produto de uma lista:

```
(define list-product
  (lambda (s)
    (let more ((s s))
      (if (null? s) 1
          (* (car s) (more (cdr s)))))))
```

Se um dos elementos da lista é zero, não há motivo para continuar. Pode-se incluir uma verificação:

```
(define list-product
  (lambda (s)
    (let recur ((s s))
      (if (null? s) 1
          (if (zero? (car s))
              0
              (* (car s) (recur (cdr s))))))))
```

No entanto, mesmo com esta verificação o programa pode precisar realizar muitos retornos de função (note que este procedimento não é recursivo na cauda).

Uma maneira de escapar deste loop é criar uma continuação antes de iniciá-lo e chamá-la quando um dos elementos for zero:

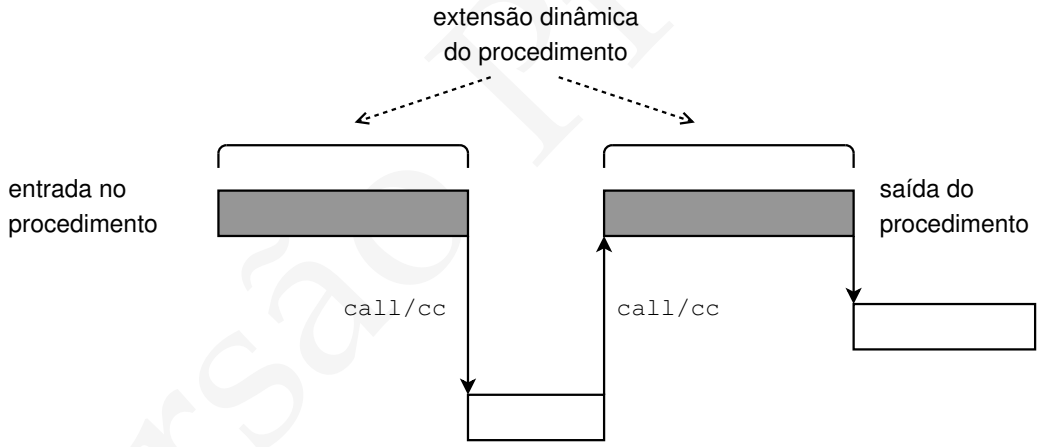
Versão Preliminar

```
(define list-product
  (lambda (s)
    (call/cc     ;; Lembramos do ponto antes de começar o loop...
      ;; Na computação seguinte (a forma lambda), "fora"
      ;; é esta continuação
      (lambda (fora)
        (let recur ((s s))
          (if (null? s) 1
              (if (= (car s) 0) (fora 0)
                  (* (car s) (recur (cdr s))))))))))
```

Se `(car s)` é zero, a computação é `(λ s 0)`.

10.3 EXTENSÃO DINÂMICA E DYNAMIC-WIND

A *extensão dinâmica* de um procedimento é o período em que seu código está sendo executado, entre sua chamada e seu retorno. Como continuações podem entrar e sair de procedimentos a qualquer momento, a extensão dinâmica de procedimentos Scheme pode não ser contínua no tempo.



A entrada na extensão dinâmica de um procedimento acontece em duas situações: quando ele é chamado ou quando alguma continuação, que havia sido criada dentro da extensão dinâmica deste procedimento, é invocada.

A saída da extensão dinâmica de um procedimento acontece quando ele retorna ou quando uma continuação, criada fora da extensão dinâmica deste procedimento, é invocada.

Versão Preliminar

O procedimento `dynamic-wind` força a execução de trechos de código durante qualquer entrada ou saída da extensão dinâmica de um procedimento, inclusive quando a entrada ou saída se dá via continuações. Este procedimento aceita três argumentos:

```
(dynamic-wind
  before
  code
  after)
```

Os argumentos `before`, `code` e `after` devem ser procedimentos sem argumentos. O programa a seguir é usado no padrão R^{5}RS para ilustrar o funcionamento de `dynamic-wind`:

```
(letrec ((path '())
         (c #f)
         (add (lambda (s)
                (set! path (cons s path)))))

  (dynamic-wind
    ;; before:
    (lambda () (add 'connect))

    ;; code goes here:
    (lambda ()
      (add (call/cc
             (lambda (c0)
               (set! c c0)
               'talk1))))

    ;; after:
    (lambda () (add 'disconnect)))

  (if (< (length path) 4)
      (c 'talk2)
      (reverse path)))
(connect talk1 disconnect connect talk2 disconnect)
```

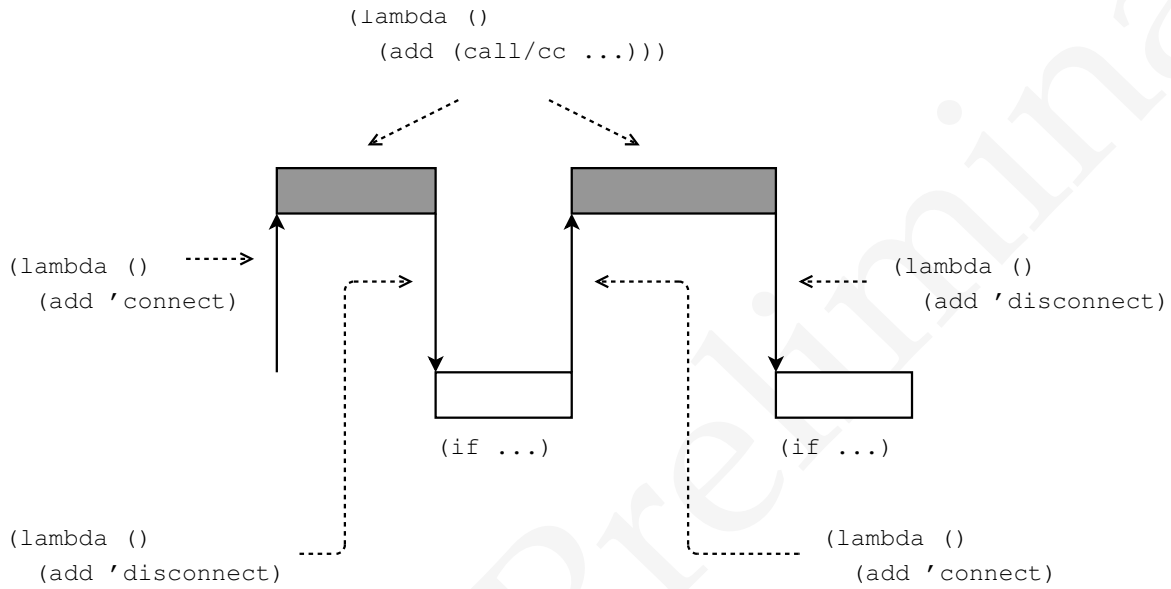
A variável `path` é inicializada com uma lista vazia; `c` será usada para guardar uma continuação; `add` adiciona um elemento a `path`. O programa chama então o procedimento `(lambda () (add (call/cc ...))` *dentro* de uma forma `dynamic-wind`, determinando

Versão Preliminar

que sempre que o programa entrar e sair do escopo dinâmico deste procedimento, deve executar `(lambda () (add 'connect))` e `(lambda () (add 'disconnect))`.

Inicialmente, o trecho é executado uma única vez; no entanto, como o `if` termina por chamar a continuação mais uma vez, o trecho é executado novamente – mas os dois procedimentos são novamente executados, e incluem `connect` e `disconnect` à lista `path`.

A figura a seguir mostra a extensão dinâmica do procedimento `(lambda () (add (call/cc ...)))` e ilustra quando os outros procedimentos são chamados.



10.4 SISTEMAS DE EXCEÇÕES

Continuações podem ser usadas para implementar sistemas de tratamento de exceções. Uma macro simples usando `syntax-rules` é suficiente:

Versão Preliminar

```
(define-syntax try
  (syntax-rules (on-error:)
    ((_ on-error: handler f1 ...)
     (begin
       (let* ((condicao #f)
              (resultado-final #f) ;; presumindo falha
              (talvez
               (call/cc
                (lambda (k)
                  (set! throw (lambda (co)
                                ;; handler deve ser chamado c/
                                ;; argumento de throw, então
                                ;; é necessário guardá-lo:
                                (set! condicao co)
                                ;; força o retorno com false:
                                (k #f)))
                  (set! resultado-final (begin f1 ...))
                  #t)))) ;; não houve throw, retorne true
         (if talvez resultado-final (handler condicao)))))))
```

A macro `try` aceita um procedimento de tratamento de erros e outro procedimento, que será chamado.

```
(try on-error: proc-trata-erro
  proc-a-chamar
```

Quando o procedimento `throw` é chamado e a continuação usada o controle é transferido para o início do procedimento `talvez`. A computação que estava em curso é abandonada e volta-se ao `if`; como o valor de retorno do `throw` será `#f`, a forma `(handler condicao)` é executada.

Se `throw` não for usada, `talvez` retornará `#t` e o `if` retornará o valor de `resultado-final`.

O procedimento `proc-trata-erro` deve aceitar um argumento, e o programador deverá cuidar para que o tipo de dados passado como argumento para `throw` funcione corretamente com `proc-trata-erro`.

Versão Preliminar

```
(try
  on-error: (lambda (e)
              (display (format "--- ERRO: ~a ---~%" e)))
  (display 'tudo-bem)
  (newline)
  (throw 'catastrofe)
  (display 'nunca-chegarei-aqui)
  (newline))
tudo-bem
-- ERRO: catastrofe --
```

Há um pequeno problema com a macro `try`: ela usa `set!` para modificar algumas variáveis locais (`condicao`, `talvez` e `resultado-final`) – o que não tem efeito fora do escopo do `let*` – mas também modifica `handler`. A modificação de `handler` é visível após o uso do try:

```
(try
  on-error: (lambda (e)
              (display (format "--- ERRO: ~a ---~%" e)))
  (throw 'catastrofe))
-- ERRO: catastrofe --

(throw "Uh?")
-- ERRO: uh? --
```

O procedimento de tratamento de erros passado para o `try` permaneceu vinculado à variável global `handler` – e não há sequer como prever quando esta modificação ocorrerá, porque ela só acontece quando uma exceção é levantada!

Há uma solução simples para este problema: a macro `try` pode se encarregar de guardar o valor anterior da variável `throw` – e para que isto faça sentido, deve *haver* um valor anterior de `throw`! Uma chamada a `throw` fora de algum `try` deve resultar em uma mensagem minimamente informativa. A nova versão de `try` mostrada a seguir implementa estas mudanças.

Versão Preliminar

```
(define throw
  (lambda (arg)
    (format "Throw chamado fora de try!")
    (newline)
    (display (format "Argumento de throw: ~a~%" arg))))

(define-syntax try
  (syntax-rules (on-error:)
    ((_ on-error: handler f1 ...)
     (begin
       (let* ((old-throw throw) ;; antigo throw guardado
              (condicao #f)
              (resultado-final #f) ;; presumindo falha
              (talvez
               (call/cc
                (lambda (k)
                  (set! throw (lambda (co)
                                ;; handler deve ser chamado c/
                                ;; argumento de throw, então
                                ;; é necessário guardá-lo:
                                (set! condicao co)
                                ;; força o retorno com false:
                                (k #f)))
                  (set! resultado-final (begin f1 ...))
                  #t)))) ;; não houve throw, retorne true
         ;; restaure throw:
         (set! throw old-throw)
         (if talvez resultado-final (handler condicao)))))))
```

O sistema de tratamento de exceções desenvolvido nesta seção é minimalista, mas ainda assim útil. Há diversos outros modelos de tratamento de exceção possíveis, mas este texto não os abordará.

Ao desenvolver estes sistemas de exceções tivemos que usar uma variável no ambiente global. Isso foi necessário porque o trecho de código a ser executado, `(begin f1 ...)`, pode conter chamadas a outros procedimentos que por sua vez chamam `throw`, e não tínhamos como incluir facilmente vínculo para `throw` em todos estes ambientes a não ser através do ambiente global.

Versão Preliminar

10.5 CO-ROTINAS

Subrotinas são trechos de código com um ponto de entrada e um ponto de saída; aceitam parâmetros na entrada e retornam valores ao terminar. Os procedimentos na linguagem Scheme oferecem toda a funcionalidade de subrotinas (e mais que elas). Quando mais de uma subrotina é chamada, suas invocações ficam sempre empilhadas, e a última a ser iniciada deve ser a primeira a retornar.

Co-rotinas são semelhantes a procedimentos: trechos de código que aceitam argumentos ao iniciar. No entanto, co-rotinas não obedecem a mesma disciplina de pilha que subrotinas: pode-se entrar e sair de uma co-rotina várias vezes, e co-rotinas diferentes podem ser intercaladas no tempo. Uma co-rotina pode retornar valor mais de uma vez (e neste caso é chamada de *gerador*).

Como continuações capturam o contexto atual de um processo, parece natural usá-las para implementar co-rotinas.

```
(define-syntax coroutine
  (syntax-rules ()
    ((coroutine arg resume body ...)
     (letrec ((local-control-state
               (lambda (arg) body ...))
              (resume
               (lambda (c v)
                 (call/cc
                  (lambda (k)
                    (set! local-control-state k)
                    (c v))))))
       (lambda (v)
         (local-control-state v))))))
```

A macro `coroutine` usa dois argumentos seguidos de várias formas: o primeiro argumento, `arg`, é o argumento inicial para a co-rotina; o segundo, `resume`, é o nome do procedimento que será usado para transferir o controle para outra co-rotina; após os dois argumentos vem o corpo da co-rotina.

Dentro do `letrec` criamos uma variável `local-control-state`, que sempre terá como valor o ponto de entrada corrente para esta co-rotina. Inicialmente, este ponto de entrada é o procedimento `(lambda (arg) body ...)`, mas cada vez que o controle deixar esta co-rotina, o valor desta variável passará a ser a continuação corrente. A outra variável local, `resume`, será trocada pelo nome que o programador der ao procedimento de transferência

Versão Preliminar

de controle; este procedimento aceita dois argumentos, `c` e `v` – o primeiro é o procedimento para o qual queremos transferir o controle e o segundo é o argumento que passaremos para este outro procedimento. Após chamar `call/cc`, o procedimento `resume` guarda a continuação corrente em `local-control-state`, para que seja possível retomar mais tarde, e chama `c` com argumento `v`. Em seguida, `local-control-state` é chamado.

Quando outra co-rotina chamar retornar a esta, também chamará o procedimento `local-control-state`, retornando assim ao ponto onde esta rotina havia sido interrompida.

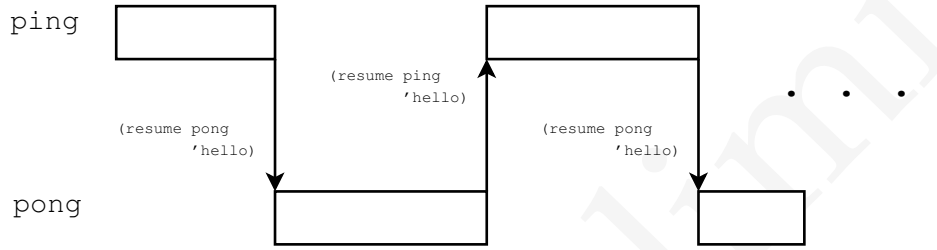
O exemplo a seguir mostra duas co-rotinas, `ping` e `pong`, transferindo controle entre si várias vezes.

```
(define write-line
  (lambda (line)
    (display line)
    (newline)))

;; Como faremos referência a pong antes de defini-lo como
;; procedimento, é boa prática dar-lhe algum valor antes:
(define pong #f)

(define ping
  (coroutine value resume
             (write-line "PING 1")
             (resume pong value)
             (write-line "PING 2")
             (resume pong value)
             (write-line "PING 3")
             (resume pong value)))

(define pong
  (coroutine value resume
             (write-line "Pong 1")
             (resume ping value)
             (write-line "Pong 2")
             (resume ping value)
             (write-line "Pong 3")
             (resume ping value)))
```

Versão Preliminar

```
(ping 'hello)
PING 1
Pong 1
PING 2
Pong 2
PING 3
Pong 3
hello
```

As linhas são mostradas alternadamente pelas duas threads; a última linha é o valor de retorno de `ping`. A figura a seguir ilustra a extensão dinâmica das duas co-rotinas.



10.6 MULTITAREFA NÃO-PREEMPTIVA

Uma das mais importantes aplicações de continuações é a implementação de `threads`. Nesta seção desenvolveremos um sistema multitarefa simples usando continuações.

Guardaremos os processos em uma lista:

```
(define processos '())
```

Para adicionar o argumento à lista processos definimos `enfilera-processo`.

```
(define enfilera-processo
  (lambda (proc)
    (set! processos (append processos (list proc)))))
```

O procedimento `start` retira um processo da lista e o executa.

Versão Preliminar

```
(define start
  (lambda ()
    (let ((p (car processos)))
      (set! processos (cdr processos))
      (p))))
```

`Pause` se lembra de onde estava, inclui o contexto atual no final da fila, depois chama `start` (pega o próximo).

```
(define pause
  (lambda ()
    (call/cc
      (lambda (k)
        (enfilera-processo (lambda () (k #f)))
        (start)))))
```

O procedimento `enfilera-processo` inclui uma função que adiciona à lista o (let ...) que:

- Primeiro, inclui a si mesmo no fim da fila e passa o controle para o primeiro da fila;
- Depois faz algo;
- Por último, chama a si mesmo (entra em loop).

Versão Preliminar

```
(enfilera-processo
  (lambda ()
    (let f ()
      (pause)
      (display "0")
      (f))))
```

O procedimento `enfilera-processo` é usado para incluir tarefas na lista:

```
(for-each enfilera-processo
  (list (lambda () (let f () (pause) (display "1") (f)))
        (lambda () (let f () (pause) (display "a'") (f)))
        (lambda () (let f () (pause) (display "!") (f)))
        (lambda () (let f () (pause) (newline) (f)))))
```

E o procedimento `start` começa a processar as tarefas:

```
(start)
```

Na primeira chamada, cada processo chama `pause` e reinclui a si mesmo na fila; na segunda, já começam os `displays`.

Esta seção trata apenas do mecanismo que intercala os processos, dando a ilusão de que executam em paralelo. Um sistema de threads precisa também oferecer mecanismos para garantir que os processos trabalhem de maneira ordenada, sem que o estado das variáveis globais fique inconsistente. O Capítulo 14 tratará de threads com mais profundidade.

10.7 O GOTO FUNCIONAL E CUIDADOS COM CONTINUAÇÕES

A seguinte chamada a `call/cc` ilustra um fato importante a respeito de continuações.

```
((call-with-current-continuation
  (lambda (k)
    (letrec ((um
              (lambda ()
                (display "Um")
                (newline)
                (k dois)))
             (tres
              (lambda ()
                (display "Tres")
                (newline)
                (k fim)))
             (dois
              (lambda ()
                (display "Dois")
                (newline)
                (k tres)))
             (fim
              (lambda ()
                (display "Fim!")
                (newline)
                #f)))
      um))))
```

Versão Preliminar

Este programa mostra claramente a relação entre continuações em programas funcionais e GOTO em programas imperativos (basta trocar o identificador "k" por "goto" e reler o programa). Embora haja, como o exemplo mostra, claras similaridades entre continuações e o comando GOTO, há também diferenças importantes: continuações trocam de *estado* em um programa, mudando o contexto corrente e se lembrando de algum contexto que faz sentido. O GOTO muda o fluxo de controle arbitrariamente entre *posições no fonte do programa*, podendo inclusive levar o programa a estados inconsistentes ou que não fazem sentido.

Ainda que os problemas com continuações sejam menos graves que aqueles relacionados ao GOTO, elas devem, assim como este, ter seu uso controlado e assim como macros, devem ser usadas apenas quando funções não forem suficientes e isoladas em poucas partes de um programa. Após a verificação que um determinado padrão de uso de continuações se repete e poderia ser abstraído, primitivas são elaboradas usando `call/cc` para implementar estes padrões e isoladas em poucas macros ou funções, como nos exemplos dados (`try`, `coroutine` e `pause` abstraem padrões de uso de continuações).

10.8 NÃO-DETERMINISMO

(esta seção está incompleta)

Suponha que tenhamos três listas de números e queiramos encontrar uma tripla (a, b, c), com um número de cada lista, que possam ser os lados de um triângulo (não degenerado). Por exemplo, se as listas são

```
lista1 = (1 9 3)
lista2 = (2 4 3)
lista3 = (1 2 2)
```

Podemos formar triângulos com $(1, 2, 2)$, $(3, 2, 2)$, $(3, 4, 2)$, $(3, 3, 1)$ e $(3, 3, 2)$ – mas não com $(1, 2, 1)$, que é degenerado, ou $(1, 4, 2)$, que não pode ser triângulo.

Podemos testar uma a uma as possibilidades até encontrar uma. Inicialmente escrevemos um predicado que determina se tres valores podem ser lados de um triângulo.

```
(define tri?
  (lambda (a b c)
    (and (< a (+ b c))
         (< b (+ a c))
         (< c (+ a b)))))
```


Versão Preliminar

```
(tri? 1 2 1)
#f
(tri? 3 2 2)
#t
(tri? 1 3 1)
#f
```

Agora construímos o procedimento `acha-triangulo`, que recebe tres listas, l_1, l_2 e l_3. A lista l_1 contém todas as possibilidades para o primeiro lado; a lista l_2 tem as possibilidades para o segundo lado, e a lista l_3 as possibilidades para o terceiro lado. O procedimento usa tres variáveis, `i`, `j` e `k` para percorrer cada lista, tentando todas as possibilidades. Note que usamos uma continuação para escapar quando encontrarmos um triângulo válido.

```
(define acha-triangulo
  (lambda (l1 l2 l3)

    ;; quando encontrar um triangulo,
    ;; chame (out lista-de-lados)
    (call/cc
     (lambda (out)

       (do ((i 0 (+ i 1)))
           ((= i (length l1)))
         (do ((j 0 (+ j 1)))
             ((= j (length l2)))
           (do ((k 0 (+ k 1)))
               ((= k (length l3)))
             (let ((a (list-ref l1 i))
                   (b (list-ref l2 j))
                   (c (list-ref l3 k)))

               (cond ((tri? a b c) ;; <== triangulo encontrado!
                      (out (list a b c))))))))))))

(acha-triangulo '(1 9 3)
                '(2 4 3)
                '(1 2 2))
```

Versão Preliminar

```
(1 2 2)
```

Se removermos o `call/cc` e acumularmos os resultados em uma lista, obteremos todos os valores.

Este programa, no entanto, é um tanto deselegante e parece demasiado complexo para uma tarefa que deveria ser simples. Gostaríamos de poder especificar o programa da seguinte maneira: recebemos as tres listas de possíveis tamanhos para lados, dizemos que `a` deve vir da primeira lista, `b` da segunda e `c` da terceira, e que estes devem satisfazer `(tri? a b c)`. Tendo informado isso, o programa deve sozinho encontrar os valores para `a`, `b` e `c`.

```
(define amb-tri
  (lambda (l1 l2 l3)
    (let ((a (choose-from-list l1))
          (b (choose-from-list l2))
          (c (choose-from-list l3)))
      (require (tri? a b c))
      (display (list a b c)))))

(amb-tri '(1 9 3)
         '(2 4 3)
         '(1 2 2))
(1 2 2)
```

O código acima não especifica *como* a solução será encontrada; isto será feito automaticamente por `choose-form-list` e `require` (que farão a busca exaustiva, tentando uma a uma as soluções). Algoritmos descritos desta forma são chamados de *não-determinísticos*.

Até agora descrevemos computações através de procedimentos *determinísticos:* se olharmos para todo o ambiente e para a próxima instrução (a próxima forma Scheme), não haverá dúvida quanto ao próximo passo da execução.

Desenvolveremos uma macro `amb` e um procedimento `require`. A macro `amb` escolherá não-deterministicamente algum dos seus argumentos e o retornará. O procedimento `require` será usado para impor restrições adicionais.

Por exemplo, `(define x (list (amb 'alfa 'beta) (amb 1 2)))` poderá dar a `x` um dentre quatro valores: `(alfa 1)` `(alfa 2)` `(beta 1)` `(beta 2)`. Se em seguida determinarmos `(require (even? (cadr x)))`, teremos garantido que `x` vale `(alfa 2)` ou `(beta 2)`.

Versão Preliminar

```
(define x (list (amb 'alfa 'beta) (amb 1 2)))
x
(beta 1)
(require (even? (cadr x))))
x
(beta 2)
```

Definiremos `amb` da seguinte maneira:

- Quando `amb` for chamado sem argumentos, não há o que escolher, e um erro deverá ser gerado (este é o comportamento *inicial* de `(amb)`, que poderá ser modificado posteriormente).
- Quando chamado com um único argumento, `amb` deve retorná-lo, porque é a única alternativa.
- Quando chamado com vários argumentos, `amb` deve retornar um deles, mas modificar o comportamento de `(amb)` sem argumentos, para que não gere erro, e ao invés disso retorne outro elemento da lista.

Primeiro definimos o procedimento `amb-fail`:

```
(define amb-fail
  (lambda args
    (error "Non-deterministic search exhausted")))
```

É comum usar a mensagem "Amb tree exhausted" para esta falha. Preferimos falar de "busca não-determinística".

Como exemplo usaremos a forma `(define x (amb 10 20 30))`. Estamos passando a `amb` uma lista de tres opções, 10, 20 e 30. Inicialmente, qualquer valor pode ser atribuído a `x`, mas queremos poder escolher outro ao usar `(amb)`, até que todos tenham sido usdos – e depois disso qualquer chamada a `(amb)` deverá gerar erro.

A expansão desta forma poderia ser o seguinte código.

Versão Preliminar

```
(define x  (let ((saved-fail amb-fail))

             ;; lembrando onde começamos, para poder
             ;; retornar aqui e refazer o "define x"
             ;; com outros valores:
             (call/cc
               (lambda (k-success)

                 (call/cc
                   (lambda (k-failure)
                     ;; a próxima chamada a
                     ;; (amb) cairá aqui:
                     (set! amb-fail
                           (lambda ()
                             (set! amb-fail saved-fail)
                             (k-failure (quote boo))))

                     ;; define o valor de x como 10:
                     (k-success 10)))

                 (call/cc
                   (lambda (k-failure)
                     (set! amb-fail
                           (lambda ()
                             (set! amb-fail saved-fail)
                             (k-failure (quote boo))))
                     (k-success 20)))

                 (call/cc
                   (lambda (k-failure)
                     (set! amb-fail
                           (lambda ()
                             (set! amb-fail saved-fail)
                             (k-failure (quote boo))))
                     (k-success 30)))
                 (saved-fail))))
```

Versão Preliminar

Primeiro o `let` guarda o conteúdo original de `amb-fail`. O valor do `call/cc` externo será o valor de `x`.

Os `call/cc` internos são executados em sequência, mas cada um deles interrompe a computação para retornar ao `call/cc` externo, dando um valor para `x`.

Antes de retornar o valor de `x`, cada `call/cc` interno muda o valor de `amb-fail` para a sua continuação. Assim, quando `(amb)` é chamado sem argumentos, o próximo `call/cc` interno é avaliado, e um novo valor é escolhido para `x`.

O `define` poderá ser finalizado várias vezes.

Agora que sabemos como deve ser a expansão de `amb`, escrevemos a macro.

```
(define-syntax amb
  (syntax-rules ()
    ((amb) (amb-fail))
    ((amb expression) expression)
    ((amb expression ...)
     (let ((saved-fail amb-fail))
       (call/cc
        (lambda (k-success)
          (call/cc
           (lambda (k-failure)
             (set! amb-fail (lambda ()
                              (set! amb-fail saved-fail)
                              (k-failure 'boo)))
             (k-success expression)))
           ...
           (saved-fail)))))))
```

Definimos em seguida `require`, que verifica se uma condição foi satisfeita; se não tiver sido, chama `(amb)` para tentar uma nova escolha.

O procedimento `require` deve ser definido *depois* da macro `amb`, porque usa `amb` internamente (e `amb` deve estar disponível quando `require` for *lido*).

```
(define (require p)
  (if (not p) (amb)))
```

Com `amb` e `require` (e também `amb-fail`, usado por `amb`) já podemos implementar programas não-determinísticos. Desenvolveremos também procedimentos e macros que simplificarão seu uso.

Versão Preliminar

O procedimento `amb-list` recebe suas opções em uma lista, e não como argumentos. Isto será particularmente útil quando quisermos passar para `amb` uma lista muito longa, construída em tempo de execução (não podemos usar `apply` porque `amb` é uma macro).

```
(define amb-list
  (lambda (lst)
    (let loop ((lst lst))
      (if (null? lst)
          (amb)
          (amb (car lst) (loop (cdr lst)))))))
(define opcoes '(um dois tres quatro))
(define a (amb-list opcoes))
a
um
(amb)
a
dois
(amb)
a
tres
```

A macro `amb-all` permite listar todas as possibilidades que seriam tentadas por `amb`.

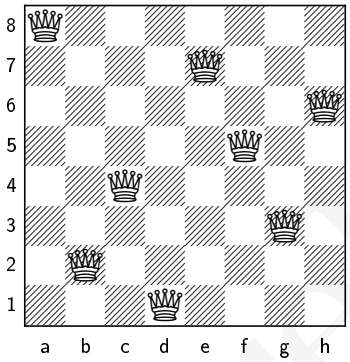
```
(define-syntax amb-all
  (syntax-rules ()
    ((_ e)
     (let ((saved-fail amb-fail)
           (results '()))
       (if (call/cc
            (lambda (k)
              (set! amb-fail (lambda () (k #f)))
              (let ((v e))
                (set! results (cons v results))
                (k #t))))
           (amb-fail))
       (set! amb-fail saved-fail)
       (reverse results)))))
```


Versão Preliminar

```
(define opcoes '(um dois tres quatro))
(amb-all (amb-list opcoes))
(um dois tres quatro)
```

10.8.1 Exemplo: n rainhas

O problema das oito rainhas consiste em encontrar uma maneira de dispor sobre um tabuleiro de xadrez oito rainhas, de forma que nenhuma delas ataque a outra. A próxima figura mostra uma solução para o problema.



A seguir discutimos como resolver o problema no tabuleiro tradicional, de tamanho 8×8 – mas pode-se também definir o problema para outros tamanhos de tabuleiro.

Como há 64 posições em um tabuleiro, o número de configurações possíveis para oito rainhas é $\binom{64}{8} = 4426165368$. Não é necessário, no entanto, tentar todas as possíveis configurações: sabemos que não poderemos posicionar mais de uma rainha por coluna (porque elas evidentemente se atacariam). Como o número de rainhas é igual ao de colunas, podemos presumir que haverá exatamente uma por coluna. Só resta escolher a linha em que cada rainha ficará. Como também não podemos posicionar duas rainhas na mesma linha, haverá oito possibilidades para a primeira rainha; para a segunda, sete (porque uma foi tomada pela primeira); na terceira, seis, e assim por diante. O número de configurações é $8! = 40320$ – muito menor que se fizéssemos a busca ingênua que descrevemos antes. Se eliminarmos as casas atacadas pela diagonal este número fica ainda menor.

Nosso programa então deverá atribuir a cada rainha uma linha, sendo que cada rainha já terá sua coluna determinada.

Duas rainhas se atacam quando estão na mesma linha diagonal, horizontal ou vertical. Construiremos dois procedimentos, `diagonal-attack?` para verificar se as duas estão na

Versão Preliminar

mesma diagonal, e `staight-attack?` para verificar se estão na mesma linha horizontal ou vertical.

Duas posições estão na mesma diagonal quando podemos chegar de uma a outra andando o mesmo número de casas na horizontal e na vertical. Isto é o mesmo que dizer que a distância entre as colunas deve ser igual à distância entre as linhas.

```
(define diagonal-attack?
  (lambda (q1-col q1-lin
           q2-col q2-lin)
    (= (abs (- q2-col q1-col))
       (abs (- q2-lin q1-lin)))))
```

Para saber se duas posições estão na mesma linha horizontal ou vertical verificamos se ambas tem alguma coordenada em comum.

```
(define straight-attack?
  (lambda (q1-col q1-lin
           q2-col q2-lin)
    (or (= q1-col q2-col) (= q1-lin q2-lin))))
```

O procedimento `safe?` verifica se uma nova rainha pode ser adicionada em uma posição, considerando as posições já tomadas por uma lista de outras rainhas.

As colunas são numeradas em ordem inversa: A última da lista tem coluna igual a um; a primeira tem coluna igual a `(length queens)`. A nova rainha ficará na coluna `(+ 1 (length queens))`. O procedimento determina estes valores (`q-col` e `q2-col`) logo no início. O laço interno verifica para cada rainha `q2` da lista se há um ataque.

Versão Preliminar

```
(define safe?
  (lambda (q-lin queens)
    (let ((q-col (+ 1 (length queens))))
      (let loop ((q2-col (length queens))
                 (rest-queens queens))
        (if (null? rest-queens)
            #t
            (let ((q2-lin (car rest-queens)))
              (cond ((diagonal-attack? q-col  q-lin
                                       q2-col q2-lin)
                     #f)
                    ((straight-attack? q-col  q-lin
                                       q2-col q2-lin)
                     #f)
                    (else
                     (loop (- q2-col 1)
                           (cdr rest-queens)))))))))))
```

Para cada rainha o procedimento `8-queens` determina não-deterministicamente uma linha (a coluna da i-ésima rainha é a coluna i). Sempre que uma nova rainha é posicionada, chamamos `require` para encontrar uma posição segura para ela.

Versão Preliminar

```
(define 8-queens
  (lambda ()
    (let ((q1 (amb 1 2 3 4 5 6 7 8))
          (q2 (amb 1 2 3 4 5 6 7 8)))
      (require (safe? q2 (list q1)))
      (let ((q3 (amb 1 2 3 4 5 6 7 8)))
        (require (safe? q3 (list q2 q1)))
        (let ((q4 (amb 1 2 3 4 5 6 7 8)))
          (require (safe? q4 (list q3 q2 q1)))
          (let ((q5 (amb 1 2 3 4 5 6 7 8)))
            (require (safe? q5 (list q4 q3 q2 q1)))
            (let ((q6 (amb 1 2 3 4 5 6 7 8)))
              (require (safe? q6 (list q5 q4 q3 q2 q1)))
              (let ((q7 (amb 1 2 3 4 5 6 7 8)))
                (require (safe? q7 (list q6 q5 q4 q3 q2 q1)))
                (let ((q8 (amb 1 2 3 4 5 6 7 8)))
                  (require (safe? q8 (list q7 q6 q5 q4 q3 q2 q1)))
                  (list q1 q2 q3 q4 q5 q6 q7 q8)))))))))))
```

Testamos o procedimento: `(8-queens)`

```
(1 5 8 6 3 7 2 4)
```

O Exercício 145 pede a construção de um procedimento que mostre o tabuleiro com as rainhas nas posições indicadas pelo procedimento `8-queens`.

```
(show-queens (n-queens 8))
```

```
+-+-+-+-+-+-+-+-+
|Q| | | | | | | |
+-+-+-+-+-+-+-+-+
| | | | |Q| | | |
+-+-+-+-+-+-+-+-+
| | | | | | | |Q|
+-+-+-+-+-+-+-+-+
| | | | | |Q| | |
+-+-+-+-+-+-+-+-+
| | |Q| | | | | |
+-+-+-+-+-+-+-+-+
| | | | | | |Q| |
```

Versão Preliminar

```
+-+-+-+-+-+-+-+-+
| |Q| | | | | | |
+-+-+-+-+-+-+-+-+
| | | |Q| | | | |
+-+-+-+-+-+-+-+-+
```

Como há mais de uma solução, podemos usar `(amb)` para forçar o *backtracking* e obter uma nova solução (a busca continuará *como se a solução corrente não tivesse sido aceita e uma falha tivesse ocorrido*:

```
(8-queens)
(1 5 8 6 3 7 2 4)
(amb)
(1 6 8 3 7 4 2 5)
(amb)
(1 7 4 6 8 2 5 3)
(amb)
(1 7 5 8 2 4 6 3)
```

Para obter todas as soluções usamos a macro `amb-all`:

```
(amb-all (8-queens))
((1 5 8 6 3 7 2 4) (1 6 8 3 7 4 2 5) (1 7 4 6 8 2 5 3)
(1 7 5 8 2 4 6 3) (2 4 6 8 3 1 7 5) (2 5 7 1 3 8 6 4)
(2 5 7 4 1 8 6 3) (2 6 1 7 4 8 3 5) (2 6 8 3 1 4 7 5)
⋮
(7 4 2 8 6 1 3 5) (7 5 3 1 6 8 2 4) (8 2 4 1 7 5 3 6)
(8 2 5 3 1 7 4 6) (8 3 1 6 2 5 7 4) (8 4 1 3 6 2 7 5))
```

Notamos que o procedimento anterior parece um tanto repetitivo: os pares de linha `let/require` se repetem com mínimas mudanças de um para outro. Além disso, gostaríamos de poder resolver o problema para tamanhos diferentes de tabuleiro – seis rainhas em um tabuleiro 6×6, por exemplo. Reusaremos então o procedimento `safe?`, e escreveremos um procedimento `n-queens` que tentará posicionar n rainhas em um tabuleiro $n \times n$.

No procedimento `8-queens` as posições de cada rainha eram escolhidas usando `(amb 1 2 3 4 5 6 7 8)`. Como queremos que o número de argumentos para `amb` seja variável, usaremos `amb-list`: dado o parâmetro n, criamos uma lista com números de 1 a n, chamada `one-to-n` (usamos o procedimento `iota` da SRFI-1, descrito na Seção 6.1).

Depois iniciamos com uma lista `queens` vazia e adicionamos uma a uma as rainhas e escolhemos as linhas das rainhas com `(amb-list one-to-n)`.

Versão Preliminar

```
(define n-queens
  (lambda (n)
    (let ((one-to-n (iota n 1)))
      (let loop ((i 1) (queens '()))
        (if (> i n)
            queens
            (let ((new-queen (amb-list one-to-n)))
              (require (safe? new-queen queens))
              (loop (+ i 1) (cons new-queen queens))))))))
(n-queens 6)
(2 4 6 1 3 5)
(show-queens (n-queens 6))
+-+-+-+-+-+-+
| |Q| | | | |
+-+-+-+-+-+-+
| | | |Q| | |
+-+-+-+-+-+-+
| | | | | |Q|
+-+-+-+-+-+-+
|Q| | | | | |
+-+-+-+-+-+-+
| | |Q| | | |
+-+-+-+-+-+-+
| | | | |Q| |
+-+-+-+-+-+-+
```

10.8.2 amb como procedimento

Antes de implementarmos `amb` como procedimento, deve ficar claro porque o fizemos inicialmente como macro. Considere o código a seguir:

Versão Preliminar

```
(define resposta
  (lambda ()
    ;; segue aqui computação extremamente demorada que,
    ;; após 7 e meio milhões de anos, resulta na resposta
    ;; para a vida, o Universo, e tudo mais.
  ))

(define x (amb (+ 2 2)
               (resposta)))
```

Se a primeira opção, `(+ 2 2)`, for suficiente, e a segunda nunca for necessária, a *macro* `amb` nunca avaliará o segundo argumento (que é demasiado lenta). Quando implementamos `amb` como procedimento, sempre que o usarmos, forçosamente teremos todos os argumetnos avaliados, antes do início da avaliação do corpo do procedimento. Isso não significa que a macro `amb` seja "melhor" que o procedimento equivalente, de forma absoluta. Como veremos adiante, a implementação de `amb` como procedimento nos dará acesso à sequencia de continuações de maneira fácil e clara (isso será usado no Capítulo 12, na implementação de um interpretador Prolog).

Representaremos as continuações como uma pilha, representada como lista.

```
(define amb+-stack '())

(define (amb+-reset)
  (set! amb+-stack '()))

(define (amb+-pop)
  (set! amb+-stack (cdr amb+-stack)))

(define (amb+-push k)
  (set! amb+-stack (cons k amb+-stack)))
```

O procedimento `amb+-fail` é semelhante ao procedimento de falha que usamos com a macro `amb`.

Versão Preliminar

```
(define (amb+-fail)
  (if (null? amb+-stack)
      (error "amb+: no more choices!")
      (let ((back (car amb+-stack)))
        (amb+-pop)
        (back back))))
```

O procedimento `amb+` captura a continuação no momento em que é chamado. De forma análoga à macro, se o procedimento é chamado sem argumentos, resulta em uma chamada a `amb+-fail`. Quando há argumentos, o procedimento lembra-se do primeiro (guarda na variável `choice`), modifica a variável local `choices` removendo seu `car`, empilha a continuação capturada no começo do procedimento e retorna `choice`. Observe que ao retornar ao procedimento quando a continuação for usada as variáveis serão modificadas para seus valores anteriores, *mas* `choices` *foi definida antes da continuação, e a remoção do* `car` *não será desfeita – o* `car` *removido já foi coletado para o lixo!* Assim, quando a continuação que foi posta no topo da pilha for usada, o segundo argumento de `amb+` será usado, e assim por diante.

```
(define (amb+ . args)
  (let ((choices args))
    (let ((cc (now)))
      (cond ((null? choices) (amb+-fail))
            ((pair? choices)
             (let ((choice (car choices)))
               (set! choices (cdr choices))
               (amb+-push cc)
               choice))
            (else (error "amb choices must be a list"))))))
```

Cada vez que `amb+` é usado com argumentos, uma continuação é incluída no topo da pilha.

Os procedimentos `amb+-list` e `require+` serão úteis.

```
(define (amb+-list lst) (apply amb+ lst))

(define (require+ p)
  (if (not p) (amb+)))
```

Versão Preliminar

A variável global `amb+-stack` contém a pilha de continuações de `amb+`, e podemos modificá-la se for necessário.

EXERCÍCIOS

Ex. 133 — Determine os contextos das subexpressões:

a) `(+ 2 3)` em `(abs (+ 2 3))`

b) `x` em `(* x (- 4 y))`

c) `(sqrt (* a b))` em `(* a (log (sqrt (* a b))) c)`

d) `x` em:

```
(cond ((> a b)
       (* 2 x))
      ((< a b)
       (* 10 y))
      (else 0))
```

Ex. 134 — Implemente um procedimento `contexto` que aceite dois argumentos, uma expressão (uma lista de símbolos) e uma subexpressão, e determine o contexto (também em forma simbólica). Se o segundo argumento não for subexpressão do primeiro, o procedimento `contexto` deve retornar `#f`

Ex. 135 — Explique detalhadamente o que este programa faz:

```
(let ((c #f))
  (call/cc (lambda (k)
             (set! c k)))
  (c #f))
```

Ex. 136 — Descreva uma macro `while` que use uma continuação para sair do laço.

Ex. 137 — Explique *como* as expressões são avaliadas, e *o que* retorna da avaliação:

Versão Preliminar

```
(call/cc call/cc)

(call/cc (lambda (x) (x x)))

(call/cc (lambda (x) x))

((call/cc call/cc) call/cc)

(call/cc (call/cc call/cc))

(call/cc (lambda (x)
           (call/cc (lambda (y)
                      (x y)))
           (print 'ok)))
```

Ex. 138 — Modifique o mini-sistema de tratamento de exceções da seção 10.4 para que o tratador de erros possa aceitar um número arbitrário de argumentos. Assim seria possível usar `try` da seguinte forma:

```
(try on-error:
     (lambda (erro status)
       (display (format "erro: ~a~%" erro))
       (display (format "status após erro: ~a~%"
                        status)))
  ...
  (throw 'erro-conexao
         "selecionei outro host"))
```

Ex. 139 — No final da Seção 10.4 mencionamos porque tivemos que usar o ambiente global para guardar o procedimento `throw`. Desenhe um diagrama de ambientes para ilustrar o problema.

Ex. 140 — Mostre que na verdade é possível resolver o problema descrito no final da Seção 10.4.

Ex. 141 — É possível implementar uma primitiva que constrói co-rotinas usando procedimentos ao invés da macro usada na seção 10.5. Mostre como fazê-lo, e explique detalhadamente os procedimentos.

Versão Preliminar

Ex. 142 — Reimplemente o pequeno sistema de threads usando a primitiva `coroutine`.

Ex. 143 — Escreva procedimentos semelhantes ao que soluciona o problema das n rainhas, para outras peças de xadrez (reis, bispos, cavalos e torres).

Ex. 144 — Reescreva o gerador de soluções para o problema das n rainhas sem usar não-determinismo.

Ex. 145 — Escreva o procedimento `show-queens`, mencionado na Seção 10.8.1

Ex. 146 — Reescreva `amb` usando outro sistema de macros.

Ex. 147 — Modifique a implementação de `amb` dada neste Capítulo para que faça busca em largura.

Ex. 148 — Modifique a implementação de `amb` dada neste Capítulo para que faça um misto de busca em largura e em profundidade (escolha alternadamente – ou aleatoreamente – para cada parâmetro de `amb` o tipo de busca.

Ex. 149 — Modifique o interpretador do Capítulo 7, tornando-o não-determinístico (a forma especial `amb` deve ser incluída no interpretador, sem que seja necessário para o usuário usar macros e continuações).

Ex. 150 — A (extremamente confusa) linguagem INTERCAL foi projetada como uma paródia das linguagens da década de 70, com inúmeras *features*. Um dos comandos de INTERCAL é o "`COME FROM`", paródia do `GOTO`. Enquanto `GOTO x` faz o controle mudar para o local onde `x` foi definido, o comando `COME FROM x` faz o programa pular imediatamente para a linha do `COME FROM` assim que chegar ao rótulo `x`.
Tente implementar `come-from` em Scheme, e diga quais dificuldades você encontrou.

```
(begin
  (print "Da tribo pujante")
  (aqui)
  (print "Que agora anda errante")
  (print "Por fado inconstante")
  (come-from aqui)
  (print "Guerreiros, nasci;"))
Da tribo pujante
Guerreiros, nasci;
```

Versão Preliminar

```
(define is-number-list?
  (lambda (lst)
    (map (lambda (x)
           (if (not (number? x))
               (out)))
         #t)))

(let ((lista '(1 2 3 a 5)))
  (if (is-number-list? lista)
      #t
      (begin (come-from out)
             #f)))
```

a) Inicialmente, presuma que haverá exatamente um `come-from` com cada rótulo.

b) Faça funcionar agora o caso em que há mais de um `(come-from x)` para o *mesmo* `x`. Como há dois pontos para onde o programa pode ir, você pode escolher o critério para decidir qual caminho tomar. Por exemplo, o primeiro ou último `(come-from x)` do programa. Ou ainda, *qualquer* um dos `(come-from x)`, escolhido aleatoreamente.

Ex. 151 — Porque não podemos usar `amb` aninhados? Como modificar `amb` para que posamos fazê-lo?

RESPOSTAS

Resp. (Ex. 143) — Faça procedimentos que verifiquem se uma peça (um bispo, por exemplo) está sob ataque; depois passe este procedimento como parâmetro para o solucionador.

Resp. (Ex. 144) — Use o procedimento `with-permutations` definido na Seção 6.1.1.

Resp. (Ex. 145) — Uma possível implementação é mostrada a seguir.

Versão Preliminar

```
(define show-queens
  (lambda (queens)
    (define display-line
      (lambda (n)
        (display "+")
      (do ((i 0 (+ i 1)))
          ((= i n))
        (display "-+"))
      (newline)))

    (let ((n (length queens)))
      (display-line n)
      (do ((lin 1 (+ 1 lin)))
          ((> lin n))
        (display "|")
        (do ((col 1 (+ 1 col)))
            ((> col n))
          (if (= col (list-ref queens (- lin 1)))
              (display 'Q)
              (display #\space))
          (display "|"))
        (newline)
        (display-line n)))))
```

Resp. (Ex. 146) — Macros com renomeação explícita:

Versão Preliminar

```
(define-syntax amb
  (lambda (exp r cmp)
    (let ((args (cdr exp)))
      (cond ((null? args)
             `(,(r 'fail)))
            ((= (length args) 1)
             (cadr args))
            (else
             (let ((value-cases
                    (map (lambda (the-value)
                           `(,(r 'call/cc)
                             (,(r 'lambda) (k-failure)
                               (,(r 'set!) amb-fail (lambda ()
                                                      (,(r 'set!) fail
                                                                  saved-fail)
                                                      (k-failure 'boo)))
                               (k-success ,the-value))))
                         args)))

               `(let ((saved-fail amb-fail))
                  (,(r 'call/cc)
                   (,(r 'lambda) (k-success)
                     ,@value-cases
                     (saved-fail)))))))))
```

Resp. (Ex. 147) — Uma possível solução usaria algo parecido com os procedimentos a seguir.

Versão Preliminar

```
(define really-fail
  (lambda ()
    (error "amb: no more choices")))

(define amb-fail
  (lambda ()
    (if (null? backtrack-points)
        (really-fail)
        (let ((k (prox backtrack-points)))
          (set! backtrack-points (remove-prox backtrack-points))
          (k)))))
```

Os procedimentos `prox` e `remove-prox` determinam como a busca é feita.

Resp. (Ex. 149) — O livro de Abelson e Sussman [AS96] contém um interpretador meta-circular (diferente do apresentado neste livro). Tente entendê-lo e em seguida adaptar algumas de suas ideias para o nosso interpretador.

Resp. (Ex. 150) — Não é possível implementar `come-from` apenas com continuações. Também não é possível fazê-lo interpretando uma forma por vez, como normalmente se faz em Scheme. A única maneira de fazê-lo é em tempo de compilação (ou de leitura), tendo acesso tanto ao rótulo `(aqui)` e ao comando `(come-from aqui)`. Isto não é um problema para INTERCAL, porque nela o programa é lido de uma só vez.

Versão Preliminar

Versão Preliminar

11 PREGUIÇA

Ao encontrar uma expressão um interpretador pode avaliá-la imediatamente e determinar seu valor (como em todos os exemplos vistos até agora) ou pode esperar até que o valor da expressão seja usado em alguma outra computação. O primeiro caso é chamado de *avaliação estrita*, e o segundo de *avaliação preguiçosa*.

Em Scheme a avaliação é estrita por *default*:

```
(define a (quotient 1 0))
Error: (quotient) division by zero
```

O valor de `a` não era ainda necessário (ele não seria mostrado pelo REPL, e nem usado em alguma computação). Mesmo assim o interpretador Scheme tentou calcular o valor imediatamente – e como houve uma tentativa de dividir um por zero, o REPL devolveu uma mensagem de erro.

11.1 DELAY E FORCE

Se, no entanto, pedirmos ao interpretador que guarde a expressão para avaliar mais tarde, ele o fará. Para isto usamos a macro `delay`:

```
(define a (delay (quotient 1 0)))
a
#<promise>
```

O valor de `a` é uma "promessa": `a` é um procedimento que poderá retornar o valor de `(quotient 1 0)`, mas somente quando for necessário. O procedimento `force` toma uma promessa e tenta obter seu valor:

```
(force a)
Error: (quotient) division by zero
```

No próximo exemplo, definimos duas variáveis com o valor `(* 10 10 10 10)` – uma usando avaliação estrita e outra usando avaliação preguiçosa.

```
(define d (* 10 10 10 10))
(define e (delay (* 10 10 10 10)))
```


Versão Preliminar

```
d
10000
e
#<promise>
```

O resultado de `delay` pode ou não ser um procedimento. Muitas implementações de Scheme definem um tipo `promise`. Se `e` não é um procedimento, não podemos aplicá-lo, e é necessário usar force:

```
(+ e 1)
Error: (+) bad argument type: #<promise>
(+ (force e) 1)
10001
```

O próximo exemplo mostra que o ambiente de uma expressão ainda não avaliada é aquele onde ela foi definida – mas podemos modificá-lo antes de usar `force`.

Definimos uma variável `zz` que faz referência a outras duas variáveis que ainda não definimos:

```
(define nome-completo
               (delay (string-append nome
                                     sobrenome)))
Warning: the following toplevel variables
are referenced but unbound:
nome
sobrenome
```

Tentar forçar `nome-completo` é evidentemente um erro:

```
(force nome-completo)
Error: unbound variable: nome
```

Podemos tentar criar um ambiente léxico onde `nome` e `sobrenome` existam e forçar `nome-completo` neste ambiente, mas isto também resultará em erro:

```
(let ((nome "Richard") (sobrenome "Wagner")) (force nome-completo))
Error: unbound variable: nome
```

O ambiente de `nome-completo` é o ambiente léxico onde a definimos. Como `nome-completo` foi definida no ambiente global, podemos criar os vínculos faltantes ali.

Versão Preliminar

```
(define nome "Richard")
(define sobrenome "Wagner")
(force nome-completo)
"Richard Wagner"
```

11.1.1 Como implementar *delay* e *force*

A implementação de `delay` e `force` é surpreendentemente simples. Aqui os definiremos com os nomes `depois` e `agora`:

```
(define-syntax depois
  (syntax-rules ()
    ((depois x)
     (lambda () x))))

(define agora
  (lambda (x) (x)))
```

A macro `depois` aceita uma S-expressão e devolve um procedimento que, ao ser executado, avalia a S-expressão. Este procedimento retornado por `depois` é a "promessa".

O procedimento `agora` toma a promessa (o procedimento) e o executa, produzindo o valor desejado.

Há um problema com esta implementação de `depois/agora`. Quando executamos o procedimento `agora` várias vezes, ele avalia novamente a expressão cada vez que é executado.

```
(define a (depois
            (begin
              (display "Calculando")
              (newline)
              (+ 10 10))))
a
#<procedure (a)>
(a)
Calculando
20
(a)
Calculando
```

Versão Preliminar

```
20
```

No entanto, o padrão R^{5}RS exige que o valor seja calculado uma única vez.

```
(define a (delay   ;; <== Mudamos para o delay do Scheme
            (begin
              (display "Calculando")
              (newline)
              (+ 10 10))))
(force a)
Calculando
20
(force a)
20
```

A segunda chamada a `(force a)` não mostrou a string "Calculando", porque um valor para a expressão já havia sido calculado.

Resolveremos o problema alterando a macro `depois`:

```
(define-syntax depois
  (syntax-rules ()
    ((depois x)
     (let ((done #f)
           (value #f))
       (lambda ()
         (cond ((not done)
                (set! value x)
                (set! done #t)))
         value)))))
```

Esta implementação é bastante simplista, porque a macro `depois` gera um procedimento que não é distinguível de outros procedimentos:

```
(define x (depois (* 10 20)))
x
#<procedure (?)>
(x)
200
```


Versão Preliminar

Desta forma o procedimento `agora` não seria necessário. No entanto, é importante que promessas sejam tratadas como um tipo especial de procedimento e que o procedimento `force` existe para manter a clareza conceitual do código. Implementações de Scheme normalmente definem promessas como uma estrutura especial, e não como procedimento comum:

```
(define b (delay (* 10 20)))
b
#<promise>
(b)
Error: call of non-procedure: #<promise>
```

11.2 ESTRUTURAS INFINITAS

A implementação de listas em Scheme envolve três predicados, `cons`, `car` e `cdr`. Se todo o tratamento da cauda da lista passar a ser feito de forma preguiçosa, pode-se definir listas infinitas.

```
(define random-list
  (lambda ()
    (cons (random 10)
          (delay (random-list)))))
```

O procedimento `random-list` gera um número aleatóreo para a primeira posição da stream, mas deixa uma promessa no lugar do "resto":

```
(define l (random-list))
l
(5 . #<promise>)
```

O procedimento `lazy-take` força a computação dos n primeiros elementos da stream:

Versão Preliminar

```
(define lazy-take
  (lambda (lst n)
    (if (positive? n)
        (cons (car lst)
              (lazy-take (force (cdr lst))
                         (- n 1)))
        '())))
(lazy-take l 5)
(5 4 4 1 1)
```

Subsequentes chamadas a `lazy-take` não mudam a sequencia de números já gerados, porque `force` não fará a mesma computação duas vezes:

```
(lazy-take l 10)
(5 4 4 1 1 0 6 2 1 0)
```

Uma variante da lista de infinitos números aleatóreos poderia aceitar um intervalo para geração dos números:

```
;; Gera uma lista infinita de números aleatóreos. Cada
;; número estará no intervalo [a,b).
(define random-list
  (lambda (a b)
    (cons (+ a (random-integer (- b a)))
          (delay (random-list a b)))))
```

Da mesma forma que listas, é possível definir árvores e outras estruturas infinitas usando avaliação preguiçosa.

11.3 STREAMS

Há situações em que listas e vetores não são adequados para a representação de sequências de dados muito grandes ou infinitas. *Streams* são semelhantes a listas, mas da mesma forma que a lista de números aleatóreos na seção anterior, os elementos de uma *stream* somente são computados quando necessários.

A partir de poucas primitivas é possível implementar *streams* que se comportam como "listas preguiçosas". Uma stream será definida a seguir como um par (assim como uma

Versão Preliminar

lista) – mas embora o lado esquerdo do par seja tratado da mesma forma que o `car` de uma lista, o lado direito será construído sempre usando `delay`:

```
(define-syntax stream-cons
  (syntax-rules ()
    ((_ a b)
     (cons a (delay b)))))

(define stream-car car)

(define stream-cdr
  (lambda (s)
    (force (cdr s))))
```

O símbolo `empty-stream` (ou qualquer outro objeto que não possa ser resultado de `stream-cons`) pode ser usado para representar a stream vazia; um predicado `stream-null?` testa se a stream é igual a `empty-stream`:

```
(define stream-null?
  (lambda (s)
    (eqv? s 'empty-stream)))
```

Os procedimentos para listas desenvolvidos no capítulo 6 não funcionarão com streams naquela forma, mas é simples desenvolver variantes que funcionem com streams. Os exemplos a seguir mostram a implementação de `filter` e `map` para streams.

```
(define stream-filter
  (lambda (pred? s)
    (cond ((stream-null? (s))
           'empty-stream)
          ((pred? (stream-car s))
           (stream-cons (stream-car s)
                        (stream-filter pred? (stream-cdr s))))
          (else
           (stream-filter pred? (stream-cdr s))))))
```

Ao contrário de `filter`, o procedimento `stream-filter` não percorre a estrutura inteira aplicando `pred?` e selecionando elementos (se o implementássemos assim ele nunca terminaria!) O que `stream-filter` retorna é uma nova stream cujo `car` é o primeiro elemento da stream que satisfaça `pred?` (veja o segundo caso do `cond`) e cujo `car` é uma

Versão Preliminar

promessa. A computação prometida ali é de *aplicar novamente* `stream-filter` *no resto da stream original*.

```
(define stream-map
  (lambda (f s)
    (if (stream-null? (s))
        'empty-stream
        (stream-cons (f (stream-car s))
                     (stream-map f (stream-cdr s))))))
```

o funcionamento de `stream-map` é parecido com o de `stream-filter`: o `car` é `f` aplicado ao `car` da stream original, e o `cdr` é a promessa de aplicar `f` a todos os outros elementos da stream.

Um procedimento particularmente útil é o que transforma uma stream em lista. Obviamente não é possível transformar uma stream infinita em lista, mas um procedimento que tome os n primeiros elementos de uma stream é bastante útil, e sua implementação é muito simples:

```
(define stream-take
  (lambda (s n)
    (if (positive? n)
        (cons (stream-car s)
              (stream-take (stream-cdr s)
                           (- n 1)))
        '())))
```

Na seção 1.6.1 construímos um pequeno programa que aproxima a razão áurea. Podemos encarar a série de aproximações como uma stream.

Cada vez que um novo elemento da stream deve ser calculado usaremos `next-phi`:

```
(define next-phi
  (lambda (valor)
    (+ 1 (/ 1 valor))))
```

A stream inicia com 2.0; usamos `stream-map` para determinar como o resto da stream é construída.

Versão Preliminar

```
(define phi-stream
  (lambda ()
    (define approx-stream
      (stream-cons 2.0
            (stream-map next-phi approx-stream)))
    approx-stream))
```

O procedimento `phi-stream` retorna a stream de aproximações, e podemos usar `stream-take` para extrair partes dela.

```
(define phi-approx (phi-stream))
(stream-take phi-approx 10)
(2.0
1.5
1.6666666666666665
1.6
1.625
1.6153846153846154
1.619047619047619
1.6176470588235294
1.6181818181818182
1.6179775280898876)
```

11.4 ESTRUTURAS DE DADOS COM CUSTO AMORTIZADO

(esta seção é um rascunho)

Ao implementar certas estruturas de dados pode ser vantajoso usar avaliação preguiçosa [Oka99; Oka96].

11.5 RECURSÃO NA CAUDA E PROCEDIMENTOS PREGUIÇOSOS

Discutiremos um problema da implementação ingênua de `delay` e `force`, usando o exemplo da SRFI-45.

A seguir temos um filtro para listas.

Versão Preliminar

```
(define (stream-filter p? s)
  (if (null? s) '()
      (let ((h (car s))
            (t (cdr s)))
        (if (p? h)
            (cons h (stream-filter p? t))
            (stream-filter p? t)))))
(stream-filter odd? '(2 4 5 6 7 1 0))
(5 7 1)
```

A versão preguiçosa deste filtro pode ser obtida fazendo a seguinte transformação:

- Use `delay` sempre que um objeto é construído (por exemplo, ao redor de `'()` e de `cons`).
- Use `force` em argumentos de procedimentos que "desconstróem", como `car`, `cdr`, `null?`.
- Use `(delay (force ...))` no início do procedimento.

```
(define (stream-filter p? s)
  (delay (force
          (if (null? (force s)) (delay '())
              (let ((h (car (force s)))
                    (t (cdr (force s))))
                (if (p? h)
                    (delay (cons h (stream-filter p? t)))
                    (stream-filter p? t)))))))
(stream-filter odd? '(2 4 5 6 7 1 0))
#<promise>

(force (stream-filter odd? '(2 4 5 6 7 1 0)))
(5 . #<promise>)
```

Agora faremos um experimento.

Versão Preliminar

O procedimento `from` constrói uma lista infinita preguiçosa, contendo `(n n+1 n+2 ...)`

```
(define (from n)
  (delay (cons n (from (+ n 1)))))
(from 10)
#<promise>
(force (from 10))
(10 . #<promise>)
(force (cdr (force (from 10))))
(11 . #<promise>)
(force (cdr (force (cdr (force (from 10))))))
(12 . #<promise>)
```

- Construimos uma lista infinita começando do zero, `(from 0)`;
- Usamos `stream-filter` para obter uma lista com somente um elemento, muito grande (por exemplo, 1000000000);
- Usamos `force` para forçar a computação desde elemento, e `car` para obtê-lo:

```
(define large-number 1000000000)

(car (force (stream-filter (lambda (n) (= n large-number))
                           (from 0))))
```

O código executado não será recursivo na cauda, e uma grande quantidade de memória será usada. Para perceber o que acontece, definimos um laço simples usando a construção "`(delay force ...)`":

```
(define (loop) (delay (force (loop))))
(loop)
#<promise>
```

Olhamos agora para o que acontece quando forçamos esta promessa com `(force (loop))`

```
(force (loop)) =
  mude promessa1: (delay (force (loop)))
       c/valor (force (LOOP))
  retorne valor em promessa1
```

Agora forçamos o loop interno (marcado com maiúsculas):

versão preliminar

```
(force (loop)) =
  atualize promessa1: (delay (force (loop)))
      c/valor [ atualize promessa2 : (delay (force (loop)))
                      c/valor (force (LOOP))
                retorne valor em promessa2 ]
 retorne valor em promessa1
```

Novamente,

```
(force (loop)) =
  atualize promessa1: (delay (force (loop)))
      c/valor [ atualize promessa2 : (delay (force (loop)))
                      c/valor [ atualize promessa3 : (delay (force (loop)))
                                      c/valor (force (LOOP))
                                retorne valor em promessa3 ]
                retorne valor em promessa2 ]
  retorne valor em promessa1
```

A cada iteração, uma nova promessa é criada, e fica pendente.

A forma especial R^{7}RS `delay-force` pode ser usada no lugar de `(delay (force ...))`, garantindo que as chamadas a force serão recursivas na cauda, e que o uso de memória será limitado:

```
(define (stream-filter p? s)
  (delay-force                              ;; <== delay-force aqui!
     (if (null? (force s)) (delay '())
         (let ((h (car (force s)))
               (t (cdr (force s))))
           (if (p? h)
               (delay (cons h (stream-filter p? t)))
               (stream-filter p? t))))))
```

EXERCÍCIOS

Ex. 152 — A macro `depois`, definida na seção 11.1.1, foi modificada para que o valor de expressões não fossem computados novamente a cada chamada de `agora`. É possível manter a versão original de `depois` e alterar o procedimento `agora` para que ele se lembre do resultado e não calcule a expressão duas vezes? Quão difícil seria fazê-lo?

Ex. 153 — Implemente versões dos procedimentos do Capítulo 6 para streams.

Ex. 154 — A avaliação preguiçosa usando `delay` e `force` tem um problema: não se pode usar um combinador para estruturas preguiçosas e não preguiçosas (em outras palavras, a preguiça implementada como biblioteca implica em uso de tipos diferentes para estruturas preguiçosas). Elabore mais sobre este tema.

Versão Preliminar

Ex. 155 — Mostre como implementar o procedimento `stream-ref`, que retorna o n-ésimo elemento de uma stream. Faça duas versões: uma usando `stream-take` e outra não.

Ex. 156 — Construa uma versão de `delay` que aceite múltiplos argumentos.

Versão Preliminar

Versão Preliminar

12 PROGRAMAÇÃO EM LÓGICA

Da mesma forma como funções são um conceito fundamental no estilo funcional de programação e as classes e objetos são fundamentais na programação orientada a objetos, *relações entre objetos* são o fundamento da programação em Lógica.

No paradigma da programação em Lógica (e falamos de um "paradigma" idealizado, mas que sempre se paresenta de forma híbrida na prática – veja o comentário na página 4, Capítulo Paradigmas?), um programa determina *relações entre objetos*. Estas relações são descritas de duas maneiras: há *fatos*, que determinam diretamente que uma determinada relação é válida para alguns objetos, e *regras*, que permitem deduzir fatos a partir de outros.

Um exemplo simples é o problema das torres de Hanói. Um programa que resolva este problema usando linguagem imperativa deve explicitar quais passos devem ser tomados para mover os discos de uma haste a outra. Se usarmos programação em Lógica, descreveremos relações lógicas, e pediremos ao programa que *prove que é possível mover os discos de uma para outra haste*. Para provar que é possível, nosso programa deverá produzir as instruções necessárias para realizar a tarefa – e esta instruções serão a prova de que é possível fazê-lo.

12.1 DEDUÇÃO COM PROPOSIÇÕES SIMPLES

Antes de tratar de relações entre objetos será útil uma pequena discussão sobre como mecanizar inferência lógica. No próximo exemplo temos oito sentenças, `a`, ..., `h`; três regras afirmando relação de implicação lógica entre as sentenças e três fatos (os fatos são uma lista de sentenças que declaramos serem verdadeiras – no exemplo a seguir, são "a", "c" e "h").

`a`: Paulo é violinista
`b`: Paulo gosta de música
`c`: violinistas gostam de música
`d`: poetas gostam de ler
`e`: Paulo gosta de ler
`f`: Luiza é poeta
`g`: Luiza gosta de ler

Versão Preliminar

`h`: quem gosta de música também gosta de ler.
Regra: `a`, `c` $\rightarrow$ `b`.
Regra: `d`, `f` $\rightarrow$ `g`.
Regra: `h`, `b` $\rightarrow$ `e`.
Fato: `a`.
Fato: `c`.
Fato: `h`.

As primeiras oito linhas acima são uma codificação das sentenças em variáveis (demos a cada sentença um "nome"). Depois delas há três implicações, que chamamos de regras, e três fatos.

Gostaríamos de poder especificar apenas estas proposições, perguntar se Paulo gosta de ler, e obter como saída de nosso programa o raciocínio:

```
Fato: violinistas gostam de música
Fato: Paulo é violinista
Temos que: Paulo gosta de música porque [Paulo é violinista E violinistas
gostam de música]
Fato: quem gosta de música também gosta de ler
Temos que: Paulo gosta de ler porque [quem gosta de música também gosta de
ler E Paulo gosta de música]
```

Começamos com a codificação de sentenças. Podemos usar uma simples lista de associações.

```
(define cod-sentences
          '((a "Paulo é violinista")
            (b "Paulo gosta de música")
            (c "violinistas gostam de música")
            (d "Poetas gostam de ler")
            (e "Paulo gosta de ler")
            (f "Luiza é poeta")
            (g "Luiza gosta de ler")
            (h "quem gosta de música também gosta de ler")))
```

Quando a sentença for encontrada, queremos somente o texto dela (e portanto precisamos do `cadr` do que `assq` retornar quando realizarmos a busca). O procedimento `explain-proposition` extrai o texto de uma sentença codificada.

```
(define explain-proposition
  (lambda (p db)
    (let ((found (assq p db)))
```

Versão Preliminar

```
      (if found (cadr found) #f))))
(explain-proposition 'g cod-sentences)
"Luiza gosta de ler"
(explain-proposition 'z cod-sentences)
#f
```

Representaremos as regras como listas, onde a cabeça é o consequente e a cauda é uma lista de antecedentes. Assim, a regra $a \wedge c \to b$ será representada como `(b (a c))`.

Um fato será representado como uma regra sem antecedentes: $\to c$ significa que c "pode ser deduzido sem usar outras proposições", ou seja, é um axioma.

As regras já mostradas serão representadas então da seguinte maneira.

```
(define regras
          '((b a c)
            (g f d)
            (e h b)
            (a )
            (c )
            (h )))
```

Para conseguirmos o raciocínio que mostramos como exemplo, precisamos de um procedimento que tome a sentença desejada (em nosso exemplo, `e`), as regras, e identifique o raciocínio

i) a, c (fatos)

ii) $a \wedge c \to b$ (deduz-se b)

iii) h (fato)

iv) $b \wedge h \to e$ (deduz-se e)

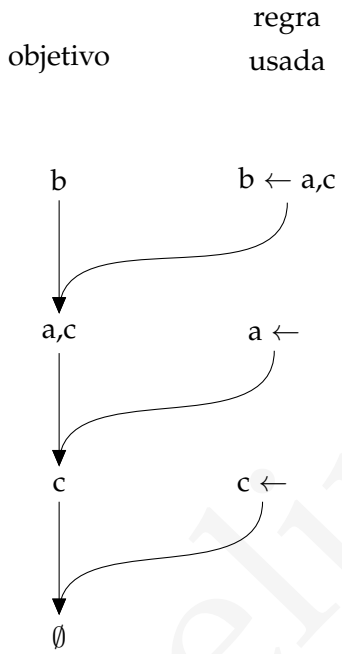
Como iniciamos com a sentença que desejamos provar (e), podemos raciocinar "para trás". O seguinte algoritmo é usado para realizar este raciocínio.

- Comece com a lista de objetivos contendo apenas p, a sentença que se quer demonstrar.
- Enquanto a lista de objetivos não estiver vazia:
  - Se p é fato, remova da lista de objetivos.
  - Se p é consequência de uma regra $a, b, \ldots \to p$, troque p por $a, b, \ldots$ na lista de objetivos.

Versão Preliminar

– Caso contrário abandone o laço e retorne `#f`.

Por exemplo, quando pedirmos ao sistema que mostre que Paulo gosta de música (proposição b), o processo realizado é ilustrado a seguir.



O objetivo começa com b. Como o objetivo é igual à cabeça da regra $b \leftarrow a, c$, trocamos b no objetivo por a, c. Em seguida, selecionamos o próximo item do objetivo, a. Como ele é igual ao consequente da regra "$a \leftarrow$" (o mesmo que o fato a), podemos trocá-lo pela cauda da regra, que é vazia – na prática, o a some do objetivo, restando apenas c. Da mesma forma, com a regra "$c \leftarrow$" podemos eliminar c do objetivo. Quando o objetivo fica vazio, terminamos e mostramos a sequência de regras usadas, na sequência reversa: $c; a; (b \leftarrow a, c); b$ – ou seja, *temos* a *e* c*, e temos também* $a, c \rightarrow b$*, portanto* b.

O procedimento `raciocinio-tras` implementa este algoritmo.

Versão Preliminar

```
(define raciocinio-tras
  (lambda (p r)
    (let loop ((x (list p))
               (exp '()))
      (if (null? x)
          exp
          (let ((goal (car x)))
            (let ((conc (assq goal r)))
              (if conc
                  (loop (append (cdr conc) (cdr x))
                        (cons conc exp))
                  #f)))))))
(raciocinio-tras 'b regras)
((c) (a) (b a c))
```

Conseguimos mecanizar o raciocínio: o procedimento nos lista dois fatos, a e c, e também nos informa que podemos concluir b a partir deles.

Precisamos agora de um procedimento que transforme o raciocínio, representado desta forma, nas frases codificadas.

Versão Preliminar

```
(define explain
  (lambda (r db)
    (define explain-and
      (lambda (lst)
        (let ((s (explain-proposition (car lst) db)))
          (if (= (length lst) 1)
              s
              (string-append s " E " (explain-and (cdr lst)))))))
    (define explain-each
      (lambda (x)
        (if (null? (cdr x))
            (string-append "Fato: "
                           (explain-proposition (car x) db) "\n")
            (string-append "Temos que: "
                           (explain-proposition (car x) db)
                           " porque [" (explain-and (cdr x))
                           "]\n"))))
    (if r
        (reduce string-append "" (reverse (map explain-each r)))
        "Nenhuma conclusão")))

(display (explain (raciocinio-tras 'b regras)
                  cod-sentences))
```

Fato: violinistas gostam de música
Fato: Paulo é violinista
Temos que: Paulo gosta de música porque [Paulo é violinista E violinistas gostam de música]

Para obter exatamente o raciocínio que dissemos que gostaríamos de obter, pedimos `'e` ao invés de `'b`:

```
(display (explain (raciocinio-tras 'e r)
                  cod-sentences))
```

Fato: Violinistas gostam de música
Fato: Paulo é violinista
Temos que: Paulo gosta de música porque [Paulo é violinista E Violinistas gostam de música]

Versão Preliminar

```
Fato: Quem gosta de música também gosta de ler
Temos que: Paulo gosta de ler porque [Quem gosta de música também gosta de
ler E Paulo gosta de música]
```

Se não for possível concluir o objetivo, a explicação será "Nenhuma conclusão". A partir dos fatos na base de dados que descrevemos é impossível concluir que "Luiza gosta de ler" – e nosso sistema de raciocínio de fato reporta "nenhuma conclusão".

```
(display (explain (raciocinio-tras 'g regras) d))
Nenhuma conclusão
```

12.2 PROLOG: DEDUÇÃO COM VARIÁVEIS

Esta Seção descreve as idéias básicas da programação em Lógica, usando a linguagem Prolog. Nas Seções subsequentes voltamos a usar Scheme para desenvolver um pequeno interpretador Prolog e alguns programas Prolog – no entanto, é necessária esta introdução à programação em Lógica antes de podermos prosseguir.

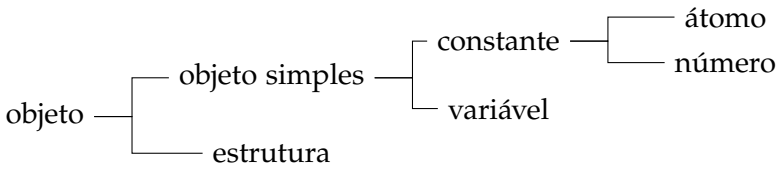
Até o momento nos restringimos à Lógica Proposicional, o que limita muito a utilidade do nosso sistema de "automação de raciocínio".

Se desenvolvermos a mesma ideia de raciocínio automático para a Lógica de Predicados, chegaremos à essência da *Programação em Lógica*. Relembramos que o ambiente Scheme recebe formas do usuário e usa `eval` para avaliar as formas (possivelmente com efeitos colaterais e mudanças de estado), e finalmente devolve ao usuário o *valor* da forma que ele passou ao ambiente Scheme.

Já um ambiente Prolog (uma linguagem que implementa Programação em Lógica) espera que o usuário dê uma expressão lógica contendo variáveis, que será interpretada como uma pergunta. O sistema Prolog tentará encontrar valores para as variáveis que tornem a expressão verdadeira e sua resposta conterá esses valores. Por exemplo, podemos perguntar ao sistema Prolog "Existe x tal que $\text{sen}(\pi) = x$?" – e a resposta será "Sim, $x = 0$". O valor de x encontrado pelo Prolog é um *exemplo* de que "existe um x tal que $x = \text{sen}(\pi)$".

Em Prolog há um único tipo de dado, chamado de *termo*. Apesar de haver um único tipo de dado, ele é subdividido em alguns subtipos: um termo pode ser um átomo, que é semelhante a um símbolo em Scheme; um número; uma variável; ou um termo composto (uma estrutura). Dizemos que há um só tipo de dados, apesar dessa subdivisão, porque não há tipo associado com variáveis. A figura a seguir mostra esta classificação.

Versão Preliminar



Átomos cujos nomes são compostos somente de letras são representados diretamente, como símbolos em Scheme: `torre` e `bispo` são átomos. Átomos com outros caracteres no nome devem ser escritos entre aspas simples: `'nome composto'` é um átomo. Como átomos podem conter qualquer tipo de caractere, cadeias de caracteres não são um tipo de dados especial em Prolog: elas são representadas como átomos.

Uma variável em Prolog é uma entidade cujo valor não é conhecido *a priori*, mas que só pode ser definido uma única vez. Nisso Prolog assemelha-se a uma linguagem de programação funcional pura, onde não se permite mutação de variáveis[1]. Uma vez que a variável X tenha sido vinculado ao valor 5, esse permanece sendo seu valor: não há como modificá-lo.

Em Scheme, usamos registros para que representar variáveis que são naturalmente agregadas, e para representar hierarquia. Neste texto, antes de usarmos registros usamos listas rotuladas (veja a Seção 8.7). Em Prolog usamos *estruturas*. Uma estrutura é representada em Prolog usando a notação de função: filmes poderia ser descrito pela estrutura Prolog `filme`, onde as partes da estrutura representam título, diretor, ano de produção, situação (na estante ou emprestado) e local atual (em qual de nossas estantes ele está, ou para quem emprestamos).

```
filme('The Royal Tenenbaums',
      'Wes Anderson',
      2001,
      comedia,
      estante,
      b3).
filme('Sweeney Todd',
      'Tim Burton',
      2007,
      musical,
      emprestado,
      'Fulano de Tal').
```

1 O exemplo proeminente de uma tal linguagem é Haskell. Scheme, se restrita a procedimentos sem mutação de variáveis (como no Capítulo 1), também exemplifica o conceito.

Versão Preliminar

Não há método especial para criar estruturas, como o procedimento para criar estruturas (o `maker` na macro `define-structure` – veja a página 255). Simplesmente usamos a estrutura.

É importante observar que em Prolog, a expressão `cos(0)` *não* é a aplicação da função cosseno ao número zero. É uma estrutura de nome `cos`. É necessário forçar o ambiente Prolog a interpretar uma estrutura como se fosse uma expressão matemática, se o quisermos. O mesmo vale para "`3*(A + 2)`", que é uma simplificação sintática para "`*(3,+(A,2))`", representando uma estrutura que *pode* ser avaliada como expressão matemática se necessário.

Termos são objetos sem valor booleano: as variáveis e estruturas. A definição de termo é dada a seguir.

- Uma variável é um termo.
- $f(t_1, t_2, \ldots, t_k)$ é um termo, desde que f seja um símbolo de função e todos os t_i sejam termos.

Em Prolog podemos expressar *relações* entre objetos, que também podem ser chamadas de *predicado*.

Um *objetivo* pode ser visto como um objeto com valor booleano, que usamos em inferência lógica. Objetivos são átomos ou termos compostos[2].

Uma *pergunta* é uma sequência finita de objetivos.

Uma *cláusula* ou *regra* é uma implicação da forma

$$p(x_1, x_2, \ldots, x_n) \leftarrow A_1, A_2, \ldots A_k$$

onde os A_i são objetivos.

Um *programa* Prolog é um conjunto finito de cláusulas. Em sistemas Prolog, a implicação ("$\leftarrow$") é representada pelo símbolo `:-`.

Por exemplo, a seguir temos a declaração de alguns fatos a respeito de filmes e duas regras que expressam a relação "leve": um filme é seguramente considerado "leve" se for comédia ou musical – em outros casos não podemos ter certeza[3].

2 Na literatura de programação em Lógica (não especificamente da linguagem Prolog), o termo "átomo" muitas vezes é usado para o que chamamos de "objetivo".

3 Este é um exemplo simplificado, apenas para ilustrar como programas Prolog funcionam. De fato, o trabalho de Stanley Kubrik é evidência clara de que não há como isolar filmes em gêneros da forma como fizemos.

Versão Preliminar

```
drama('Blood and Bones').          /* (1) */
horror('O Iluminado').             /* (2) */
comedia('Dr Strangelove').         /* (3) */
comedia('O Corte').                /* (4) */
musical('Sweeney Todd').           /* (5) */

japones('Blood and Bones').
coreano('Blood and Bones').
americano('O Iluminado').
americano('Dr Strangelove').
frances('O Corte').
americano('Sweeney Todd').

leve(F) ← comedia(F).
leve(F) ← musical(F).
```

Tendo codificado esta pequena base de dados a respeito de filmes junto com a relação `leve`, poderemos perguntar ao sistema Prolog se há algum filme americano e que seja "leve":

```
?- leve(X), americano(X)
X = 'Dr Strangelove'
yes
```

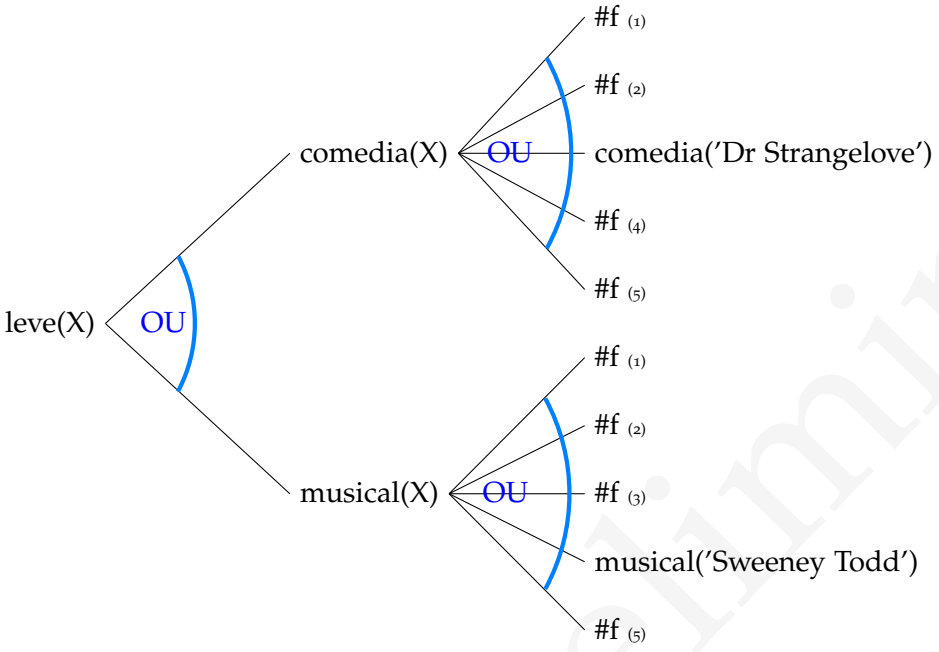
Podemos pedir, inclusive, que o interpretador Prolog nos dê outra resposta[4]:

```
?- leve(X)
X = 'Dr Strangelove'?
;
X = 'O Corte'?
yes
;
X = 'Sweeney Todd'?
yes
```

4 É tradicional que sistemas Prolog permitam que o usuário peça mais respostas digitando um caractere ponto-e-vírgula.

Versão Preliminar

Para chegar a esta resposta, o interpretador Prolog iniciou com a pergunta, `leve(X)` e tentou trocar X por `comedia(X)`; depois, unificou X com 'Dr Strangelove', chegando à primeira solução.

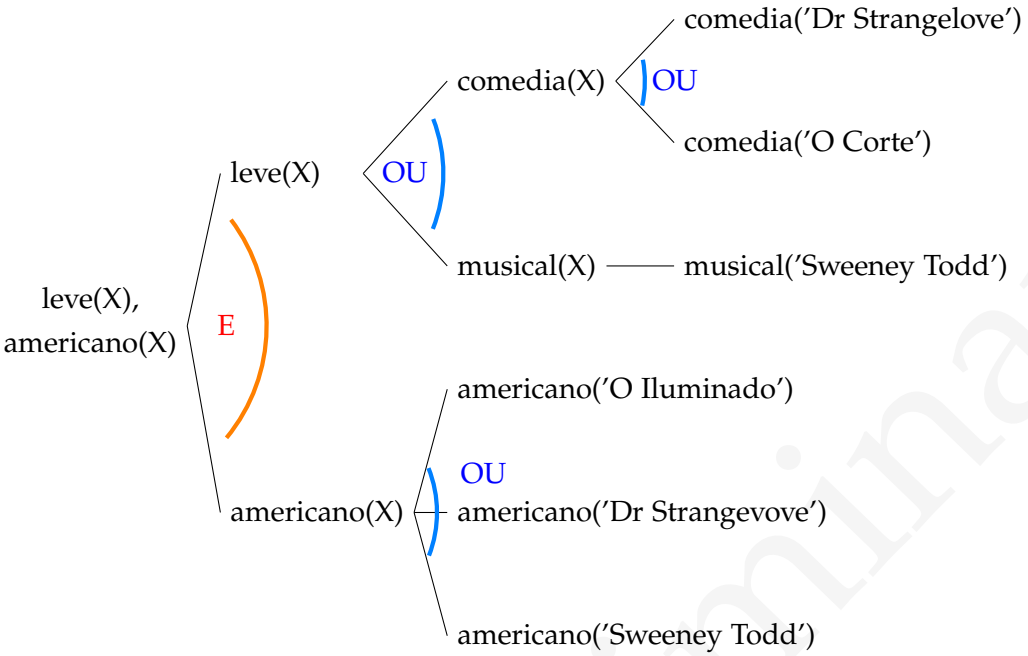


Se pedirmos ao interpretador Prolog para satisfazer *dois* objetivos, a árvode de busca passará a ter dois dipos de bifurcação: as to tipo "ou" e as do tipo "e"[5]. Por exemplo, podemos perguntar a respeito de filmes leves e que também sejam americanos:

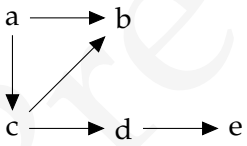
```
leve(X), americano(X)
```

5 *OR-branches* e *AND-branches* em Inglês.

Versão Preliminar



Damos agora outro exemplo. Observe o grafo a seguir.



Este grafo pode ser representado em Prolog simplesmente listando a relação que define suas arestas. Daremos a esta relação o nome de `edge`.

```
edge(a,b).
edge(a,c).
edge(c,b).
edge(c,d).
edge(d,e).
```

Um nó Y é alcançavel a partir de outro nó X se há uma aresta que os liga diretamente ou se há uma aresta de X a algum nó intermediário I, que por sua vez alcança Y. Esta definição pode ser traduzida diretamente em Prolog.

```
reach(X,Y) ← edge(X,Y).
reach(X,Y) ← edge(X,I),
             reach(I,Y).
```


Versão Preliminar

Este programa nos permite perguntar ao sistema Prolog se dois nós do grafo são alcançáveis:

```
reach(a,b)
yes
reach(c,e)
yes
reach(d,b)
no
```

Damos outro exemplo. A definição de fatorial em Prolog pode ser descrita pelas seguintes duas cláusulas[6]

```
fat(N,1) ← N < 2.
fat(N,X) ← N' is N-1,
           fat(N',X'),
           X is N · X'.
```

Usamos a notação "A is B" para indicar que a variável A deve ter o valor igual a B.

Estas linhas dizem ao interpretador Prolog que:

- Se $N < 2$, o fatorial de N é um.
- Em outros casos,
  - Seja $N' = N - 1$,
  - o fatorial de N' é X',
  - seja $X = N \cdot X'$,

  então o fatorial de N é X.

Este programa coincide com a descrição da definição recursiva da função fatorial (exceto que computa fatorial de números negativos, retornando -1).

6 Em uma implementação real de Prolog, teríamos que trocar ← por `:-` e os nomes N' e X' por NN e XX, e o programa seria codificado da seguinte maneira:

```
fat(N,1) :- N < 2.
fat(N,X) :- NN is N-1,
            fat(NN,XX),
            X is N * XX.
```

Notamos que não é usual calcular o fatorial de um número diretamente – usa-se, ao invés disso, o logaritmo da função Γ.

Versão Preliminar

12.2.1 Modelo de execução

Apresentamos dois modelos de execução para a linguagem Scheme: um na Seção 1.3.3 (página 14), sem a previsão de mutação de variáveis, e outro na Seção 3.2 (página 104), onde incluímos ambientes e mutação. Apresentamos também aqui um modelo de execução para Prolog. Na Seção 12.3 desenvolveremos um interpretador Prolog baseado nesse modelo de execução.

Em nosso modelo de execução prsumiremos que há um programa Prolog (uma sequência de cláusulas) e uma pergunta (uma sequência de objetivos). O resultado da execução será uma substituição que torne a pergunta verdadeira ou *falso* caso a substituição não possa ser encontrada.

Neste algoritmo,

- P é a pergunta.
- R é o resolvente (a lista de objetivos que ainda devem ser satisfeitos). Cada iteração do laço teta remover um objetivo da lista, mas pode adicionar outros.
- Θ é a substituição que está sendo gradualmente construída (cada itaração do laço pode aumentar a substituição).
- θ é a substituição encontrada em uma única iteração do laço.
- $A \leftarrow a_1, a_2, \ldots$ é uma cláusula do programa.

```
R ← P
loop(Θ, R):
    se R = {} retorne Θ
    senao:
        escolha o próximo objetivo g de R
        escolha uma cláusula A ← a1, a2, ...,
            tal que A, g unificam com mgu θ
        se a cláusula não existe, retorne falso

        // temos agora θ, uma substituição e
        // a1, a2, ..., a cauda de A.

        seja A' ← a1', a2', ... a cláusula A renomeada
        troque g por a1', a2', ... no resolvente
        aplique θ em Θ
        loop(θ ∪ Θ, R)
```


Versão Preliminar

Cada cláusula do programa escolhida para uso no algoritmo é *renomeada*: todos os átomos tem seus nomes trocados por outros, cujos nomes nunca foram usados antes nesta computação. Mais adiante esclareceremos o motivo disso.

Em programação em Lógica, os objetivos e cláusulas são escolhidos não-deterministicamente, sem ordem definida. Já em Prolog os objetivos são escolhidos da esquerda para a direita na pergunta, e as cláusulas do programa também são escolhidas na ordem em que aparecem.

Exemplificamos o modelo de execução detalhando a maneira como um interpretador Prolog responderia as perguntas que demos de exemplo na Seção 12.2 ("Prolog: Dedução com variáveis").

Para o primeiro exemplo, usamos a base de dados da página 348. Analisamos agora cada passo da computação realizada por um interpretador prolog ao processar a pergunta `leve(X), americano(X)`. Os passos são detalhados na próxima tabela.

resolvente	Θ	
`leve(X), americano(X)`	{}	
`comedia(X), americano(X)`	{}	cauda de `leve(x)` ← `comedia(X)`
`comedia('Dr Strangelove'), americano('Dr Strangelove'')`	{X →`'Dr Strangelove'`}	unificando X com `'Dr Strangelove'`
`americano('Dr StrangeLove')`	{X →`'Dr Strangelove'`}	`comedia('Dr Strangelove')` satisfeito
{}	{X →`'Dr Strangelove'`}	`americano('Dr Strangelove')` satisfeito

Cada linha da tabela é detalhada a seguir.

linha 1 Determina que o objetivo é, inicialmente, a lista ⟨`leve(X), americano(X)`⟩.

linha 2 Toma o primeiro objetivo do resolvente, `leve(X)` e verifica se ele unifica com a cabeça de alguma regra. Encontra a regra "`leve(F)` ← `comedia(F)`", e modifica o objetivo para "comedia(X), americano(X)". A substituição encontrada, F → X, é aplicada tanto no objetivo como na substituição anterior.

linha 3 Procurando novamente satisfazer o primeiro objetivo, o interpretador tentará usar a cláusula `comedia('Dr Strangelove'')`, e X será unificada com `'Dr Strangelove'`. Nesta linha a substituição já foi aplicada no resolvente.

linha 4 Depois tenta satisfazer o (novo) primeiro objetivo do resolvente, "comedia(X)", encontrando `comedia('Dr Strangelove')`. Como esta regra não tem cauda, este objetivo é removido do resolvente, que passa a ser `americano('Dr StrangeLove')`. A substituição X →`'Dr Strangelove'` é aplicada no resolvente e em Θ, e depois adicionada a Θ.

Versão Preliminar

linha 5 Como o único objetivo no resolvente unifica com um fato, ele é removido.

linha 5 O interpretador retorna $\Theta = \{X \rightarrow$ `'Dr Strangelove'` $\}$.

Usando agora os predicados `reach` e `edge` para verificar a alcançabilidade em grafos, construimos outro exemplo. Supona que a pergunta seja `reach(a,d)`.

objetivo	Θ	
`reach(a,d)`	{}	(1)
`edge(a,d)`	{}	usando `reach(X,Y)` ← `edge(X,Y)`
`edge(a,d)`	{}	falha, retorna a **(1)**
`edge(a,I), reach(I,d)`	{}	usando `reach(X,Y)` ← `edge(X,I), ...`
`edge(a,b), reach(b,d)`	{I → b}	usando o fato `edge(a,b)` **(2)**
`reach(b,d)`		`edge(a,b)` satisfeito
`edge(b,d)`		falha, retorne a (2)
`edge(a,c), reach(c,d)`	{I → c}	usando o fato `edge(a,c)`
`reach(c,d)`	{}	`edge(a,c)` satisfeito
`edge(c,d)`	{}	usando `reach(X,Y)` ← `edge(X,Y)`
{}	{}	`edge(c,d)` satisfeito

A substituição I → ... não envolve variáveis presentes na pergunta, por isso a descartamos assim que não é mais necessária.

Na primeira falha, o interpretador desiste de satisfazer `reach(a,d)` usando a cláusula `reach(X,Y)` ← `edge(X,Y)`, para tentar usar a segunda cláusula do predicado `reach`. A linha seguinte mostra que o interpretador unificou X com a e Y com d, e a cauda da regra foi inserida no resolvente.

12.3 IMPLEMENTANDO PROGRAMAÇÃO EM LÓGICA

Nesta Seção implementaremos um interpretador Prolog. Começaremos com uma versão simples do interpretador de Prolog puro, e faremos modificações até chegar a um interpretador completo (mas muito pouco eficiente) para Prolog.

Antes de mais nada estabelecemos a representação de cláusulas e do resolvente.

Um objetivo será representado por uma lista: `(f 1 2 ?a)` representa o objetivo $f(1, 2, A)$.

Uma cláusula ou regra $A \leftarrow A_1, A_2, \ldots, A_k$ é representada como uma lista de objetivos `(a a1 a2 ... ak)` onde o primeiro objetivo representa a cabeça da regra (a conclusão) e os outros constituem os antecedentes (a cauda). Podemos então simplesmente usar `car` e `cdr`.

Versão Preliminar

```
(define rule-head car)
(define rule-tail cdr)
```

O resolvente será uma lista de objetivos. Criamos uma camada de abstração que encapsule o método de escolha do próximo objetivo: teremos procedimentos `select-goal` e `next-goals`. Como escolheremos sempre o próximo da esquerda para a direita, estes são `car` e `cdr`.

```
(define select-goal car)
(define next-goals cdr)
```

Como precisamos renomear as variáveis das cláusulas do programa, construímos dois procedimentos: `rename-one`, que renomeia uma única variável, adicionando a ela um rótulo, e `rename-vars`, que renomeia todas as variáveis em uma estrutura.

```
(define rename-one
  (lambda (var tag)
    (string->symbol
     (string-append
      (symbol->string var) "_"  (number->string tag)))))


(define rename-vars
  (lambda (vars tag)
    (cond ((pair? vars)
           (cons (rename-vars (car vars) tag)
                 (rename-vars (cdr vars) tag)))
          ((matching-symbol? vars)
           (rename-one vars tag))
          (else vars))))
```

Será necessário um procedimento que, a partir de um objetivo e uma regra, nos retorne o *mgu* do objetivo com a cabeça da regra e a cauda da regra. Quando o objetivo é a lista vazia não há como escolher uma regra, e o procedimento retorna `#f`. Quando a unificação é possível, retorna uma lista com a substituição e a cauda da regra. Quando não é possível, retorna `#f`.

Versão Preliminar

```
(define unifying-clause
  (lambda (a rule)
    (if (null? rule)
        #f
        (let ((sub (unify a (rule-head rule))))
          (if sub
              (list sub (rule-tail rule))
              #f)))))
```

O resultado de `unifying-clause` é uma lista com a substituição e a cauda da regra. Por exemplo, se há no programa uma regra `f(X,Y) ← g(X), a, h(Y).` o objetivo é $f(2, B)$, então `(unifying-clause obj rule)` retorna uma lista com dois elementos: o primeiro é a substituição $\{X \to 2, B \to Y\}$ e o segundo é a cauda da regra, "$g(X), a, h(Y)$". Usando a representação em Scheme, a lista é

```
( ((?x 2) (?b ?y))   ((g ?x) (h ?y) a) )
```

Para tornar o programa mais claro, criamos procedimentos para extrair substituição e cauda dessa lista.

```
(define sub/tail->tail cadr)
(define sub/tail->sub car)
```

Quando dermos a resposta final ao usuário, será interessante mostrar apenas a substituição das variáveis que constavam na pergunta original, e não aquelas feitas como parte da computação. O procedimento `select-sub` filtra[7]. a substituição, mantendo apenas as trocas $A \to B$ onde A ocorre no *goal*.

```
(define select-sub
  (lambda (sub goal)
    (define relevant?
      (lambda (s)
        (occurs? (car s) goal '())))
    (filter relevant? sub)))
```

7 O procedimento `filter` foi definido na página 191.

Versão Preliminar

Usaremos o procedimento `amb+-list` (descrito na Seção 10.8.2), porque mais adiante manipularemos diretamente a pilha de continuações[8]. No entanto, o procedimento `amb+-fail` sempre termina com erro quando não há mais opções, e não queremos que nosso interpretador Prolog termine a execução com um erro somente porque não conseguiu satisfazer o objetivo. Modificaremos `amb+-fail` para retornar `#f` quando não houver mais opções:

```
(define (amb+-fail)
  (if (null? amb+-stack)
      #f
      (let ((back (car amb+-stack)))
        (amb+-pop)
        (back back))))
```

O procedimento `resolve` recebe o resolvente, a substituição determinada até o momento, um rótulo que determina o que adicionar às variáveis quando renomeá-las, o objetivo e o programa. Em seguida selecionará o próximo objetivo e tentará unificá-lo com a cabeça de alguma regra. Quando conseguir, aplicará a nova substituição sobre a cauda da regra, sobre a substituição anterior e sobre o resolvente anterior. Depois retornará a nova substituição e o novo resolvente.

8 A macro `amb` não nos daria uma pilha explícita. Além disso, não há prejuízo em usar `amb+`, já que os argumentos são apenas listas representando cláusulas, e não computações que possam demorar para serem avaliadas.

Versão Preliminar

```
(define resolve
  (lambda (res sub tag goal prog)
    (if (null? res)
        (list (select-sub sub goal) (apply-sub sub res))
        ;; Escolha não-deterministicamente uma cláusula:
        (let ((rule (rename-vars (amb+-list prog) tag)))
          (if rule
              (begin
                ;; A cabeça da cláusula deve unificar com o objetivo:
                (require+ (unify (select-goal res) (rule-head rule)))
                ;; Obtenha a substituição e cauda, aplique a
                ;; substituição e retorne o resultado:
                (let ((sub+tail
                       (unifying-clause (select-goal res) rule)))
                  (let ((tail  (sub/tail->tail sub+tail))
                        (theta (sub/tail->sub sub+tail)))
                    (let ((new-sub (append theta (apply-sub theta sub))))
                      (let ((new-res
                             (apply-sub theta
                                        (append tail
                                                (next-goals res)))))
                        (list new-sub new-res))))))
              #f)))))
```

Criamos também uma camada de abstração para extrair a substituição e o resolvente da lista retornada por `resolve`.

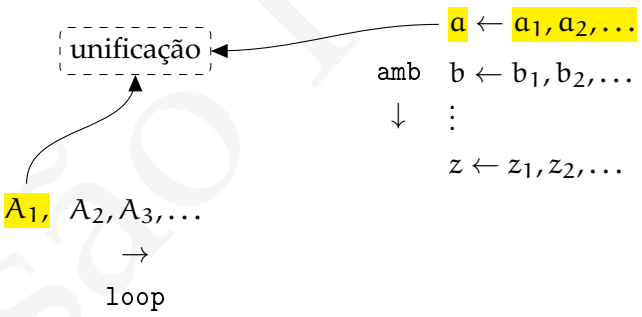
```
(define resolved->sub car)
(define resolved->res cadr)
```

Nosso interpretador Prolog recebe um programa e um objetivo. Ele percorre todos os objetivos, tentando resolvê-los. Quando não houver mais objetivos para resolver, retorna a substituição corrente.

Versão Preliminar

```
(define pure-prolog
  (lambda (prog goal)
    (amb+-reset)
    (let loop ((res goal)
               (sub '())
               (tag 1))
      (if (null? res)
          (select-sub sub goal)
          (let ((resolved (resolve res sub tag goal prog)))
            ;; Se este objetivo foi resolvido, passe para
            ;; o próximo. Senão, retorne falso
            (if resolved
                (loop (resolved->res resolved)
                      (resolved->sub resolved)
                      (+ tag 1))
                #f))))))
```

O procedimento `amb-reset+` é chamado logo no início de `pure-prolog`, para eliminar qualquer resíduo de chamadas anteriores a `amb+`. O laço "`loop`" itera sobre os objetivos do resolvente, mantendo três variáveis: `res` é o resolvente atual; `sub` é a substituição obtida até o momento; e `tag` é o rótulo a ser usado na próxima vez que uma cláusula for renomeada.

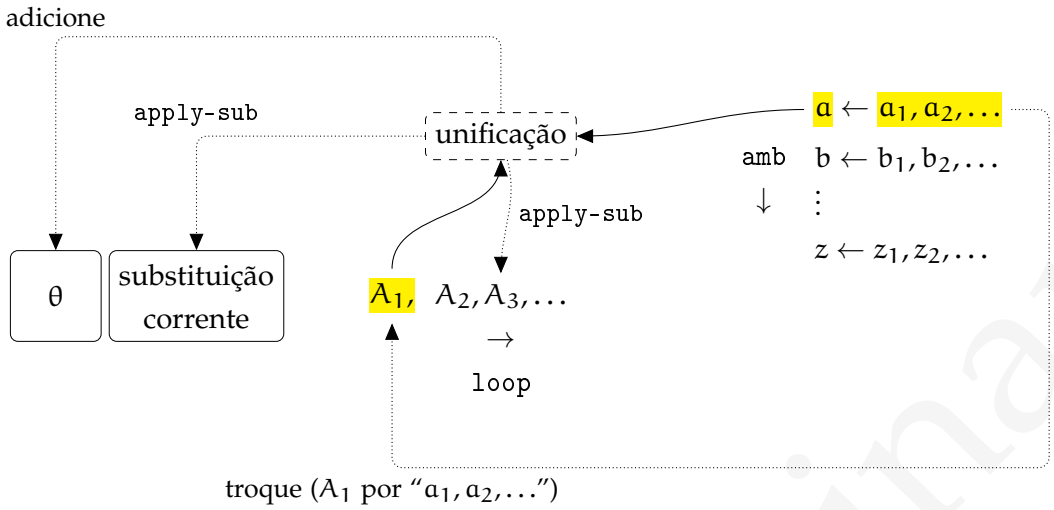


Quando `res` é a lista vazia, não há mais objetivos a resolver, e a substituição atual é suficiente – o Prolog então calcula a parte da substituição que é relevante para a pergunta original e a devolve.

Quando ainda há objetivos no resolvente, nosso Prolog chama `resolve`, que devolverá a nova substituição e o novo resolvente – e em seguida retorna para uma nova iteração de `loop`.

A próxima figura ilustra o funcionamento do interpretador Prolog.

Versão Preliminar



Usaremos o seguinte programa Prolog como exemplo.

```
f(0).
f(Z) ← g(Z).
h(3).
h(4).
q(4).
p(Z,Y,S) ← f(Z), g(Y), h(S).
g(10).
```

A tradução do programa para nosso Prolog-em-Scheme é dada a seguir.

```
(define prolog-example
  '(( (f 0) )
    ( (f ?z) (g ?z))
    ( (h 3) )
    ( (h 4) )
    ( (q 4) )
    ( (p ?z ?y ?s) (f ?z) (g ?y) (h ?s))
    ( (g 10) )))
```

A pergunta que faremos é `p(10,D,A), q(A).`

```
(pure-prolog prolog-example '( (p 10 ?d ?a) (q ?a) ))
((?d . 10) (?a . 4))
```

O Prolog nos respondeu com a substituição $\{\mathtt{D} \to 10, \mathtt{A} \to 4\}$.

Versão Preliminar

12.3.1 Prolog com funções Scheme

O Prolog puro que desenvolvemos tem utilidade prática muito limitada:

- A entrada e saída de dados fica restrita a perguntas na forma de termos e respostas, apresentadas como substituições;
- Não podemos realizar operações em números e cadeias. Os valores usados pelo Prolog são apenas símbolos, e a única coisa que podemos fazer com eles é unificá-los com variáveis.

Nesta Seção desenvolveremos para nosso Prolog uma interface para predicados Scheme. Poderemos usar quaisquer procedimentos booleanos Scheme no lugar de predicados Prolog. Desta forma teremos disponíveis `display` e diversos outros procedimentos Scheme.

Nossos objetivos Prolog são representados como símbolos ou como listas Scheme. Usaremos listas-de-listas para representar chamadas a procedimentos Scheme.

```
((f ?a ?b) (g ?a) ((display ?b)))
```

Neste exemplo, `(g ?a)` é um termo que o Prolog deve resolver; já `((display ?b))` é uma chamada a um procedimento Scheme. O procedimento `display` terá o efeito colateral de mostrar a variável `?b` quando for "executado" como objetivo, e *seu valor de retorno será interpretado como booleano, sendo esse o valor-verdade do predicado*.

O interpretador precisa verificar o primeiro elemento do átomo para decidir se é um objetivo comum ou se é uma chamada a procedimento Scheme. Quando o `car` do átomo for uma lista, ele é chamada externa (por exemplo, "`((newline))`"). Para isso criaremos o procedimento `atom-start`, que determina o primeiro elemento de um átomo (e que será o mesmo que `car`).

```
(define atom-start car)
```

O interpretador Prolog com interface para Scheme é listado a seguir.

Versão Preliminar

```
(define prolog+scheme
  (lambda (prog goal)
    (amb+-reset)
    (let loop ((res goal)
               (sub '())
               (tag 1))
      (cond ((null? res)
             (select-sub sub goal))
            ((list? (atom-start (car res)))
             (let ((truth (eval (apply-sub sub
                                           (atom-start (car res))))))
               (require+ truth)
               (if truth
                   (loop (cdr res) sub tag)
                   #f)))
            (else (let ((resolved (resolve res sub tag goal prog)))
                    (if resolved
                        (loop (resolved->res resolved)
                              (resolved->sub resolved)
                              (+ tag 1))
                        #f)))))))

(define prolog-family '(((father john johnny))
                        ((father old-john john))))

(prolog+scheme prolog-family '( (grandpa old-john ?who)
                                ((display "Grandson is: " ?who)) ))
```

```
Grandson is: johnny
((?who . johnny))
```

É necessário lembrar, no entanto, que procedimentos de entrada e saída em Scheme não tem valor de retorno definido, e de acordo com o padrão da linguagem a implementação pode escolher qualquer valor de retorno – inclusive `#f`. Assim, se no exemplo anterior `display` retornar `#f`, o predicado falhará (pode-se remediar este problema usando uma função ao redor de display: `(lambda args (aplly display args) #t)`).

Versão Preliminar

No exemplo a seguir, a forma Scheme `(= 5 5)` retorna `#t`, e portanto o predicado `((= 5 5))` terá valor verdadeiro (como o predicado é verdadeiro independente de outras cláusulas ou fatos, podemos usá-lo com o programa vazio).

```
(prolog+scheme '( )
               '( ((= 5 5)) ))
```
```
()
```

Se uma forma externa retornar `#f`, o predicado falha.

```
(prolog+scheme '( )
               '( ((= 2 5)) ))
```
```
#f
```

12.3.2 Instanciando variáveis temporárias

O seguinte programa Prolog deveria calcular o dobro de um número. No entanto, quando tentamos usá-lo para calcular o dobro de 3, nos deparamos com um erro:

```
(define dobro '(((dobro ?x ?y)
                 ((= ?y (* 2 ?x)))
                 ((display "O dobro de " ?x " é " ?y)))))
(prolog+cut dobro '((dobro 3 ?a)))
```
```
Error: unbound variable: ?y_1
```

Analisamos então o motivo do erro, verificando os passos do interpretador até tentar resolver `((= ?y (* 2 ?x))`. A cláusula teve suas variáveis renomeadas:

```
(dobro ?x_1 ?y_1)
 ((= ?y_1 (* 2 ?x_1)))
 ((display "O dobro de " ?x " é " ?y_1))
```

e sua cauda incluída no resolvente. Quando `((= ?y_1 (* 2 ?x_1)))` foi selecionado, a substituição corrente não tinha valor para `?y_1` – *esta variável nunca havia sido unificada com outro termo, porque não estava na cabeça de qualquer regra*. O interpretador Prolog verificou que se trata de uma lista, aplicou a substituição corrente nela (que trocou `?x_1` por 3) e

Versão Preliminar

pediu ao interpretador Scheme que avaliasse `(= ?y_1 (* 2 3))`. Como não há `?y_1` no ambiente Scheme, ocorreu um erro.

Devemos observar também que o procedimento Scheme "=" não vincula variáveis – ele apenas testa igualdade de números. No entanto, mesmo que usemos `unify`, o problema persiste:

```
(define dobro '(((dobro ?x ?y)
                 ((unify ?y (* 2 ?x)))
                 ((display "O dobro de " ?x " é " ?y)))))
(prolog+cut dobro '((dobro 3 ?a)))
Error: unbound variable: ?y_1
```

Isso porque a chamada a `unify` não cria vínculos para as variáveis dentro do nosso programa Prolog. As chamadas a Scheme nos permitem apenas:

- Obter alguns efeitos colaterais (podemos imprimir valores ou alterar *variáveis do ambiente Scheme*, por exemplo;
- Usar o valor de retorno do procedimento chamado, *que será interpretado como verdadeiro ou falso* e usado como valor-verdade.

Na linguagem Prolog o predicado padrão `is` pode ser usado para instanciar variáveis. Por exemplo, `X is (A+B)/2` avaliará `(A+B)/2` e instanciará X com o resultado, se estes forem números (se não forem, o sistema reportará um erro). O predicado `is` exige que quaisquer variáveis do lado direito estejam instanciadas – de outra forma não há como instanciar X.

Criaremos um predicado `is` que instancia variáveis: `(is ?a forma-scheme)` unificará `?a` com `forma-scheme` e aplicará a substituição resultante no resolvente e na substituição corrente.

O predicado `is` é semelhante a atribuições de valores a variáveis em outras linguagens, e é tradicional, para estas construções, usar os nomes *lvalue* para o lado esquerdo e *rvalue* para o lado direito. Para acessarmos a expressão em um predicado `is` usaremos `is->rvalue`.

```
(define is->rvalue cadr)
(is->rvalue (?x (sqrt (* 2 x))))
(sqrt (* 2 x))
```


Versão Preliminar

Para exigirmos que a unificação em `is` tenha sucesso, usamos `(let ((s (unify ...))) (require+ s) ...`

```
(define do-is
  (lambda (res sub)
    (let ((s (unify
              (atom-start (cdr res))
              (eval (apply-sub sub (is->rvalue (cdr res)))))))
      (require+ s)
      s)))
```

Ao encontrar um predicado `is`, o interpretador chama `do-is` para avaliar a expressão, e aplica a substituição usada para unificar a nova variável com seu valor.

```
(define prolog+is
  (lambda (prog goal)
    (amb+-reset)
    (let loop ((res goal)
               (sub '())
               (tag 1))
      (cond ((null? res)
             (select-sub sub goal))

            ((eq? (atom-start (car res)) 'is)
             (let ((s (do-is (car res) sub)))
               (loop (apply-sub s (cdr res))
                     (append s (apply-sub s sub)) tag)))

            ((list? (atom-start (car res)))
               ...)
            (else
             (let ((resolved
                    (resolve res sub tag goal prog)))
               (if resolved
                   (loop (resolved->res resolved)
                         (resolved->sub resolved)
                         (+ tag 1))
                   #f)))))))
```

Versão Preliminar

Com este novo interpretador o código anterior funciona como esperávamos, trocando apenas a chamada ao procedimento Scheme "=" pelo uso do novo predicado "is".

```
(define prolog-dobro '( ((dobro ?x ?y)
                         (is ?y (* 2 ?x))
                         ((display "O dobro de " ?x " é
" ?y)))))
(prolog+local prolog-dobro '((dobro 3 ?a)))
O dobro de 3 é 6
((?a 6))
```

Em Prolog o predicado `is` é usado apenas para avaliar expressões numéricas. Nosso interpretador faz mais que isso, avaliando *qualquer forma Scheme*, constituindo não apenas uma ferramenta para computação numérica, mas uma FFI (*Foreign Function Interface*, "interface para funções externas", ou uma forma de permitir o uso de uma linguagem de programação a partir de outra).

12.3.3 Predicados meta-lógicos

De acordo com o padrão Prolog, o predicado `listing` pode ser usado para obter a listagem do código fonte de um predicado – desde que ele esteja disponível (há predicados que são parte de bibliotecas padrão, e que podem inclusive ter sido feitos em linguagens diferentes de Prolog – estes não podem ser listados).

Os predicados `asserta(C)` e `assertz(C)` adicionam a cláusula C à base de dados do Prolog – o primeiro predicado adiciona no início, e o segundo no final da base de dados. Já o predicado `retract(C)` remove uma cláusula C da base de dados.

Para poder usar `asserta/assertz/retract` com um predicado P, é necessário declarar P como predicado "dinâmico" (um predicado é dinâmico se pode ser alterado durante a execução do programa). A declaração de `p/2` como dinâmico é

```
:- dynamic p/2.
```

Agora é possível usar `asserta` e `assertz` para adicionar cláusulas ao predicado `p/2`.

```
asserta(p(1,2)).
assertz(p(1,3)).
p(X,3).
X=1
```


Versão Preliminar

Usamos também `retract` para remover uma das cláusulas. `retract(p(1,3)).`
`p(X,3).`
no.

Implementaremos a seguir `asserta` e `retract`. Estes predicados simplesmente mudam a variável local `prog` no interpretador: `asserta` usa `cons` para adicionar uma cláusula, e `retract` usa `remove` para filtrar o predicado escolhido. Ambas usam `(set! prog ...)` para modificar o programa.

```
(define asserta
  (lambda (res)
    (let ((fact (cdar res)))
      (set! prog (cons fact prog)))))

(define retract
  (lambda (res)
    (let ((fact (cadar res)))
      (set! prog (remove (lambda (x) (equal? x fact))
                         prog)))))
```

Criamos um fecho sobre `prog` e `goal` para que os procedimentos `asserta` e `retract` possam modificar `prog`.

Versão Preliminar

```
(define prolog+meta
  (lambda (prog goal)
    (define asserta ...)
    (define retract ...)
    (amb+-reset)
    (let loop ((res goal)
               (sub '())
               (tag 1))
      (cond ((null? res)
             (select-sub sub goal))

            ((eq? (atom-start (car res)) 'asserta)
             (asserta res)
             (loop (cdr res) sub tag))

            ((eq? (atom-start (car res)) 'retract)
             (retract res)
             (loop (cdr res) sub tag))

            ((eq? (atom-start (car res)) 'is)
             ...)

            ((list? (atom-start (car res)))
             ...)

            (else
             (let ((resolved (resolve res sub tag goal prog)))
               (if resolved
                   (loop (resolved->res resolved)
                         (resolved->sub resolved)
                         (+ tag 1))
                   #f)))))))
```

Tanto `asserta` como `retract` tem efeitos colaterais, e portanto o local do programa em que são usados faz diferença. Considere o seguinte exemplo:

Versão Preliminar

```
(define prolog-assert-ok '( ((f ?x) (g ?x)
                                    (asserta (h 3))
                                    (h ?x))
                            ((g 2))
                            ((g 3))))

(prolog+meta prolog-assert-ok '((f ?a)))
((?a 3))
```

O interpretador conseguiu provar `(h ?3)` porque o fez depois de executar o `asserta`. Se invertermos a ordem desses dois objetivos, a computação falha, *mesmo com retrocesso*:

```
(define prolog-assert-fails '( ((f ?x) (g ?x)
                                       (h ?x)
                                       (asserta (h 3))
                               ((g 2))
                               ((g 3))))

(prolog+meta prolog-assert-fails '((f ?a)))
#f
```

O retrocesso aconteceu logo que o interpretador tentou provar `(h 2)` – mas como naquele momento não havia qualquer fato para `h` na base de dados, o interpretador sequer chegou a tentar provar o objetivo `(asserta (h 3))` – apenas tentou `(h 3)`, que também falhou, e depois disso esgotaram-se suas escolhas.

Para os exemplos de `retract` usaremos o programa abaixo.

```
(define prolog-retract '( ((f 2))
                          ((f 3))
                          ((g 1) (retract ((f 2))))
                          ((g 1) (f 3))))
```

Podemos perguntar `(f ?x)`, e obteremos a substituição `((?x 2))`:

```
(prolog+meta prolog-retract '((f ?x)))
((?x 2))
```

Versão Preliminar

Se nossa pergunta exigir tanto `(g 1)` como `(f ?x)`, o Prolog tentará primeiro a cláusula contendo o `retract`, falhará ao tentar provar `(f 2)`, retrocederá e usará a segunda cláusula para `(g 1)`, substituindo `?x` por 3.

```
(prolog+meta prolog-retract '((g 1) (f ?x)))
((?x 3))
```

O próximo exemplo é importante: se incluirmos um corte entre `(g 1)` e `(f ?x)`, o interpretador *não falhará!* Isso porque o o corte não impede completamente o retrocesso, ele apenas impede que substituições anteriores sejam refeitas. Como o interpretador não havia feito nenhuma substituição ao chegar ao corte, ele não tem efeito prático.

```
(prolog+meta prolog-retract '((g 1) ! (f ?x)))
((?x 3))
```

A ordem em que pedimos os objetivos no objetivo faz diferença: se o Prolog tentar provar `(f 2)` antes de `(g 1)`, terá sucesso.

```
(prolog+meta prolog-retract '((f 2) (g 1)))
()
```

No entanto, se trocarmos a ordem dos objetivos na pergunta, o interpretador eliminará `(f 2)` antes de tentar prová-lo.

```
(prolog+meta prolog-retract '((g 1) (f 2)))
#f
```

12.3.4 Corte

Um corte em uma cláusula impede que novas alternativas *para aquele predicado* possam ser tentadas. No entanto, quando as cláusulas são adicionadas ao resolvente, perdemos a informação de qual corte se refere a qual predicado. Podemos remediar esta situação, incluímos uma anotação em cada corte, logo antes da cauda da cláusula ser incluída no resolvente. Como já temos um procedimento que percorre a cauda renomeando variáveis, podemos usá-lo para identificar os cortes e gravar junto a eles o ponto de retrocesso (na verdade, gravamos a pilha inteira de `amb+`).

Versão Preliminar

```
(define rename-vars
  (lambda (vars tag)
    (cond ((pair? vars)
           (cons (rename-vars (car vars) tag)
                 (rename-vars (cdr vars) tag)))
          ((matching-symbol? vars)
           (rename-one vars tag))

          ((eq? vars '!)
           (list '! (cdr amb+-stack)))

          (else vars))))
```

Onde havia um corte passará a haver uma lista da forma `(! STACK)`, onde `STACK` é a *cauda* da pilha de continuações (somente a cauda, porque estamos justamente cortando o último ponto de escolha!).

Criamos um procedimento para identificar um corte entre os objetivos do resolvente: `cut?` apenas verifica se o objeto é um par, e se seu `car` é `!`.

```
(define cut?
  (lambda (x)
    (and (pair? x)
         (eqv? (car x) '!))))
```

A nova versão do interpretador contém mais um caso: quando um corte é encontrado, a pilha que ele traz é posta no lugar da que estava sendo usada.

Versão Preliminar

```
(define prolog+cut
  (lambda (prog goal)
    (amb+-reset)
    (let loop ((res goal)
               (sub '())
               (tag 1))
      (cond ((null? res)
             (select-sub sub goal))

            ((cut? (car res))
             (set! amb+-stack (cadar res))
             (loop (cdr res) sub tag))

            ((eq? (atom-start (car res)) 'asserta)
             ...)
            ((eq? (atom-start (car res)) 'retract)
             ...)
            ((eq? (atom-start (car res)) 'is)
             ...)
            ((list? (atom-start (car res)))
             ...)

            (else (let ((resolved (resolve res sub tag goal prog)))
                    (loop (resolved->res resolved)
                          (resolved->sub resolved)
                          (+ tag 1))))))))
```

Para o programa a seguir, a pergunta $a(Z)$ resulta em sucesso com a substituição $Z \to 2$. A primeira escolha do interpretador foi $Z \to 1$ por ter encontrado a cláusula $b(1)$ – mas como não foi possível satisfazer $c(1)$, houve a necessidade de *backtracking* e a escolha pasou a ser $Z \to 2$.

```
(prolog+cut '(( (a ?x) (b ?x) (c ?x) )
              ( (b 1) )
              ( (b 2) )
              ( (c 2) ))
            '((a ?z)))
```


Versão Preliminar

```
((?z 2))
```

No entanto, se incluirmos um corte na primeira cláusula, entre $b(X)$ e $c(X)$, o interpretador terá de se comprometer com as escolhas que fez desde que entrou na cláusula $a(\cdot)$ (inclusive a escolha desta regra para $a(\cdot)$ e a escolha do valor de X). Ele terá escolhido substituir $X \to 1$, e como não há como satisfazer $c(1)$, a computação falhará.

```
(prolog+cut '(( (a ?x) (b ?x) ! (c ?x) )
                ( (b 1) )
                ( (b 2) )
                ( (c 2) ))
              '((a ?x)))
#f
```

É muito comum associar o uso de corte com o uso de *falha*: podemos forçar o interpretador a falhar na tentativa de satisfazer um objetivo. Quando corte e falha são combinados, pode-se obter diversas formas concisas de controle de fluxo.

O único objeto Scheme que tem valor falso para a forma `if` é `#f` – todos os outros objetos são, para estes efeitos, equivalentes a `#t`. Podemos criar um procedimento `pfail` ("Prolog fail") que sempre retornará `#f`, e usá-lo quando for necessário determinar que uma computação falhe em um dado ponto.

```
(define pfail (lambda () #f))

(prolog+cut '(((f ?x)
               ((display "Falharemos: "))
               ((pfail))))
            '((f 1)))
Falharemos:
#f
```

Nosso último interpretador já é suficiente para resolver o problema das Torres de Hanói, descrito na Seção 1.7.2.1. O problema pode ser resolvido pelo seguinte programa Prolog.

Versão Preliminar

```
hanoi(N) ← hanoi_aux(N, 3, 1, 2).

hanoi_aux(0, _, _, _) ← !.
hanoi_aux(N, A, B, C) ← N_1 is N-1,
                        hanoi_aux(N_1, A, C, B),
                        mova(A, B),
                        hanoi_aux(N_1, C, B, A).

mova(F, T) ← write([F, -->, T]), nl.
```

O programa, traduzido para nosso Prolog-em-Scheme, é mostrado a seguir.

```
(define prolog-hanoi
  '(((hanoi ?n) (hanoi-aux ?n 3 1 2))
    ((hanoi-aux 0 ?a ?b ?c) ! )
    ((hanoi-aux ?n ?a ?b ?c)
     (is ?nn (- ?n 1))
     (hanoi-aux ?nn ?a ?c ?b)
     (mova ?a ?b)
     (hanoi-aux ?nn ?c ?b ?a))
    ((mova ?de ?para)
     ((display ?de))
     ((display " --> "))
     ((display ?para))
     ((newline)))))
(prolog+cut prolog-hanoi '((hanoi 3)))
3 -> 1
3 -> 2
1 -> 2
3 -> 1
2 -> 3
2 -> 1
3 -> 1
()
```

A última linha é o valor de retorno de `prolog+local`: uma substituição vazia.

Versão Preliminar

12.3.5 Negação

Na programação em Lógica (e em Prolog) não podemos usar a forma negativa de átomos e termos: podemos declarar `a` ou `f(b)`, mas não podemos declarar que "`c` é falso" ou "`f(d)` é falso". A teoria da programação em Lógica depende disso (é possível incluir negação nos átomos, mas o procedimento de resolução que descrevemos não funcionaria).

Em Prolog usamos a *hipótese do mundo fechado*: presumimos que tudo o que é verdade pode ser deduzido da base de dados Prolog. O que não pode ser deduzido é falso.

Assim, a negação em Prolog é descrita pela regra de inferência, "quando não se pode provar P, conclui-se 'não P'"

A negação como falha não é equivalente à negação em Lógica. Considere a seguinte base de dados Prolog:

```
pintor(picasso).
pintor(cezanne).
```

Podemos perguntar `pintor(cezanne)` e obteremos a resposta correta "sim". Também podemos perguntar `pintor(X)` e obteremos as respostas "picasso" e "cezanne". No entanto, o problema da negação como falha fica claro quando perguntamos `pintor(monet)`. Da maneira como a concebemos, nossa base de dados trazia *conhecimento parcial* a respeito de pintores (ou seja, da relação `pintor`). O fato de alguém não estar listado não significa que ele não seja pintor. Em Lógica temos predicados que são demonstráveis, os que são *refutáveis* e os *indecidíveis* (estes são os que não conseguimos demonstrar nem refutar). Para um interpretador Prolog, no entanto, o mundo é "fechado", e as proposições são "verdadeiras" ou "falsas". Ao não conseguir concluir "`pintor(monet)`", conclui imediatamente que "não `pintor(monet)`".

A negação como falha pode ser implementada em Prolog usando corte e falha.

```
not(X) ← X, !, fail.
not(X).
```

Suponha que X pode ser satisfeito pelo interpretador Prolog, e que pedimos que o Prolog responda `not(X)`. A primeira cláusula será escolhida, e o resolvente será `X, !, fail`. Como o interpretador conseguirá satisfazer X, passará então pelo corte e falhará. Como não pode mais retroceder, o resultado é um "não".

Quando X é falso, logo que a primeira cláusula é escolhida, X falha antes do corte, e o Prolog retrocede para a segunda cláusula, que sempre tem sucesso, retornando uma substituição vazia.

Versão Preliminar

Usando a sintaxe de nosso interpretador, o predicado `not` é:

```
((not ?x) ?x ! ((pfail)))
((not ?x))
```

No próximo exemplo, incorporamos o predicado `not` a um programa onde só há uma outra cláusula, `f(a)`. Quando pedimos ao Prolog que tente provar a negação de `F(Z)` ele falha, porque consegue provar `f(Z)`.

```
(prolog+cut '( ((not ?x) ?x ! ((pfail)))
               ((not ?x))
               ((f a)) )
             '( (not (f ?z)) ))
#f
```

Ao negarmos um objetivo que falharia, obtemos sucesso e uma substituição vazia:

```
(prolog+cut '( ((not ?x) ?x ! ((pfail)))
               ((not ?x))
               ((f a)) )
             '( (not (f b)) ))
()
```

A dupla negação pode ser usada para eliminar os vínculos de variáveis, já que quando a segunda cláusula de `not` é usada, a substituição vazia é retornada.

```
(prolog+cut '(((not ?x) ?x ! ((pfail)))
              ((not ?x))
              ((f a)) )
            '((not (not (f ?z)))))
()
```

12.4 UM METAINTERPRETADOR PROLOG

Há um predicado definido no padrão prolog que permite ter acesso a cláusulas do programa. Para usar `clause(A,B)`, `A` deve estar ao menos parcialmente instanciado, e e então o interpretador Prolog tentará unificar `A` com a cabeça de alguma cláusula e `B` com

Versão Preliminar

a respectiva cauda. Por exemplo, se houver na base de dados a regra

```
f(1) ← g(2).
```

então a pergunta `clause(f(X),B)` retornará

```
B = g(2),
X = 1.
```

Em Prolog um predicado pode ser público ou privado. Apenas predicados públicos podem ser acessados por `clause/2`.

O predicado `clause/2` pode ser usado para construir um metainterpretador de Prolog puro.

```
interpret(A,B) ← !,
                 interpret(A),
                 interpret(B).
interpret(true) ← !.
interpret(A) ← clause(A,B),
               interpret(B).
```

As cláusulas usadas devem ser todas públicas. Em Prolog podemos declarar que uma lista de cláusulas é pública usando o predicado `public/1`:

```
:- public([a/0,x/1,y/1]).
```

Agora damos a definição dos predicados `a/0`, `x/1`, e `y/1`:

```
a.
x(Z) ← y(Z).
y(5).
```

Finalmente, podemos interpretá-los.

```
interpret(a).
yes.
interpret(x(Q)).
Q=5
```

Versão Preliminar

12.5 PROGRAMANDO EM PROLOG

Esta Seção trata de algumas técnicas de programação em Prolog. Não é um tratamento exaustivo, e sim uma breve exploração do assunto. Na Seção 12.7 há indicações de livros que tratam de programação em Prolog de maneira mais extensa.

12.5.1 Repetição e acumuladores

12.5.2 Listas

A linguagem Prolog padrão permite a construção de listas da mesma maneira que nas linguagens do tipo Lisp. O funtor "." tem papel análogo ao procedimento `cons` e ao operador ".". Em Scheme, a forma `(a . (b . (c . (d . '()))))` é uma lista, que pode ser abreviada como `(a b c d)`. Em Prolog, esta mesma lista é escrita como `.(a,.(b,.(c,.(d,[]))))`, ou `[a|[b|[c|[d]]]]` e também pode ser abreviada como `[a,b,c,d]` ou `[a|[b,c,d]]`.

Como nosso interpretador foi desenvolvido em Scheme, podemos usá-lo para manipular listas Scheme diretamente, já que nosso procedimento de unificação funciona com listas e pares.

O predicado `append` poderia ser descrito da seguinte maneira em Prolog.

```
append((X . Y), Z, (X . W)) ← append(Y, Z, W).
append([],X,X).
```

Traduzimos o predicado para nosso interpretador Prolog:

```
(define prolog-append '( ((append (?x . ?y) ?z (?x . ?w))
                          (append ?y ?z ?w))
                         ((append () ?x ?x))))
(prolog+cut prolog-append '( (append (0) ?a (0 1)) ))
((?a 1))
```

A resposta está correta, porque `(?a 1)` é o mesmo que `(?a . (1))` – a substituição indica a troca de `?a` pela lista `(1)`.

```
(prolog+cut prolog-append '( (append (0) ?a (?d 1) )))
((?d . 0)) (?a 1))
```


Versão Preliminar

12.5.3 Listas-diferença

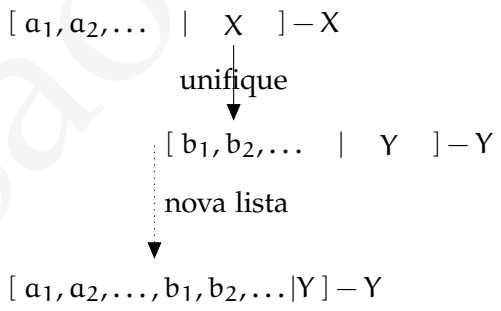
O predicado `append` que desenvolvemos na Seção 12.5.2 precisa percorrer toda a primeira lista, portanto seu tempo de execução não é constante. O mesmo vale para o procedimento `append` de Scheme. No entanto, em Scheme podemos manter um ponteiro para o final da lista, como fizemos ao desenvolver a estrutura de fila na Seção 3.3.3 (página 116). Há uma técnica semelhante em Prolog, que exploraremos nesta Seção.

Em Prolog podemos também "manter um ponteiro para o final da lista". Isso é o mesmo que criar uma lista cuja cauda seja uma variável T, por exemplo. A lista `[1,2,3|T]` tem o marcador T no final, e se instanciarmos T como `[4,5]`, a lista passará a ser `[1,2,3,4,5]`.

Lembramos que em Prolog operadores matemáticos são apenas *funtores*, e sua função é apenas descrever *sintaticamente* uma estrutura: $A - b$ representa a estrutura "$-(A, b)$", e não necessariamente uma subtração.

Podemos representar a lista `[1,2,3]` como `[1,2,3|X]-X`. Nossa interpretação para isso será "a lista que começa com $1, 2, 3$, tem cauda X, mas sem a cauda (interpretamos $-X$ como a operação de remoção de X da cauda). A vantagem de representar a lista dessa forma é que a variável X funciona como "marcador de final". Para concatenar listas-diferença usamos o predicado a seguir.

Como exemplo de uso de listas-diferença, mostramos como realizar a operação de concatenação sem ter que percorrer toda a lista. Supomos agora que temos duas listas representadas como lista-diferença: $A - X$ e $B - Y$. Ou seja, A é representada como $[a_1, a_2, \ldots | X] - X$ e B é $[b_1, b_2, \ldots | Y] - Y$. Usualmente as variáveis X e Y não estarão instanciadas. Elas estão ali somente para "marcar o lugar" do final de cada lista. Se quisermos concatenar as duas listas, podemos simplesmente unificar X com $[b_1, b_2, \ldots | Y]$ e usar a cauda Y que já estava em B como cauda da nova lista resultante:



O código Prolog para `append_dl` consiste então de uma única linha:

```
append_dl(A-B, B-C, A-C).
```

Além de curto e elegante, `append_dl` é mais eficiente do que um predicado `append` que

Versão preliminar

percorresse a primeira lista: `append_dl` realiza apenas um número fixo de unificações, independente do tamanho da lista.

O próximo exemplo, dado no livro de Leon Sterling e Ehud Shapiro [SS94], mostra como listas-diferença podem ser usadas para evitar operações de concatenação.

O predicado `flatten`, mostrado a seguir, pode usado para transformar uma lista hierárquica (uma árvore) em uma lista plana. Para "aplanar" uma lista [X|T], usamos recursão duas vezes: aplanamos tanto X como T, resultando nas listas X1 e T1, e depois devolemos a concatenação de T1 e T2. Para aplanar X que seja constante e diferente da lista vazia, simplesmente retornamos a lista [X]. Para aplanar a lista vazia devolvemos ela mesma.

```
flatten([X|T], Y) ← flatten(X,T1),
                    flatten(T,T2),
                    append(T1, T2, Y).
flatten(X,[X]) ← constant(X), X ≠ [].
flatten([],[]).
```

Usando listas-diferença, o predicado pode ser reescrito como segue.

```
flatten_dl([X|T],Y-Z) ← flatten_dl(X,A-B),
                        flatten_dl(T,C-D),
                        append_dl(A-B, C-D, Y-Z).
flatten_dl(X,[X|T]-T) ← constant(X), X ≠ [].
flatten_dl([],X-X).
```

Observamos no entanto que a chamada a `append_dl` apenas unifica algumas das variáveis, e podemos incorporar isto diretamente no programa, reescrevendo a cláusula de acordo com a definição de `append_dl`.

```
flatten_dl([X|T],Y-Z) ← flatten(X,A-B),
                        flatten(T,B-C).
flatten_dl(X,[X|T]-T) ← constant(X), X ≠ [].
flatten_dl([],X-X).
```

Finalmente, damos um último exemplo de uso de listas-diferença. Para contar os elementos em uma lista-diferença, podemos construir indutivamente o predicado `count`:

- Para a lista $X - X$ (ou $X - Y$, com X unificando com Y), a contagem é zero.
- Em outros casos, a contagem para $[H|T] - T1$ é a contagem de $T - T1$ mais um.

Em Prolog o predicado `count` é especificado como segue.

Versão Preliminar

```
count(X-X1,0) :- unify_with_occurs_check(X,X1), !.
count([H|T]-T1,N) :- count(T-T1,M), N is M+1.
```

O predicado `unify_with_occurs_check` realiza unificação com o *occurs check*.

A tradução do predicado `count` para nosso Prolog-em-Scheme é dada a seguir.

```
(define prolog-difflist '( ((count (- ?x ?x1) 0)
                            ((unify '?x '?x1))
                            !)
                           ((count (- (?h . ?t) ?t1) ?n)
                            (count (- ?t ?t1) ?m)
                            (is ?n (+ ?m 1)))))

(prolog+cut prolog-difflist
            '( (count (- (1 2 3 4 . ?a) ?a) ?n) ))
((?n . 4))
```

12.5.4 Usando cortes

Cortes tem em Prolog a mesma relevância que continuações em Scheme: são uma ferarmenta fundamental a partir da qual diversas outras podem ser criadas eficientemente (nesta Seção ilustramos como descrever `if`; nas Seções anteriores usamos uma abstração de `not`). Diversas construções de controle foram propostas para a linguagem Prolog que na verdade poderiam ser descritas usando cortes.

Inicialmente faremos uma distinção entre dois usos do corte, chamados de "cortes verdes" e "cortes vermelhos", e em seguida abordamos alguns dos usos comuns do corte em programas Prolog.

12.5.4.1 *Tipos de corte*

Os cortes em um programa Prolog podem ser classificados em dois tipos, normalmente conhecidos como *cortes verdes* e *cortes vermelhos*. Um corte verde existe apenas para tornar o programa mais eficiente – sua remoção não muda o resultado da computação. Já o corte vermelho é usado para modificar o sentido do programa.

Um exemplo muito simples de corte vermelho está no uso de corte-e-falha, como o fizemos para implementar a negação:

Versão Preliminar

```
not(X) :- X, !, fail.
not(_).
```

A remoção do corte mudaria completamente a semântica deste programa.

Já no seguinte programa, os cortes são verdes: há duas cláusulas iniciais, e estas impõem inicialmente condições complementares. O primeiro corte apenas permite que o interpretador Prolog deixe de chamar `positivos` recursivamente quando `X>0` (a última cláusula trata da lista vazia e de objetos que não são listas).

```
positivos([X|T1],[X|T2]) ← X > 0, ! positivos(T1,T2).
positivos([X|T1],[H|T2]) ← X <= 0, ! positivos(T,[H|T2]).
positivos(_,[]).
```

A remoção dos dois cortes não altera o significado do programa – apenas pode terná-lo menos eficiente.

Como sabemos que na última cláusula a condição `X <= 0` sempre será verdadeira, poderíamos omití-la, assim como o corte que a acompanha:

```
positivos2([X|T1],[X|T2]) ← X > 0, ! positivos2(T1,T2).
positivos2([_|T1],[H|T2]) ← positivos2(T1,[H|T2]).
positivos2(_,[]).
```

O programa `positivos2` é equivalente ao programa `positivos`. No entanto, nesta nova versão do programa, o corte na segunda cláusula passa a ser vermelho, porque sua remoção mudaria o significado do programa. Ao removermos o corte de `positivos2` obtemos o programa `positivos_errado`:

```
positivos_errado([X|T1],[X|T2]) ← X > 0, positivos_errado(T1,[H|T2]).
positivos_errado([_|T1],[H|T2]) ← positivos_errado(T1,[H|T2]).
positivos_errado(_,[]).
```

Para verificar que esta versão não daria respostas corretas, basta observar que sem o corte a segunda cláusula seria usada quando a primeira passar da condição $X > 0$ mas falhar em `p(T1,[HT2])|`. Pedimos por exemplo os positivos da lista `[-1,1]`:

```
positivos_errado([-1,1],A).
A = [1,[]]?
```

Além do predicado `not`, podemos usar cortes vermelhos para simular a construção "se/então/senão":

Versão Preliminar

```
if(COND, THEN, ELSE) ← COND, !, THEN.
if(COND, THEN, ELSE) ← ELSE.
```

Se simplesmente removermos o corte, mudamos o significado do predicado `if`. Para obter o mesm efeito, teríamos que adicionar uma condição `not(COND)` no início da segunda cláusula, e isto pode ser custoso.

```
if(COND, THEN, ELSE) ← COND, THEN.
if(COND, THEN, ELSE) ← not(COND), ELSE.
```

É consenso entre programadores Prolog que o uso excessivo de cortes vermelhos normalmente indica problemas na arquitetura do programa.

12.5.4.2 *Usos comuns de corte*

Normalmente os cortes em um programa Prolog são usados com um dos objetivos listados a seguir.

i) *Confirmar a escolha de uma cláusula de um predicado*, impedindo que outras sejam usadas se houver retrocesso.

ii) *Fazer falhar imediatamente um objetivo*, sem tentar outras cláusulas. Esta é a combinação corte-falha, que já usamos na definição do predicado de negação.

iii) *Interromper a geração de alternativas* em um algoritmo do tipo "gerar-e-testar".

12.5.4.2.1 Confirmando a escolha de uma cláusula

Para nosso próximo exemplo usaremos um predicado que falha quando tentamos satisfazê-lo na primeira vez, mas depois sempre tem sucesso. Este predicado, `fail_first`, pode parecer demasido artificial, mas na verdade ilustra uma situação que pode facilmente acontecer na prática quando um objetivo pode falhar dependendo do estado de bases de dados e outras condições externas ao programa.

```
fail_first ← not(done), !, asserta(done), fail.
fail_first.
```

Neste exemplo temos um trecho de programa que calcula o crédito bancário disponível a clientes. Há duas modalidades de crédito: o consignado (nesta modalidade as prestações são descontada diretamente na folha de pagamento do cliente), e comum (nesta modalidade as prestações são descontadas na conta-corrente). Há uma regra que determina que

Versão Preliminar

nenhum cliente deve poder obter empréstimo se estiver inadimplente com o banco, ou se for devedor de impostos.

```
deve_impostos('Henry Thoreau').

credito_consignado('Henry Thoreau', 1000).
credito_comum('Henry Thoreau',2000).

credito_consignado('John Doe', 500).
credito_comum('John Doe',1500).

credito(Cliente,0) ← inadimplente(Cliente).
credito(Cliente,0) ← deve_impostos(Cliente).
credito(Cliente,C) ← credito_consignado(Cliente,Consignado),
                     credito_comum(Cliente,Comum),
                     C is Consignado + Comum.
```

Aparentemente o programa funciona:

```
credito('Henry Thoreau',C)
C=0
yes.
credito('John Doe,C)
C=2000
yes.
```

No entanto, pode ser que algum predicado após `credito` falhe e o interpretador tente resatisfazer `credito`. A pergunta a seguir mostra o que pode acontecer.

```
credito('Henry Thoreau',C),fail_first.
C = 3000
yes.
```

Inicialmente, a segunda cláusula (que dá zero de crédito a devedores de impostos) foi usada. Mas quando `fail_first` falhou, o interpretador retrocedeu e tentou satisfazer novamente `credito`, usando a terceira cláusula, retornando o crédito de 3000.

O problema está em permitirmos que o interpretador tente mais cláusulas depois da segunda. Podemos simplesmente adicionar um corte no final dela (e da primeira, que funciona de forma semelhante) para que o programa funcione corretamente.

Versão Preliminar

```
credito(Cliente,0) ← inadimplente(Cliente), !.
credito(Cliente,0) ← deve_impostos(Cliente), !.
credito(Cliente,C) ← credito_consignado(Cliente,Consignado),
                     credito_comum(Cliente,Comum),
                     C is Consignado + Comum.
```

```
credito('Henry Thoreau',C),fail_first.
no.
```

O corte, no entanto, pode dificultar a leitura do programa. É possível usar, na última cláusula, a negação das condições restritivas, tornando o programa mais claro.

```
credito(Cliente,0) ← inadimplente(Cliente).
credito(Cliente,0) ← deve_impostos(Cliente).
credito(Cliente,C) ← not ( inadimplente(Cliente) ),
                     not ( deve_impostos(Cliente) ),
                     credito_consignado(Cliente,Consignado),
                     credito_comum(Cliente,Comum),
                     C is Consignado + Comum.
```

De maneira geral programadores Prolog preferem esta solução por ser mais clara, mas observamos que aqui temos que tentar satisfazer os predicados `deve_impostos` e `inadimplentes` duas vezes (uma nas duas primeiras cláusulas, e outra na terceira, anes de negá-los).

Damos outro exemplo deste uso de corte, na definição de um preedicado que calcula números de Fibonacci.

```
fib(0,0).
fib(1,1).
fib(N,F) ← N1 is N-1,
           N2 is N-2,
           fib(N1,F1),
           fib(N2,F2),
           F is F1 + F2.
```

O predicado `fib` aparentemente funciona, mas se tentarmos a pergunta "`fib(1,F), fail_first`", o interpretador entrará em um laço infinito que somente parará quando a pilha estourar.

Para remediar a situação podemos usar as duas soluções que usamos para o predicado `credito`. A primeira (e não recomendada) é o uso direto do corte.

Versão Preliminar

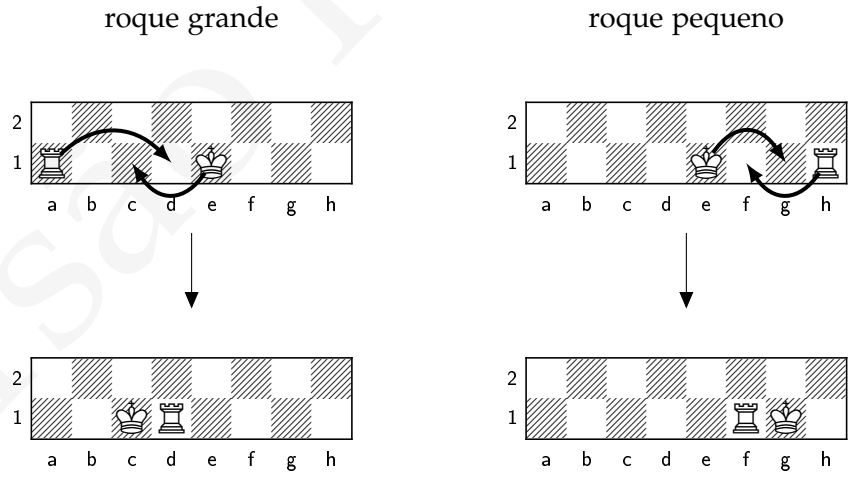
```
fib(0,0) ← !.
fib(1,1) ← !.
```

A segunda solução é negar as condições anteriores na terceira cláusula de `fib`. Poderíamos usar `not` duas vezes, mas como desta vez tratamos de uma condição numérica, é ainda melhor usar o operador "`>`".

```
fib(N,F) ← N > 2,
           N1 is N-1,
           N2 is N-2,
           fib(N1,F1),
           fib(N2,F2),
           F is F1 + F2.
```

12.5.4.2.2 Corte-e-falha

Já expusemos o uso da combinação corte-falha na construção das abstrações `not` e `if`. O corte seguido de falha pode também ser usado para especificar exceções a uma regra. Por exemplo, no jogo de xadrez há uma jogada especial chamada "roque", que envolve o rei e uma das torres (o rei "esconde-se atrás" da torre). As duas figuras a seguir mostram esta jogada (quando a torre da esquerda é usada, diz-se que foi feito um "roque grande"; quando a torre da direita é usada, foi um "roque pequeno").



As regras do xadrez definem que o roque só é possível em certas condições:

- O rei não pode ter sido movido até o momento do roque

Versão Preliminar

- A torre não pode ter sido movida até o momento do roque.
- O rei não pode estar em cheque.
- O rei não pode terminar em cheque.
- A casa por onde o rei passará (d1 no roque grande ou f1 no roque pequeno) deve estar livre: não deve estar ocupada nem atacada.
- O rei e a torre devem estar na mesma linha[9].

Podemos traduzir estas regras para Prolog usando corte e falha nalgumas das regras.

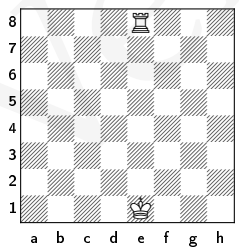
```
castling_ok(Player,Rook) ← moved(Player, king), !, fail.
castling_ok(Player,Rook) ← moved(Player, Rook), !, fail.
castling_ok(Player,Rook) ← check(Player), !, fail.
castling_ok(Player,Rook) ← attacked(castle, Player, Rook), !, fail.
castling_ok(Player,Rook) ← free_path(Player, king, Rook),
                           same_row(Player, king, Rook).
```

Assim como no uso de corte para escolha da cláusula certa (visto na Seção 12.5.4.2.1), podemos trocar a combinação corte-falha pela negação.

```
castling_ok(Player,Rook) ← not (moved(Player, king)).
...
```

O efeito será o mesmo, porque o predicado `not` inclui o corte.

9 Se um peão atravessar o tabuleiro, pode ser promovido a torre, e se esta torre estiver na coluna e, o jogador poderia tentar fazer um roque "vertical", que esta regra proíbe:



Versão Preliminar

12.5.4.2.3 *Evitando a geração de soluções alternativas*

12.6 MÁQUINAS ABSTRATAS E IMPLEMENTAÇÕES DE PROLOG

É evidente que nosso interpretador Prolog não é eficiente. Há duas grandes evidências disso: primeiro, o interpretador representa o programa como uma lista de associação (quando o programa é muito grande a busca por cláusulas será lenta); além disso, cada vez que a escolha de uma cláusula é feita, uma continuação é guardada – e cada continuação contém todo o ambiente sendo usado naquele momento, inclusive variáveis que provavelmente não precisaríamos guardar.

Implementações eficientes de Prolog normalmente usam métodos eficientes para armazenar o programa, e traduzem o programa Prolog para código especial que opera em uma máquina virtual (normalmente para a *Máquina Abstrata de Warren* – WAM[10] ou a *Máquina Abstrata de Vienna* – VAM[11]).

12.7 MAIS SOBRE PROLOG

O livro de William Clocksin e Christopher Mellish [CM03] é uma boa introdução à programação em Prolog. O de Ivan Bratko [Bra11] apresenta técnicas mais elaboradas e aplicações em Inteligência Artificial. Os livros de Covington, Nute e Vellino [CNV98], e de Richard O'Keefe [OKe09] abordam diversas técnicas avançadas de programação em Prolog. O de Sterling e Shapiro [SS94], além de explorar técnicas avançadas de programação, apresenta o modelo de execução da programação em Lógica. O livro de Ulf Nilsson e Jan Małuszyński [NM95] é mais formal e rigoroso, e enfatiza fortemente os fundamentos da Programação em Lógica. Há também uma coletânea de aplicações mostrando o uso de diversas técnicas de programação Prolog, editada por Leon Sterling [Ste03]

A sintaxe de Prolog usada neste Capítulo é completamente diferente dos sistemas Prolog atuais, mas é parecida com a sintaxe usada, por exemplo, no sitema Micro-Prolog [CM84; CME83] (desenvolvido para computadores de 8 bits na década de 80), e exatamente a mesma sintaxe usda no LM-Prolog [Car84] (escrito em ZetaLisp, para máquinas Lisp, nos anos 80). O LM-Prolog também usava a marca de interrogação para diferenciar variáveis de funtores.

10 *"Warren Abstract Machine"*

11 *"Vienna Abstract Machine"*

Versão Preliminar

EXERCÍCIOS

Ex. 157 — Há pequenos programas Prolog neste Capítulo que não foram traduzidos para o nosso interpretador Prolog-em-Scheme. Faça a tradução e teste os programas.

Ex. 158 — Neste Capítulo mostramos o predicado `fail_first`, que falha na primeira vez que é usado, mas sempre é satisfeito com sucesso depois. Construa os predicados:

- `alternate_fail`, que falha anternadamente (o predicado deve alternar sucesso e falha cada vez que é usado).
- `fail_when(F)`. A variável `F` deve ser instanciada com uma função de uma variável. O predicado será chamado várias vezes, e na n-ésima vez, falhará se e somente se `F(n)` for verdadeiro. Por exemplo, `fail_when( prime )` falhará na n-ésima chamada se n for primo; `fail_when( odd )` falhará quando n for ímpar.

Ex. 159 — Reescreva o jogo de poquer desenvolvido na Parte I usando Prolog.

Ex. 160 — Escreva predicados para manipulação de filas usando listas-diferença em Prolog.

Ex. 161 — O predicado padrão Prolog `findall/3` encontra todos os termos que satisfazem um predicado: na pergunta `findall(X,Objetivo,Resultado)`, normalmente `X` ocorre no objetivo. O interpretador instanciará `Resultado` com uma lista de todos os `X` que satisfaçam `Objetivo`. Por exemplo,

```
f(1).
f(2).
f(X) ← g(X), not h(X).

g(3).
g(4).

h(4).

findall(X,f(X),L).
L = [1, 2, 3]
```

Implemente o predicado `findall/3`, sem modificar o interpretador Prolog (ou seja, implemente-o em Prolog).

Ex. 162 — O interpretador `prolog+cut` tem dois trechos de códigos quase idênticos:

Versão Preliminar

```
((eqv? (atom-start (car res)) 'asserta)
 (do-asserta res)
 (loop (cdr res) sub tag))
((eqv? (atom-start (car res)) 'retract)
 (do-retract res)
 (loop (cdr res) sub tag))
```

Se houver mais predicados padrão que aueiramos incluir, o código do interpretador ficará muito longo. Mostre como modificar esse trecho de maneira que o código fique menor.

Ex. 163 — Criamos procedimentos `rename-one` e `rename-vars` para renomear as variáveis de uma cláusula no interpretador Prolog. Porque não poderíamos simplesmente ter trocado `rename-one` por `gensym` (descrito na Seção 8.9.1)?

Ex. 164 — Para forçar a computação a falhar, chamamos um procedimento Scheme `fail`, que sempre rtorna `#f`. Modifique o interpretador Prolog para que possamos usar diretamente `#f`, ou algum outro símbolo, como `#fail`.

Ex. 165 — Implemente `assertz` e `abolish`. O predicado `assertz` é semelhante a `asserta`, exceto por incluir a regra no final da base de dados. O predicado `abolish` remove *todas* as regras de um predicado: `abolish(p/2)` eliminará todas as regras `p(.,.) :- ...`

Ex. 166 — Implemente `clause/2` em nosso interpretador Prolog, e depois implemente o metainterpretador da Seção 12.4.

Ex. 167 — Pesquise a definição dos predicados padrão Prolog `consult`, `var`, `nonvar`, `atom`, `number`, `atomic`, e implemente-os em nosso interpretador Prolog.

Ex. 168 — Nosso Prolog não é robusto: se a entrada não tiver o formato correto, as mensagens de erro dadas vem de procedimentos como `car`, `cdr`, etc. Implemente um verificador de sintaxe para ele.

Ex. 169 — Nosso Prolog permite corte apenas em cláusulas do programa, e não no objetivo. Explique porque isso acontece e modifique o interpretador para que cortes possam ser usados no objetivo. Por exemplo, considere o programa Prolog a seguir:

```
f(1).
f(2).
g(2).
```

A pergunta a seguir não usa corte, e o interpretador a responderá com a substituição $X = 2$.

Versão Preliminar

```
?- f(X),g(X).
```

Já a próxima pergunta não tem como ser respondida pelo interpretador, que não saberá o que fazer com o corte na lista de objetivos.

```
?- f(X),!,g(X).
```

Queremos que o corte seja corretamente interpretado (e que esta última pergunta seja respondida negativamente, já que após escolher `f(1)` não é possível retroceder para escolher `f(2)`).

Ex. 170 — A linguagem Prolog permite o uso de variáveis anônimas. Estas são variáveis que usamos ao escrever a pergunta, mas que não nos interessam na resposta. Uma variável anônima é denotada por "_".

```
member(_,[]).
no

member(_,[1,2,3]).
yes
```

Observe que duas ocorrências de _ não correspondem à mesma variável:

```
f(a,b).

?- f(_,_).
yes
```

Implemente variáveis anônimas em nosso interpretador Prolog.

Ex. 171 — Quando nosso sistema Prolog precisa encontrar uma cláusula que unifique com algum termo, buscamos todas as cláusulas da base de dados, mas isso é ineficiente. Modifique o sistema para que ele passe a usar *indexação de termos*, tornando o *backtracking* mais rápido.

Ex. 172 — Experimente com idéias diferentes para `select-goal` e `next-goals`.

Ex. 173 — Nosso Prolog não é muito eficiente. Considere o seguinte trecho:

```
(require (unify (car res) (car rule)))
(let ((sub-and-tail (unifying-clause (car res) rule tag)))
```


Versão Preliminar

Estas duas linhas realizam unificação duas vezes – uma na verificação do `require` e outra para obter a substituição e a cauda da regra.
Encontre uma maneira de tornar nosso Prolog mais eficiente – *sem* abrir mão de não-determinismo e *sem* usar mutação de variáveis.

Ex. 174 — Nossa implementação do predicado `is`, como mencionamos no texto, funciona para qualquer forma Scheme, e não apenas para expressões numéricas. Mostre como restringir o `is` de forma que só permita cálculos com resultado numéricos.

Ex. 175 — Nosso Prolog somente retorna uma resposta paar cada pergunta, ignorando outras possíveis respostas. Tente modificá-lo para que ele retorne todas as respostas (e todas as substituições) possíveis, como um sistema Prolog tradicional. Por exemplo, o seguinte programa:

```
p(a).
p(b).
```

tem duas respostas possíveis, e sistemas Prolog oferecem ambas ao usuário.

```
p(X).
X=a ?
X=b ?
```

Ex. 176 — Tente implementar Prolog usando busca em largura ao invés de busca em profundidade. Comente o resultado.

Ex. 177 — O prolog que implementamos usa não-determinismo com `amb` para escolher as cláusulas do programa a serem usadas. Também é possível usar *streams* para isso. Mostre como.

Ex. 178 — Troque a lista de associação no interpretador Prolog por uma árvore balanceada (AVL ou vermelho-preto), e verifique a diferença no desempenho quando o programa Prolog sendo interpretado tem muitas regras.

Ex. 179 — Se seu ambiente Scheme oferece interface com algum servidor de banco de dados, construa uma base de dados SQL para representar o programa e compare novamente o desempenho (ao usar SQL você terá que rever o uso de continuações no interpretador)

Ex. 180 — Há uma máquina virtual concebida especificamente para a implementação de sistemas Prolog – a *Máquina Abstrata de Warren*[12], ou *WAM*. Leia o tutorial de Hassan

12 Warren Abstract Machine

Versão Preliminar

Aït-Kaci sobre a WAM [Aït91] e a implemente em Scheme. Depois construa um sistema Prolog sobre esta máquina virtual.

RESPOSTAS

Resp. (Ex. 161) — Use predicados meta-lógicos (`asserta` e `retract`).

```
findall(X,Obj,_) ← asserta(achei(z)),
                   Obj,
                   asserta(achei(res(X))),
                   fail.

findall(_,_,L) ← group([],M),!,L=M.

group(S,L) ← next(X),
             !,
             group([X|S],L).
group(L,L).

next(Y) ← retract(achei(X)), !, X=res(Y).
```

Resp. (Ex. 162) — Use `eval` e `call` para chamá-los.

```
(set! asserta ...)
(set! retract ...)
...
        ((memq (atom-start (car res)) '(asserta retract))
         (call (eval (atom-start (car res))) (list res))
         (loop (cdr res) sub tag))
```

Resp. (Ex. 163) — Cada instância de uma variável na cláusula teria um nome diferente:

Versão Preliminar

```
(define rename-vars-w/gensym
  (lambda (vars)
    (cond ((pair? vars)
           (cons (rename-vars (car vars) tag)
                 (rename-vars (cdr vars) tag)))
          ((matching-symbol? vars)
           (gensym vars))
          (else vars))))
```

Ao usarmos este procedimento: `(rename-vars-w/gensym '( (f ?a) ((g ?a) (g ?a))))`

```
((f ?a1085) ((g ?a1086) (g ?a1087)))
```

Percebemos que a variável `?a` foi renomeada tres vezes, e o sistema Prolog passará a crer que são tres variáveis diferenetes.

Resp. (Ex. 169) — Quando implementamos o corte em nosso interpretador Prolog, modificamos o procedimento `rename-vars` para trocar o símbolo `!` por uma lista `(! (cdr AMB-STACK))`, onde `AMB-STACK` é a pilha atual de continuações. Como apenas cláusulas do programa são renomeadas, o símbolo de exclamação aparecerá isolado no objetivo, e o interpretador não lidará corretamente com ele.
Podemos modificar o procedimento `rename-vars` para não renomear variáveis quando o tag for `#f`. O efeito de `(rename-vars V #f)` será de apenas modificar os cortes em `V`.

```
(define rename-vars
  (lambda (vars tag)
    (cond ((pair? vars)
           (cons (rename-vars (car vars) tag)
                 (rename-vars (cdr vars) tag)))
          ((and tag (matching-symbol? vars))
           (rename-one vars tag))
          ((eq? vars '!)
           (list '! (cdr amb+-stack)))
          (else vars))))
```

No interpretador, precisamos apenas renomear o objetivo antes de iniciar o laço principal:

Versão Preliminar

```
(let loop ((res (rename-vars goal #f))
           (sub '())
           (tag 1))
```

Resp. (Ex. 174) — Basta verificar, após avaliar a expressão, se ela é um número. Podemos usar `number?` para isso. Note que *não* devemos permitir variáveis não instanciadas na expressão a ser avaliada.

Versão Preliminar

Versão Preliminar

13 TIPOS: VERIFICAÇÃO E INFERÊNCIA

(Este Capítulo abordará verificação e inferência de tipos em Scheme)

Versão Preliminar

Versão Preliminar

Parte III.

Programação Concorrente

Versão Preliminar

Versão Preliminar

14 CONCORRÊNCIA

Programas concorrentes são projetados para realizar mais de uma tarefa ao mesmo tempo (ou para passar ao usuário a impressão de que o faz): são similares a diversos programas sequenciais funcionando em paralelo.

Funções são mecanismos de abstrações para processos; com variáveis não-locais, é possível construir fechos, que são abstrações de processos e dados a eles relacionados; continuações são a abstração da noção de "estado" de uma computação, e podem ser usadas para implementar abstrações para o comportamento de um programa na presença de falhas, computação não-determinística etc. Mecanismos para programação concorrente oferecem outra forma de abstração, que permite modelar diversas tarefas diferentes que são realizadas simultaneamente[1].

Programas concorrentes podem ser compostos de processos e threads.

> Um *processo* é um trecho de programa com um espaço de endereçamento próprio (não compartilhado com outros processos);
>
> Uma *thread* é um trecho de programa que compartilha o espaço de endereçamento com outras threads.

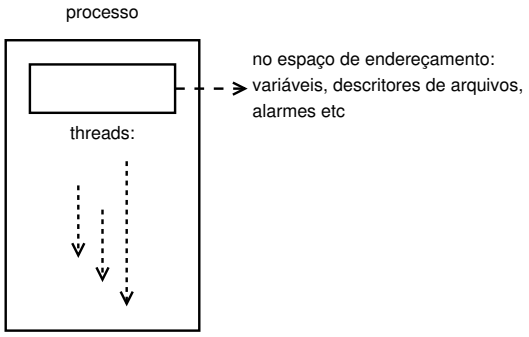
O leitor poderá encontrar definições ligeiramente diferentes de threads e processos: na terminologia tradicional de Sistemas Operacionais, um processo é um programa com todos os seus recursos (espaço de endereçamento, arquivos abertos, alarmes etc) – e uma thread é uma "linha de controle" de um processo[2]. O sistema operacional então mantém vários processos (programas) em execução, e cada programa pode ter mais de uma thread ("linha de controle"). Um componente do sistema operacional chamado de *escalonador*[3] reveza as threads de todos os processos nas CPUs disponíveis.

1 Alguns mecanismos para programação concorrente foram abordados superficialmente no Capítulo 10, na discussão de aplicações de continuações.

2 No entanto, com o surgimento do sistema Linux esta distinção deixou de ser tão clara.

3 "Scheduler" em Inglês.

Versão Preliminar



Nossa discussão sobre concorrência não entrará nos detalhes da implementação de concorrência em sistemas operacionais; o leitor encontrará discussões mais profundas e detalhadas em livros sobre Sistemas Operacionais [Tan10; Sta07; Sil10]. Nos interessa apenas que threads compartilham (ao menos parcialmente) seus recursos (em especial o espaço de endereçamento), enquanto processos não o fazem. Usaremos então as seguintes abstrações:

> Uma *instrução atômica* é uma instrução que pode ser executada por um processo com a garantia de que não será executada parcialmente, de forma intercalada com outra.

> Um programa concorrente é a intercalação das instruções atômicas de um conjunto de programas sequenciais.

> Há uma entidade escalonadora que decide de qual processo será a próxima instrução atômica a ser executada; o programador não tem como influenciar esta entidade nas suas decisões.

Estas abstrações são úteis para modelar programas concorrentes executando em uma única CPU (com multitarefa), com múltiplas CPUs e também programas distribuídos.

Há três mecanismos importantes para o suporte a programação concorrente do ponto de vista do programador:

- *Criação de threads ou processos:* toda linguagem ou framework com suporte a programação concorrente oferece algum meio para criação de threads e processos;
- *Comunicação:* a criação de múltiplos processos e threads não é interessante se não puder haver alguma forma de comunicação entre eles. Isso normalmente é feito por regiões compartilhadas de memória ou por troca de mensagens;

Versão Preliminar

- *Sincronização:* embora não seja possível interferir diretamente no comportamento do escalonador, deve haver algum método para restringir, de maneira limitada, a ordem de execução das instruções atômicas. (Por exemplo, quando dois usuários pedem para comprar o último ingresso para um evento esportivo, as diferentes threads devem executar de maneira ordenada para que não aconteça de ambos obterem o ingresso, e nem da venda ser negada a ambos).

Linguagens e bibliotecas que suportam programação concorrente diferem em como implementam estes mecanismos. É comum classificar linguagens e *frameworks* para programação concorrente de acordo com seus mecanismos de comunicação e sincronização em duas grandes classes: a das linguagens com *memória compartilhada* e as que suportam *passagem de mensagens*.

14.1 CRIAÇÃO DE THREADS EM SCHEME

Para criação de threads em Scheme usaremos o procedimento `make-thread`, que cria uma thread a partir de um procedimento. A thread não é iniciada automaticamente – é preciso usar `thread-start!`.

```
(define x (make-thread (lambda ()
                         (display 'hello))))

(thread-start! x)
```

Para esperar até que uma thread termine usaremos `thread-join!`.

```
(thread-join! x)
```

O próximo exemplo mostra um procedimento `create-n-threads` que recebe um número `n` e um procedimento `proc`, e cria `n` threads diferentes, de forma que a `n`-ésima thread execute `(proc n)`.

Versão Preliminar

```
(define proc
  (lambda (n)
    (display "thread ")
    (display n)
    (newline)))

(define create-n-threads
  (lambda (n proc)
    (if (zero? n)
        '()
        (cons (make-thread (lambda ()
                             (proc n)))
              (create-n-threads  (- n 1) proc)))))

(define threads (create-n-threads 5 proc))
```

A variável `threads` agora contém uma lista de threads. A representação usada ao imprimí-la depende da implementação de Scheme.

```
threads
(#<thread: thread9> #<thread: thread10> #<thread: thread11>
#<thread: thread12> #<thread: thread13>)
```

Iniciamos todas as threads e esperamos que terminem:

```
(let ((started-threads (map thread-start! threads)))
  (for-each thread-join! started-threads))
thread 5
thread 4
thread 3
thread 2
thread 1
```

O procedimento `thread-sleep!` faz a thread chamadora dormir por determinado tempo. Se mudarmos o procedimento `proc` para incluir uma chamada a `thread-sleep!` como no próximo exemplo, notaremos que o *tempo de término* entre os `displays` cresce gradualmente. Como o tempo decorre linearmente e o tempo que cada thread dorme

Versão Preliminar

cresce com $2.5n^2$, o *intervalo* entre os `displays` também aumenta. Note também que a primeira thread termina primeiro, porque dorme menos.

```
(define proc
  (lambda (n)
    (thread-sleep! (/ (* n n) 2.5))
    (for-each display (list "thread " n " slept " (/ (* n n) 2.5)))
    (newline)))
```

```
thread 1 slept 0.4
thread 2 slept 1.6
thread 3 slept 3.6
thread 4 slept 6.4
thread 5 slept 10.0
```

Em pseudocódigo, usaremos a notação "thread_start! $(a, b, \cdots)$" para iniciar uma lista de threads $(a, b, \cdots)$ (e o mesmo para thread_join!).

A criação e *start* de muitas threads pode tornar o código confuso. Usaremos o seguinte procedimento para criar e iniciar n threads.

```
(define n-thread-start!
  (lambda args
    (map thread-start!
         (map make-thread args))))
```

Não criamos um procedimento `n-thread-join!` porque tentar usar `thread-join!` em várias threads repetidamente poderia causar problemas: estaríamos presumindo que nenhuma das threads depende de outra para terminar – de outra forma um deadlock seria possível, se tentarmos esperar as threads em uma ordem inapropriada.

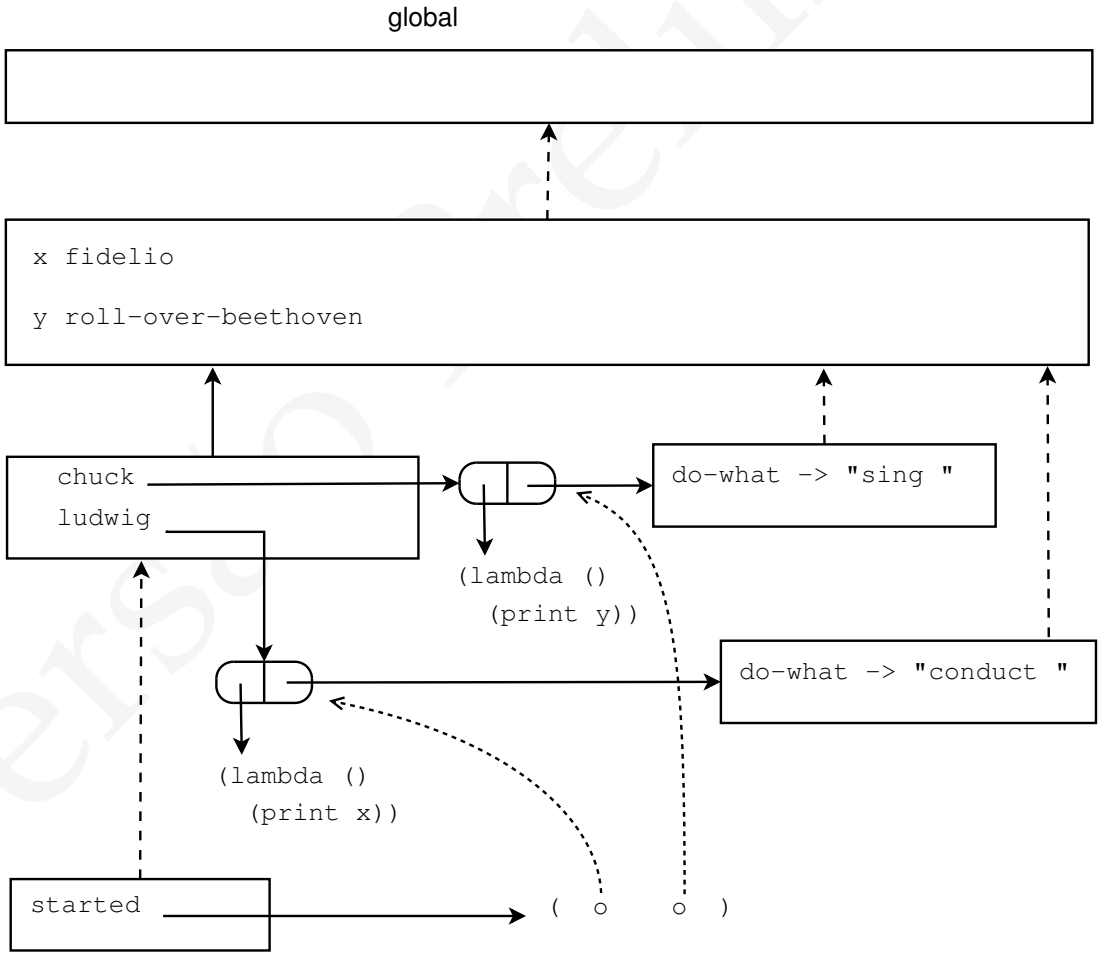
14.1.1 Ambientes de threads

Como Scheme adota escopo léxico, o ambiente da thread criada por `(make-thread proc)` é composto do ambiente local de `proc` e o ambiente onde `make-thread` foi chamado. Assim, threads diferentes podem compartilhar parte de seus quadros, como ilustrado pelo programa a seguir.

Versão Preliminar

```
(let ((x 'fidelio)
      (y 'roll-over-beethoven))

  (let ((chuck  (make-thread (lambda ()
                               (let ((do-what "sing "))
                                 (print do-what y)))))
        (ludwig (make-thread (lambda ()
                               (let ((do-what "conduct "))
                                 (print do-what x))))))

    (let ((started (map thread-start! (list chuck ludwig))))
      (for-each thread-join! (list ludwig chuck)))))
```

O diagrama de ambientes deste programa é mostrado a seguir. Usamos para threads a mesma representação que havíamos usado antes para procedimentos.



Cada uma das threads tem seu ambiente, com vínculos diferentes para `do-what`; mas compartilham o quadro do primeiro `let` com os vínculos para `x` e `y`.

14.2 COMUNICAÇÃO ENTRE THREADS

O compartilhamento de variáveis é uma forma de comunicação entre threads, que detalharemos no Capítulo 15. O uso de uma mesma variável por mais de uma thread gera a necessidade de mecanismos explícitos de sincronização que impeçam o acesso e modificação desordenados daquela variável.

Outra forma de comunicação entre threads é usando troca de mensagens: diferentes threads trocam mensagens, sem compartilhar variáveis.

Para ilustrar a diferença entre as duas abordagens, pode-se imaginar um contador que diferentes threads devem atualizar (um contador de acessos a algum recurso, por exemplo). Usando memória compartilhada, cada thread acessa diretamente o contador e o atualiza; as threads devem usar algum mecanismo de sincronização que garanta que o acesso será feito de forma a preservar a consistência dos dados. Já usando mensagens, o contador não é compartilhado – ele é acessado diretamente por apenas uma thread, e a sua atualização é feita através de troca de mensagens entre as threads.

14.3 PROBLEMAS INERENTES À PROGRAMAÇÃO CONCORRENTE

Esta Seção descreve alguns problemas relacionados à programação concorrente. Estes problemas serão retomados nos próximos Capítulos.

14.3.1 Corretude

Programas sequenciais se comportam de maneira determinística: iniciando em um mesmo estado, o programa sempre se comporta da mesma maneira e termina no mesmo estado final. Pode-se então reproduzir o comportamento de um programa sequencial e usar baterias de testes para detectar problemas nele.

Já um programa concorrente pode se comportar de maneira diferente cada vez que é executado, mesmo iniciando em um mesmo estado.

Não sendo possível usar testes para verificar programas concorrentes, é possível tentar provar formalmente a corretude dos algoritmos subjacentes a estes programas.

Versão Preliminar

Há dois tipos de propriedade interessantes relacionadas à corretude de programas concorrentes:

- *Safety (segurança);* uma propriedade deste tipo determina que o programa nunca entre em determinados estados. Por exemplo, um sistema que contabiliza créditos de telefonia implementa operações para adicionar crédito (feita quando o usuário compra mais créditos) e de descontar crédito (que acontece sempre que o usuário usa o telefone). Uma propriedade de segurança determinaria que a soma dos créditos comprados, usados e disponíveis deve ser zero (ou, equivalentemente, que as operações adicionar e descontar possam executar concorrentemente, *sempre* resultando no mesmo número de créditos que resultariam da execução de ambas uma após a outra, sem intercalação);
- *Liveness:* uma propriedade de *liveness* determina que certos estados do programa serão em algum momento alcançados. Por exemplo, um servidor *web* pode criar uma nova thread para cada conexão. Uma propriedade de *liveness* poderia ser a exigência de que cada thread consiga enviar ao cliente uma página HTML em algum momento (e não ter que esperar indefinidamente).

14.3.2 Dependência de velocidade

O tempo que cada instrução atômica executa em um processo varia de acordo com muitos fatores, como atividade de entrada e saída no computador, presença de outros processos e prioridade dada a cada processo. Não se pode determinar *a priori* quantas instruções atômicas um processo poderá executar antes que outro processo também o faça. Quando estas diferenças na velocidade de execução de diferentes processos causam diferenças no resultado esperado de uma computação, há uma *condição de corrida*.

No exemplo a seguir há uma variável global `saldo`, modificada por três threads concorrentemente. Definimos um procedimento `deposita`, que adiciona um dado valor a uma variável global `saldo`:

Procedimento deposita(valor)

```
s ← saldo
saldo ← s + valor
```

Criamos também o procedimento `repete_deposita`, que repete o procedimento `deposita` vinte vezes, adicionando 10 em cada vez:

Versão Preliminar

Procedimento repete_deposita

```
para todo 1 ⩽ i ⩽ 20:
    deposita 10
```

Finalmente, iniciamos tres threads diferentes, cada uma executando `repete_deposita`.

condicao_de_corrida

```
x ← make_thread repete_deposita
y ← make_thread repete_deposita
z ← make_thread repete_deposita
thread_start (x,y,z)
thread_join (x,y,z)
```

O valor de `saldo` após a execução dos três processos concorrentemente dependerá da velocidade de execução de cada processo, e poderá mudar cada vez que o programa for executado: apesar de sessenta depósitos de valor dez serem efetuados, o valor do saldo final dificilmente será igual a seiscentos.

A seguir traduzimos este exemplo para Scheme usando a SRFI 18. Usaremos `thread-sleep!` com um número aleatóreo como argumento nas threads que fazem depósitos. Desta forma aumentamos a probabilidade de que os resultados sejam diferentes cada vez que o programa for executado. Mesmo sem este artifício, o programa ainda seria suscetível à condição de corrida e o saldo seria muito provavelmente menor do que seiscentos.

O procedimento `deposita` é onde acontece a leitura e atribuição da variável global, e é apenas ali que a execução concorrente se torna um problema. Trechos de programa como este são chamados de *regiões (ou seções) críticas*.

```
(define saldo 0)

(define deposita
  (lambda (valor)
    (let ((s saldo))
      (thread-sleep! (* (random-real) 0.020))
      (set! saldo (+ valor s))
      (thread-sleep! (* (random-real) 0.070)))))

(define repete-deposita
  (lambda ()
    (do ((i 0 (+ 1 i)))
        ((= i 20))
```

Versão Preliminar

```
      (deposita 10))))

(define cria-threads
  (lambda (proc n)
    (if (zero? n)
        '()
        (cons (make-thread proc)
              (cria-threads proc (- n 1))))))

(let ((threads (cria-threads repete-deposita 3)))
    (for-each thread-start! threads)
    (for-each thread-join! threads)
    (display "Saldo final: ")
    (display saldo)
    (newline))
```

Um trecho de código que modifica variáveis compartilhadas entre threads é chamado de *seção crítica*.

Este exemplo expressa o problema através de um mecanismo de memória compartilhada, mas o mesmo problema pode ocorrer usando passagem de mensagens: basta que se modele a variável global como um processo e que o acesso a ela seja feito através de mensagens.

Normalmente o problema das seções críticas é resolvido impondo-se a exigência de que apenas uma thread acesse a variável de cada vez. A isso se dá o nome de *exclusão mútua* (é comum usar a contração *mutex*, do Inglês *"mutual exclusion"*).

Este mecanismo no entanto traz outros problemas, descritos na próxima seção.

14.3.3 Deadlocks

Quando um conjunto de processos fica impossibilitado de prosseguir executando, tem-se um *deadlock*. Os deadlocks acontecem quando processos diferentes tentam adquirir recursos gradualmente usando exclusão mútua e chegam a um impasse. A Figura 14.3.3 mostra a situação em que dois processos, A e B, tentam adquirir recursos X e Y (note que os recursos são liberados por cada um dos processos na ordem inversa em que foram adquiridos):

Versão Preliminar

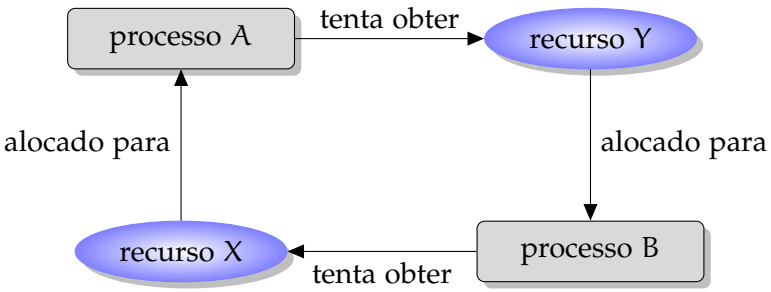


Figura 14.1.: O processo A espera pelo recurso Y, que está alocado para B. No entanto, B só liberará Y após obter X, que está alocado para A.

ProcessoA

```
adquire_recurso X
adquire_recurso Y
...
libera_recurso Y
libera_recurso X
```

ProcessoB

```
adquire_recurso Y
adquire_recurso X
...
libera_recurso X
libera_recurso Y
```

exemplo_deadlock

```
a ← make_thread processo_A
b ← make_thread processo_B
thread_start (a,b)
thread_join  (a,b)
```

Este programa, quando executado, entrará em deadlock, e nenhuma mensagem será mostrada ao usuário.

Uma maneira de implementar acesso exclusivo de threads a recursos é usando *mutexes*. Um mutex é um objeto que pode ser "travado" por uma thread; quando duas threads tentam obter o mesmo mutex, uma delas permanece bloqueada até que a outra libere o mutex.

O programa Scheme a seguir, por exemplo, sempre entrará em deadlock. Os procedimentos `make-mutex`, `mutex-lock!` e `mutex-unlock!` são usados para criar, travar e liberar mutexes.

Versão Preliminar

```
(define make-conta
  (lambda (v)
    (cons v (make-mutex))))

(define valor car)
(define set-valor! set-car!)
(define lock-conta cdr)

(define-syntax with-account
  (syntax-rules ()
    ((_ r body ...)
     (begin
       (mutex-lock! r)
       body ...
       (mutex-unlock! r)))))

(define retira
  (lambda (a v)
    (set-valor! a (- (valor a) v))))

(define deposita
  (lambda (a v)
    (set-valor! a (+ (valor a) v))))

(define transfere
  (lambda (a b v)
     (with-account (lock-conta a)
       (thread-sleep! 0.200)
       (with-account (lock-conta b)
         (retira a v)
         (deposita b v)))))

(define conta-a (make-conta 0))
(define conta-b (make-conta 0))

(let ((threads (map make-thread
                    (list (lambda ()
```

Versão Preliminar

```
                          (transfere conta-a conta-b 10))
                        (lambda ()
                          (transfere conta-b conta-a 5))))))
  (for-each thread-start! threads)
  (for-each thread-join! threads))

(print (valor conta-a) "; " (valor conta-b) "\n")
```

Neste exemplo, os recursos são contas bancárias. Para transferir um valor de A para B, uma thread obtém um mutex lock em A e depois tenta obter para B. Outra thread já obteve o mutex lock para B e então aguarda para obter o lock de A.

Pode-se *prevenir* ou *resolver* deadlocks: para prevení-los é necessário eliminar pelo menos uma das condições para sua existência:

- *Exclusão mútua:* um recurso pode ser alocado para no máximo um processo de cada vez;
- *Aquisição gradual de recursos:* um processo pode obter um recurso e manter sua posse enquanto requisita outros;
- *Não-preempção:* Um processo não terá um recurso tomado sem tê-lo liberado;
- *Espera circular:* há n processos $P_1, P_2, \cdots, P_n$ tal que cada processo $P_i \quad (i < n)$ espera por algum recurso alocado para o processo P_{i+1}, e P_n espera por um recurso alocado para P_1.

Pode-se ainda deixar a cargo do sistema que, antes de alocar um recurso a um processo, verifique que esta alocação não resultará em deadlock.

Para resolver deadlocks, é necessário detectá-los e eliminá-los. A eliminação de deadlocks normalmente se dá escolhendo um processo que já adquiriu alguns recursos e forçando-o a liberá-los.

14.3.4 *Starvation*

Se um processo fica indefinidamente esperando para executar, diz-se que ele "morre de fome" (*starvation*). Para que isto não ocorra,

- Não deve haver a possibilidade de deadlock;
- O escalonamento dos processos deve ser justo: embora cada processo possa ter uma prioridade diferente, o escalonador deve garantir que todos os processos executarão periodicamente.

Versão Preliminar

O exemplo a seguir mostra um sistema de emissão de tickets (poderiam ser passagens aéreas, entradas para cinema, teatro ou show – não importa). Há duas categorias: os VIPs tem preferência sobre os comuns, e sempre que houver um pedido de ticket VIP na fila, ele será atendido antes do pedido de ticket comum. Se a emissão de tickets for suficientemente demorada e os pedidos suficientemente frequentes, a fila de VIPs nunca estará vazia, e a fila de comuns nunca andará. O programa simula esta situação com três threads: uma que emite tickets, removendo pedidos de duas filas, e duas outras que produzem os pedidos e os incluem em duas filas (`vip` e `comum`). O programa emite tickets comuns com numeração negativa e tickets VIP com numeração positiva. O leitor pode verificar que a fila de comuns não anda, e nenhum número negativo é mostrado.

```
(load "queues.scm")

(begin (define filas (make-mutex))
       (define fila-vip (make-q))
       (define fila-comum (make-q)))

(define emite
  (lambda (ticket)
    (thread-sleep! 0.100)
    (print "Emitindo ticket: " ticket)))

(define print-loop
  (lambda ()
    (mutex-lock! filas)
    (cond ((not (empty-q? fila-vip))
           (emite (dequeue! fila-vip))
           (mutex-unlock! filas)
           (print-loop))
          ((not (empty-q? fila-comum))
           (emite (dequeue! fila-comum))
           (mutex-unlock! filas)
           (print-loop))
          (else
           (mutex-unlock! filas)
           (print-loop)))))

(define produz-vip
```

Versão Preliminar

```
  (let ((num 0))
    (lambda ()
      (set! num (+ 1 num))
      (mutex-lock! filas)
      (enqueue! num fila-vip)
      (mutex-unlock! filas)
      (produz-vip))))

(define produz-comum
  (let ((num 0))
    (lambda ()
      (set! num (- num 1))
      (mutex-lock! filas)
      (enqueue! num fila-comum)
      (mutex-unlock! filas)
      (produz-comum))))

(let ((threads (list (make-thread print-loop)
                     (make-thread produz-vip)
                     (make-thread produz-comum))))
  (let ((started (map thread-start! threads)))
    (thread-join! (car started))))
```

14.4 DOIS PROBLEMAS TÍPICOS

Aqui são descritos dois problemas clássicos de programação concorrente. Há muitos outros problemas típicos nos livros de Gregory Andrews [And99] e de Allen Downey [Dow09] (este último mostra para cada problema uma solução usando semáforos).

14.4.1 Produtor-consumidor

Um processo produz dados sequencialmente, mas sem periodicidade definida, e os deixa em um *buffer* de tamanho limitado. Outro processo usa estes dados, e os descarta após o uso. Quando o *buffer* está cheio, somente o processo consumidor pode atuar; quando o *buffer* está vazio, somente o produtor pode prosseguir. A Figura **??** ilustra esta situação. É necessário que ambos usem um mecanismo de sincronização para acessarem o *buffer* sem

Versão Preliminar

que os dados sejam incorretamente modificados e também para que cada processo saiba se pode ou não prosseguir. A próxima figura ilustra esta situação: as esferas representam itens de dados produzidos pelo processo `produtor`, e a fila onde estão (ilustrada como uma esteira) tem tamanho limitado (já há tres itens na fila, e só há espaço para mais um).

O produtor e o consumidor executam os laços simples mostrados a seguir.

Procedimento produtor

```
repita sempre:
    e ← gera_evento
    adiciona buffer, e
```

Procedimento consumidor

```
repita sempre:
    e ← retira buffer
    processa e
```

14.4.2 Jantar dos Filósofos

Para ilustrar o problema da sincronização entre processos, Edsger Dijkstra descreveu uma situação em que cinco processos tentam adquirir cinco recursos compartilhados. Para descrever esta situação usa-se como analogia um jantar de cinco filósofos. Os cinco filósofos sentam-se a uma mesa redonda, cada um com um prato à sua frente, e entre cada dois pratos há um garfo (a Figura 14.2 mostra um diagrama da mesa). Os filósofos só conseguem comer com dois garfos de cada vez, e portanto não há garfos para todos; quando um filósofo não estiver comendo, permanecerá pensando. Encontrar um algoritmo para permitir que todos os filósofos comam sem que haja deadlock ou starvation é um bom exercício, embora aparentemente um exercício inútil e distante da realidade[4]. Os problemas enfrentados e as técnicas utilizadas são os mesmos presentes na atividade de programação concorrente "no mundo real".

4 A analogia talvez não tenha sido a melhor possível – não há notícia de muitas pessoas que comam macarrão com dois garfos, menos ainda de pessoas que compartilhem garfos com dois vizinhos durante uma refeição. Pode-se ainda questionar se os filósofos nunca se saciarão, ou se não são capazes de pensar enquanto comem, por exemplo. No entanto, por mais alheia que seja à realidade (ou talvez por isso), a analogia foi bem-sucedida, uma vez que consta de quase todo livro a respeito de programação concorrente.

Versão Preliminar

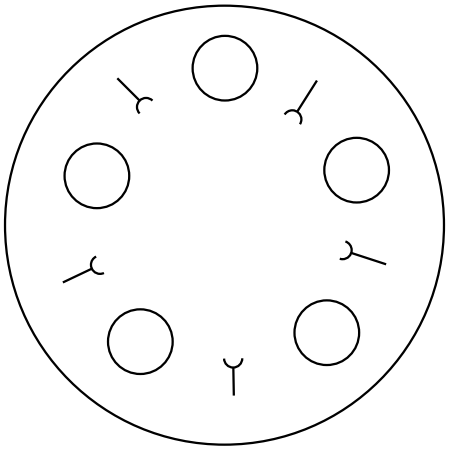


Figura 14.2.: O jantar dos filósofos.

EXERCÍCIOS

Ex. 181 — Conserte os programas das seções 14.3.2, 14.3.3 e 14.3.4.

Versão Preliminar

Versão Preliminar

15 MEMÓRIA COMPARTILHADA

Quando a comunicação entre processos se dá por memória compartilhada é necessário usar algoritmos para garantir que seja possível acessar variáveis compartilhadas sem condições de corrida. Há trechos de código que realizam acesso a variáveis compartilhadas entre threads de tal maneira que se mais de uma thread executar tal trecho, uma condição de corrida pode levar as variáveis a um estado inconsistente. Estes trechos de código são chamados de *seções críticas*.

SRFI-18

Neste Capítulo faremos uso da SRFI 18, que propõe uma API para uso de threads em Scheme.

15.1 TRAVAS (*locks*) DE EXCLUSÃO MÚTUA

Uma solução para exclusão mútua é exigir que cada thread que pretenda usar um recurso compartilhado adquira uma trava, ou *lock* antes de acessar o recurso, e a libere logo depois:

```
(define deposita
  (lambda (conta valor)
    (mutex-lock! (get-mutex conta))
    (set-saldo! conta (+ (saldo conta) valor))
    (mutex-unlock! (get-mutex conta))))
```

Os procedimentos `mutex-lock!` e `mutex-unlock!` seriam usados para travar e destravar o acesso à variável conta: quando uma thread executa `(lock! x)`, ela passa a deter o acesso à trava `x`, e outras threads só conseguirão acesso a esta trava quando a primeira thread liberá-la com `mutex-unlock!`

`Mutex-lock!` e `mutex-unlock!` são grafados com o ponto de exclamação no final porque necessariamente causam efeito colateral, modificando o estado (oculto) de alguma estrutura.

Há diferentes maneiras de implementar as operações para adquirir e liberar locks. Os procedimentos `mutex-lock!` e `mutex-unlock!` podem ser construídos usando diferentes algoritmos – por exemplo o de Peterson e o de Lamport – mas é mais comum que sejam implementados mais diretamente com suporte de hardware.

Versão Preliminar

Em Scheme, a SRFI 18 define os seguintes procedimentos para criar e usar mutexes:

- `(make-mutex)` aloca e retorna um novo mutex destravado;
- `(mutex? obj)` determina se um objeto é um mutex;
- `(mutex-state m)` retorna o estado do mutex `m`, que pode ser:
  - Um objeto do tipo thread, que neste caso será a thread que detém o mutex (e a única que pode destravar o mutex);
  - `not-owned` se o mutex está travado mas pode ser destravado por qualquer thread;
  - `abandoned` se foi abandonado (foi travado por uma thread que terminou sem liberá-lo);
  - `not-abandoned` se está liberado.
- `(mutex-lock! m [timeout [thread]])` tenta travar o mutex `m`. Se o mutex já está travado, a thread dorme até que possa adquirí-lo ou até que `timeout` se esgote. Quando o timeout se esgota, o procedimento retorna `#f`; quando o mutex é adquirido com o parâmetro `thread` igual a `#t`, o mutex fica no estado travado mas sem dono (`not-owned`); de outra forma o mutex fica travado por `thread`.
- `(mutex-unlock! m [condition-variable [timeout]])` Destrava o mutex `m`. O parâmetro adicional `condition-variable` será detalhado na seção sobre variáveis de condição.

Para atomicamente tentar bloquear o mutex e retornar `#t` em caso de sucesso ou `#f` em caso de falha, podemos usar o timeout igual a zero:

Versão Preliminar

```
(define m (make-mutex))

(for-each thread-start!
          (map make-thread
               (list (lambda ()
                       (mutex-lock! m)
                       (thread-sleep! 1)
                       (mutex-unlock! m)
                       (print 'releasing...))
                     (lambda ()
                       (thread-sleep! 0.5)
                       ;; will try and fail:
                       (if (mutex-lock! m 0)
                           (print 'I-GOT-THE-MUTEX!)
                           (print 'sorry-no-deal))))))
```

```
sorry-no-deal
releasing...
```

Os mutexes definidos pela SRFI 18 não são recursivos: uma thread não pode bloquear duas vezes o mesmo mutex. Se tentar fazê-lo, ficará bloqueada na segunda tentativa até que outra thread libere o mutex.

```
(define m (make-mutex))
(mutex-lock! m 0)
#t
(mutex-state m)
#<thread: primordial>
(mutex-lock! m 0)
#f
```

15.1.1 Discussão

O tempo usado por threads para adquirir e liberar locks é chamado de *overhead de lock*.

- A disputa por um recurso pode bloquear um número excessivo de threads;

Versão Preliminar

- O overhead de lock existe mesmo quando a probabilidade de conflito entre threads é muito baixa, e pode levar a um problema de eficiência;
- O uso de locks tem como efeito colateral a possibilidade de deadlocks.

Uma thread está em *espera ocupada* quando, ao invés de ser interrompida para ceder seu tempo a outra enquanto espera, permanece executando um laço à espera de que uma condição se modifique[1].

15.2 VARIÁVEIS DE CONDIÇÃO

Há situações em que além de garantir exclusão mútua entre threads, precisamos evitar que threads permaneçam em espera ocupada. Nestas situações usaremos variáveis de condição.

Usaremos novamente como exemplo o problema produtor/consumidor. Um servidor que processa requisições em um buffer poderia usar o algoritmo a seguir para consumir requisições.

Procedimento queue_lock

```
repita sempre:
    lock queue_lock
    se há requisições no buffer:
        r ← desenfilera buffer
        trata r
    unlock queue_lock
```

Enquanto não há requisições disponíveis, este algoritmo permanecerá iterando e verificando se o buffer está vazio ou não, e desta forma consumirá tempo de processamento que poderia ser usado por outros processos. A versão Scheme deste algoritmo é mostrada a seguir.

1 A espera ocupada com *spinlocks* é útil na implementação de sistemas operacionais, em situações em que cada thread permanece muito pouco tempo na seção crítica, e desta forma o tempo que outras threads permanecerão em espera ocupada é pequeno. Em outras situações a espera ocupada pode causar problemas, e deve-se optar por outro mecanismo.

Versão Preliminar

```
(load "queues.scm")

(define lock (make-mutex))
(define q (make-q))

(define produce
  (let ((i 0))
    (lambda ()
      (let ((x (list i (* 2 i))))
        (mutex-lock! lock)
        (enqueue! x q)
        (mutex-unlock! lock)
        (set! i (+ 1 i))
        (produce)))))

(define consume
  (lambda ()
    (let wait-for-it ()
      (mutex-lock! lock)
      (if (empty-q? q)
          (begin
            (mutex-unlock! lock)
            (wait-for-it))))
    (let ((z (dequeue! q)))
      (display z)
      (newline))
    (mutex-unlock! lock)
    (consume)))

(thread-join! (car (n-thread-start! produce consume)))
```

Idealmente, esta thread deveria ser bloqueada quando o buffer estivesse vazio e acordada novamente somente quando houver requisições no buffer.

Variáveis de condição permitem bloquear threads até que alguma condição seja satisfeita – por exemplo, que um buffer não esteja vazio, ou que uma requisição seja feita. Uma thread pode *esperar* por uma variável de condição; quando o fizer, será bloqueada até que outra thread sinalize a variável.

Versão Preliminar

Usamos mutexes para conseguir exclusão mútua entre threads, e variáveis de condição para que as threads coordenem seu trabalho

No próximo exemplo servidor inicia obtendo o mutex `mutex_buffer_nao_vazio`, e em seguida esperando que a variável de condição associada a este mutex seja sinalizada. Se o servidor precisar chamar `wait`, *o mutex será automaticamente liberado* quando esta thread for bloqueada. Quando a thread do servidor for desbloqueada, *ela automaticamente readquirirá o mutex*.

Procedimento consumidor

```
repita sempre:
    lock mutex_buffer_nao_vazio
    enquanto não buffer_nao_vazio:
        wait condvar_buffer_nao_vazio

    r ← desenfilera buffer
    trata r
    se não há mais requisições:
        buffer_nao_vazio ← F

    unlock mutex_buffer_nao_vazio
```

As threads que produzem requisições e as enfileram no buffer sinalizam a variável `buffer_nao_vazio` para que alguma thread consumidora possa processar a requisição.

Procedimento produtor

```
lock mutex_buffer_nao_vazio
enfilera buffer req
buffer_nao_vazio ← V
sinaliza buffer_nao_vazio
unlock mutex_buffer_nao_vazio
```

Implementaremos agora uma solução para o problema do produtor/consumidor usando variáveis de condição. Em Scheme usaremos variáveis de condição como definidas na SRFI 18.

O procedimento `make-condition-variable` retorna uma nova variável de condição; o predicado `condition-variable?` verifica se um objeto é variável de condição. Para sinalizar uma variável de condição usamos `condition-variable-signal!`.

Para esperar por uma condição chamamos `mutex-unlock!` com um terceiro argumento: `(mutex-unlock! mutex variavel)` libera `mutex` e bloqueia a thread até que `variavel` seja sinalizada.

Versão Preliminar

A SRFI 18 dá como exemplo de uso típico de variáveis de condição o trecho de código a seguir:

```
(let loop ()
    (mutex-lock! m)
    (if (condition-is-true?)
        (begin
          (do-something-when-condition-is-true)
          (mutex-unlock! m))
        (begin
          (mutex-unlock! m cv)
          (loop))))
```

Neste trecho de código, após a thread adquirir um mutex e verificando que uma condição é verdadeira, realiza algo e libera o mutex. Se a condição não for verdadeira, a thread *não* tenta imediatamente repetir o processo: ela pede para ser avisada por outra thread quando tiver que tentar novamente – e aí sim, volta ao início do laço. A chamada de `(mutex-unlock! m cv)` libera o mutex `m` para que outra thread possa trabalhar e no futuro sinalizar esta thread; ao mesmo tempo em que libera o mutex, a thread bloqueia a si mesma, esperando pela variável de condição `cv`.

Para implementarmos a solução do problema produtor/consumidor Usaremos a implementação de fila da Seção 3.3.3.

```
(load "queues.scm")
```

Criamos uma fila `q` para servir de buffer, um mutex e uma variável de condição.

```
(define q (make-q))
(define lock (make-mutex))
(define not-empty (make-condition-variable "not-empty"))
```

O produtor cria listas da forma (i 2i), e segue incluindo itens no buffer e incrementando i.

Após criar um item, adquire o mutex, enfilera o item, e sinaliza a variável de condição `not-empty`, desbloqueando a thread consumidora se ela estiver bloqueada. Logo em seguida, libera o mutex, incrementa seu contador e volta ao início do laço.

Versão Preliminar

```
(define produce
  (let ((i 0))
    (lambda ()
      (let ((x (list i (* 2 i))))
        (mutex-lock! lock)
        (enqueue! x q)
        (condition-variable-signal! not-empty)
        (mutex-unlock! lock)
        (set! i (+ 1 i))
        (produce)))))
```

A thread consumidora começa adquirindo o mutex. Em seguida verifica se a fila está vazia – se estiver, chama `mutex-unlock!`, liberando o mutex para o produtor *e bloqueando a si mesma até que* `not-empty` seja sinalizada. Ao ser acordada, readquire o mutex e verifica se a fila ainda está vazia.

Quando a fila não estiver vazia, o consumidor retira um item, mostra, e libera o mutex. Depois recomeça o laço.

```
(define consume
  (lambda ()
    (let wait-for-it ()
      (mutex-lock! lock)
      (if (empty-q? q)
          (begin
            (mutex-unlock! lock not-empty)
            (wait-for-it))))
    (let ((z (dequeue! q)))
      (display z)
      (newline))
    (mutex-unlock! lock)
    (consume)))
```

Agora basta que criemos as threads produtora e consumidora.

```
(n-thread-join! (n-thread-start! produce consume))
```

Os procedimentos que usamos permitem fazer uso básico de variáveis de condição. Há também outros, que são listados aqui sem exemplo de uso.

Versão Preliminar

`make-condition-variable` pode aceitar um parâmetro opcional com um nome para a variável de condição; `(condition-variable-name)` retorna o nome de uma variável de condição.

Variáveis de condição podem conter memória local. `(condition-variable-specific condition-variable)` obtém o objeto guardado na memória local da variável de condição; `(condition-variable-specific-set! condition-variable obj)` guarda o objeto na memória local da variável de condição.

`(condition-variable-broadcast! condition-variable)` sinaliza *todas* as threads que aguardam uma condição.

15.2.1 Encontro (rendez-vous) de duas threads

Um algoritmo concorrente exige que duas threads A e B se encontrem (façam um *rendez-vous*): há um ponto no algoritmo de A e outro no algoritmo de B até onde ambas devem chegar, para só então poder continua executando. Por exemplo um jogo de ação pode manter duas threads executando: uma prepara o cenário do próximo quadro enquanto outra mostra o cenário atual na tela. A primeira a terminar deve esperar a outra.

No pseudocódigo a seguir a thread A não pode executar a_3 antes que B tenha executado b_2:

Thread A	Thread B
a_1	b_1
a_2	•
•	b_2
a_3	

O ponto marcado nas duas threads é um *ponto de encontro*, e queremos então algum mecanismo que possa ser usado nesse ponto de encontro em cada thread que só permita que a thread continue se a outra também já chegou ali. Este mecanismo será um par de fechos, cada um usado em uma das threads.

Usaremos um mutex `m` para ambas as threads; para cada uma delas, teremos também uma variável booleana indicando se já chegaram ao ponto de encontro (`A_arrived`, `B_arrived`) e uma variável de condição usada para que a primeira thread a chegar espere pela segunda (`A_condvar`, `B_condvar`).

Versão Preliminar

Thread A

```
a1
a2
A_arrived ← t
signal A_condvar
lock m
se not B_arrived
    wait B_condvar m
a3
```

Thread B

```
b1
B_arrived ← t
signal B_condvar
lock m
se not A_arrived
    wait A_condvar m
b2
```

Quando a primeira thread chega ao ponto de encontro, ela imediatamente modifica o seu estado indicando que já chegou. Em seguida avisa a outra thread e adquire o mutex. Depois disso verifica se a outra chegou, e caso isso não tenha ocorrido, destrava o mutex e espera que a outra thread a acorde.

O procedimento `make-rendez-vous` retorna uma lista com os dois procedimentos a serem usados nas duas threads.

Versão Preliminar

```
(load "boxing.scm")

(define make-rendez-vous
  (lambda ()
    (define lock (make-mutex))
    (define cv-a (make-condition-variable))
    (define cv-b (make-condition-variable))
    (define a-arrived (box #f))
    (define b-arrived (box #f))

    (define a-meets-b
      (lambda (i-arrived he-arrived my-cv his-cv)
        (setbox! i-arrived #t)
        (condition-variable-signal! my-cv)
        (mutex-lock! lock)
        (if (not (unbox he-arrived))
            (mutex-unlock! lock his-cv)
            (mutex-unlock! lock))))

    (list (lambda () (a-meets-b a-arrived b-arrived
                                cv-a cv-b))
          (lambda () (a-meets-b b-arrived a-arrived
                                cv-b cv-a)))))
```

A lista retornada por `make-rendez-vous` tem dois procedimentos, a serem usados por duas threads diferentes. As duas threads sincronizarão na posição do código onde usarem estes procedimentos.

Os dois procedimentos retornados são construídos chamando `a-meets-b`: na segunda vez, com argumentos trocados. `A-meets-b` constrói a lógica do encontro da mesma forma que no pseudocódigo.

As duas threads que usaremos para ilustrar um encontro listam os números de zero a nove, esperam uma pela outra, e só então identificam-se para o usuário.

Versão Preliminar

```
(define thread-a
  (lambda ()
    (do ((i 0 (+ i 1)))
        ((= i 10))
      (thread-sleep! 0.1)
      (print i))
    (rendez-vous-a)
    (print "------> A")))

(define thread-b
  (lambda ()
    (do ((i 0 (+ i 1)))
        ((= i 10))
      (thread-sleep! 0.1)
      (print i))
    (rendez-vous-b)
    (print "------> B")))
```

Agora usamos `make-rendez-vous` para criar os dois procedimentos `rendez-vous-a` e `rendez-vous-b`, iniciamos as duas threads e as iniciamos:

```
(begin
  (define rendez-vous-list (make-rendez-vous))
  (define rendez-vous-a (car rendez-vous-list))
  (define rendez-vous-b (cadr rendez-vous-list)))

  (thread-join! (car (n-thread-start! thread-a
                                      thread-b)))
0
0
1
1
...
8
8
9
9
----> B
```


Versão Preliminar

```
----> A
```

15.3 SEMÁFOROS

Semáforos, inventados por Edsger Dijkstra [Dij65], podem ser vistos como uma generalização da ideia de mutex locks: enquanto um lock permite que uma thread tenha exclusividade de acesso a um recurso, um semáforo determina um número máximo de threads que poderão utilizar um recurso de cada vez[2].

Um semáforo tem como componentes um número inteiro e dois procedimentos, chamados de P e V[3] (ou `signal` e `wait`), usados para controlar o acesso a um recurso.

O número no semáforo representa a quantidade de threads que ainda pode conseguir acesso ao recurso. O procedimento `wait` de um semáforo é chamado por threads que queiram acesso ao recurso, e o procedimento `signal` é usado para liberar o acesso ao recurso.

Quando uma thread chama `signal`, o contador interno do semáforo é incrementado de uma unidade.

Quando uma thread chama `wait`, o contador é verificado. Se este for igual a zero, a thread deverá dormir até que o contador seja maior que zero. Se o contador for maior que zero, ele é diminuído e a thread ganha acesso ao recurso compartilhado.

A SRFI 18 não inclui procedimentos para uso de semáforos, mas mostra a implementação de um semáforo como exemplo, usando mutexes e variáveis de condição, que é reproduzido a seguir.

Procedimento wait(sem)

```
lock sem.mutex
se sem.contador > 0:
    sem.contador ← sem.contador −1
    unlock sem.mutex
senao:
      unlock sem.mutex
      cond_wait sem.cond
```

2 Ou ainda, mutex locks são um caso particular de semáforo onde o número de threads que pode adquirir o recurso é um.

3 Os nomes P e V foram dados por Dijkstra, usando a primeira letra de palavras em Holandês que descrevem as operações: V para *verhogen* (incremente) e P para a palavra inventada *prolaag*, que teria o significado de "tente decrementar" *(probeer te verlagen)*.

versão preliminar

Procedimento signal(sem)

```
lock sem.mutex
sem.contador ← sem.contador +1
se sem.contador > 0:
    cond_broadcast sem.cond
```

```
(define (make-semaphore n)
  (vector n (make-mutex) (make-condition-variable)))

(define (semaphore-wait! sema)
  (mutex-lock! (vector-ref sema 1))
  (let ((n (vector-ref sema 0)))
    (if (> n 0)
        (begin (vector-set! sema 0 (- n 1))
               (mutex-unlock! (vector-ref sema 1)))

        (begin (mutex-unlock! (vector-ref sema 1)
                              (vector-ref sema 2))
               (semaphore-wait! sema)))))

(define (semaphore-signal-by! sema increment)
  (mutex-lock! (vector-ref sema 1))
  (let ((n (+ (vector-ref sema 0) increment)))
    (vector-set! sema 0 n)
    (if (> n 0)
        (condition-variable-broadcast! (vector-ref sema 2)))
    (mutex-unlock! (vector-ref sema 1))))
```

Semáforos podem ser implementados como bibliotecas, como parte de linguagens de programação ou como funções internas de um sistema operacional (os sistemas operacionais do tipo Unix implementam semáforos e expõem sua API para usuários, por exemplo).

O nível de abstração dos semáforos é relativamente baixo, exigindo que o programador se lembre de liberar todos os semáforos que adquirir, e garanta que cada thread adquirirá e liberará semáforos de forma ordenada a fim de evitar deadlocks. Há ideias de nível de abstração conceitual mais alto, como monitores e memória transacional, que liberam o programador de parte destes problemas. A linguagem Java, por exemplo, permite declarar

Versão Preliminar

que o acesso a certos componentes do programa deve ser sincronizado; este mecanismo é implementado internamente com semáforos.

15.3.1 Exemplo de uso: *rendezvous*

Podemos usar dois semáforos para implementar um encontro entre duas threads, com o mesmo efeito do rendezvous que implementamos com mutexes e variáveis de condição. Usaremos os dois semáforos para indicar que as duas threads já chegaram ao ponto de encontro. A thread que chegar primeiro informa a outra (chamando `signal` em seu próprio semáforo) e espera que o semáforo da outra seja incrementado:

Thread A	Thread B
a_1	b_1
a_2	signal_sema bOK
signal_sema aOK	wait_sema aOK
wait_sema bOK	b_2
a_3	

Em Scheme podemos novamente escrever um procedimento que cria pares de procedimentos para rendezvous. Desta vez o fecho conterá apenas os dois semáforos, e os procedimentos são mais simples porque a complexidade foi abstraída no mecanismo do semáforo.

```
(define make-sema-rendezvous
  (lambda ()
    (let ((a-ok (make-semaphore 0))
          (b-ok (make-semaphore 0)))
      (list (lambda ()
              (semaphore-signal-by! a-ok 1)
              (semaphore-wait! b-ok))
            (lambda ()
              (semaphore-signal-by! b-ok 1)
              (semaphore-wait! a-ok))))))
```

Podemos também modificar diretamente as threads e incluir os signal e wait:

Versão Preliminar

```
(define a-OK (make-semaphore 0))
(define b-OK (make-semaphore 0))

(define thread-a
  (lambda ()
    (do ((i 0 (+ i 1)))
        ((= i 10))
      (thread-sleep! 0.05)
      (print 'a #\space i))
    (semaphore-signal-by! a-OK 1)
    (semaphore-wait! b-OK)
    (print "------>>> A")))
```

Apesar da simplicidade da solução com semáforos, é normalmente possível conseguir uma solução mais eficiente para um problema usando variáveis de condição e mutexes. Semáforos normalmente são melhores em situações onde há a necessidade de controlar contadores.

15.3.2 Exemplo: produtor-consumidor

A solução que apresentamos para o problema do produtor/consumidor com mutexes e variáveis de condição não leva em conta o tamanho do buffer. Na verdade, presumimos que ele é ilimitado – ou que é grande o suficiente para nunca se esgotar. Da mesma forma que aquela implementação força o consumidor a dormir quando o buffer está vazio, seria interessante forçar o produtor a dormir se o buffer estiver cheio. Isso pode também ser resolvido com mutexes e variáveis de condição (veja o Exercício 184), mas como agora precisaremos de um contador de recursos, será natural usarmos semáforos.

Usaremos um mutex e dois semáforos: um para indicar a quantidade de lugares e outro para indicar o número de itens disponíveis. Inicialmente, o número de itens é zero, e o número de lugares é igual ao tamanho do buffer:

---

mutex ← make_mutex
itens ← Semaphore(0)
lugares ← BUFFER_SIZE

O produtor espera que haja um lugar no buffer, espera até que o buffer esteja disponível, inclui um item, libera o buffer e finalmente sinaliza que há mais um item disponível:

Versão Preliminar

Procedimento produtor – com sincronização

```
repita sempre:
    e ← gera_evento
    wait_sema lugares
    lock mutex
    add buffer, (e)
    unlock mutex
    signal_sema itens
```

O consumidor espera até que haja itens no buffer e decrementa o contador de itens, adquire acesso exclusivo ao buffer e retira um item. Depois sinaliza o semáforo `lugares`, indicando que agora há mais um lugar disponível:

Procedimento consumidor – com sincronização

```
repita sempre:
    wait_sema itens
    lock mutex
    e ← retira buffer
    unlock mutex
    signal_sema lugares
    processa e
```

Traduziremos agora estes algoritmos para Scheme. Nosso buffer terá tamanho igual a quatro, e será representado usando nossa implementação de fila.

```
(begin
  (define buffer-size 4)
  (define lock    (make-mutex))
  (define lugares (make-semaphore buffer-size))
  (define itens   (make-semaphore 0))
  (define q (make-q)))
```

Os produtor, mostrado a seguir, é uma tradução simples do pseudocódigo mostrado antes. Cada vez que um item é inserido no buffer um asterisco é mostrado (usaremos isto para verificar que o buffer ficou cheio).

Versão Preliminar

```
(define produce
  (let ((i 0))
    (lambda ()
      (let ((x (list i (* 2 i))))
        (semaphore-wait! lugares)
        (mutex-lock! lock)
        (enqueue! x q)
        (mutex-unlock! lock)
        (semaphore-signal-by! itens 1)
        (set! i (+ 1 i))
        (print '*)
        (produce)))))
```

O consumidor retira itens e os mostra:

```
(define consume
  (lambda ()
    (semaphore-wait! itens)
    (thread-sleep! 0.3)
    (mutex-lock! lock)
    (let ((z (dequeue! q)))
      (mutex-unlock! lock)
      (semaphore-signal-by! lugares 1)
      (print z)
      (consume))))
```

Como usamos `thread-sleep!` para tornar a thread consumidora mais lenta que a produtora o buffer ficará cheio, como podemos notar na saída do programa: quatro asteriscos são mostrados (um para cada item enfileirado), e o quinto somente é mostrado depois que o primeiro item é processado. Deste ponto em diante, o buffer ficará sempre com três ou quatro itens, porque a thread consumidora é muito lenta. Cada vez que ela consome um item, a produtora rapidamente enche o buffer novamente com outro.

```
(thread-join! (car (n-thread-start! consume produce)))
*
*
*
*
(0 0)
*
```


Versão Preliminar

```
(1 2)
*
(2 4)
*
...
```

15.3.3 Exemplo: jantar dos filósofos

Há duas soluções muito simples e elegantes para o jantar dos filósofos, baseadas em duas observações:

- Se apenas quatro filósofos comerem de cada vez, não haverá deadlock nem starvation;
- Se dois dos filósofos tentarem obter os garfos em ordem oposta, não haverá deadlock nem starvation.

Usando a primeira observação, podemos dar a cada garfo um semáforo binário e usar também um semáforo `podem_comer` para indicar quem pode tentar comer.

Procedimento pega-garfos() com semáforos: solução com quatro comendo de cada vez

```
repita sempre:
    wait podem_comer
    wait garfo[esquerda i]
    wait garfo[direita i]
```

Procedimento deixa-garfos() com semáforos: solução com quatro comendo de cada vez

```
repita sempre:
    signal podem_comer
    signal garfo[esquerda i]
    signal garfo[direita i]
```

A segunda solução impõe que filósofos diferentes usem ordens diferentes para adquirir os garfos (o uso de assimetria na ordem de aquisição dos recursos é uma técnica comum no desenvolvimento de algoritmos concorrentes com recursos compartilhados):

Versão Preliminar

Procedimento pega-garfos() com semáforos: solução com ordem na aquisição de garfos

```
repita sempre:
    se i é par:
        wait garfo[esquerda i]
        wait garfo[direita i]
    senão
        wait garfo[direita i]
        wait garfo[esquerda i]
```

15.4 TRAVA PARA LEITORES E ESCRITOR

Quando uma estrutura pode ser lida por várias threads mas modificada por apenas uma, pode ser interessante usar um mecanismo que permita que vários leitores acessem uma estrutura simultaneamente, mas que dê acesso exclusivo a um escritor.

Faremos uma analogia com uma sala onde as threads entram: quando a primeira leitora entra, ela liga a luz. Quando a última leitora sai, ela apaga a luz. A escritora só entrará na sala quando a luz estiver apagada.

A primeira thread leitora bloqueará o acesso da thread escritora; usaremos um contador para lembrar quantas threads leitoras estão ativas. Quando este contador for zero (ou seja, quando a última thread sair), o acesso é liberado para a thread escritora.

Assim, o escritor só precisa esperar até que a sala esteja vazia. Quando estiver, ele tranca a porta (`writer_enter sala`) e entra em sua região crítica. Ao sair, ele abre a porta (`writer_leave sala`).

Procedimento escritor

```
repita sempre:
    e ← gera_evento
    lock sala.writer_lock
    add buffer, (e)
    unlock sala.writer_lock
```

O leitor só precisa chamar `reader_enter` e `reader_leave` ao entrar e sair da região crítica, passando para estas funções o semáforo que indica que há alguém ali dentro:

Versão preliminar

Procedimento leitor

```
repita sempre:
    reader_enter sala
    leia buffer
    reader_leave sala
```

A implementação de `reader_enter` é simples: a função obtém acesso exclusivo, incrementa o contador (ela é "mais uma" thread na sala) e verifica se o contador agora é um. Se for, ela é a primeira thread – e neste caso deve "acender a luz", indicando à escritora que agora a sala está ocupada:

Procedimento reader_enter(sala)

```
lock sala.reader_lock
    cont ← cont +1
    se cont = 1:
        lock sala.writer_lock
    unlock sala.reader_lock
```

Ao sair, a leitora chamará `reader_leave`, que verifica se é a última. Se for, ela "apaga a luz", indicando à escritora que agora não há mais leitoras na sala:

Procedimento reader_leave(sema)

```
wait sala.reader_lock
    cont ← cont −1
    se cont = 0:
        unlock sala.writer_lock
    unlock sala.reader_lock
```

Este algoritmo supõe que o escalonamento é justo, e que, havendo competição entre as threads leitoras e a escritora para entrar na sala, a escritora em algum momento conseguirá o mutex lock e entrará. Mesmo assim, é possível que haja uma quantidade de leitoras muito grande, de forma que a thread escritora espere tempo demais para entrar (ou que nunca entre). Isso pode ser resolvido dando prioridade à escritora no momento da requisição do mutex lock `sala`. A implementação desta variação do *lightswitch* fica como exercício.

Versão Preliminar

```
(define make-readers-writer-lock
  (lambda ()
    (let ((readers 0)
          (writer-lock (make-mutex))
          (reader-lock (make-mutex)))

      (define reader-enter (lambda ()
                             (mutex-lock! reader-lock)
                             (set! readers (+ readers 1))
                             (if (= readers 1)
                                 (mutex-lock! writer-lock))
                             (print "readers: " readers)
                             (mutex-unlock! reader-lock)))

      (define reader-leave (lambda ()
                             (mutex-lock! reader-lock)
                             (set! readers (- readers 1))
                             (if (= readers 0)
                                 (mutex-unlock! writer-lock))
                             (mutex-unlock! reader-lock)))

      (define writer-enter (lambda ()
                             (mutex-lock! writer-lock)))

      (define writer-leave (lambda ()
                             (mutex-unlock! writer-lock)))

      (list reader-enter
            reader-leave
            writer-enter
            writer-leave))))
```

Usaremos esta trava para controlar threads que acessam um vetor: uma delas escolhe aleatoreamente duas posições do vetor, incrementa uma e decrementa outra. Outras duas threads leem o vetor e calculam a média de seus elementos. Como a thread gravadora não deve mudar a média, as leitoras sinalizarão um erro se o valor da média mudar.

A gravadora trabalhará em intervalos de pelo menos 0.008 segundos:

Versão Preliminar

```
(define writer
  (lambda ()
    (thread-sleep! 0.008)
    (write-enter!)
    (print "writing!")
    (let ((i (random-integer 10))
          (j (random-integer 10)))
      (vector-set! v i (+ (vector-ref v i) 1))
      (vector-set! v j (- (vector-ref v j) 1)))
    (print "done writing...")
    (write-leave!)
    (writer)))
```

As leitoras trabalham em intervalos mais rápidos (0.001 segundos):

```
(define reader
  (lambda ()
    (thread-sleep! 0.001)
    (reader-enter!)
    (let ((avg (/ (vector-fold (lambda (i acc x)
                                 (+ acc x))
                               0 v)
                  10.0)))
      (if (not (= avg 50.0))
          (error "Vector average is not 50? " avg v)))
    (reader-leave!)
    (reader)))
```

Criamos agora duas threads leitoras e uma gravadora:

Versão Preliminar

```
(let ((procs (make-readers-writer-lock))
      (v (make-vector 10 50)))
  (define reader-enter! (list-ref procs 0))
  (define reader-leave! (list-ref procs 1))
  (define write-enter!  (list-ref procs 2))
  (define write-leave!  (list-ref procs 3))

  (define writer ...)
  (define reader ...)

  (thread-join! (car (apply n-thread-start! (list writer
                                                  reader
                                                  reader)))))
```

```
readers: 1
readers: 1
writing!
done writing...
readers: 1
readers: 1
readers: 1
readers: 2
readers: 1
readers: 1
...
```

Em alguns momentos há uma thread leitora trabalhando, em outros há duas, mas nunca há leitoras quando a gravadora detém o lock (nunca há nada entre as linhas "`writing!`" e "`done writing...`"). O leitor pode tentar mudar os valores usados nas duas chamadas a `thread-sleep!` e a quantidade de leitoras para verificar o comportamento do programa.

15.4.1 Mais sobre semáforos

Semáforos são pouco usados em aplicações concorrentes, mas muito usados em núcleos de sistemas operacionais e outros programas concorrentes de baixo nível. Uma excelente exposição de muitos padrões de uso de semáforos (tanto os mais comuns como diversos usos menos conhecidos) é o livro de Allen Downey, "The Little Book of

Versão Preliminar

Semaphores" [Dow09]. O livro de Gregory Andrews [And99] também aborda o uso e implementação de semáforos.

15.5 BARREIRAS

Pode ser necessário que várias threads façam um único rendezvous em um certo momento – nenhuma thread pode passar de um certo ponto no código até que todas tenham chegado. A isso chamamos de barreira.

Podemos usar barreiras para implementar, por exemplo, programas que precisam atualizar objetos em uma tela – diferentes threads trabalham em diferentes objetos, e somente após todas chegarem a uma barreira cada uma delas desenha seu objeto.

Há diferentes maneiras de implementar barreiras; mostraremos aqui como implementá-las usando semáforos. Podemos usar um contador de threads que já tenham chegado à barreira; um *mutex lock* `m` para controlar o acesso ao contador; e um semáforo "barreira", que inicia com zero e somente receberá um `signal` quando a última thread chegar à barreira:

```
contador ← 0
m ← semaforo 1
barreira ← semaforo 0
```

Cada thread então executa:

```
antes da barreira

wait m
contador ← contador + 1
signal m

se contador = max
      signal barreira

wait barreira
signal barreira

depois da barreira
```

O procedimento `make-barrier` produz fechos: ele retorna um procedimento que pode ser usado em uma barreira para `n` threads. Como o procedimento a ser usado é idêntico para todas as threads, apenas um é retornado.

Versão Preliminar

```
(define make-barrier
  (lambda (n)
    (let ((count 0)
          (m (make-mutex))
          (barrier (make-semaphore 0)))
      (lambda ()
        (mutex-lock! m)
        (set! count (+ count 1))
        (mutex-unlock! m)
        (if (= count n)
            (semaphore-signal-by! barrier 1))

        (semaphore-wait! barrier)
        (semaphore-signal-by! barrier 1)))))
```

Criamos agora uma barreira para quatro threads.

```
(define barrier (make-barrier 4))
```

Agora criaremos quatro threads: cada uma conta de zero a nove, em velocidades diferentes, depois aguardam em uma barreira e finalmente se identificam.

```
(define make-writer
  (lambda (n)
    (lambda ()
      (do ((i 0 (+ i 1)))
          ((= i 10))
        (thread-sleep! (* n 0.1))
        (print n '--> i))
      (barrier)
      (print "------>>> " n))))
```

Mesmo com a diferença de velocidades (forçada aqui com `thread-sleep!`), as threads terminam suas contagens e identificam-se juntas, porque tiveram que aguardar na barreira:

Versão Preliminar

```
(define writers (list (make-writer 0)
                      (make-writer 1)
                      (make-writer 2)
                      (make-writer 3)))

(thread-join! (car (apply n-thread-start!
                          writers)))
```

```
0->0
0->1
0->2
0->3
0->4
0->5
0->6
0->7
0->8
0->9
1->0
...
3->8
3->9
---->>> 3
---->>> 0
---->>> 1
---->>> 2
```

15.6 MONITORES

Semáforos são uma primitiva de sincronização importante e muito versátil (o livro de Allen Downey, citado na Seção sobre semáforos, demonstra soluções para uma vasta gama de problemas de sincronização usando apenas semáforos) – no entanto, são uma primitiva "de baixo nível": as chamadas a `wait_sema` e `signal_sema` podem ter que ser feitas de maneira pouco simples, e o esquecimento ou uso incorreto de uma delas pode causar problemas que só se revelam quando um sistema já está em produção. Além disso,

Versão Preliminar

estas primitivas estão relacionadas ao funcionamento interno dos semáforos, e não ao problema que o programador está atacando; elas não expressam claramente sua função no programa.

Os monitores foram propostos por Tony Hoare e Per Brinch Hansen[Hoa74; Bri75] como uma alternativa de mais alto nível aos semáforos. Toda linguagem de programação de alto nível oferece mecanismos de abstração para processos e para dados. Monitores são uma única forma de abstração para dados, processos e concorrência.

Um monitor consiste de variáveis, procedimentos e variáveis de condição. As variáveis somente são visíveis a partir dos procedimentos do monitor, e somente um procedimento do monitor pode estar executando em cada momento (como se houvesse um mutex lock para acesso ao monitor)[4]

Além de prover automaticamente um mecanismo de exclusão mútua através da exigência de que somente uma thread por vez execute qualquer procedimento, os monitores também suportam sincronização através de suas variáveis de condição, que somente são acessíveis a procedimentos do monitor. Assim, a exclusão mútua em um monitor é *implícita*, porque segue sempre o mesmo padrão, mas a sincronização é *explícita*, porque é diferente para cada caso de uso de monitor.

Variáveis de condição em monitores funcionam da mesma maneira como foram descritas na Seção 15.2, mas é importante observar que: uma thread que detenha o monitor, ao chamar `wait K`, desocupa temporariamente o monitor e permite que outras threads o ocupem. A thread que chamou `wait` ficará em uma fila associada com a condição K; quando K passar a ser verdade, a primeira thread da fila voltará a ocupar o monitor.

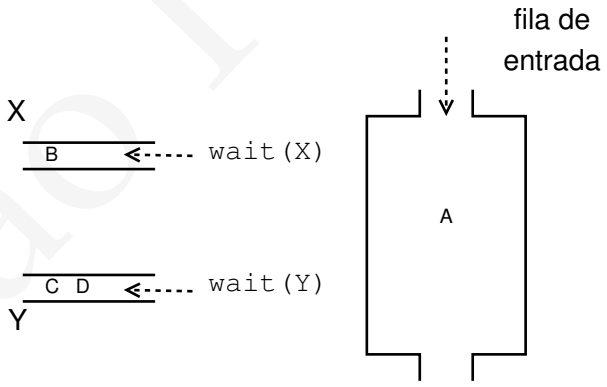


Figura 15.1.: Um monitor.

4 São semelhantes a classes em linguagens orientadas a objetos, mas onde o acesso aos atributos dos objetos é sempre feito apenas por métodos da classe, e somente por uma única thread de cada vez – e não havia originalmente o conceito de herança para monitores.

Versão Preliminar

A Figura 15.1 ilustra o funcionamento de um monitor: a thread *A* detém (sozinha) o monitor; as threads B, C e D já adquiriram o monitor em algum momento, mas executaram `wait` e agora aguardam pelas variáveis de condição X e Y.

Quando uma thread chama `signal K`, ela pode continuar executando ou bloquear. Se continuar executando, a próxima thread da fila de K será a primeira a executar depois dela.

Um exemplo simples de monitor sem variáveis de condição é mostrado a seguir. Este

```
monitor conta
  var:
    real saldo

  procedure:
    deposit v
      saldo = saldo + v

    withdrawal v
      saldo = saldo - v
```

O programador não precisa ocupar-se com exclusão mútua, uma vez que `deposit` e `withdrawal` nunca poderão executar ao mesmo tempo para uma mesma instância deste monitor.

15.6.1 Disciplina de sinalização

A descrição original de monitores determinava que quando uma thread chama `signal`, ela deve ser bloqueada e aguardar para adquirir novamente o monitor. Ela pode ou não ter prioridade sobre as outras threads que tentam entrar no monitor.

Versão Preliminar

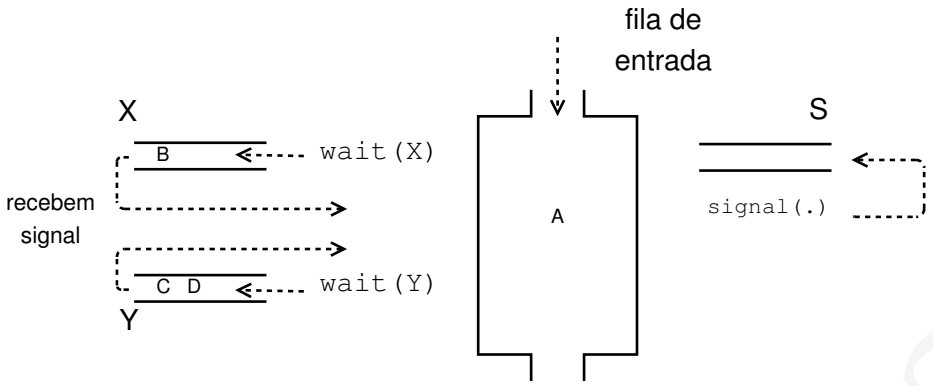


Figura 15.2.: Um monitor com variáveis de condição bloqueantes.

Na Figura 15.2, a thread A ocupa o monitor e as threads B, C e D esperam por variáveis de condição. Se A chamar `signal Y`, uma das threads na fila Y acordará e passará imediatamente a ocupar o monitor; a thread A ficará então em uma fila S, que tem prioridade sobre a fila de entrada.

Pode-se modificar a ideia original de monitores para permitir que uma thread continue executando após chamar `signal`. Neste caso, as threads que são liberadas para adquirir o monitor passam para a fila de entrada. O diagrama da Figura 15.3 mostra um monitor onde `signal` não bloqueia: quando A chamar `signal Y`, C e D voltarão para a fila de entrada, e A continuará executando.

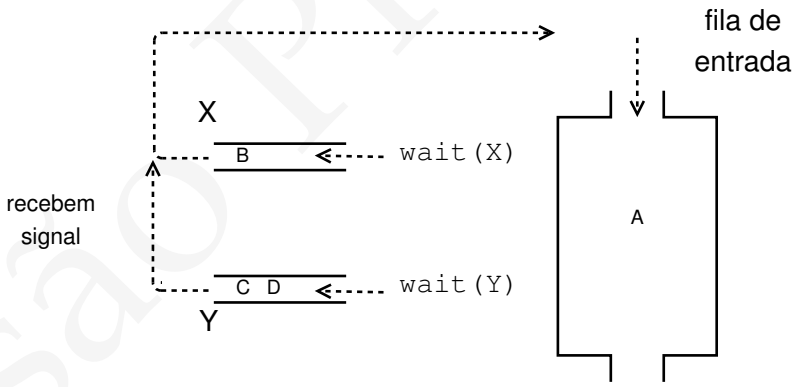


Figura 15.3.: Um monitor com variáveis de condição não bloqueantes.

15.6.2 Em Scheme

Não implementaremos monitores em Scheme. Os monitores tem duas características importantes: uma é a de abstração (exclusão mútua implícita e agrupamento de dados,

Versão Preliminar

variáveis de condição e procedimentos relacionados); a outra é a de isolamento (somente procedimentos do monitor podem acessar seus dados e variáveis de condição – e procedimentos de um monitor não podem acessar variáveis externas a ele). O leitor pode, como exercício, usar tipos definidos pelo usuário, fechos e macros para implementar monitores.

15.6.3 Exemplo: produtor-consumidor

Um monitor deverá encapsular o buffer e um contador que indicará a quantidade de itens no buffer. Além disso, usará também duas variáveis de condição, `cheio` e `vazio`.

A função `monitor/adiciona` verifica se o buffer já está cheio (MAX é o tamanho do buffer); se estiver, espera pela variável de condição `cheio`. Em seguida, adiciona o elemento ao buffer e incrementa o contador. Se o valor do contador, após o incremento, tiver o valor um, então o buffer estava vazio antes da inserção – e neste caso, chama `signal vazio` para indicar à thread consumidora que o buffer não está mais vazio.

Procedimento monitor/adiciona(e)

```
se contador = MAX:
    wait cheio

adiciona buffer e
contador ← contador +1
se contador = 1:
    signal vazio
```

A função `monitor/remove` espera até que haja itens no buffer, remove um item e decrementa o contador. Se o contador, após o decremento, tiver o valor $MAX - 1$, então o buffer estava cheio antes da remoção. Neste caso `signal cheio` é chamado para indicar à thread produtora que já há espaço no buffer para novos itens.

Procedimento monitor/remove

```
se contador = 0
    wait vazio

e ← remove buffer
contador ← contador −1
se contador = MAX − 1
    signal cheio
```

O código do produtor e do consumidor fica mais simples, uma vez que a complexidade do gerenciamento do buffer foi isolada no monitor:

Versão Preliminar

Procedimento produtor – com monitor

```
repita sempre:
    e ← gera_evento
    monitor/adiciona e
```

Procedimento consumidor – com monitor

```
repita sempre:
    e ← monitor/remove
    processa e
```

15.6.4 Exemplo: barreira

Um monitor pode implementar uma barreira, como a já mostrada com semáforos. O monitor inicia um contador com zero e o incrementa cada vez que uma thread chama um método `monitor/espera_barreira`:

monitor/espera_barreira

```
se contador = MAX:
    para toda thread esperando por barreira:
      signal barreira
senão:
    contador ← contador +1
    wait barreira
```

Cada thread chama `monitor/espera_barreira` na posição do código onde deve haver a sincronização. As $n-1$ primeiras threads chamarão `wait` e esperarão; quando a n-ésima thread chegar, chamará `signal` e acordará todas as outras[5].

O algoritmo a seguir mostra uma thread onde a barreira está entre as instruções a_2 e a_3:

```
a1
a2
monitor/espera_barreira
a3
```

5 Em Java o laço "para todo" não é necessário: basta usar `notifyAll` ao invés de `notify`

Versão Preliminar

15.6.5 Exemplo: jantar dos filósofos

Uma solução simples para o problema do jantar dos filósofos usando monitores é emular semáforos com um monitor por garfo. Assim é possível implementar as duas soluções já vistas para o problema (a solução com quatro filósofos comendo e a solução com filósofos assimétricos).

15.6.6 Monitores em Java

A classe `Object` em Java implementa um monitor – e portanto qualquer objeto Java pode ser usado como monitor. Uma thread que chama o método `wait` de um objeto é bloqueada imediatamente e colocada em uma fila de threads associada àquele objeto. Quando outra thread chama o método `notify` deste mesmo objeto, *uma* das threads em sua fila é acordada. Se o método `notifyAll` de um objeto é chamado, *todas* as threads da fila de um objeto são acordadas.

Em Java não há variáveis de condição. Cada objeto tem uma única fila, e os métodos `wait` e `notify` operam sobre esta fila.

15.7 MEMÓRIA TRANSACIONAL

Usar mutexes, semáforos e monitores de maneira eficiente e correta é difícil: a eficiência de um programa concorrente aumenta com granularidade de locks mais fina, mas com isso também aumenta a complexidade da tarefa do programador.

Sistemas gerenciadores de bancos de dados permitem que diversas consultas (tanto de leitura como de escrita) sejam feitas simultaneamente. Cada consulta é desenvolvida sem que o programador tenha que se preocupar com outras consultas sendo feitas simultaneamente. Isto é possível porque o programador pode marcar sequências de comandos em sua consulta que devem ser executadas de maneira indivisível. Estas sequências de comandos são chamadas de *transações*. O programador poderia marcar o início e o fim de uma transação, por exemplo, com `begin-transaction` e `end-transaction`:

Versão Preliminar

```
BEGIN-TRANSACTION
 s ← GET saldo, conta_a
 j ← GET saldo, conta_b
 PUT saldo_a: saldo_a - valor
 PUT saldo_b: saldo_b + valor
END-TRANSACTION
```

Não haverá condição de corrida quando diversas transações executarem, porque o efeito será semelhante ao da execução da transação sem que outras executem simultaneamente. Em Bancos de Dados normalmente exigimos que transações apresentem quatro propriedades, conhecidas por *ACID:*

- *Atomicidade:* uma transação pode atomicamente terminar e tornar suas modificações visíveis ao resto do sistema ou abortar. Se abortar não deve ter efeitos colaterais, deixando o sistema no mesmo estado em que ficaria se a transação não tivesse executado;
- *Consistência:* uma transação deve ter uma visão consistente dos dados durante toda a sua execução;
- *Isolamento:* as mudanças realizadas por uma transação não são visíveis fora dela até que consiga terminar com sucesso;
- *Durabilidade:* após o término com sucesso da transação, as mudanças feitas por uma transação não são perdidas.

Transações são ferramentas de abstração muito convenientes, e podem ser adaptadas para que possam ser usadas fora de sistemas gerenciadores de bancos de dados.

Sistemas que suportam memória transacional oferecem primitivas para iniciar e terminar transações com semântica semelhante à de *begin-transaction* e *end-transaction*, comuns em sistemas gerenciadores de bases de dados [Kni86; HM93]. Usando memória transacional podemos construir sistemas concorrentes sem qualquer preocupação com mecanismos complexos de sincronização, usando apenas transações. Em memória transacional não estamos interessados em todas as propriedades das transações em bancos de dados – na verdade os sistemas de memória transacional *não* oferecem durabilidade.

Tim Harris, James Larus e Ravi Rajwar enumeram [HLR10] algumas diferenças importantes entre transações em bancos de dados e em memória:

- O acesso a dados em um SGBD é lento, e portanto o tempo de processamento usado para gerenciar transações é irrelevante. Em memória transacional isto não é verdade;

Versão Preliminar

- Os dados armazenados em um SGBD são duráveis, mas em memória normalmente isto não é necessário;
- Sistemas de memória transacional são a transposição de mecanismos do contexto de SGBDs para o de Linguagens de programação, onde há uma plenitude de paradigmas e formas de abstração.

15.7.1 Memória transacional por software

Os primeiros sistemas propostos para memória transacional funcionavam em hardware. Em 1997 Shavit e Touitou propuseram implementar memória transacional sem suporte por hardware [ST97] e sem o uso de locks. Posteriormente trabalhos experimentais mostraram que o uso de locks torna algoritmos de memória transacional mais simples e eficientes.

Há muitas maneiras de implementar memória transacional por software, e grande parte delas funciona de maneira muito próxima do hardware. Implementaremos aqui uma variante simplificada do algoritmo TL2 [**tl2**] que não é muito eficiente mas ilustra o uso de memória transacional como abstração em programas concorrentes.

O TL2 determina que cada variável que possa ser modificada em transações tenha um número de versão e um mutex associado a ela, incrementado cada vez que seu valor é modificado, e que seja mantido um contador global, incrementado cada vez que qualquer variável é modificada.

Há dois algoritmos descritos no TL2: um para escrita e leitura de variáveis em transações onde há modificações de variáveis, e outro para leitura de variáveis em transações que não modificam variáveis.

- *Início:* Leia o valor do contador e guarde em uma variável local da transação `rv` ("read version");
- *Escrita:* escreva o valor em um conjunto temporário (não diretamente em seu lugar na memória);
- *Leitura (dentro da transação):* após ler uma variável, verifique se ela está no conjunto de escrita desta thread. Se estiver, retorne dali o valor. Caso contrário, verifique se sua versão é $\leqslant rv$ e se seu lock está liberado – caso contrário, aborte a transação.
- *Commit:*
  - Tente travar todas as variáveis do conjunto de escrita; se não conseguir, aborte;
  - Incremente o contador global de versões;

Versão Preliminar

- Valide o conjunto de leitura: para cada variável lida pela transação, verifique se sua versão é menor ou igual a `rv`. Caso uma delas não seja, aborte a transação. Verifique também se alguma delas está travada para leitura por outra thread. Se estiver, aborte a transação.
- Grave os valores das variáveis de escrita em seus locais definitivos e libere os locks.

Para transações que não modificam variáveis, é necessário apenas verificar se as variáveis lidas foram modificadas depois do início da transação, comparando a versão de cada uma no momento do commit com o valor do contador no início da transação. Se alguma variável tiver versão maior que a do contador, a transação deve abortar.

15.7.1.1 *Uma implementação em Scheme*

Implementaremos em Scheme um pequeno sistema de memória transacional baseado no TL2. Não se trata de uma implementação de alto desempenho, mas é útil como uma forma conveniente de abstração que nos libera do trabalho com mutexes, semáforos e variáveis de condição.

Uma implementação de Scheme com suporte a memória transacional[6] poderia oferecer a forma especial `atomically`:

```
(atomically
  (let ((x (standard-deviation vec))
        (y (* x z)))
    (set! a (+ a x))))
```

Esta forma Scheme descreveria uma transação que lê valores consistentes de `x`, `y`, `z` e `vec`, depois modifica e altera atomicamente o valor de `a`.

Outra possibilidade é exigir que o programador indique explicitamente as variáveis usadas em cada transação:

```
(let ((x #f)
      (y #f))
  (atomically-with (x y)
    (set! x (standard-deviation vec))
    (set! y (* x z))
    (set! a (+ a x))))
```

6 Independente de haver ou não suporte por hardware.

Versão Preliminar

A forma especial `atomically-with` marcaria as variáveis `x` e `y`, já existentes no contexto léxico daquela forma, de forma que todas as modificações nelas façam parte da transação, mas deixaria de fora a variável `a`.

Seria interessante também definir sintaxe para o caso em que as variáveis estão sendo introduzidas, com uma variante de `let`:

```
(atomically-let ((x (standard-deviation vec))
                 (y (* x z)))
  (set! a (+ a x)))
```

Não implementaremos formas especiais, mas sim algumas primitivas sobre as quais estas formas especiais podem ser implementadas:

- `(stm-start tr)` inicia uma transação;
- `(stm-read/w tr var)` lê o valor da variável `var`. Deve ser usado *dentro* da transação (a leitura será do valor interno, possivelmente modificado, de `x`);
- `(stm-write! tr var val)` escreve um novo valor na variável `val`. Evidentemente, também deve ser usado dentro da transação;
- `(stm-commit/w! tr)` termina uma transação que modificou variáveis;
- `(stm-commit/r tr)` termina uma transação que apenas leu variáveis.

As variáveis usadas em transações precisarão ser armazenadas em estruturas de dados especiais, por isso também implementaremos os procedimentos:

- `(make-transactional x)` cria uma estrutura de variável transacional com o valor inicial `x`;
- `(trans-value x)` lê o valor da variável transacional `x`; deve ser usado *fora* da transação e reflete a visão externa à transação.

Precisaremos passar valores por referência; usaremos nossa implementação de caixas:

```
(load "boxing.scm")
```

Para usar uma variável em uma transação usaremos o procedimento `make-transactional` que agrega à variável um número de versão e um mutex. A descrição original do TL2 mostra como usar uma única palavra de máquina para armazenar tanto o mutex como o número de versão, mas depende do uso de instruções assembly[7]. Nesta implementação usaremos um mutex e um inteiro.

7 O algoritmo usa `compare-and-swap` para mudar um bit que representa o mutex ou incrementar a versão, codificada nos outros bits.

Versão Preliminar

```
(define make-transactional
  (lambda (x)
    (list x 0 (make-mutex))))
```

Precisamos de procedimentos para extrair o valor, a versão e o mutex:

```
(define trans-value car)
(define trans-version cadr)
(define trans-mutex caddr)
```

Nosso algoritmo precisará também de primitivas para incrementar o número de versão e para obter uma lista com os números de versão de uma lista de variáveis.

```
(define trans-inc!
  (lambda (x)
    (set-car! (cdr x) (+ (cadr x) 1))))

(define trans-versions
  (lambda (set)
    (map cadr set)))
```

Também usaremos procedimentos para verificar se uma variável está travada para escrita, para tentar travar uma variável e para destravá-la:

```
(define trans-locked?
  (lambda (x)
    (let ((state (mutex-state (list-ref x 2))))
      (not (or (eq? state 'not-abandoned)
               (eq? state 'abandoned))))))

(define trans-try-lock!
  (lambda (x)
    (mutex-lock! (trans-mutex x) 0)))

(define trans-unlock!
  (lambda (x)
    (mutex-unlock! (trans-mutex x))))
```

Versão Preliminar

Como teremos que travar um conjunto inteiro de variáveis de cada vez, precisamos de um procedimento que tente fazê-lo uma variável por vez, retornando `#t` em caso de sucesso e `#f` quando algum dos mutexes não puder ser adquirido:

```
(define trans-try-lock-set!
  (lambda (set)
    (cond ((null? set)
           #f)
          ((trans-try-lock! (car set))
           (if (not (trans-try-lock-set! (cdr set)))
               (trans-unlock! (car set))
               #t))
          (else #f))))
```

Não há a possibilidade de deadlock, porque se `trans-try-lock-set!` não conseguir adquirir um dos mutexes ele *imediatamente* retornará `#f`.

Para iniciar uma transação simplesmente copiamos o valor do contador global:

```
(define start
  (lambda ()
    (set! rv (unbox counter-box))))
```

Para ler uma variável em uma transação onde há escritas,

```
(define trans-read/w
  (lambda (x)
    (if (or (> (trans-version x) rv)
            (trans-locked? x))
        (signal 'abort-transaction))

    (let ((x-in-write-set (assq x uncommited-writes)))
      (if (and x-in-write-set
               (not (null? (cdr x-in-write-set))))
          (cdr x-in-write-set)
          (trans-value x)))))
```

Para escrever em uma variável durante uma transação encontramos seu valor em `uncommited-writes` e o modificamos:

Versão Preliminar

```
(define trans-write!
  (lambda (x value)
    (let ((x-in-write-set (assq x uncommited-writes)))
      (set-cdr! x-in-write-set value))))


(define commit/w!
  (lambda ()
    (if (trans-try-lock-set! write-set)
        (begin (for-each trans-inc! write-set)
               (for-each (lambda (x) (set-car! (car x)
                                               (cdr x)))
                         uncommited-writes)
               (setbox! counter (+ 1 (unbox counter)))
               (for-each trans-unlock! write-set)
               #t)
        #f)))
```

A leitura em uma transação apenas de leitura precisa apenas verificar se as versões das variáveis lidas não são maiores do que o valor do contador no início da transação e que o lock da variável não está travado:

```
(define trans-read/r
  (lambda (x)
    (if (or (> (trans-version x) rv)
            (trans-locked? x))
        (signal 'abort-transaction)
        (trans-value x))))
```

Finalmente, o fecho com as variáveis `counter-box`, `rv`, `uncommited-writes`, `read-set` e `write-set`:

Versão Preliminar

```
(define make-transaction
  (lambda (counter-box read-set write-set)
    (let ((rv 0)
          (uncommited-writes (map list write-set)))

      (define start ...)
      (define trans-read/w ...)
      (define trans-read/r ...)
      (define trans-write! ...)
      (define commit! ...)

      (lambda (msg)
        (case msg
          ((start)     start)
          ((read/w)    trans-read/w)
          ((read/r)    trans-read/r)
          ((write!)    trans-write!)
          ((commit!)   commit!))))))
```

Definiremos procedimentos para tornar o uso do fecho mais conveniente:

```
(begin
  (define (stm-start  tr)        ((tr 'start)))
  (define (stm-read/w tr var)    ((tr 'read/w) var))
  (define (stm-write! tr var val) ((tr 'write!) var val))
  (define (stm-commit/w! tr)     ((tr 'commit/w!)))
  (define (stm-commit/r tr)      ((tr 'commit/r))))
```

Criaremos agora três variáveis transacionais `a`, `b` e `c`.

```
(begin
  (define counter (box 0))
  (define a (make-transactional 10))
  (define b (make-transactional 20))
  (define c (make-transactional 30)))
```

Definimos duas transações e as iniciamos.

Versão Preliminar

```
(define t (make-transaction counter
                            (list a b)
                            (list c)))

(define t2 (make-transaction counter
                             (list c)
                             (list)))

(stm-start t)
(stm-start t2)
(stm-read/w t b)
20
(stm-write! t c 50)
(trans-value c)
30
```

O comportamento do sistema está correto: como não fizemos ainda o commit, o valor de `c` visível fora da transação ainda é 30. Dentro da transação, no entanto, seu valor é 50:

```
(stm-read/w c)
50
```

Após o commit, o novo valor ficará visível fora da transação, e as versões de `c` e do contador serão incrementadas:

```
(stm-commit! t)
(trans-value c)
50
(trans-version c)
1
(unbox counter)
1
```

Como iniciamos a transação `t2` quando a versão de `c` era zero, não conseguiremos ler seu valor dentro da transação:

```
(stm-read/r t2 c)
Error: uncaught exception: abort-transaction
```

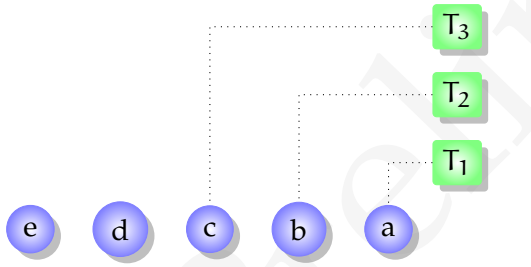
Versão Preliminar

15.8 THREAD POOLS

Quando threads vivem por pouco tempo e são criadas em grande quantidade, há uma sobrecarga de tempo e recursos relacionada ao processo de criação e destruição de threads. Uma técnica de implementação (ou "padrão de projeto") importante é o uso de *pools de threads*.

Um número de threads é criado; estas threads *não* executam diretamente código relacionado ao trabalho que deve ser realizado. Estas threads buscam itens de trabalho em uma fila e os executam. Um item de trabalho consiste de um procedimento e seus dados (em Scheme isto se traduz naturalmente para fechos).

Assim, uma mesma thread pode executar um procedimento A, retornar ao pool e mais tarde executar um procedimento B. A próxima figura ilustra um *pool* com tres threads (T_1, T_2 e T_3) e cinco tarefas (a, b, c, d e e). As tres primeiras tarefas já foram tomadas pelas tres threads; a primeira thread que terminar tomará a próxima tarefa da fila (d).



Nossa implementação de exemplo usa um mutex para controlar o acesso à fila de tarefas e um semáforo para contar o número de itens na fila (as threads trabalhadoras fazem um `semaphore-wait!` neste semáforo). Usaremos nossas implementações de fila e de semáforo:

```
(load "semaphore.scm")
(load "queues.scm")
```

Aparentemente isto pode levar a starvation da thread escritora, que poderia ser indefinidamente preterida quando da disputa pelo lock. No entanto, as leitoras no pool de threads também removem itens da fila – e quando a fila estiver vazia, a thread escritora necessariamente conseguirá o lock.

Para obter um item de trabalho, uma thread trabalhadora deve primeiro esperar até que haja itens disponíveis. Ela *não* pode excluir a thread produtora ainda, ou haveria um deadlock. Em seguida, havendo itens na fila, esta thread exclui a produtora e depois adquire exclusividade para obter o item de trabalho:

Versão Preliminar

```
(define get-work-item!
  (lambda ()
    (semaphore-wait! task-available)
    (mutex-lock! lock)
    (let ((item (dequeue! work-queue)))
      (mutex-unlock! lock)
      item)))
```

Depois de obter o item, a thread abre mão da exclusividade e libera o mutex.

Para adicionar itens de trabalho, a thread produtora adquire o mutex, adiciona o item na fila, e avisa as consumidoras que há mais um item disponível e permite que elas entrem:

```
(define add-work-item!
  (lambda (item)
    (mutex-lock! lock)
    (enqueue! item work-queue)
    (semaphore-signal-by! task-available 1)
    (mutex-unlock! lock)))
```

Um `worker` retira um item de trabalho da fila, chama `interage` e fecha a porta TCP. Em seguida, recomeça com outro item de trabalho. Se a fila de trabalho estiver vazia, o `worker` ficará bloqueado ao chamar `get-work-item!`.

```
(define worker
  (lambda ()
    (let loop ((item (get-work-item!)))
      (call/cc
       (lambda (k)
         (with-exception-handler
          (lambda (e)
            (print (thread-name (current-thread))
                   " got exception: " e "\n")
            (k #f))
          item)))
      (loop (get-work-item!)))))
```

O pool de threads é uma lista. Para cada elemento inicial da lista, o procedimento `init-proc` é chamado.

Versão Preliminar

```
(define make-list
  (lambda (n init-proc)
    (let loop ((i (- n 1))
               (l '()))
      (if (< i 0)
          l
          (loop (- i 1)
                (cons (init-proc) l))))))

(define make-pool
  (lambda ()
    (make-list num-threads
               (lambda ()
                 (make-thread worker)))))
```

O fecho descrito abaixo contém diversos pequenos procedimentos locais e um exportado (`add-work-item!`).

```
(define make-work-queue
  (lambda (num-threads)
    (let ((work-queue (make-q))
          (task-available (make-semaphore 0))
          (lock (make-mutex)))

        (define get-work-item! ...)
        (define add-work-item! ...)
        (define worker ...)
        (define make-pool ...)

        (for-each thread-start! (make-pool))
        add-work-item!))))
(define add-task! (make-work-queue 2))
(add-task! (lambda () (display 'hello) (newline)))
hello
```

O procedimento interno `worker` precisa usar `with-exception-handler` porque se uma exceção for levantada durante o período em que uma tarefa estiver sendo executada, a

Versão Preliminar

thread morrerá. Este mecanismo garante que haverá uma entrada nos logs e que a thread simplesmente abandonará a tarefa, mas continuará viva e retornará para o pool:

```
(add-task! (lambda ()
             (print 'ok)
             (error 'tragedy!)
             (print 'will-never-get-here)))
```

```
ok
thread #2 got exception: Error: tragedy!
```

Após mostrar a notificação de que houve uma exceção, `worker` continua vivo, e é possível continuar enviando tarefas para ele.

Se tivéssemos criado um worker mais simples como este:

```
(define worker
  (lambda ()
    (let loop ((item (get-work-item!)))
      (item)
      (loop (get-work-item!)))))
```

qualquer exceção ou erro produzido por `(item)` terminaria a thread, e as threads poderiam morrer uma a uma até que o pool se esvaziasse completamente.

Podemos querer obter o valor de retorno de uma tarefa enviada ao pool de threads, mas não podemos fazê-lo diretamente, porque o procedimento que usamos para enfileirar a tarefa não pode esperar até que a tarefa tenha terminado (se assim fosse, poderíamos dispensar as threads e executar a tarefa sequencialmente). Podemos, no entanto, usar memória compartilhada (como um fecho) para permitir que uma thread veja o resultado da computação de outra.

Versão Preliminar

```
(let ((result #f)
      (lock (make-mutex))
      (cv   (make-condition-variable)))
  (let ((produce-result (lambda ()
                          (thread-sleep! 1)
                          (set! result 'this-is-the-result)
                          (condition-variable-signal! cv)))
        (read-result    (lambda ()
                          (print "waiting for result...")
                          (mutex-unlock! lock cv)
                          (print result))))
    (mutex-lock! lock)
    (add-task! read-result)
    (add-task! produce-result)))
```

```
waiting for result...
this-is-the-result
```

15.8.1 Deadlocks e starvation

Um pool de threads traz uma nova possibilidade de deadlock: se todas as threads em execução no pool estiverem esperando por recursos que só podem ser liberados por threads que estão aguardando na fila teremos um deadlock.

Versão Preliminar

```
(define add (make-work-queue 2))

(define m (make-mutex))
(define cv (make-condition-variable))

(define acquire
  (lambda ()
    (print 'will-get)
    (mutex-unlock! m cv)
    (print 'got-it)))

(define free
  (lambda ()
    (condition-variable-signal! cv)))
(add acquire)
will-get
(add acquire)
will-get
(add free)
(add free)
```

As duas únicas threads executando no pool são `acquire`; as threads `free`, que foram incluídas depois, ficam esperando para executar. Temos um deadlock que não teríamos se tivéssemos iniciado todas as tarefas com `make-thread`, sem usar o pool de threads.

O programador deve garantir que este tipo de deadlock não acontecerá, usando com cautela variáveis de condição e ajustando o tamanho do pool de threads.

Além do deadlock que descrevemos, um pool de threads pode potencializar outros problemas como escalonamento injusto e livelock, caso muitas das threads em execução sejam lentas ou permaneçam muito tempo bloqueadas.

Pools de threads são úteis para processos curtos que não precisam aguardar uns pelos outros sincronamente – por exemplo, processos que atendam requisições em um servidor (HTTP, DNS ou de qualquer outro tipo).

Versão Preliminar

15.8.2 Exemplo: um servidor HTTP

O procedimento `copia` recebe uma porta de entrada, uma de saída, e copia linhas de uma para outra até que a leitura resulte em fim de arquivo. O procedimento `check-http` toma uma lista de strings (recebidas em uma linha) e verifica se elas podem ser uma requisição HTTP.

```
(define copia
  (lambda (in out)
    (let loop ( (linha (read-line in)) )
      (cond ((not (eof-object? linha))
             (display linha out)
             (newline out)
             (loop (read-line in)))))))

(define check-http
  (lambda (lista)
    (cond ((not (= (length lista) 3))
           (print "Not a real HTTP request")
           #f)
          ((not (string-ci= (car lista) "get"))
           (print "Request not understood: "
                  (car lista))
           #f)
          ((not (= (string-prefix-length "HTTP"
                                         (caddr lista))
                   4))
           (print "Not a real HTTP request")
           #f)
          (else
           (print "Serving " (cadr lista))
           #t))))
```

Quando o usuário tentar acessar URLs como `http://exemplo.com/`, o browser enviará a requisição "`GET /`". O procedimento `trata-caminho` transforma a string `"/"` em `"/index.html"` Outras regras de reescrita poderiam ser incluídas aqui.

Versão Preliminar

```
(define trata-caminho
  (lambda (str)
    (if (string=? str "/")
        "/index.html"
        str)))
```

O procedimento `send-ok-headers` envia o cabeçalho da resposta quando o servidor tiver encontrado o arquivo HTML a ser transferido:

```
(define send-headers
  (lambda (out)
    (display "HTTP/1.0 200 OK\n" out)
    (display "Content-Type: text/html\n" out)))
```

O procedimento `copia-arq` recebe um nome de arquivo e uma porta de saída, verifica se o arquivo existe, e envia a resposta adequada pela porta de saída.

```
(define copia-arq
  (lambda (name out)
    (let ((arquivo (string-append base name)))
      (if (file-exists? arquivo)
          (let ((porta-arq (open-input-file arquivo)))
            (send-headers out)
            (newline out)
            (copia porta-arq out)
            (close-input-port porta-arq))
          (let ((porta-arq (open-input-file
                            (string-append base "/"
                                           not-found-file))))
            (display "HTTP/1.0 404 Not Found\n" out)
            (display "Content-Type: text/html" out)
            (newline out)
            (newline out)
            (copia porta-arq out)
            (close-input-port porta-arq))))))
```

Um procedimento que interage com o usuário usando duas portas (entrada/saída):

Versão Preliminar

```
(define interage
  (lambda (in out)
    (let ((linha (string-tokenize (read-line in))))
      (if (check-http linha)
          (let ((url (trata-caminho (cadr linha))))
            (copia-arq url out))))
    (flush-output out)
    (close-input-port  in)
    (close-output-port out)))
```

SRFI-13

O servidor usa as SRFIs 18 e 13, além de procedimentos não padrão para acesso à rede. rede

```
(load "pool.scm")

(define web-host "127.0.0.1")
(define web-port 9008)
(define base "/home/jeronimo/web")
(define number-of-workers 4)
(define not-found-file "404.html")

(define  add-work-item! (make-work-queue number-of-workers))
```

O procedimento `trata` aceita conexão TCP e manda as portas de entrada e saída do socket para a fila de trabalho, e o procedimento `inicia-servidor` começa a ouvir em uma porta, passando para `trata` o socket.

```
(define trata
  (lambda (s)
    (let-values (((in out) (tcp-accept s)))
      (add-work-item! (lambda () (interage in out)))
    (trata s)))

(define inicia-servidor
  (lambda ()
    (let ((socket (tcp-listen web-port)))
      (trata socket))))

(inicia-servidor)
```

Versão Preliminar

15.8.3 Thread Pools em Java

A classe `java.util.concurrent.ThreadPoolExecutor` implementa pools de threads.

O exemplo minimalista a seguir mostra duas threads inseridas em um pool. Usamos apenas `java.util.Random` além das classes relacionadas a programação concorrente em `java.util.concurrent.*`:

```
import java.util.concurrent.*;
import java.util.Random;
```

Duas classes, `FazAlgo` e `FazOutraCoisa`, implementam a interface `Runnable`. Uma delas mostra `Hello` repetidamente, mas em intervalos aleatóreos de tempo, e a outra conta de um em um, mostrando os números também em intervalos aleatóreos.

```
class FazAlgo implements Runnable {
    public void run() {
        Random randomGenerator = new Random();
        Thread eu = Thread.currentThread();
        while(true) {
            System.out.println("Hello!");
            try {
                eu.sleep(randomGenerator.nextInt(500));
            }
            catch(InterruptedException ie){}
        }
    }
}
```

Versão Preliminar

```
class FazOutraCoisa implements Runnable {
    public void run() {
        Random randomGenerator = new Random();
        int i= 0;
        Thread eu = Thread.currentThread();
        while(true) {
            i++;
            System.out.println(i);
            try {
                eu.sleep(randomGenerator.nextInt(500));
            }
            catch(InterruptedException ie){}
        }
    }
}
```

A classe `Pool` implementa o pool de threads usando `ArrayBlockingQueue` e `ThreadPoolExecutor`. Não entraremos nos detalhes de implementação de um pool de threads em Java neste texto.

```
public class Pool {
    public static void main (String[] args) {
        ArrayBlockingQueue<Runnable> queue =
              new ArrayBlockingQueue<Runnable>(10);
        ThreadPoolExecutor t =
              new ThreadPoolExecutor(5,10,10,
                                     TimeUnit.SECONDS,queue);
        t.execute (new FazAlgo());
        t.execute (new FazOutraCoisa());
    }
}
```

Versão Preliminar

15.9 THREADS E CONTINUAÇÕES

EXERCÍCIOS

Ex. 182 — Um jogo em rede contabiliza os scores dos jogadores em um servidor; cada jogador pertence a um de vários times. Faça o código que recebe os scores. Deve haver procedimentos para gerar relatórios da situação, mostrando os scores de todos os times, incluindo a proporção entre eles. Para testar o servidor você precisará criar pequenos programas que simulam os jogadores, enviando scores para o servidor.

Ex. 183 — Modifique nossa solução com semáforos para o problema do produtor/consumidor, usando apenas semáforos (e não mutexes). Quantos semáforos a mais você precisa? O que pode dizer a respeito dos valores de seus contadores?

Ex. 184 — Modifique a implementação do problema do produtor/consumidor com mutexes e variáveis de condição para que funcione com buffer de tamanho limitado.

Ex. 185 — Implemente uma versão do Quicksort ou Mergesort usando duas threads. Discorra sobre a vantagem que seu algoritmo tem sobre a versão sequencial. Depois, implemente o mesmo algoritmo usando n threads, onde n é um parâmetro, e rode *benchmarks* com n variando de um até um número ligeiramente maior que a quantidade de CPUs disponíveis no sistema.

Ex. 186 — Modifique a implementação de pool de threads neste capítulo de forma que seja possível modificar o tratador de exceções. O procedimento `make-work-queue` deverá receber, além do número de threads, um tratador de exceções que deverá ser chamado cada vez que uma exceção for capturada por uma das threads trabalhadoras.

Ex. 187 — Mostre que a ideia do exercício anterior é suficientemente geral, e que não seria necessário permitir que o programador redefina o tratador default de exceções.

Ex. 188 — Modifique a implementação de pool de threads para que seja possível aumentar ou diminuir a quantidade de threads ativas. Será necessário que `make-work-queue` retorne mais de um procedimento: um para adicionar um item de trabalho (isto já é feito) e outro, para modificar o número de threads ativas.

Ex. 189 — Modifique o algoritmo do jantar dos filósofos com semáforos para n filósofos e mostre que sua solução é livre de deadlock e de starvation.

Ex. 190 — Implemente um servidor de arquivos que aceita conexões via TCP. Cada cliente inicia mandando uma linha, que pode ser:

Versão Preliminar

- `ENVIA nome` – depois da linha inicial o cliente manda um arquivo para o servidor;
- `TAMANHO nome` – depois de mandar esta linha o cliente recebe um número do servidor (o tamanho do arquivo em bytes);
- `RECEBE nome` –logo após esta linha o cliente poderá ler byte-a-byte o arquivo;
- `REMOVE nome` – o servidor silenciosamente descarta o arquivo.

Quando um arquivo está sendo lido, o cliente deve recebê-lo inteiro, como ele era quando a transmissão iniciou (mesmo que alguém tenha pedido `RECEBE` ou `REMOVE` para ele). Os comandos devem ser enfileirados e executados em sequência.

Ex. 191 — Mostre como implementar barreiras usando apenas variáveis de condição e mutexes, sem o uso explícito de semáforos. Após construir sua solução, você consegue visualizar o semáforo implícito nela?

Ex. 192 — Mostre que semáforos podem ser implementados usando monitores, sem que seja necessária nenhuma outra primitiva de sincronização e sem usar espera ocupada. Faça também o oposto (que monitores podem ser implementados usando semáforos, sem espera ocupada).

Ex. 193 — Modifique o servidor web, adicionando server-side scripting em Scheme (use tags `<?scheme` e `?>`).

Ex. 194 — Modifique o servidor web, adicionando um cache para páginas estáticas.

Versão Preliminar

Versão Preliminar

16 PASSAGEM DE MENSAGENS

Programas concorrentes podem ser elaborados sem o uso de memória compartilhada, mantendo para cada processo um espaço de endereçamento próprio e um comportamento relativamente independente. A sincronização entre processos se dá, então, por troca de mensagens. Este Capítulo trata desta abordagem.

Usando semáforos, construiremos diferentes mecanismos para troca de mensagens, e depois mudaremos o foco para o uso destes mecanismos. No final do Capítulo discutiremos duas ideias que podem, dependendo do ponto de vista, serem descritas como paradigmas de programação: o CSP e o modelo Actor. O leitor encontrará uma discussão de programação com passagem de mensagens em maior profundidade no livro de Andrews [And99].

16.1 MENSAGENS ASSÍNCRONAS

Mensagens são enviadas de um processo a outro através de *canais*. Em um canal as mensagens permanecem na ordem em que foram incluídas, até que sejam retiradas. Quando a leitura e escrita de mensagens podem ser feitas por quaisquer processos, chamaremos os canais de *mailboxes*.

Usaremos as seguintes primitivas para trabalhar com mailboxes:

- `(mailbox-send! mailbox msg)` envia a mensagem `msg` para `mailbox`. O processo que enviou a mensagem não precisa ficar bloqueado;
- `(mailbox-receive! mailbox [timeout [default]])` retira uma mensagem de `mailbox` e a retorna. Se não houver mensagem disponível, a thread ficará bloqueada até que uma mensagem chegue naquela caixa postal. Se `timeout` for definido, a thread ficará bloqueada somente até que o prazo se esgote. Quando um prazo se esgotar, o valor `default` será retornado ou, se não houver default especificado, uma exceção será levantada;

Nossa implementação de mailbox é bastante simples, sem preocupação com eficiência, porque sua função é ajudar a compreender o mecanismo.

Versão Preliminar

Implementamos uma mailbox usando nossa implementação de fila, que ficará em um fecho junto com um mutex e uma variável de condição. Usaremos um único mutex, portanto somente uma operação poderá ser realizada em uma mailbox de cada vez.

```
(load "queues.scm")

(define make-mailbox
  (lambda ()
    (let ((messages (make-q))
          (messages-lock (make-mutex))
          (messages-available (make-condition-variable)))

      (define send! ...)
      (define receive! ...)

      (lambda (msg)
        (case msg
          ((send!) send!)
          ((receive!) receive!))))))

(define (mailbox-send! m obj) ((m 'send!) obj))
(define (mailbox-receive! m) ((m 'receive!)))
```

Para receber uma mensagem, precisamos adquirir o mutex, verificar se a fila está vazia, retirar a mensagem e liberar o mutex. Se a fila estiver vazia, chamamos `mutex-unlock!` para esperar até que outra thread inclua uma mensagem e sinalize a variável de condição `messages-available`.

Versão Preliminar

```
(define receive!
  (lambda ()
    (let loop ()
      (mutex-lock! messages-lock)
      (if (empty-q? messages)
          (begin
            (mutex-unlock! messages-lock messages-available)
            (loop))
          (begin
            (let ((msg (dequeue! messages)))
              (mutex-unlock! messages-lock)
              msg))))))
```

A implementação de `send!` é bastante simples. Após enfileirar a mensagem, sinaliza a variável de condição.

```
(define send!
  (lambda (msg)
    (mutex-lock! messages-lock)
    (enqueue! msg messages)
    (condition-variable-signal! messages-available)
    (mutex-unlock! messages-lock)))
```

Primeiro testamos nossa mailbox com uma única thread. Criamos uma mailbox `m` e enviamos três mensagens:

```
(define m (make-mailbox))
(mailbox-send! m 1)
(mailbox-send! m 20)
(mailbox-send! m 300))
```

Tentamos então receber as três mensagens.

```
(mailbox-receive! m)
1
(mailbox-receive! m)
20
(mailbox-receive! m)
300
```

Versão Preliminar

Se tentássemos receber mais uma mensagem, a única thread ativa (que está executando o REPL) ficaria bloqueada aguardando que alguém sinalizasse a variável `messages-available` – mas isso nunca acontecerá, porque não há outra thread que possa fazê-lo.

Agora realizamos um pequeno teste com duas threads. Uma thread enviará três mensagens, com um intervalo de três segundos entre a segunda e a terceira; a outra receberá e mostrará as mensagens.

```
(let ((mbox (make-mailbox)))
  (define thread-a
     (lambda ()
       (print "sender will send first...")
       (mailbox-send! mbox 'the-message-1)

       (print "sender will send second...")
       (mailbox-send! mbox 'the-message-2)

       (thread-sleep! 3)
       (print "sender will send third...")
       (mailbox-send! mbox 'the-message-3)
       (print "third sent!")))
  (define thread-b
    (lambda ()
      (print "---> " (mailbox-receive! mbox))
      (print "---> " (mailbox-receive! mbox))
      (print "---> " (mailbox-receive! mbox))))
  (let ((a (make-thread thread-a))
        (b (make-thread thread-b)))
    (thread-start! a)
    (thread-start! b)
    (thread-join! a)
    (thread-join! b)))
sender will send first...
sender will send second...
--> the-message-1
--> the-message-2
--> sender will send third...
the-message-3
```

Versão Preliminar

```
third sent!
```

A thread B chegou a mostrar a string `"---> "`, mas ficou bloqueada antes de mostrar a mensagem, e só voltou a executar depois que a thread A fez o último envio.

16.1.1 Exemplo: produtor/consumidor

O problema do produtor/consumidor é resolvido trivialmente usando uma mailbox. O exemplo a seguir implementa uma thread produtora que repetidamente envia listas da forma `("data" i)`, incrementando o valor de `i` a cada mensagem, e uma thread consumidora, que lê uma mensagem e a mostra.

```
(let ((mbox (make-mailbox)))

  (define produce
    (lambda ()
      (let loop ((i 0))
        (mailbox-send! mbox (list "data" i))
        (loop (+ i 1)))))

  (define consume
    (lambda ()
      (let ((x (mailbox-receive! mbox)))
        (display "received")
        (display (list-ref x 1))
        (newline))
      (consume)))

  (let ((p (make-thread produce))
        (c (make-thread consume)))
    (thread-start! c)
    (thread-start! p)
    (thread-join! c)
    (thread-join! p)))
```

Versão Preliminar

16.1.2 Exemplo: filtros e redes de ordenação

(esta seção é um rascunho)

Um filtro é um processo que lê mensagens de um ou mais canais e envia mensagens para outro canal, sendo as mensagens enviadas relacionadas de alguma forma às recebidas.

Um exemplo de filtro é um procedimento que receba mensagens em uma mailbox e as mande em outra apenas se satisfizerem um determinado predicado.

Outro exemplo de filtro é um procedimento de intercalação que recebe números ordenados em dois canais e envia por um terceiro canal a intercalação das duas sequências.

Para construir o intercalador precisaremos de um procedimento que copie n itens de uma mailbox para outra:

```
(define mailbox-copy!
  (lambda (m1 m2 n)
    (do ((i 0 (+ 1 i)))
        ((= i n))
      (mailbox-send! m2 (mailbox-receive! m1)))))
```

Ao processarmos um mailbox sempre enviaremos a quantidade de itens antes dos itens, para que seja possível saber quanto parar de ler.

O procedimento `merge` lê nas mailboxes de entrada dois números n_1 e n_2; Após enviar $n_1 + n_2$ para a mailbox de saída, recebe um item de cada mailbox de entrada e depois determina o que fazer em quatro casos.

Quanto $x_1 < x_2$ o número x_1 deve ser enviado para a saída. No entanto, se além disso $n_1 = 1$, acabamos de ler (e enviar) o último elemento do mailbox m_1, e portanto podemos copiar o que falta de m_2. Se $n_1 \neq 1$, recomeçamos o loop com $n_1 - 1$ e com o próximo elemento do mailbox m_1.

Os dois últimos casos são simétricos, para $x_1 \geqslant x_2$.

Versão Preliminar

```
(define merge
  (lambda (m1 m2 m3)
    (let ((n1 (mailbox-receive! m1))
          (n2 (mailbox-receive! m2)))
      (mailbox-send! m3 (+ n1 n2))
      (let loop ((n1 n1)
                 (n2 n2)
                 (x1 (mailbox-receive! m1))
                 (x2 (mailbox-receive! m2)))
        (cond ((and (< x1 x2) (= 1 n1))
               (mailbox-send! m3 x1)
               (mailbox-send! m3 x2)
               (mailbox-copy! m2 m3 (- n2 1)))
              ((< x1 x2)
               (mailbox-send! m3 x1)
               (loop (- n1 1) n2 (mailbox-receive! m1) x2))
              ((= 1 n2)
               (mailbox-send! m3 x2)
               (mailbox-send! m3 x1)
               (mailbox-copy! m1 m3 (- n1 1)))
              (else
               (mailbox-send! m3 x2)
               (loop n1 (- n2 1) x1 (mailbox-receive! m2)))))))))
```

Testamos nosso intercalador:

```
(let ((a1 '(5 1 6 9 10 20))
      (a2 '(4 2 3 8 15 ))
      (m1 (make-mailbox))
      (m2 (make-mailbox))
      (m3 (make-mailbox)))
  (for-each (lambda (x) (mailbox-send! m1 x))
            a1)
  (for-each (lambda (x) (mailbox-send! m2 x))
            a2)
  (merge m1 m2 m3)
  (print (mailbox->list m3)))
```

Versão Preliminar

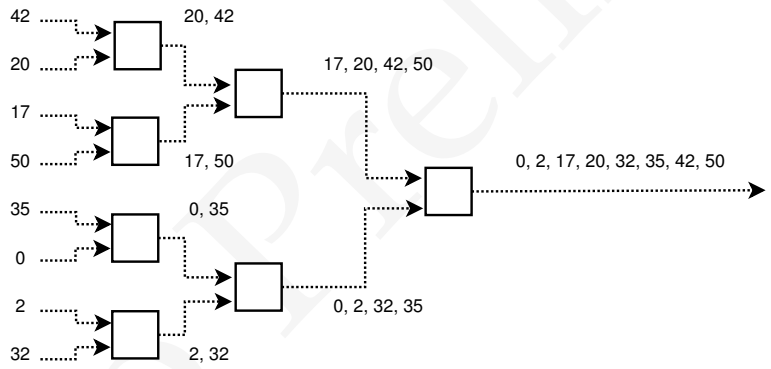
```
(9 1 2 3 6 8 9 10 15 20)
```

Note que este intercalador não funciona quando uma das mailboxes estiver vazia (mesmo que envie zero antes de iniciar). Isto acontece porque antes do `cond` já são feitos `mailbox-receive!` nas duas mailboxes de entrada.

16.1.2.1 *Exemplo: rede de ordenação por intercalação*

Podemos usar o intercalador para construir uma rede de ordenação: organizamos vários intercaladores de forma a receberem via mensagens, cada um, duas listas ordenadas, e enviar a intercalação das duas como saída.

A figura seguir ilustra uma rede de ordenação por intercalação com três camadas. As entradas na primeira camada são $42, 20, 17, 50, 35, 0, 2, 32$. Na segunda camada há quatro processos que fazem a intercalação de duas entradas cada um, enviando os valores intercalados para uma saída. A camada seguinte tem dois processos que operam de forma semelhante, e a última camada tem um único processo que faz a intercalação final, produzindo em um mailbox de saída a sequência $0, 2, 17, 20, 32, 35, 42, 50$.



Com três camadas podemos intercalar $2^3 = 8$ elementos (também precisaríamos de três camadas para qualquer número de elementos entre 5 e 7) – precisamos de $\lceil \log_2 n \rceil$ capadas para n entradas. A quantidade de mailboxes usada é, seguindo as camadas da esquerda para a direita: $8 + 4 + 2 + 1 = 15$. O número de mailboxes necessário para construir uma rede deste tipo será sempre $2n - 1$, onde n é o número de entradas.

O número de processos necessários é $4 + 2 + 1$, ou $2^2 + 2^1 + 2^0$. Como $\sum_{i=0}^{k} = 2^{k+1} - 1$, este é o número de intercaladores que esta rede usa para k camadas.

Será útil criarmos procedimentos para calcular o logaritmo na base dois (usaremos $\log_2 x = (\ln x)/(\ln 2)$) e a menor potência de dois maior ou igual a um inteiro (para isso calcularemos $2^{\lceil \log_2 x \rceil}$):

Versão preliminar

```
(define log2
  (lambda (x)
    (/ (log x) (log 2))))

(define next-power-of-two
  (lambda (n)
    (inexact->exact (expt 2 (ceiling (log2 n))))))
```

O procedimento que constrói a rede de intercalação somente cria um vetor e o preenche com mailboxes vazias. O tamanho do vetor é calculado como já descrevemos: primeiro toma-se a potência de 2 imediatamente acima do número de entradas; depois, o vetor é criado com tamanho $2k - 1$.

```
(define make-marge-sort-net
  (lambda (inputs)
    (let ((total-inputs (next-power-of-two inputs)))
      (let ((n (- (* 2 total-inputs) 1)))
        (let ((v (make-vector n)))
          (do ((i 0 (+ 1 i)))
              ((= i n))
            (vector-set! v i (make-mailbox)))
          v)))))
```

Precisamos de um procedimento para determinar quantos elementos do vetor são entradas da rede de ordenação, e quantos representam elementos internos. O número de entradas é $\lceil n/2 \rceil$ (onde n é o tamanho do vetor).

```
(define merge-sort-net-inputs
  (lambda (sn)
    (inexact->exact
     (ceiling (/ (vector-length sn) 2)))))
```

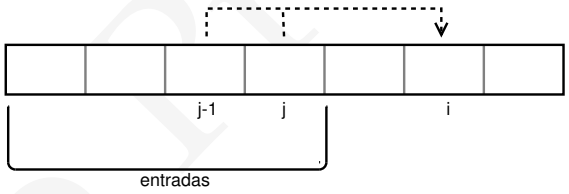
Para iniciar a rede de ordenação criamos várias threads – uma para cada processo de intercalação. Percorremos o vetor com dois índices, i e j; o índice i marca o destino de um processo de intercalação, e os índices j e $j - 1$ marcam as origens.

Versão Preliminar

```
(define merge-sort-net-start
  (lambda (sn)
    (let ((inputs (merge-sort-net-inputs sn)))
      (let loop ((i inputs)
                 (j 1))
        (let ((out (thread-start!
                    (make-thread (lambda ()
                                   (merge (vector-ref sn j)
                                          (vector-ref sn (- j 1))
                                          (vector-ref sn i)))))))
          (if (= i (+ j 1))
              out
              (loop (+ i 1) (+ j 2)))))))))
```

Como o procedimento `merge-sort-net-start` retorna a thread na última posição do vetor, poderemos mais tarde usar `thread-join!` neste objeto, que nos retornará o valor retornado pelo procedimento `merge` – que é justamente o mailbox de saída da última intercalação.

A figura a seguir ilustra como os mailboxes de índice i, j e $j-1$ são atribuídos às entradas e à saída de um processo de intercalação em uma rede para quatro entradas.



O procedimento `feed-sn` recebe uma rede de ordenação (o vetor, já com as mailboxes criadas) e envia para as mailboxes de entrada diversos valores de uma lista.

Versão Preliminar

```
(define merge-sort-net-feed
  (lambda (sn lst)
    (let ((size (length lst))
          (inputs (merge-sort-net-inputs sn)))

      (do ((i 0 (+ i 1)))
          ((= i inputs))
        (if (>= i size)
            (mailbox-send! (vector-ref sn i) 0)
            (mailbox-send! (vector-ref sn i) 1)))

      (let loop ((data lst)
                 (i 0))
        (if (not (null? data))
            (begin (mailbox-send! (vector-ref sn i) (car data))
                   (loop (cdr data) (+ i 1)))))))))
```

Depois, `feed-sn` percorre a lista enviando um elemento para cada nó de entrada.

Será útil termos um procedimento que mostra toda a rede de ordenação.

```
(define merge-sort-show
  (lambda (sn)
    (vector-for-each (lambda (i x)
                       (display i)
                       (display ": ")
                       (write (mailbox->list x))
                       (newline))
                     sn)
    (values)))
```

Definiremos uma rede de intercalação para sete elementos. Como a próxima potência de dois oito, a rede terá $2 \times 8 - 1 = 15$ mailboxes:

```
(define n (merge-sort-net-make 7))
(merge-sort-show n)
0: ()
1: ()
2: ()
...
```


Versão Preliminar

```
13: ()
14: ()
```

Agora alimentamos a rede com sete números: `(merge-sort-net-feed n '(3 4 6 5 2 1 0))`

```
(merge-sort-show n)
0: (1 3)
1: (1 4)
2: (1 6)
3: (1 5)
4: (1 2)
5: (1 1)
6: (1 0)
7: (0)
8: ()
9: ()
10: ()
11: ()
12: ()
13: ()
14: ()
```

Note que o mailbox de índice sete ficou vazio (seu primeiro e único elemento é zero).

Iniciamos a rede, obtendo uma thread que devemos esperar:

```
(define t (merge-sort-net-start n))
```

O `thread-join!` nos retornará o resultado da última intercalação, que é um mailbox. Usamos `mailbox->list` para mostrá-lo.

```
(define result (thread-join! t))
(mailbox->list result)
(7 0 1 2 3 4 5 6)
```

Podemos também mostrar toda a rede de intercalação, que está vazia a não ser pelo último mailbox.

```
(merge-sort-show n)
0: ()
1: ()
...
```


Versão Preliminar

```
13: ()
14: (7 0 1 2 3 4 5 6)
```

16.1.3 Seleção de mensagens por predicado

Em algumas situações podemos querer obter as mensagens da caixa postal fora da ordem em que foram enviadas.

```
(define fetch!
  (lambda (pred?)
    (mutex-lock! messages-lock)
    (let loop ((x (queue-extract! messages pred?)))
      (if (not x)
          (begin
            (mutex-unlock! messages-lock messages-available)
            (loop (queue-extract! messages pred?)))
          (let ((msg (car x)))
            (mutex-unlock! messages-lock)
            msg)))))
```

Usando `mailbox-fetch!` podemos extrair uma mensagem do meio da fila, sem alterar a entrega das outras mensagens:

```
(let ((m (make-mailbox)))
  (mailbox-send! m 'x)
  (mailbox-send! m 'y)
  (mailbox-send! m "I am NOT a symbol!")
  (mailbox-send! m 'z)
  (print (mailbox-fetch! m string?))
  (print "----------")
  (print (mailbox-receive! m))
  (print (mailbox-receive! m))
  (print (mailbox-receive! m)))
```

```
I am NOT a symbol!
-------
x
y
```


Versão Preliminar

z

Este mecanismo pode ser útil quando um processo precisa selecionar a próxima mensagem que venha de um outro processo (mas não outros), ou quando é necessário filtrar mensagens de acordo com algum critério: por exemplo, o processo pode selecionar a próxima requisição para processar alguma tarefa, ou pode selecionar a próxima mensagem vinda de seu processo supervisor.

16.1.4 Seleção de mensagens por casamento de padrões

16.1.5 Timeout

16.1.6 Exemplo: Programação genética

Um sistema de programação genética pode ser implementado usando troca de mensagens: quando um indivíduo da população, um processo é criado. Este processo pode sofrer mutação e realizar *cross-over* com outros através de mensagens. Quando há diversos hosts disponíveis, uma quantidade de processos pode ser enviada para cada host; a seleção dos melhores indivíduos e o cross-over pode ser feita localmente em cada host ou entre indivíduos em hosts diferentes, possivelmente dando prioridade a indivíduos locais.

16.1.7 O Modelo Actor

16.2 MENSAGENS SÍNCRONAS

O envio de mensagens pode ser síncrono: podemos determinar que um processo que queira envia uma mensagem a outro tenha que esperar até que a mensagem seja lida, e somente então possa prosseguir. Do ponto de vista de interface, a única diferença é que a primitiva para envio de mensagens passa a ser bloqueante. Internamente, passaremos a ter também um limite no espaço necessário para armazenamento de mensagens: enquanto mensagens assíncronas podem acumular em filas sem um limite definido, apenas uma mensagem síncrona pode ficar pendente por processo.

Construiremos inicialmente primitivas para envio e recebimento de mensagens síncronas, e depois trataremos da seleção de mensagens de diferentes mailboxes.

Como não há a possibilidade de acúmulo de mensagens, não precisamos de uma fila; usaremos uma variável local `data` para armazenar a mensagem, além de uma variável `full?` para indicar se o mailbox está cheio.

Versão Preliminar

Precisaremos de três variáveis de condição: duas que determinam quando um `send` ou `receive` podem iniciar (só podemos iniciar o envio quando o mailbox estiver vazio, e só podemos ler quando estiver cheio), e uma que determina quando o remetente pode continuar (só após o destinatário ter lido a mensagem e esvaziado a mailbox).

```
(define (make-empty-mailbox)
      (let ((mutex (make-mutex))
            (put-condvar (make-condition-variable))
            (get-condvar (make-condition-variable))
            (message-was-read (make-condition-variable))
            (full? #f)
            (data #f))

        (define sync-send! (lambda (msg) ...)
        (define sync-receive! (lambda () ...)

        (lambda (msg)
          (case msg
            ((sync-send!) sync-send!)
            ((sync-receive!) sync-receive!)
            (else (error "unknown message"))))))

(define (mailbox-sync-send! m obj) ((m 'sync-send!) obj))

(define (mailbox-sync-receive! m) ((m 'sync-receive!)))
```

Para enviar uma mensagem, é necessário esperar até que o mailbox esteja vazio e adquirir o mutex. Depois, modificar o conteúdo de `data`, indicar que o mailbox está cheio e notificar alguma thread que possa estar esperando para ler, e em seguida liberar o mutex.

Versão Preliminar

```
(define (sync-send! obj)
  (mutex-lock! mutex)
  (if full?
      (begin
        (mutex-unlock! mutex put-condvar)
        (sync-send! obj))
      (begin
        (set! data obj)
        (set! full? #t)
        (condition-variable-signal! get-condvar)
        (mutex-unlock! mutex message-was-read))))
```

Para receber, esperamos até que o mailbox esteja cheio e adquirimos o mutex. Depois lemos o conteúdo de `data`, marcamos o mailbox como vazio e sinalizamos *duas* variáveis de condição: uma para alguma thread que esteja esperando para enviar nova mensagem, e outra para a thread que estava esperando após enviar a mensagem que acabamos de ler. Finalmente, liberamos o mutex e retornamos o valor da mensagem.

```
(define (sync-receive!)
  (mutex-lock! mutex)
  (if (not full?)
      (begin
        (mutex-unlock! mutex get-condvar)
        (sync-receive!))
      (let ((result data))
        (set! data #f)
        (set! full? #f)
        (condition-variable-signal! message-was-read)
        (condition-variable-signal! put-condvar)
        (mutex-unlock! mutex)
        result)))
```

Para ilustrar o funcionamento do mailbox síncrono, criaremos duas threads. A primeira envia duas mensagens, e avisa logo após o envio de cada uma. A segunda dorme três segundos, recebe uma mensagem, depois dorme mais três segundos, e recebe a outra.

Versão Preliminar

```
(define m (make-empty-mailbox))

(define thread-a
  (lambda ()
    (mailbox-put! m 'xyz)
    (print "--- done 1 ---")
    (mailbox-put! m 'abc)
    (print "--- done 2 ---")))

(define thread-b
  (lambda ()
    (thread-sleep! 3)
    (print "*** " (mailbox-get! m) " ***")
    (thread-sleep! 3)
    (print "*** " (mailbox-get! m) " ***")))

(let ((a (make-thread thread-a))
      (b (make-thread thread-b)))
    (print '--------)
    (thread-start! a)
    (thread-start! b)
    (thread-join! a)
    (thread-join! b))
-----
                    ;; pausa (3s)
*** xyz ***
-- done 1 --
                    ;; pausa (3s)
*** abc ***
-- done 2 --
```

16.2.1 Seleção de mensagens

Na seção anterior conseguimos implementar de maneira bastante simples as primitivas síncronas para troca de mensagens. No entanto, estas primitivas operam em um mailbox

Versão Preliminar

de cada vez, e não há como um processo decidir receber a mensagem que vier primeiro, independente do canal por onde ela vier.

Queremos poder também expressar a escolha de mensagem dependendo do canal por onde ela vem, da seguinte forma:

```
(select ((mailbox1 x)
         ;; faz algo com x)
        ((mailbox2 y)
         ;; faz algo com y))
```

O `select` mostrado acima espera que alguma mensagem chegue em algum dos dois mailboxes, e então a recebe, passando a executar o trecho de código correspondente (de forma parecida com um `cond`).

Não podemos simplesmente permanecer em loop verificando os mailboxes, e se fizermos a thread dormir precisaremos que um `send` para qualquer um dos mailboxes (mailbox1 e mailbox2) a acorde. No entanto, ao enviar uma mensagem para o mailbox1, por exemplo, não sabemos quem notificar.

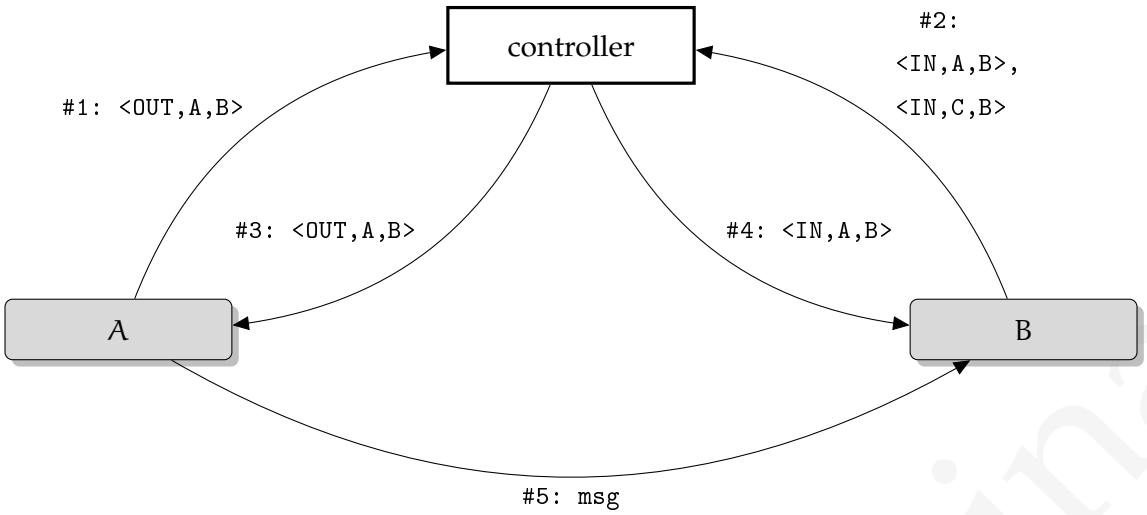
Este problema pode ser resolvido usando um controlador de mensagens[1] a quem os processos devem se dirigir antes de trocar mensagens entre si.

Cada vez que um processo quiser enviar uma mensagem, antes deverá enviar um *template* da sua mensagem ao controlador, contendo a direção, o remetente e o destinatário. Para receber uma mensagem, um processo envia uma *lista* de templates, informando assim todas as mensagens que poderia receber.

A Figura a seguir ilustra um sistema de mensagens síncronas usando um controlador: o processo A pede ao controlador para enviar uma mensagem para B, enviando o template `<OUT, A, B>`; como não há processo que tenha informado querer receber tal mensagem, o controlador armazena a mensagem em sua lista de pendencias. Mais tarde o processo B informa o controlador que pretende receber uma comunicação de A *ou* de C, enviando os templates `<IN, A, B>, <IN, C, B>`; agora o controlador percebe que um destes templates casa com o template pendente `<OUT, A, B>`, e então envia aos dois processos mensagens informando a eles que agora podem prosseguir. Por último o processo A envia a B a mensagem de dados.

1 Fazemos aqui uma analogia com controladores de tráfego aéreo. Gregory Andrews [And99] usa o termo *clearinghouse*, que poderíamos traduzir como "câmara de compensação".

Versão Preliminar



Cada processo tem dois mailboxes: um para receber dados (no exemplo acima é onde a mensagem `msg` chegou na mailbox de entrada do processo B) e outro para controle, onde recebem notificações do controlador.

O controlador precisa enviar não apenas o mailbox, mas o template completo para ambos os processos – por exemplo, na Figura anterior o controlador enviou o template `#1: <OUT,A,B>` para o A, e não apenas "B".

FIXME: porque?

Implementaremos um controlador e primitivas para enviar e receber mensagens síncronas em Scheme, usando mailboxes síncronas.

Usaremos as listas de associação mutáveis que desenvolvemos na Seção 3.3.4.

Cada processo tem um nome e duas caixas postais: uma para dados e uma para controle.

```
(define-record-type process
  (make-process-from-mailboxes name data-in control-in)
  process?
  (name process-name)
  (data-in process-data)
  (control-in process-control))

(define-record-printer (process x out)
  (print "#<process name: " (process-name x)
         " data: "  (process-data x)
         " control: " (process-control x)
         ">"))
```


Versão Preliminar

Não queremos ter que criar as mailboxes de cada processo explicitamente, por isso construímos um procedimento `make-process` que criará as duas mailboxes e nomeará o processo

```
(define make-process
  (let ((counter 1))
    (lambda args
      (let ((data (make-mailbox))
            (control (make-mailbox))
            (name (if (null? args)
                      (string-append "process-"
                                     (number->string counter))
                      (car args))))
        (set! counter (+ counter 1))
        (make-process-from-mailboxes name data control)))))
```

Um template de mensagem contém a direção, os processos remetente e destinatário.

```
(define-record-type message-template
  (make-message-template direction
                         sender
                         receiver)
  message-template?
  (direction msg-direction)
  (sender    msg-sender)
  (receiver  msg-receiver))

(define-record-printer (message-template x out)
  (display "#<msg-tpl " out)
  (display (msg-direction x) out)
  (display " s::" out)
  (display (msg-sender x) out)
  (display " r::" out)
  (display (msg-receiver x) out)
  (display " >" out))
```

O procedimento `build-match` recebe um template e devolve outro semelhante mas com a direção trocada. Este template será usado para casar pares de pedidos de comunicação.

Versão Preliminar

```
(define build-match
  (lambda (template)
    (make-message-template (case (msg-direction template)
                             ((IN)  'OUT)
                             ((OUT) 'IN))
                           (msg-sender template)
                           (msg-receiver template))))
(build-match (make-message-template 'IN, 'a, 'b)
(#<msg-tpl OUT a b>)
```

Dados um template e um mapa de processos em templates, o procedimento `find-match-in-alist` retornará o primeiro template na lista que casa com o template dado.

```
(define find-match-in-alist
  (lambda (template pending)
    (let ((match-tpl (build-match template)))
      ;; find who the peer is:
      (let ((peer (case (msg-direction template)
                    ((IN)  (msg-sender template))
                    ((OUT) (msg-receiver template)))))
        ;; get peer's pending list:
        (let ((peer-pending (alist-find peer pending)))
          ;; find a match in peer's pending list:
          (if peer-pending
              (member match-tpl (cdr peer-pending))
              #f))))))
```

Dados uma *lista* de templates e um mapa de processos em templates, o procedimento `find-match` retornará o primeiro template no mapa que casa com um template da lista.

Versão Preliminar

```
(define find-match
  (lambda (pending templates)
    (if (null? templates)
        #f
        (let ((matches (find-match-in-alist (car templates)
                                            pending)))
          (if matches
              (car matches)
              (find-match pending (cdr templates)))))))
```

Quando o controlador encontrar dois templates que casam, ele enviará mensagens aos dois processos informando-os que podem se comunicar. Isso é feito pelo procedimento `send-control-messages`.

```
(define send-control-messages
  (lambda (direction sender dest)
    (let ((dest-ctl    (process-control dest))
          (sender-ctl  (process-control sender)))
      (case direction
        ((OUT) (mailbox-send! sender-ctl (list 'OUT dest))
               (mailbox-send! dest-ctl   (list 'IN  sender)))
        ((IN)  (mailbox-send! sender-ctl (list 'IN  dest))
               (mailbox-send! dest-ctl   (list 'OUT sender)))
        (else  (error "Direction neiter IN nor OUT"))))))
```

O procedimento `loop` dentro de `make-message-controller` recebe templates de mensagens e verifica em sua lista de templates pendentes se o novo template recebido casa com algum já armazenado.

Versão Preliminar

```
(define make-message-controller
  (lambda ()
    (let ((pending (make-alist))
          (tpl-in (make-mailbox)))
      (define loop
          (lambda ()
            (let ((template (mailbox-receive! tpl-in)))
              (let ((sender    (car template))
                    (direction (msg-direction (cadr template))))
                (let ((match (find-match pending template)))
                  (if (not match)
                      (alist-set! pending sender template)
                      (begin (send-control-messages direction
                                                    sender
                                                    (msg-receiver match))
                             (alist-set! pending match '()))))))
            (loop)))

      (thread-start! (make-thread loop))
      tpl-in)))
```

O procedimento `sync-receive!` constrói um template de mensagem, envia ao controlador, depois aguarda por uma mensagem do controlador e finalmente recebe a mensagem.

```
(define sync-receive!
  (lambda (ch sender receiver)
    (mailbox-send! ch
                   (list receiver
                         (make-message-template 'IN
                                                sender
                                                receiver)))
    (mailbox-receive! ch)
    (let ((result (mailbox-receive! (process-data receiver))))
      result)))
```

Versão Preliminar

```
(define sync-send!
  (lambda (ch sender receiver msg)
    (mailbox-send! ch
                   (list sender
                         (make-message-template 'OUT
                                                sender
                                                receiver)))
    (mailbox-receive! ch)
    (mailbox-send! (process-data receiver) msg)))
```

16.2.2 Communicating Sequential Processes

EXERCÍCIOS

Ex. 195 — Modifique a implementação de rede de ordenação para que aceite mais de um número por entrada.

Ex. 196 — O algoritmo de Strassen para multiplicação de matrizes [Cor+09] pode ser modificado para usar trocas de mensagens.
Para multiplicar duas matrizes A e B, tal que

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix} \text{ e}$$

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix},$$

onde $X_{1,1}, X_{1,2}, X_{2,1}, X_{2,2}$ representam os quatro blocos resultantes da partição de X em quatro partes iguais, usamos o algoritmo de Strassen: definimos as matrizes

$$\begin{aligned}
\mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\
\mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\
\mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\
\mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\
\mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\
\mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\
\mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2})
\end{aligned}$$

Versão Preliminar

e calculamos a matriz C:

$$\begin{aligned}
\mathbf{C}_{1,1} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{1,2} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{2,1} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{2,2} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

Com isso fazemos sete multiplicações de submatrizes ao invés de oito (que é o que um algoritmo ingenuo faria). Uma implementação usando troca de mensagens poderia criar um novo processo cada vez que uma chamada recursiva é feita.

a)Modifique o algoritmo de Strassen, produzindo um algoritmo concorrente usando mensagens assíncronas em pseudo-código, e prove que ele está correto.

b)Implemente o algoritmo que você desenvolveu no item anterior.

c)Faça agora uma versão síncrona do algoritmo de Strassen, e descreva-o usando o CSP.

d)Prove que o algoritmo do item anterior está correto.

e)Implemente o algoritmo do item (c) em alguma linguagem que suporte passagem síncrona de mensagens (ou usando uma biblioteca que permita fazê-lo).

f)Modifique uma das implementações para que ela funcione com matrizes onde as entradas podem ser simbólicas, como esta:

$$\begin{bmatrix} \mathbf{x} & \mathbf{2x} & \mathbf{a} \\ \mathbf{4.0} & \mathbf{2.5} & \mathbf{3b} \\ \mathbf{-x} & \mathbf{0.0} & \mathbf{a} \end{bmatrix}$$

que poderia ser lida de um arquivo como este:

```
x         (* 2 x)   a
4.0       2.5       (* 3 b)
(* -1 x)  0.0       (* a 2)
```

Nas entradas da matriz você pode permitir apenas `+`, `-`, `*` e `/`.

O resultado evidentemente será outra matriz no mesmo formato de entrada, misturando símbolos e números. Por exemplo, se

$$\mathbf{A} = \begin{bmatrix} \mathbf{x} & \mathbf{2x} \\ \mathbf{4.0} & \mathbf{2.5} \end{bmatrix} \text{ e } \mathbf{B} = \begin{bmatrix} \mathbf{1} & \mathbf{2} \\ \mathbf{-y} & \mathbf{0} \end{bmatrix}$$

então

$$\mathbf{AB} = \begin{bmatrix} \mathbf{x - 2xy} & \mathbf{2x} \\ \mathbf{4 - 2.5y} & \mathbf{8} \end{bmatrix}$$

Versão Preliminar

A matriz AB deve ser representada da mesma forma que os arquivos de entrada (cada posição é uma lista que poderíamos calcular usando `eval`):

```
(- x (* 2 (* x y)))  (* 2 x)
(- 4 (* 2.5 y))      8
```

Não se preocupe em fatorar ou simplificar as expressões, exceto quando houver formas onde *todos* os elementos são números.

Ex. 197 — Os procedimentos `sync-send!` e `sync-receive!` só permitem escolher entre um de muitos canais para envio ou um de muitos canais de saída. Escreva um único procedimento `sync-comm!` que permita a um processo escolher dentre vários eventos:

```
(sync-comm! ch (list (make-message-template 'OUT a b)
                     (make-message-template 'OUT a c)
                     (make-message-template 'IN  alarm a)))
```

Ex. 198 — Escreva uma macro `select` que permita usar o procedimento do Exercício 197 mais convenientemente:

```
(select ch ((! b x)
            (print 'mandei-x))
           ((! c y)
            (print 'mandei-y))
           ((? alarm z)
            (print "alarme:" z)))
```

No exemplo acima `!` significa "envie" e `?` significa "receba". O significado deste código é: tente enviar x para o mailbox b ou y para o mailbox c, mas não ambos (a seleção dependerá de qual leitor chegará primeiro para receber a mensagem – o de b ou o de c). Se, no entanto, algo chegar pelo mailbox `alarm` antes de alguma mensagem ser mandada, guarde o alarme na variável `z` e execute o `print` correspondente.

Ex. 199 — É possível criar dois procedimentos `sync-send!` e `sync-receive!` (ou um procedimento `symc-comm!`, como proposto no Exercício 197) que não precisem receber o controlador como parâmetro, sem que seja necessário criar o controlador como variável global? Mostre porque não é possível ou mostre como implementar.

Versão Preliminar

Parte IV.

Projetos Sugeridos

Versão Preliminar

Versão Preliminar

17 PROJETOS SUGERIDOS

Os pequenos projetos a seguir envolvem conceitos abordados em diferentes Capítulos do livro.

Ex. 200 — Complete o framework de testes unitários da Seção 1.9.1, incluindo, além de testes com `equal?`, testes para conexões TCP, testes que verificam se um erro ou exceção foi gerado (comparando com o erro ou a exceção correta), testes para computação em tempo real (onde uma função passa no teste somente se terminar antes de um tempo especificado).

Ex. 201 — Implemente um interpretador metacircular para Scheme que avalie todas as formas de maneira preguiçosa.

Ex. 202 — Implemente um sistema preguiçoso de objetos (um sistema de objetos onde o processamento e resposta de mensagens se dá de forma preguiçosa).

Ex. 203 — Modifique o sistema de objetos para que o recebimento de mensagens use casamento de padrões.

Ex. 204 — Modifique o sistema de objetos para que funcione de maneira distribuída em rede.

Ex. 205 — Implemente um sistema de objetos onde as mensagens trocadas são perguntas Prolog. Inclua a parte "não pura" do Prolog.

Ex. 206 — Implemente verificação de tipos no interpretador Prolog.

Ex. 207 — Modifique o interpretador Prolog para que use threads internamente, a fim de acelerar o processamento das perguntas. Isso pode ser feito usando "paralelismo OU" (quando há mais de uma cláusula que possa ser usada, tente ambas em paralelo, e verifique *na ordem correta* se alguma delas teve sucesso) ou "paralelismo E" (quando não há variáveis compartilhadas em uma sequência de objetivos, processe-os em paralelo).

Ex. 208 — Implemente predicados que permitam ao programador Prolog criar e usar threads.

Ex. 209 — Construa um interpretador ITERCAL com sintaxe de Scheme.

Versão Preliminar

Ex. 210 — Implemente um jogo de batalha naval com um servidor e vários clientes. O cliente pode rodar em linha de comando, mostrando o mar como uma grande tabela ASCII. Pense em como resolver os problemas de (1) mar muito amplo e (2) jogadores demais.

Ex. 211 — Faça uma interface web para o jogo de batalha naval. Não para o servidor. No entanto, não faça um jogo somente no servidor. O cliente é parecido com o que foi feito no projeto sugerido anterior, e se comunica com o servidor da maneira usual. No entanto, ao invés de mostrar uma tabela ASCII representando o mar, ele roda um servidor HTTP local, com páginas mostrando o mar em gráficos.

Ex. 212 — Implemente um mini banco de dados distribuído

Versão Preliminar

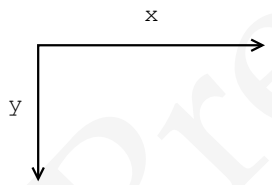
A FORMATOS GRÁFICOS

Este Apêndice traz a descrição dos formatos para imagens gráficas usado em alguns trechos do texto.

Normalmente estes formatos não são manipulados diretamente pelo programador, e sim via bibliotecas especializadas; no entanto, o espírito deste texto nos direciona a entrar nestes detalhes a fim de compreender mais profundamente o assunto.

Os formatos Netpbm e SVG são bastante simples. O Netpbm é um formato para descrição de imagens por mapas de bits, e o SVG (Scalable Vector Graphics) é um formato para descrição de imagens por vetores.

O sistema de coordenadas nestes formatos é semelhante ao plano Cartesiano, com os pontos refletidos no eixo das abscissas: `(x, y)` representa o ponto que dista `x` à direita da origem e `y` *para baixo* dela:



A.1 NETPBM

Netpbm é uma família de formatos para imagens gráficas. Todos os formatos netpbm representam a imagem como um mapa de pixels. Há formatos para imagens em preto-e-branco (onde a informação armazenada por pixel é zero ou um); para tons de cinza (onde a informação armazenada por pixel é um número correspondente a algum tom de cinza) e para imagens coloridas (onde são armazenados três números por pixel, indicando a quantidade relativa de vermelho, verde e azul).

Os formatos Netpbm permitem armazenar a imagem em formato legível por pessoas, onde cada dado relativo a um pixel é escrito usando sua representação numérica em ASCII, ou em formato binário ilegível e mais compacto.

A tabela a seguir lista os seis formatos Netpbm e suas características:

Versão Preliminar

formato	informação por pixel	representação por pixel
PBM (P1)	bits (0 ou 1)	1 caracter ASCII
PGM (P2)	tons de cinza	1 número representado em ASCII
PPM (P3)	cores	3 números representados em ASCII
PBM (P4)	bits (0 ou 1)	1 bit
PGM (P5)	tons de cinza	8 bits
PPM (P6)	cores	24 bits

Todos os formatos iniciam com uma linha onde há apenas o nome do formato, em ASCII (P1, P2, etc).

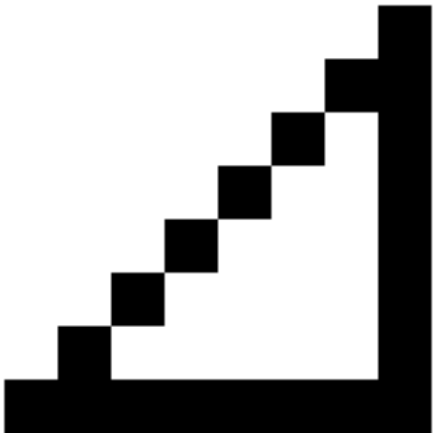
A.1.1 P1: preto e branco, legível

Para o formato P1, a segunda linha contém o número de colunas e o número de linhas da imagem, e os próximos números (separados por espaços) são zero ou um, representando os bits da imagem. O exemplo a seguir é uma imagem de um triângulo:

```
P1
10 9
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1 1 0
0 0 0 0 0 0 1 0 1 0
0 0 0 0 0 1 0 0 1 0
0 0 0 0 1 0 0 0 1 0
0 0 0 1 0 0 0 0 1 0
0 0 1 0 0 0 0 0 1 0
0 1 1 1 1 1 1 1 1 0
```

O arquivo acima representa a seguinte figura, em escala maior:

Versão Preliminar



A.1.2 P2: tons de cinza, legível

O formato P2 é semelhante ao P1, exceto que:

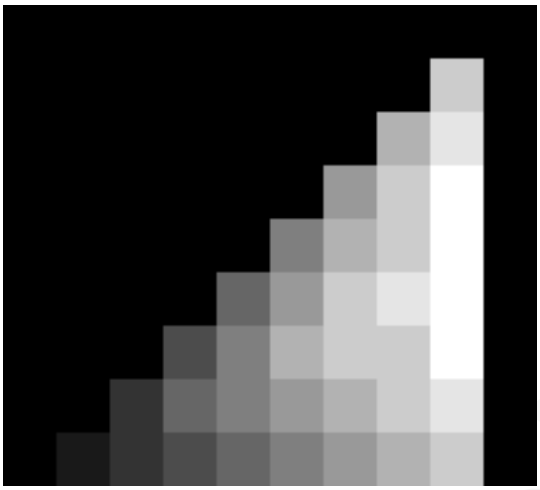
- Há mais uma linha entre as dimensões da imagem e os pixels, onde existe um único número. Este é o maior número usado para representar tons de cinza na imagem;
- Ao invés de uns e zeros representando preto e branco, cada entrada é um número representando um tom de cinza: zero representa preto e o maior número representa branco.

O exemplo a seguir é semelhante ao usado na seção anterior, mas desta vez usando tons de cinza.

```
P2
10 9
10
0 0 0 0 0 0 0 0 0  0
0 0 0 0 0 0 0 0 8  0
0 0 0 0 0 0 0 7 9  0
0 0 0 0 0 0 6 8 10 0
0 0 0 0 0 5 7 8 10 0
0 0 0 0 4 6 8 9 10 0
0 0 0 3 5 7 8 8 10 0
0 0 2 4 5 6 7 8 9  0
0 1 2 3 4 5 6 7 8  0
```

Versão Preliminar

O arquivo acima representa a seguinte figura, em escala maior:



A.1.3 P3: em cores, legível

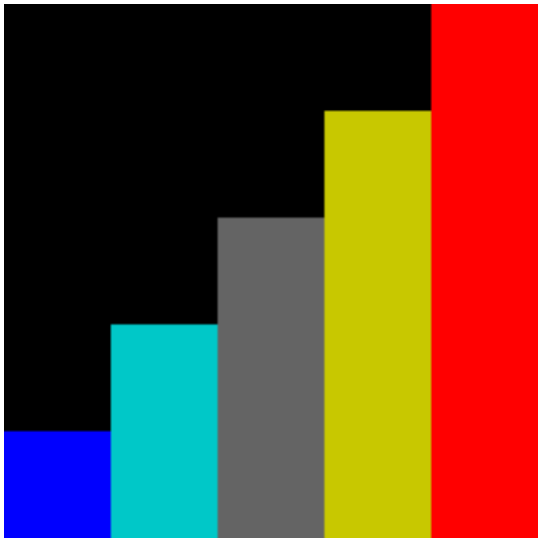
O formato P3 é semelhante ao formato P2, com as seguintes diferenças:

- O número antes dos pixels representa o maior número usado para representar a quantidade de cada cor;
- Cada entrada na matriz de pixels é composta por *três* números em sequência, representando as quantidades de vermelho, verde e azul.

```
P3
5 5
255
0 0 0    0 0 0    0 0 0        0   0   0  255 0 0
0 0 0    0 0 0    0 0 0      200 200 0  255 0 0
0 0 0    0 0 0    100 100 100  200 200 0  255 0 0
0 0 0    0 200 200 100 100 100  200 200 0  255 0 0
0 0 255  0 200 200 100 100 100  200 200 0  255 0 0
```

O arquivo acima representa a seguinte figura, em escala maior:

Versão Preliminar



A.1.4 Netpbm binário

As variantes binárias dos formatos acima são mais compactas, e portanto ocupam menos espaço e podem ser lidas e escritas em disco mais rapidamente. Os cabeçalhos são semelhantes, mudando apenas a identificação de cada formato (P4, P5 e P6). A matriz de pixels é representada da seguinte maneira:

- PBM (P4) representa oito bits em cada byte;
- PGM (P5) representa um pixel em cada byte;
- PPM (P6) usa três bytes por pixel.

Os bytes representando pixels *não* são separados por espaços.

A.2 SVG

(esta seção está incompleta)

SVG é um formato pare descrição de imagens vetoriais em duas dimensões desenvolvido pelo W3C. A especificação completa do formato SVG é demasiado longa para ser incluída neste Apêndice, que traz apenas uma descrição básica.

Versão Preliminar

A.2.1 SVG é XML

Gráficos SVG são armazenados em um arquivo XML, e todo arquivo SVG deve ser semelhante a este:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC
         "-//W3C//DTD SVG 1.1//EN"
         "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
     xmlns="http://www.w3.org/2000/svg">

<!-- a descrição do gráfico vai aqui -->

</svg>
```

Figuras e texto são descritos em SVG como elementos XML. Cada elemento é desenhado na ordem em que é encontrado no arquivo, e quando há sobre posição entre elementos o efeito será como se o segundo fosse desenhado acima do primeiro. Se o segundo elemento não for completamente opaco, as cores serão misturadas.

A.2.2 Tamanho da imagem

Como SVG é um formato para gráficos vetoriais, as distâncias e tamanhos podem ser relativos (sem unidade definida). O tamanho da imagem *pode* ser determinado para cada gráfico SVG usando alguma unidade de medida:

```
<svg xmlns='http://www.w3.org/2000/svg'
      width="100px" height="200px" version="1.1">
```

As unidades suportadas são em, ex, px, pt, pc, cm, mm, in, e porcentagens.

Se estes atributos forem omitidos, o tamanho da imagem será determinado apenas quando ela for embarcada em algum meio que o determine.

A especificação do formato SVG inclui mais de um sistema de coordenada; não trataremos disso neste Apêndice.

Versão Preliminar

A.2.3 Estilo

Há uma grande quantidade de atributos de estilo que podem ser usadas em elementos SVG. Listamos algumas delas:

- `stroke`: a cor do traçado;
- `stroke-width`: a largura do traçado;
- `fill`: a cor da parte interna da figura, se ela for fechada;
- `font-family`: a família da fonte (para elementos de texto);
- `font-size`: o tamanho da fonte (para elementos de texto).

Cores podem ser especificadas usando seus nomes ou seus componentes de vermelho, verde e azul. Por exemplo, `fill="blue"` é o mesmo que `fill="rgb(0,0,255)"` e `fill="#0000ff"`.

A.2.4 Elementos básicos

O elemento `line` desenha um segmento de reta entre dois pontos `(x_1, y_1)` e `(x_2, y_2)`.

```
<line x1="0" y1="0" x2="300" y2="300"
      stroke="rgb(99,99,99)"
      stroke-width="2" />
```

Desenhamos retângulos com o elemento `rect`.

```
<rect width="300" height="100"
      fill="rgb(0,0,255)"
      stroke-width="2"
      stroke="rgb(0,0,0)" />
```

Para desenhar um círculo, usamos o elemento `circle` com atributos `cx`, `cy` para o centro e `r` para o raio.

```
<circle cx="100" cy="50" r="40"
        stroke="black"
        stroke-width="2"
        fill="red" />
```

Versão Preliminar

Elipses podem ser desenhadas com o elemento `ellipse`, usando `cx, cy` para o centro e `rx, ry` para os dois raios (nos eixos x e y).

```
<ellipse cx="300" cy="150" rx="200" ry="80"
         fill="rgb(200,100,50)"
         stroke="rgb(0,0,100)"
         stroke-width="2" />
```

Um polígono é desenhado com o elemento `polygon`. O atributo `points` é a lista de pontos que definem o polígono.

```
<polygon points="220,100 300,210 170,250"
         fill="#cccccc"
         stroke="#000000"
         stroke-width="1" />
```

Uma *polyline* é uma linha que passa por vários pontos. Sua descrição em SVG é semelhante à de um polígono, mas não haverá segmento de reta ligando o último ponto ao primeiro.

```
<polyline points="0,0 0,20 20,20 20,40 40,40 40,60"
          fill="white"
          stroke="red"
          stroke-width="2" />
```

O elemento `path` contém um atributo `d` que contém os dados do caminho: `M` é um `moveto`, `L` é um `lineto` e `Z` termina o caminho.

```
<path d="M250 150 L150 350 L350 350 Z" />
```

Para incluir texto em um arquivo SVG há o elemento `text`:

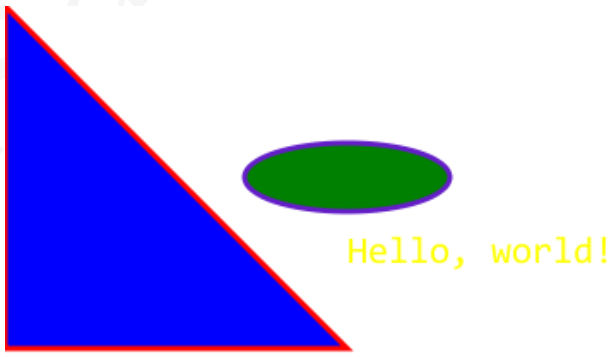
```
<text x="200" y="150"
      font-family="Courier"
      font-size="22"
      fill="yellow" >
  Hello, world!
</text>
```

Um exemplo completo é mostrado a seguir:

Versão Preliminar

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
         "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

  <svg
       version="1.1"
       xmlns="http://www.w3.org/2000/svg">

    <text x="200" y="150"
          font-family="Consolas"
          font-size="22"
          fill="yellow" >
    Hello, world!
  </text>

  <polygon points="0,0 200,200 0,200"
           fill="blue"
           stroke="rgb(255,0,0)"
           stroke-width="3" />

  <ellipse cx="200" cy="100" rx="60" ry="20"
           fill="green"
           stroke="rgb(100,30,200)"
           stroke-width="3" />

</svg>
```

A figura descrita no arquivo acima é:



Versão Preliminar

Versão Preliminar

B RESUMO DE SCHEME

(este Apêndice está incompleto)

Procedimentos R^{7}RS ainda não incluídos neste apêndice:

```
read-bytevector, read-bytevector!, write-bytevector, write-partial-bytevector
utf8->string, string->utf8
define-library
eager
lazy
environment
digit-value
string-ni<=? (compara strings normalizadas)
string-ni<?
string-ni=?
string-ni>=?
string-ni>?
case-lambda
open-binary-input-file, open-binary-output-file
open-input-bytevector, open-output-bytevector, get-output-bytevector
get-environment-variable, get-environment-variables
```

Este Apêndice traz um resumo da linguagem Scheme.

B.1 SINTAXE

O padrão R^{7}RS define que Scheme deve diferenciar maiúsculas e minúsculas por default em símbolos. Este comportamento pode ser modificado – implementações Scheme podem oferecer as seguintes diretivas:

```
#!fold-case
#!no-fold-case
```

A primeira diretiva determina que os símbolos sejam transformados em caixa baixa ao serem lidos, e a segunda determina que os símbolos sejam lidos como estão, sem modificação. Estas diretivas podem aparecer em qualquer lugar onde um comentário

Versão Preliminar

poderia estar. Seu efeito é apenas este (ademais, são tratadas como comentários) e vale para as leituras feitas após o leitor encontrá-las.

B.1.1 Comentários

Há três maneiras de comentar código:

- Comenta-se uma linha com ponto-e-vírgula:
  `(parte não comentada)  ; resto da linha comentada...`
- Comenta-se uma S-expressão com `#;( ... )`
- Comenta-se um trecho de código com `#| ... |#`

B.1.2 Estruturas cíclicas

Na representação externa de objetos Scheme,

- `#n#` marca um ponto de referência;
- `#n=` é uma referência a um ponto já definido.

Por exemplo,

```
(define x (list 'a 'b 'c))
x
(a b c)
(set-cdr! (cdr x) x)
x
#0=(a b . #0#)
```

B.2 TIPOS DE DADOS E SUAS REPRESENTAÇÕES

- *Booleano*: representados por `#t` e `#f`;
- *Caracter*: individualmente representados em programas pelo prefixo `#\` (por exemplo, `#\a`, `#\b`). Quebra de linha e espaço tem representações especiais, `#\newline` e `#\space`;
- *Número* (veja discussão sobre números a seguir);

Versão Preliminar

- *Par*: Células compostas de duas posições, `car` e `cdr`. Usados para construir listas. A representação externa de um par é `(car . cdr)`;
- *Porta*: retornado por procedimentos de entrada e saída, e não representado diretamente em código.
- *Procedimento*: criado por `lambda`, e não representado diretamente em código.
- *String*: sequências de caracteres, entre aspas;
- *Símbolo*: nomes, representados sem aspas;
- *Vetor*: sequências de objetos, normalmente criados por procedimento. A representação externa é `#(obj1 obj2 ...)`. Há um tipo distinto de vetor:
  - *Bytevector*: um vetor de *bytes*. Um *byte* é um inteiro entre zero e 255.

Além destes tipos de dados, há promessas (criadas com `delay`) e valores indefinidos (retornados por procedimentos com efeitos colaterais).

Listas são pares que não formam estrutura circular e cujo último `cdr` é `'()`.

Pode-se usar a repreentação externa de pares, listas e vetores em programas, usando `quote`, mas estes serão imutáveis: `'(a . b)`, `'(a b c)`, `'#(a b c)`.

Números podem ser exatos ou inexatos.

- *Exato*
- *Inexato*

Dentre os inexatos há `+inf.0`, `-inf.0`, e `+nan.0`.

Números também podem ser:

- *Complexo*: representado como `a+bi`, `a-bi`, `-a+bi` ou `-a-bi`, onde `a` e `b` são representações de números não complexos;
- *Real*: representado como inteiro, racional ou como número de ponto flutuante;
- *Racional*: representados da mesma forma que inteiros ou números de ponto flutuante, ou no formato `n/d`, onde `n` é um numerador inteiro e `d` é um denominador inteiro;
- *Inteiro*

Versão Preliminar

B.3 DIVISÃO

Todos os procedimentos de divisão, `/`, `quotient`, `reminder` e `modulo` tem, além da versão comum, outras cinco, prefixadas por `ceiling`, `floor`, `truncate`, `round` e `euclidean`.

- `ceiling-` aplicará `ceiling` após a divisão, retornando o menor inteiro maior que o resultado.
- `floor-` aplicará `floor` após a divisão, retornando o maior inteiro menor que o resultado.
- `truncate-` aplicará `truncate` após a divisão, arredondando o resultado na direção do zero (como se `floor` fosse aplicado a positivos e `ceiling` para negativos).
- `round-` aplicará `round` após a divisão, retornando o inteiro mais próximo do resultado. Se o resultado estiver no ponto médio entre dois inteiros, o inteiro par será retornado.
- `euclidean-` Se $d > 0$, $q = \texttt{floor}(n/d)$; se $d < 0$, $q = \texttt{ceiling}(n/d)$.

Por exemplo,

```
(/ 5.0 6.0)
1.66666666666667
(ceiling/ 5.0 6.0)
2

(quotient 5 3)
1
(remainder 5 3)
2
(ceiling-quotient 5 3)
2
(remainder 5 3)
-1
```

B.4 FEATURES

Estas são as *features* que, de acordo com o padrão R^{7}RS, podem ser usadas com `cond-expand`.

Versão Preliminar

- `r7rs`: a implementação declara implementar corretamente o padrão R^{7}RS.
- `exact-closed`: todas as operações algébricas com exatos (exceto `/`) resultam em números exatos.
- `ratios`: A operação `/` com argumentos exatos produz um resultado exato quando o divisor não é zero (ou seja, a implementação suporta frações exatas).
- `exact-complex`: a implementação suporta números complexos exatos.
- `ieee-float`: números *inexatos* são representados como no padrão IEEE 754.
- `full-unicode`: todos os *codepoints* Unicode são suportados como caracteres.
- `windows`: a implementação de Scheme está rodando em um sistema Windows.
- `posix`: a implementação de Scheme está rodando em um sistema POSIX.
- `unix, darwin, linux, bsd, freebsd, solaris, ...`: descreve o tipo de sistema operacional.
- `i386, x86-64, ppc, sparc, jvm, clr, llvm, ...`: descvreve a arquitetura da CPU.
- `ilp32, lp64, ilp64, ...`: descvreve o modelo de memória C.
- `big-endian, little-endian`: descreve a ordem de *bytes* ao representar palavras.
- *nome*: o nome da implementação Scheme.
- *nome-versao*: nome e versão da implementação Scheme.

B.5 MÓDULOS

Módulos isolam nomes em espaços diferentes, e foram definidos no padrão R^{7}RS.

Um novo módulo é declarado da seguinte forma:
`(define-library nome` *declaração*`)`
Cada declaração pode ser:

- `(export` *especificação-de-export*`...)`.
- `(import` *conjunto-de-import*`...)`.
- `(begin ...)` para definir as variáveis, macros e procedimentos internos do módulo.

Versão Preliminar

- `(include arq1 arq2 ...)` para icluír no corpo do módulo o conteúdo dos arquivos listados (é semelhante ao `begin`, mas lê o conteúdo de arquivos).
- `(include-ci arq1 arq2 ...)` é semelhante a `include`, mas ignora diferença entre caixa alta e baixa.
- `(cond-expand` *cláusula-cond-expand*`)` para expandir código dependendo de *features* presentes no momento de leitura do código.

Um `(import ...)` pode ser usado livremente no nível base (por exemplo, no REPL ou em um arquivo que não define módulos).

- *especificação-de-export:* pode ser um de dois casos:
  - `nome` *(um único símbolo):* neste caso, o nome que existe dentro do módulo (ou porque foi importado ou porque foi definido no módulo) é exportado *com o mesmo nome*.
  - `(nome1 nome2)` *(uma lista com dois nomes):* neste caso o primeiro nome deve ser um símbolo existente no módulo, que será exportado com o segundo nome.
- *conjunto-de-import:* cada um destes define que o módulo atual deve importar símbolos de um outro módulo. A forma do conjunto de import pode ser:
  - `nome-modulo` *(o nome de um módulo):* neste caso todos os símbolos definidos no módulo `nome-modulo` são importados no módulo atual.
  - `(only` *conjunto-de-import* `nome ...)` – após determinar como realizar o `import` (via conjunto-de-import dado), filtra a lista de nomes e importa de fato *apenas* os nomes dados na lista.
  - `(except` *conjunto-de-import* `nome ...)` – após determinar como realizar o `import` (via conjunto-de-import dado), filtra a lista de nomes e importa de fato *apenas* os nomes *não dados* na lista.
  - `(prefix` *conjunto-de-import* `pref)` – após determinar como realizar o `import` (via conjunto-de-import dado), importa os nomes, prefixando com `pref`
  - `(rename` *conjunto-de-import* `(nome1 nome2) ...)` – após determinar como realizar o `import` (via conjunto-de-import dado), troca os nomes ao importar (`nome1` $\rightarrow$ `nome2`, etc).
- *cláusula-cond-expand:* deve ser da forma
  `(` *feature declaração* `...)`.
  Cada feature será testada, e o código expandido será a decaração que segue a primeira feature encontrada. Uma *feature* pode ser:

Versão Preliminar

- O nome de uma feature
- `(define-library nome)`, onde `nome` é o nome de um módulo. Se o módulo puder ser improtado, esta cláusula será expandida.
- `(and` *feature* `...)`
- `(or` *feature* `...)`
- `(not` *feature*`)`

B.5.1 Módulos padrão

- `(scheme base)`
- `(scheme inexact)`
- `(scheme complex)`
- `(scheme division)`
- `(scheme lazy)`
- `(scheme eval)`
- `(scheme repl)`
- `(scheme process-context)`
- `(scheme load)`
- `(scheme io)`
- `(scheme file)`
- `(scheme read)`
- `(scheme write)`
- `(scheme char)`
- `(scheme char normalization)`
- `(scheme time)`

Versão Preliminar

B.6 PROCEDIMENTOS E FORMAS ESPECIAIS PADRÃO

A lista a seguir é uma referencia rápida de procedimentos e formas especiais definidas pelo padrão Scheme, agregados por assunto.

Todos os procedimentos são definidos no padrão R^{5}RS, exceto aqueles marcados com "R^{7}RS", que muito provavelmente serão definidos no padrão R^{7}RS.

B.6.1 Controle e ambiente

`(and test1 ...)`
(Forma especial) Avalia as formas uma a uma, parando quando uma delas resultar em `#f`. O valor da forma `and` é o *"e"* lógico dos testes.

`(apply proc args)`
Aplica `proc` com a lista de argumentos `args`. Seção 1.8 e Capítulo 7.

`(begin form1 ...)`
(Forma especial) Avalia as formas em ordem e retorna o valor da última.

`(boolean? obj)`
Verifica se o objeto `obj` é booleano.

`(call/cc proc)`
`(call-with-current-continuation proc)`
Chama o procedimento `proc`, que deve aceitar um único argumento (a continuação corrente). O procedimento `proc` poderá chamar a continuação corrente, que é aquela de quando `call-with-current-continuation` foi chamado. A versão curta, `call/cc`, só será definida oficialmente no padrão R^{7}RS. Capítulo 10.

`(call-with-values f g)`
Chama `f` (que não deve receber argumentos) e usa seus valores de retorno como parâmetro para chamar `g`. Seção 1.8.

`(case key clause1 ...)`
(Forma especial) Cada cláusula é da forma `(lst form1 ...)` ou `(else form1 ...)`. A chave será buscada em cada lista `lst`; quando for encontrada, as formas à frente da lista serão avaliadas. A cláusula `else` sempre causa a avaliação de suas formas. As cláusulas são verificadas na ordem, e somente uma é escolhida para avaliação. Capítulo 1.

`(command-line)`
R^{7}RS Retorna a linha de comando que foi passada ao processo Scheme como uma lista de strings. A primeira string corresponde ao nome do processo, e as outras são os argumentos. Estas strings não são mutáveis.

Versão Preliminar

`(cond clausula1 clausula2 ... [ clausula-else ])`
(Forma especial) Cada cláusula é uma lista cujo primeiro elemento é um teste. Os outros elementos são formas a serem avaliadas se o teste não retornar `#f`. Os testes são avaliados na ordem em que aparecem, e apenas uma cláusula é escolhida para avaliação. A `clausula-else` tem a forma `(else forma1 forma2)`; as formas da cláusula `else` serão avaliadas se nenhuma outra for. Se não há `else` e todos os testes resultam em `#f`, `cond` não retorna valores.

`(define nome expr)`
`(define (nome args) corpo)`
`(define (nome . args) corpo)`
(Forma especial) Cria vínculos para nomes. Capítulo 1

`(define-syntax nome transformador)`
(Forma especial) Define uma macro. Capítulo 8.

`(define-values (nome1 nome2 ...) expr)`
(Forma especial) Cria vínculos para nomes. Os nomes são vinculados a novos locais e os valores retornados por `expr` são armazenados nestes locais. R^{7}RS

`(delay expr)`
Constrói uma *promessa* a partir de `expr`. Se `P` é o resultado de `(delay expr`, a forma `expr` só será avaliada com `(force p)`. Capítulo 11

`(delay-force expr)`
Constrói uma *promessa* a partir de `expr`. Conceitualmente semelhante a `(delay (force expr))`, mas garantindo que a chamada a `force` será recursiva na cauda, evitando que seu uso iterado consuma muita memória. Capítulo 11. R^{7}RS

`(do ((var1 init1 step1) ...) (teste1 expr1...) forma1 ...)`
(Forma especial) Inicializa cada variável com um valor, executa a sequência de formas e avalia cada teste, na ordem. Quando um teste não retorna `#f`, sua expressão é retornada. Se todos os testes retornam `#f`, as variáveis recebem os valores atualizados e o processo se inicia novamente. Capítulo 3.

`(dynamic-wind before thunk after)`
Avalia `thunk`. Cada vez que o interpretador entrar na extensão dinâmica de `thubk`, avaliará `before`. Cada vez que sair dela, avaliará `after`. Capítulo 10.

`(eval expr env)`
Avalia a expressão `expr` no ambiente `env`. Capítulo 7.

`(eq? obj1 obj2)`
`(eqv? obj1 obj2)`
`(equal? obj1 obj2)`
Verificam se dois objetos são iguais. Seção 1.5.

Versão Preliminar

`(exit [ obj ])`

R^{7}RS Termina o programa. Se `obj` é especificado, ele será usado como valor de retorno para o sistema operacional; caso contrário, o valor retornado ao S.O. é o de término normal. Se `obj` for `#f`, presume-se que o término não foi normal.

`(force prom)`

Força a avaliação da promessa `prom`. Capítulo 11.

`(interaction-environment)`

Retorna o ambiente em que as expressões são avaliadas, incluindo procedimentos e formas especiais não padrão, e vínculos criados pelo usuário.

`(lambda args forms)`

(Forma especial) Especifica um procedimento cujos argumentos são definidos pela lista `args` e cujo corpo é dado pelas formas `forms`. Capítulo 1.

`(let vinculos forma1 ...)`

`(let* vinculos forma1 ...)`

`(letrec vinculos forma1 ...)`

Executam formas com vínculos temporários para variáeis. Capítulo 1.

`(let-syntax vínculos forma1 ...)`

`(letrec vinculos forma1 ...)`

Executam formas usando macros temporárias.

`(make-parameter init)`

`(make-parameter init conv)`

R^{7}RS Cria um novo objeto-parâmetro, associado ao valor retornado por `(conv init)`. Se `conv` é omitido, a função identidade é usada.

`(make-promise expr)`

R^{7}RS Constrói uma promessa. O valor retornado é semelhante ao que seria retornado por `(delay expr)`, exceto que `expr` *será* avaliada imediatamente, porque `make-promise` é procedimento, e não forma especial. Capítulo 11.

`(not obj)`

Retorna `#t` se `obj` é `#f`. Em outros casos, retorna `#f`.

`(null-environment)`

Retorna um ambiente onde só existem as formas especiais do padrão Scheme. Capítulo 7.

`(or test1 ...)`

(Forma especial) Avalia os testes um a um, parando quando um deles for diferente de `#f`. O valor da forma `or` é o *"ou"* lógico dos testes.

`(parameterize ((par val) ...) expr1 expr2 ...)`

R^{7}RS As variáveis `par` devem ser objetos-parâmetro (variáveis de escopo dinâmico). As expressões `expr1 expre2 ...` serão avaliadas como em um `let`, mas usando os valores definidos pelos pares `(par val)`. Diferentemente do `let`, o escopo não é léxico.

Versão Preliminar

`(procedure? x)`
Verifica se `x` é um procedimento.
`(promise? x)`
Verifica se `x` é uma promessa (um objeto retornado por `delay`, `delay-force` ou `make-promise`. R^{7}RS Capítulo 11.
`(quasiquote template)`
`'template`
A expansão é expressão `template` não avaliada, exceto pelas partes precedidas por vírgula.
`(quote obj)`
Retorna `obj` sem avaliá-lo. `obj` deve ser a representação externa de um objeto Scheme.
`(scheme-report-environment v)`
Retorna o ambiente que contém apenas os procedimentos e formas especiais definidos na versão `v` do padrão Scheme, que sejam obrigatórios *ou* que sejam opcionais *e* suportados pela implementação. Capítulo 7.
`(set! var expr)`
(Forma especial) Avalia `expr` e armazena o resultado na variável `var`. A variável `var` *deve* ter sido definida antes.
`(syntax-rules palavras-chave regra1 ...)`
Especifica um transformador de sintaxe. Capítulo 8.
`(values obj1 ...)`
Retorna seus argumentos como múltiplos valores, que podem ser capturados por `call-with-values`.

B.6.2 Erros e Exceções

`(error reason [ obj1 ... ])`
Sinaliza um erro. O parâmetro `reason` deve ser uma string. Os parametros `obj1 ...` podem ser de qualquer tipo. R^{7}RS
`(guard (var (clausula1 clausula2 ...)) corpo)`
Corpo será avaliado como em um `begin`. Um tratador de exceções será instalado e quando uma exceção ocorrer, (i) a variável `var` será vinculada ao objeto levantado, (ii) as cláusulas listadas serão avaliadas como cláusulas de um `cond`. A extensão dinâmica e a continuação usadas durante a avaliação deste `cond` implícito são aquelas da cláusula `guard`.
`(error-object? obj)`
Verifica se `obj` é um objeto definido pelo ambiente Scheme[1] como um objeto de erro, *ou* se foi criado pelo procedimento `error`. R^{7}RS

1 Cada ambiente Scheme pode definir diferentes conjuntos de objetos de erro

Versão Preliminar

`(error-object-message error-object)`

R^{7}RS Retorna a mensagem encapsulada em `error-object`.

`(error-object-irritants error-object)`

R^{7}RS Retorna uma lista de "ofensas"[2], encapsulada em `error-object`.

`(raise obj)`

R^{7}RS Levanta uma exceção. O tratador de exceções corrente é chamado com `obj` como parâmetro.

`(raise-continuable obj)`

R^{7}RS FIXME:

`(syntax-error msg arg ...)`

R^{7}RS Usado na expansão de uma macro. O ambiente reportará um erro tão cedo quanto possível (durante a expansão da macro, se a implementação expandir macros antes da execução).

`(with-exception-handler trata corpo)`

R^{7}RS Executa `corpo`, que deve ser um procedimento com zero argumentos. Quando uma exceção é levantada, o procedimento `trata` é chamado e a ele é passado o objeto determinado pelo procedimento `raise`.

B.6.3 Listas

`(append l1 l2 ...)`

Retorna uma lista com os elementos de `l1`, `l2`, ...

`(assoc obj alist [pred?])`

`(assov obj alist)`

`(assoq obj alist)`

Buscam o objeto `obj` na lista de associação `alist`, usando diferentes predicados para comparar igualdade (`equal?`, `eqv?` e `eq?`). Seção 1.12. O predicado opcional `pred?` em `assoc`, quando presente, é usado nas comparações (definido em R^{7}RS).

`(car p)`

Retorna o conteúdo do `car` do par `p`.

`(cdr p)`

Retorna o conteúdo do `cdr` do par `p`.

`(cons obj1 obj2)`

Aloca e retorna uma novo par cujo `car` contém `obj1` e cujo `cdr` contém `obj2`. Há a garantia de que o novo par será diferente de qualquer outro objeto.

2 "Irritants" em Inglês – os motivos do erro.

Versão Preliminar

`(copy-list lst)`
Cria uma cópia de `lst`. R^{7}RS
`(for-each proc lst)`
Aplica o procedimento `proc` a cada elemento da lista `lst`, da esquerda para a direita. O valor de retorno é indefinido.
`(length lst)`
Retorna o tamanho da lista `lst`.
`(list obj ...)`
Aloca e retorna uma lista cujos membros são os objetos passados como parâmetro.
`(list->string lst)`
Retorna a string cujos caracteres são os mesmos que os da lista `lst`.
`(list->vector lst)`
Aloca e retorna um vetor cujos elementos são os mesmos que os da lista `lst`.
`(list-ref lst n)`
Retorna o elemnto de `lst` cujo índice é `n`. O primeiro índice é zero.
`(list-set! lst n obj)`
Armazena `obj` na `n`-ésima posição da lista `lst`. R^{7}RS
`(list-tail lst n)`
Retorna a sublista de `lst` que inicia com o elemento de índice `n`. O primeiro índice é zero.
`(list? obj)`
Verifica se o objeto `obj` é uma lista. Retorna `#t` para a lista vazia.
`(make-list n [obj])`
Cria uma lista de tamanho `n`. Se o argumento opcional `obj` existir, ele será usado como valor inicial em cada posição; caso contrário, qualquer valor poderá ser usado pela implementação de Scheme. R^{7}RS
`(map proc lst1 ...)`
Retorna a lista cujo n-ésimo elemento é o resultado da aplciação de `proc`, com a lista de argumentos igual à lista de n-ésimos elementos de cada uma das listas. A aridade de `proc` *deve* ser igual ao número de listas.
`(member obj lst [pred?])`
`(memq obj lst)`
`(memv obj lst)`
Se `obj` está na lista `lst`, retornam a sublista que começa com `obj`; caso contrário, retornam `#f` Diferem nos procedimentos usados para testar igualdade entre elementos (`equal?`, `eq?` e `eqv?`). O predicado opcional `pred?` em `member`, quando presente, é usado nas comparações (definido em R^{7}RS).
`(null? lst)`
Verifica se a lista `lst` é vazia.

Versão Preliminar

`(pair? obj)`
Verifica se `obj?` é um par.
`(reverse lst)`
Aloca e retorna uma lista com os mesmos elementos de `lst`, na ordem inversa.
`(set-car! par obj)`
Armazena `obj` no `car` de `par`.
`(set-cdr! par obj)`
Armazena `obj` no `cdr` de `par`.

B.6.4 Números

`(+ ...)`
Retorna o somatório dos argumentos, ou zero se nenhum argumento for passado.
`(* ...)`
Retorna o produtório dos argumentos, ou um se nenhum argumento for passado.
`(- x1 ...)`
Retorna $0 - x1 - x2 - \cdots - x_n$.
`(/ x1 ...)`
Retorna $(\cdots((1/x1)/x2)/\cdots x_n)$.
`(< x1 x2 ...)`
`(> x1 x2 ...)`
`(<= x1 x2 ...)`
`(>= x1 x2 ...)`
`(= x1 x2 ...)`
Verifica se a lista de números é estritamente crescente, estritamente decrescente, não decrescente, não crescente, ou se todos os argumentos são números iguais.
`(abs x)`
Retorna o valor absoluto de `x`.
`(acos x)`
Retorna o arcocosseno de `x`.
`(angle z)`
Retorna o ângulo formado entre o vetor z e o eixo das abscissas no plano complexo.
`(asin x)`
Retorna o arcosseno de `x`.
`(atan x)`
`(atan b a)`

Versão Preliminar

Retorna o arcotangente de `x` ou `(angle (make-rectangular a b))`: o ângulo formado entre o vetor (a, b) no plano complexo e o eixo das abscissas.

`(ceiling x)`

Retorna $\lceil x \rceil$, o menor inteiro maior ou igual a `x`.

`(complex? z)`

Verifica se um objeto é um número complexo. (Opcional)

`(cos x)`

Retorna o cosseno de `x`.

`(denominator q)`

Retorna o denominador do número racional `q`.

`(even? x)`

Verifica se o número `x` é par.

`(exact? x)`

Verifica se a representação do número `x` é exata.

`(exact-integer n)`

Verifica se `n` é inteiro *e* exato. R^{7}RS

`(exact-integer-sqrt n)`

Retorna $\lfloor \sqrt{n} \rfloor$, o maior inteiro menor ou igual à raiz de `n` R^{7}RS

`(exp x)`

Retorna e^x.

`(expt a b)`

Retorna a^b.

`(finite? x)`

Verifica se `x` é diferente de `+inf` e de `-inf`. R^{7}RS

`(floor x)`

Retorna $\lfloor x \rfloor$, o maior inteiro menor ou igual a `x`.

`(gcd n1 ...)`

Retorna o máximo denominador comum dos números passados como argumentos.

`inexact? x`

Verifica se a representação do número `x` é inxata.

`(integer? x)`

Verifica se o objeto `x` é um número inteiro.

`(lcm n1 ...)`

Retorna o mínimo múltiplo comum dos números passados como argumentos.

`(log x)`

Retorna o logaritmo de `x` na base *e*.

`(magnitude z)`

Retorna a magnitude do vetor que representa o número complexo `z`.

Versão Preliminar

`(make-polar mag ang)`
Retorna o número complexo cujas coordenadas polares no plano complexo são `(mag, ang)`.
`(make-rectangular a b)`
Retorna o número complexo `a` + `b`i.
`(max x1 ...)`
Retorna o maior dos números passados como argumento.
`(min x1 ...)`
Retorna o menor dos números passados como argumento.
`(modulo a b)`
Retorna `a` mod `b`. `floor-reminder` é sinônimo.
`(nan? x)`
R^{7}RS Verifica se `x` é o objeto NaN *(not a number)*.
`(negative? x)`
Verifica se o número `x` é negativo.
`(numerator q)`
Retorna o numerador do número racional `q`.
`(odd? x)`
Verifica se o número `x` é ímpar.
`(positive? x)`
Verifica se o número `x` é positivo (e diferente de zero).
`(quotient a b)`
`(truncate-quotient a b)`
Retorna o quociente inteiro da divisão `a`/`b`. `truncate-quotient` é sinônimo.
`(rational? x)`
Verifica se `x` é um número racional.
`(rationalize x y)`
Retorna o racional mais simples que difere de `x` por no máximo `y`. De acordo com o padrão R^{5}RS, `(rationalize (inexact->exact .3) 1/10)` é igual ao racional *exato* 1/3, e `(rationalize .3 1/10)` é igual ao *inexato* `#i`1/3.
`(real-part z)`
Retorna a parte real do número complexo `z`.
`(real? x)`
Verifica se um objeto é um número real.
`(remainder a b)`
`(truncate-reminder a b)`
Retorna o resto da divisão `a`/`b`. `truncate-reminder` é sinônimo.

Versão Preliminar

`(round x)`
Retorna o inteiro mais próximo de `x`. Quando há empate entre dois inteiros igualmente próximos de `x`, o par será retornado.
`(sin x)`
Retorna o seno do ângulo `x`.
`(sqrt x)`
Retorna $\sqrt{x}$.
`(tan x)`
Retorna a tangente do ângulo `x`.
`(truncate x)`
Retorna o valor de `x`, ignorando a parte fracionária. O tipo retornado é inteiro.
`(zero? x)`
Verifica se um número é zero.

B.6.5 Strings, símbolos e caracteres

`(char->integer x)`
Retorna uma representação do caracter como número inteiro. Não há a exigência de que esta representação coincida com ASCII ou Unicode, mas ela deve ser tal que possa ser revertida pelo procedimento `integer->char`.
`(char-alphabetic? c)`
Verifica se `c` é um caracter alfabético (maiúsculo ou não).
`(char-ci<=? a b)`
`(char-ci<? a b)`
`(char-ci=? a b)`
`(char-ci>=? a b)`
`(char-ci>? a b)`
Verificam se o caracter `a` precede, é precedido por ou se é igual a `b`. Não levam em conta diferença entre caixa alta e baixa.
`(char-downcase c)`
Retorna o caracter `c` em caixa baixa.
`(char-lower-case? c)`
Verifica `c` é um caracter em caixa baixa.
`(char-numeric? c)`
Verifica `c` é um caracter numérico (um dígito).
`(char-upcase c)`
Retorna o caracter `c` em caixa alta.

Versão Preliminar

`(char-upper-case? c)`
Verifica `c` é um caracter em caixa alta.
`(char-whitespace? c)`
Verifica `c` é o caracter espaço em branco.
`(char<=? a b)`
`(char<? a b)`
`(char=? a b)`
`(char>=? a b)`
`(char>? a b)`
Verificam se o caracter `a` precede, é precedido por ou se é igual a `b`.
`(char? c)`
Verifica se o objeto `c` é um caracter.
`(make-string n [ c ])`
Aloca e retorna uma string de tamanho `n`. Se o caracter `c` for especificado, será copiado em cada posição da string.
`(number->string x)`
Aloca e retorna uma sring contendo a representação textual do número `x`.
`(string c1 ...)`
Aloca e retorna uma string formada pelos caracteres passados como argumento.
`(string->list s)`
Retorna uma lista com os caracteres da string `s`.
`(string->number s [ rad ])`
Retorna o número representado na string, usando a base `rad` quando for especificada ou dez na sua ausência. A base deve ser 2, 8, 10, ou 16. Se `rad` for usado mas também houver um prefixo na string indicando a base, como em `"#o332"`, o preciso na string terá precedência.
`(string->symbol s)`
Retorna o símbolo cujo nome é `s`.
`(string->vector s)`
R^{7}RS Retorna um vetor com os caracteres de `s`.
`(string-append s1 ...)`
Aloca e retorna uma string cujo conteúdo é a concatenação das strings passadas como parâmetro.
`(string-ci<=? a b)`
`(string-ci<? a b)`
`(string-ci=? a b)`
`(string-ci>=? a b)`
`(string-ci>? a b)`

Versão Preliminar

Verificam se a string `a` precede, é precedida ou se é igual a `b` de acordo com uma ordem lexicográfica. Não levam em conta diferença entre caixa alta e baixa.

`(sting-copy str)`

Aloca e retorna uma cópia da string `s`.

`(string-fill! str c [ pos-a pos-b ])`

Armazena `c` em todas as posições de `str`. Os parâmetros opcionais `pos-a` e `pos-b` marcam o primeiro e o último índice das posições a serem preenchidas. R^{7}RS

`(string-for-each proc str)`

Aplica proc em cada caracter da string `str`, da esquerda para a direita.

`(string-length str)`

Retorna o tamanho da string `s`.

`(string-map proc str1 ...)`

Retorna a string cujo n-ésimo caracter é o resultado da aplciação de `proc`, com a lista de argumentos igual à lista de n-ésimos caracter de cada uma das strings. A aridade de `proc` *deve* ser igual ao número de strings. R^{7}RS

`(string-ref s pos)`

Retorna o caracter na posição `pos` da string `s`. O primeiro caracter tem índice zero.

`(string-set! s pos c)`

Copia o caracter `c` na posição `pos` da string `s`. O primeiro caracter tem índice zero.

`(string-ci<=? a b)`

`(string-ci<? a b)`

`(string-ci=? a b)`

`(string-ci>=? a b)`

`(string-ci>? a b)`

Verificam se a string `a` precede, é precedida ou se é igual a `b` de acordo com uma ordem lexicográfica.

`(string? obj)`

Verifica se o objeto `obj` é uma string.

`(substring str start end)`

Aloca e retorna uma string com os caracteres da string `str` da posição `start` até `end`. A posição `start` é incluída, mas `end` não.

`(symbol->string s)`

Retorna a representação do nome do símbolo `s` como string.

`(symbol? obj)`

Verifica se o objeto `obj` é um símbolo.

Versão Preliminar

B.6.6 Vetores

`(copy-vector vec)`
Retorna uma cópia do vetor `vec`.
`(make-vector n [ obj ])`
Aloca e retorna um vetor de tamanho `n`. Se `obj` for especificado, será usado como elemento inicial em cada posição do vetor.
`(vector obj1 ...)`
Aloca e retorna um vetor cujos elemntos são os argumentos.
`(vector->list vec)`
Retorna uma lista com os mesmos elementos do vetor `vec`.
`(vector->string vec)`
R^{7}RS Retorna uma string com os caracteres em `vec`.
`(vector-fill! vec obj [ pos-a pos-b ])`
Armazena `obj` em todas as posições do vetor `vec`. Os parâmetros opcionais `pos-a` e `pos-b`
R^{7}RS marcam o primeiro e o último índice das posições a serem preenchidas.
`(vector-for-each proc vec)`
Aplica `proc` em cada elemento do vetor `vec`, da esquerda para a direita.
`(vector-length vec)`
Retorna o tamanho do vetor `vec`.
`(vector-map proc vec1 ...)`
R^{7}RS Retorna o vetor cujo n-ésimo elemento é o resultado da aplciação de `proc`, com a lista de argumentos contendo os n-ésimos elementos de cada uma dos vetores. A aridade de `proc` *deve* ser igual ao número de vetores (semelhante a `map` para listas).
`(vector-ref vec pos)`
Retorna o objeto na posição `pos` do vetor `vec`.
`(vector-set! vec pos obj)`
Armazena o objeto `obj` na posição `pos` do vetor `vec`.
`(vector? obj)`
Verifica se `obj` é um vetor.

B.6.7 Bytevectors (R^{7}RS)

`(bytevector? obj)`
R^{7}RS Verifica se `obj` é um bytevector.
`(bytevector-copy bv)`
R^{7}RS Retorna uma cópia de um bytevector.

Versão Preliminar

`(bytevector-copy! bv1 bv2)`
Copia os bytes de `bv1` para `bv2` (`bv2` deve ser maior ou igual a `bv1`. R^{7}RS
`(bytevector-length bv)`
Retorna o tamanho do bytevector em bytes. R^{7}RS
`(bytevector-u8-ref bv k)`
Retorna o k-ésimo byte do bytevector. R^{7}RS
`(bytevector-u8-set! bv k b)`
Modifica o k-ésimo byte do bytevector, guardando ai o valor `b`. R^{7}RS
`(make-bytevector k)`
Retorna um novo bytevector com `k` bytes e conteúdo indefinido. R^{7}RS
`(partial-bytevector blob inicio fim)`
Retorna um novo bytevector com os bytes do bytevector original entre as posições `inicio`e `fim` (inclusive). R^{7}RS
`(partial-bytevector-copy! bv1 inicio bv2 inicio2)`
Copia os bytes entre as posições `inicio` e `fim` de `bv1` para `v2`, armazenando-os a partir da posição `inicio2` do bytevector de destino (`bv2`). R^{7}RS

B.6.8 Entrada e saída

`(binary-port? obj)`
Verifica se `obj` é uma porta binária. R^{7}RS
`(call-with-input-file str proc)`
Chama `proc`, que deve ser um procedimento sem argumentos, trocando a entrada corrente pela porta resultante da abertura do arqivo de nome `str`.
`(call-with-output-file str proc)`
Chama `proc`, que deve ser um procedimento sem argumentos, trocando a saída corrente pela porta resultante da abertura do arqivo de nome `str`.
`(textual-port? obj)`
Verifica se `obj` é uma porta de texto. R^{7}RS
`(char-ready? port)`
Verifica se um caracter pode ser lido da porta de texto `port`. R^{7}RS
`(close-input-port port)`
Fecha a porta de entrada `port`.
`(close-output-port port)`
Fecha a porta de saída `port`. Quaisquer dados em buffer são gravados antes da porta ser fechada.

Versão Preliminar

`(close-port port)`

R^{7}RS Fecha a porta `port`. Quaisquer dados em buffer são gravados antes da porta ser fechada, se a porta for de saída.

`(current-error-port)`

R^{7}RS Retorna a porta atual de saída para erros.

`(current-input-port)`

Retorna a porta atual de entrada.

`(current-output-port)`

Retorna a porta atual de saída.

`(delete-file str)`

R^{7}RS Remove o arquivo cujo nome é dado pela string `str`.

`(display obj [ port ])`

Imprime `obj` na porta `port`, ou na saída corrente se `port` for omitida. Capítulo 2.

`(eof-object? obj)`

Verifica se o objeto `obj` é o objeto que representa fim de arquivo.

`(file-exists? str)`

R^{7}RS Verifica se o arquivo cujo nome é `str` existe.

`(flush-output-port [ port ])`

R^{7}RS Esvazia quaisquer *buffers* usados na porta `port`, forçando a saída de dados. Se `port` for omitida, a saída corrente é usada.

`(get-output-string port)`

R^{7}RS Retorna a string com os caracteres escritos em `port`, que deve ser uma porta de saída para string criada por `open-output-string`.

`(load arq [ env ])`

Lê (usando `read`) e avalia todas as formas no arquivo cujo nome é `arq`. O argumento opcional `env` é o ambiente onde as formas serão avaliadas. Se `env` não é especificado, `(interaction-environment)` é usado. (Opcional)

`(newline [ port ])`

Escreve um fim-de-linha na porta `port`, ou na saída atual se `port` for omitida.

`(open-input-file str)`

Abre o arquivo cujo nome é `str` e retorna uma porta de entrada associada a ele.

`(open-input-string str)`

R^{7}RS Abre uma porta de entrada para leitura de dados da string `str`.

`(open-output-file str)`

Abre o arquivo cujo nome é `str`e retorna uma porta de saída associada a ele.

`(open-output-string)`

R^{7}RS Abre uma porta de entrada para saída de dados que serão acumulados em uma string (veja `get-output-string`).

Versão Preliminar

`(output-port? obj)`
Verifica se `obj` é uma porta de saída.
`(peek-char [ port ])`
Retorna o próximo caracter a ser lido da porta `port`, ou da entrada padrão se `port` for omitida. *Não* consome o caracter.
`(port-open? obj)`
R^{7}RS Verifica se a porta `x` está aberta.
`(port? obj)`
R^{7}RS Verifica se o objeto `obj` é uma porta.
`(read [ port ])`
Lê um objeto Scheme, na sua representação externa, de `port` e o retorna. Se `port` não for especificada, a porta atual de entrada será usada.
`(read-char [ port ])`
Consome um caracter da porta de texto `port` e o retorna. Se `port` não for especificada, a porta atual de entrada será usada.
`(read-u8 [ port ])`
Retorna o próximo byte da porta binária `port`. Se `port` não for especificada, a porta atual de entrada será usada. R^{7}RS só
`(transcript-off)` R^{5}RS
Desativa o efeito de `transcript-on`. (removido em R^{7}RS) só
`(transcript-on name)` R^{5}RS
Inicia uma transcrição da interação com o REPL no arquivo cujo nome é `name`. (removido em R^{7}RS)
`(u8-ready? [ port ])`
Verifica se um caracter pode ser lido da porta de texto `port`. R^{7}RS
`(with-input-from-file name proc)`
Executa `proc` (que deve ser um procedimento sem argumentos) trocando a porta de entrada corrente pela porta de entrada que resulta da abertura do arquivo cujo nome é `name`.
`(with-output-to-file name proc)`
Executa `proc` (que deve ser um procedimento sem argumentos) trocando a porta de saída corrente pela porta de entrada que resulta da abertura do arquivo cujo nome é `name`.
`(write obj [ port ])`
Escreve o objeto `obj` na porta `port`, ou na porta atual de saída se `port` for omitida. O objeto será escrito usando a representação externa de Scheme, e pode ser lido novamente por `read`.
`(write-char c [ port ])`
Escreve o caracter `c` na porta `port`, ou na porta atual de saída se `port` for omitida.

Versão Preliminar

`(write-simple obj [ port ])`

R^{7}RS Escreve o objeto `obj` na porta `port`, ou na porta atual de saída se `port` for omitida. O objeto será escrito usando a representação externa de Scheme, *sem* usar os rótulos de auto-referência (e portanto `write-simple` pode não terminar, se `obj` for uma estrutura cíclica. A representação pode ser lida novamente por `read`.

B.6.9 Registros

`(define-record-type nome cons pred? [ campo1 campo2 ...])`

R^{7}RS Define um novo tipo composto de dados (um "registro") com nome `nome`. O construtor é definido em `cons`, que deve ser da forma: `(cons-name campoA campoB ...)`. O predicado de tipo será `pred?`, e para testar se um objeto é deste novo tipo usa-se `(pred? obj)`. Os símbolos `campo1, ...` especificam os campos internos dos tipos, e devem cada um deve ser da forma `(nome-campo acessor)` ou `(nome-campo acessor modificador)`.

B.6.10 Tempo

`(current-jiffy)`

R^{7}RS Retorna a quantidade de *jiffies* passados desde uma época. Um *jiffy* é uma fração de segundo, dependente da implementação de Scheme.

`(jiffies-per-second)`

R^{7}RS Retorna o número de *jiffies* por segundo, que deve ser um inteiro exato.

`(current-seconds)`

R^{7}RS Retorna o número corrente de segundos de acordo com a escala de tempo atômico internacional (TAI). O valor 0.0 representa dez segundos após a meia-noite de 1º de Janeiro de 1970.

Versão Preliminar

C | THREADS POSIX

(este apêndice está incompleto)

Este Apêndice descreve brevemente a API para programação com threads definida no padrão POSIX [IEE03]. Usando exclusivamente a API definida neste padrão o programador torna seu código mais portável entre diferentes plataformas to tipo UNIX.

Além da seção do padrão POSIX a respeito de threads, é descrita aqui a API para acesso a semáforos (que no documento POSIX está na seção de comunicação interprocessos).

O conhecimento de threads POSIX pode ser útil aos interessados na implementação de interpretadores Scheme – o que é comum, dada a simplicidade do núcleo da linguagem (um interpretador simples pode ser construído por um bom programador C em menos de dos dias). Para o leitor interessado em construir interpretadores mais elaborados será interessante consultar os livros de Christian Queinnec [Que03] a respeito de implementação de Lisp, o de Lins e Jones [JL96] sobre coleta de lixo. Para a implementação de máquinas virtuais há bons livros de Ian Craig [Cra05] e de Smith e Nair [SN05]. A literatura sobre construção de compiladores é bastante madura e rica; são particularmente interessantes os livros de Cooper e Torczon [CT03], Andrew Appel [App04] e Ken Louden [Lou04].

C.1 CRIAÇÃO E FINALIZAÇÃO DE THREADS

```
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg);
```

Cria uma thread. Um identificador será gravado em `thread_id`; O ponto de entrada da thread é `start_routine`, que será chamada com um único argumento, `arg`. O argumento `pthread_attr_t` contém atributos da thread, e pode ser `NULL` (para criação de threads com atributos default) ou uma estrutura inicializada com `pthread_attr_init`.

```
int pthread_join(pthread_t thread_id, void** status);
```


Versão Preliminar

Espera até que a thread `thread_id` termine. O valor retornado por `thread_id` estará disponível em `*status`

```
void pthread_exit(void* status)
```

Termina a thread sendo executada, retornando `*status`, que pode ser obtido pela thread que estiver aguardando por esta (veja `pthread_join`).

```
int sched_yield(void);
```

A thread que chama esta função retorna o controle para o escalonador, como se seu tempo se houvesse esgotado. Retorna zero em caso de sucesso ou -1 em caso de erro.

```
int pthread_join(pthread_t thread_id, void** status);
```

Bloqueia a thread chamadora até que `thread_id` termine. Se `status` for diferente de `NULL`, o valor de retorno da thread sendo aguardada será copiado no local onde `*status` aponta. Se a thread aguardada for cancelada, `PTHREAD_CANCELED` é copiado onde `*status` aponta.
O valor de retorno de `pthread_join` é zero em caso de sucesso ou um código de erro em caso de falha. Os possíveis erros são:
`EDEADLK`: um deadlock foi detectado.
`EINVAL`: não é possível realizar *join* nesta thread.
`ESRCH`: a thread `thread_id` não foi encontrada.

C.2 SINCRONIZAÇÃO

C.2.1 Mutexes

Mutexes podem ser criados da seguinte forma:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Inicializa um mutex. `mutexattr` determina os atributos do mutex, e pode ser `NULL` quando se quer os atributos default. O valor de retorno sempre é zero.

Para travar e liberar mutexes há as funções `pthread_mutex_lock` e `pthread_mutex_unlock`.

Versão Preliminar

```
pthread_mutex_lock(pthread_mutex_t *mutex);
pthread_mutex_trylock(pthread_mutex_t *mutex);
pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutexes são desalocados com `pthread_mutex_destroy`

```
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

C.2.2 Semáforos

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

Estas funções criam semáforos com nome.

```
int sem_close(sem_t *sem);
```

Finaliza um semáforo. Retorna zero em caso de sucesso ou -1 em caso de erro (quando também modifica `errno`).

```
int sem_post(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem,
                  const struct timespec *abs_timeout);

int sem_getvalue(sem_t *sem, int *sval);
```

O valor do contador interno do semáforo `*sem` é armazenado em `*sval`. Retorna 0 em caso de sucesso; em caso de erro retorna -1 e modifica `errno`.

Semáforos com nome são particularmente interessantes porque podem ser compartilhados por diferentes programas: `sem_open` abre um semáforo com nome (o nome do

Versão Preliminar

semáforo é tratado como um nome de arquivo), retornando um ponteiro para uma estrutura do tipo `sem_t`. A assinatura da função é `sem_open (const char *nome, int flag, int modo, unsigned valor)`. Os argumentos `nome`, `flag` e `modo` funcionam exatamente como os argumentos de `open` para arquivos; o argumento `valor` é usado para inicializar o contador interno do semáforo. O exemplo a seguir mostra o uso de semáforos POSIX em C.

```
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

int main() {
        char n [50];

        /* Cria um semáforo inicializado com 1: */
        sem_t *s = sem_open("nome", O_CREAT, 0666, 1);

        /* wait: */
        sem_wait(s);

        printf("Digite algo!\n");
        scanf("%49s",n);
        printf("OK, %s\n",n);

        /* signal: */
        sem_post(s);

        exit(EXIT_SUCCESS);
}
```

Diferentes programas (escritos em diferentes linguagens) podem usar o mesmo semáforo com nome, que é acessado de maneira semelhante a um arquivo. Os programas a seguir, em Haskell e Perl, usam o mesmo semáforo do programa anterior em C.

Versão Preliminar

```
import Control.Exception
import System.Posix.Semaphore

main = do
    s <- semOpen "c" (OpenSemFlags True False) 0666 1
    semThreadWait s
    putStrLn "Type!"
    x <- getLine
    putStrLn $ "OK, " ++ x
    semPost s
```

```
#!/usr/bin/perl

use POSIX::RT::Semaphore;
use Fcntl;

my $sem = POSIX::RT::Semaphore->open("c",O_CREAT, 0666, 1);

$sem->wait;
print "Type!\n";
my $a = readline;
print "OK, $a";
$sem->post;
```

C.2.3 Variáveis de condição

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_cond_init(pthread_cond_t* var, attr);
pthread_cond_destroy(pthread_cond_t *cond);
```

O primeiro argumento de `pthread_cond_init` deve ser um ponteiro para uma estrutura do tipo `pthread_cond_t`.

```
pthread_cond_signal(pthread_cond_t *var);
```

Versão Preliminar

```
pthread_cond_wait(pthread_cond_t *var, pthread_mutex_t *mutex);
```

A função `pthread_cond_signal` sinaliza uma variável de condição.

Quando `pthread_cond_wait` for chamado, o mutex passado como segundo argumento já deve estar travado. O mutex será destravado e a thread será bloqueada (tudo atomicamente). Quando outra thread sinalizar a variável, a thread voltará a executar e o mutex será adquirido novamente por `pthread_cond_wait`.

C.2.4 Barreiras

Barreiras são do tipo `pthread_barrier_t`.

```
int pthread_barrier_init (pthread_barrier_t *restrict barrier,
                          const pthread_barrierattr_t *restrict attr,
                          unsigned count);
```

Inicializa um objeto do tipo barreira, alocando todos os recursos necessários. Pode-se inicializar atributos da barreira em `attr`, ou usar `NULL` se não serão usados. O número de threads participando da barreira é passado no argumento `count`. Retorna zero em caso de sucesso, ou um número indicando erro:
`EAGAIN`: o sistema não tem recursos para inicializar mais uma barreira.
`EINVAL`: o valor de count é zero.
`ENOMEM`: não há memória suficiente para inicializar a barreira.

```
int pthread_barrier_wait (pthread_barrier_t *barrier);
```

A thread chamadora ficará bloqueada até que todas as `n` threads tenham chamado esta função. Retorna `PTHREAD_BARRIER_SERIAL_THREAD` para uma das threads (escolhida arbitrariamente) e zero para todas as outras. Se o valor de retorno não for zero ou `PTHREAD_BARRIER_SERIAL_THREAD`, houve um erro.

```
int pthread_barrier_destroy (pthread_barrier_t *barrier);
```

Destrói e desaloca o objeto barreira. Retorna zero em caso de sucesso, ou um número indicando erro.

Versão Preliminar

C.3 MENSAGENS

```
mqd_t mq_open(const char *name, int oflag);
```

Cria uma nova fila de mensagens ou abre uma fila existente. A fila é identificada por `name`.

```
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,
              size_t msg_len, unsigned msg_prio);
```

Envia a mensagem `msg_ptr` para a fila `mqdes`. O tamanho da mensagem deve ser especificado no argumento `msg_len`, e pode ser zero. O argumento `mas_prio` determina a prioridade que esta mensagem terá na fila (valores maiores indicam prioridade mais alta). Retorna zero em caso de sucesso ou -1 em caso de erro (quando também é modificado o valor de `errno`). O tipo `mqd_t` é numérico, mas sua definição exata pode variar de um sistema para outro.

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned *msg_prio);
```

Remove a mensagem mais antiga com a maior prioridade da fila `mqdes`, deixando-a no buffer `msg_ptr`. O argumento `msg_len` determina o tamanho da mensagem. Se `msg_prio` for diferente de `NULL`, o local para onde ele aponta é modificado de forma a conter a prioridade da mensagem.

```
mqd_t mq_notify(mqd_t mqdes, const struct sigevent *notification);
```

Permite a um processo registrar-se para ser notificado quando uma mensagem chega em uma fila.

(a descrição de `mq_notify` está incompleta)

```
mqd_t mq_close(mqd_t mqdes);
```

Fecha o fila de mensagens `mqdes`.

(a descrição de `mq_close` está incompleta)

```
mqd_t mq_unlink(const char *name);
```

Versão Preliminar

Remove a fila de mensagens identificada por `name`. A fila é destruída e todos os processos que a haviam aberto fecham o descritor que tinham. Retorna zero em caso de sucesso ou -1 em caso de erro, quando também é modificado o valor de `errno`.

EXERCÍCIOS

Ex. 213 — Reveja exemplos e exercícios da parte III do livro e reescreva-os em C usando threads POSIX.

Ex. 214 — Implemente um interpretador Scheme com suporte a threads.

Ex. 215 — Implemente um *compilador* Scheme com suporte a threads.

Versão Preliminar

FICHA TÉCNICA

Este texto foi produzido inteiramente em LaTeX em sistemas Linux. Os diagramas foram criados sem editor gráfico, usando diretamente o pacote Ti*k*Z. Os programas Scheme foram desenvolvidos e testados em diversas implementações, dentre as quais as mais usadas foram Chicken Scheme, Chibi Scheme, Guile e Gauche. O ambiente Emacs foi usado para edição do texto LaTeX e também como ambiente de programação Scheme.

As fontes usadas foram Bera (no corpo do texto) e Computer Modern (`cmtt`, nos trechos com espaçamento fixo).

Versão Preliminar

Versão Preliminar

BIBLIOGRAFIA

[Agh85] Gul A. Agha. *Actors: a model of concurrent computation in distributed systems*. Rel. téc. MIT, 1985.

[Aït91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. É possível encontrar reproduções livros do livro em formato PDF. MIT Press, 1991. ISBN: 0262510588.

[And99] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 1999. ISBN: 0201357526.

[App04] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 2004.

[AS96] Harold Abelson e Gerald Sussman. *Structure and Interpretation of Computer Programs*. 2ª ed. Cambridge, Massachusetts: MIT Press, 1996.

[BH74] J.R. Bunch e J.E. Hopcroft. "Triangular factorization and inversion by fast matrix multiplication". Em: *Mathematics of Computation* 28.125 (1974), pp. 231–236.

[BP05] Kenneth A. Berman e Jerome L. Paul. *Algorithms: sequential, parallel and distributed*. Thomson, 2005. ISBN: 0-534-42057-5.

[Bra11] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2011. ISBN: 0321417461.

[Bri75] P. Brinch Hansen. "The programming language Concurrent Pascal". Em: *IEEE Transactions on Software Engineering* 2 (Junho) (1975).

[Bro04] Leo Brodie. *Thinking Forth*. Também disponível livremente em `http://thinking-forth.sourceforge.net/`. Punchy Publishing, 2004.

[Car84] M. Carlsson. *LM-Prolog - The Language and its Implementation*. Rel. téc. 30. UPMAIL, 1984.

[CM03] William F. Clocksin e Christopher S. Mellish. *Programming in Prolog: using the ISO standard*. 5ª ed. Springer, 2003. ISBN: 3-540-00678-8.

[CM84] K. Clark e F. McCabe. *Micro-Prolog: Programming in Logic*. Prentice-Hall, 1984.

[CME83] K. Clark, F. McCabe e J. R. Ennals. *Sinclair ZX Spectrum micro-Prolog Primer*. Sinclair Research Ltd, 1983.

Versão Preliminar

[CNV98] Michael A. Covington, Donald Nute e André Vellino. *Prolog Programming in Depth*. Glennview: Scott, Foresman e Company, 1998. ISBN: 0-673-18659-8.

[Con03] Pascal Constanza. "Dynamically Scoped Functions as the Essence of AOP". Em: *Proceedings of the ECOOP 2003 Workshop on Object-oriented Language Engineering for the Post-Java Era*. 2003.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2009. ISBN: 0262033844.

[Cra05] Ian Craig. *Virtual Machines*. Springer, 2005. ISBN: 1852339691.

[Cro95] Richard M. Crownover. *Introduction to Fractals and Chaos*. Jones & Bartlett, 1995.

[CT03] Keith Cooper e Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2003.

[deJ06] Kenneth A. deJong. *Evolutionary Computation: a unified approach*. MIT Press, 2006. ISBN: 0262041944.

[Dij65] E. W. Dijkstra. *Cooperating sequential processes*. Technological University, Eindhoven. 1965.

[Dow09] Allen B. Downey. *The Little Book of Semaphores*. 2ª ed. CreateSpace, 2009. ISBN: 1441418687.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, 2009.

[ES03] Agoston E. Eiben e J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003. ISBN: 3540401849.

[Fel+03] M. Felleisen et al. *How to Design Programs*. MIT Press, 2003.

[FF95] Daniel P. Friedman e Matthias Felleisen. *The Seasoned Schemer*. MIT Press, 1995. ISBN: 026256100X.

[Fol+95] James D. Foley et al. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, 1995. ISBN: 978-0201848403.

[FW08] Daniel Friedman e Mitchell Wand. *Essentials of Programming Languages*. 3ª ed. MIT Press, 2008.

[Gen03] James E. Gentle. *Random Number Generation and Monte Carlo Methods*. 2ª ed. Springer, 2003. ISBN: 0-387-0017-6.

[Gra93] Paul Graham. *On Lisp*. Prentice Hall, 1993.

[HLR10] Tim Harris, James Larus e Ravi Rajwar. *Transactional Memory*. 2ª ed. Morgan & Claypool, 2010. ISBN: 1608452352.

Versão Preliminar

[HM93] Maurice Herlihy e J. Eliot B Moss. "Transactional memory: Architectural support for lock-free data structures". Em: *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*. 1993, pp. 289–300.

[Hoa74] A. C. Hoare. "Monitors: an operating system structuring concept". Em: *Communications of the ACM* 17.10 (1974).

[Hoy08] Doug Hoyte. *Let Over Lambda*. Lulu.com, 2008. ISBN: 1435712757.

[IEE03] IEEE. *Portable Operating System Interface (POSIX)*. IEEE, 2003.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. 2ª ed. Lua.org, 2006. ISBN: 8590379825.

[JL96] Richard Jones e Rafael D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[Kni86] Tom Knight. "An architecture for mostly functional languages". Em: *Proceedings of the 1986 ACM conference on LISP and functional programming*. 1986.

[Knu05] Donald E. Knuth. *The Art of Computer Programming. Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley, 2005. ISBN: 0201853930.

[Knu98a] Donald E. Knuth. *The Art of Computer Programming*. 3ª ed. Vol. 2. Addison-Wesley, 1998. ISBN: 0-201-89684-2.

[Knu98b] Donald E. Knuth. *The Art of Computer Programming*. 2ª ed. Vol. 3. Addison-Wesley, 1998. ISBN: 0-201-89685-0.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. ISBN: 0262111705.

[Kri07] Shriram Krishnamurti. *Programming Languages: Application and Interpretation*. Shriram Krishnamurti, 2007.

[Kri08] Shriram Krishnamurti. "Teaching Programming Languages in a Post-Linnaean Age". Em: *SIGPLAN Workshop on Undergraduate Programming Language Curricula*. 2008.

[Lew95] Simon Lewis. *The Art and Science of Smalltalk*. Prentice Hall, 1995. ISBN: 0133713458.

[Lou04] Kenneth C. Louden. *Compiladores: princípios e práticas*. Thomson, 2004.

[LP10] William B. Langdon e Riccardo Poli. *Foundations of Genetic Programming*. Springer, 2010. ISBN: 3642076327.

[MR07] João Pavão Martins e Maria dos Remédios Cravo. *PROGRAMAÇÃO EM SCHEME: introdução à programação usando múltiplos paradigmas*. IST Press, 2007. ISBN: 978-972-8469-32-0.

[NM95] Ulf Nilsson e Jan Małuszyński. *Logic, Programming and Prolog*. John Wiley & Sons, 1995. ISBN: 978-0471959960.

Versão Preliminar

[Nor91] Peter Norvig. "Correcting a Widespread Error in Unification Algorithms". Em: *Software, Practice and Experience* 21 (1991), pp. 231–233.

[Nor92] Peter Norvig. *Paradigms of Artificial Inteligence Programming*. Morgan Kaufmann, 1992. ISBN: 1-55860-191-0.

[Oka96] Chis Okasaki. "The Role of Lazy Evaluation in Amortized Data Structures". Em: *Proceedings of ICFP'96*. 1996.

[Oka99] Chis Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. ISBN: 0521663504.

[OKe09] Richard O'Keefe. *The Craft of Prolog*. MIT Press, 2009. ISBN: 978-0262512275.

[PEG10] Slava Pestov, Daniel Ehrenberg e Joe Groff. "Factor: A Dynamic Stack-based Programming Language". Em: *Proceedings of DLS 2010*. ACM, 2010.

[PLM08] Riccardo Poli, William B. Langdon e Nicholas F. McPhee. *A Field Guide to Genetic Programming*. Lulu, 2008. ISBN: 978-1-4092-0073-4.

[Que03] Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 2003.

[Rei90] Clifford A. Reiter. *APL with a Mathematical Accent*. Springer, 1990. ISBN: 0534128645.

[RK07] Reuven Y. Rubinstein e Dirk P. Kroese. *Simulation and the Monte Carlo Method*. 2ª ed. Wiley, 2007. ISBN: 0470177942.

[Sil10] A. Silberschatz. *Fundamentos De Sistemas Operacionais*. 8a. LTC, 2010. ISBN: 852161747x.

[SN05] Jim Smith e Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005. ISBN: 1558609105.

[SS78] Guy Lewis Steele Jr. e Gerald Jay Sussman. *The Art of the Interpreter or, the Modularity Complex*. Rel. téc. AI Memo n. 453. MIT, 1978.

[SS94] Leon Sterling e Ehud Shapiro. *The Art of Prolog*. MIT Press, 1994. ISBN: 0-262-19338-8.

[ST97] Nir Shavit e Dan Touitou. "Software transactional memory". Em: *Distributed Computing* 10.2 (1997).

[Sta07] William Stallings. *Operating Systems*. 6a. Prentice Hall, 2007. ISBN: 0136006329.

[Ste03] Leon S. Sterling. *The Practice of Prolog*. MIT Press, 2003. ISBN: 978-0262514453.

[Sti05] Douglas R. Stinson. *Cryptography: Theory and Practice*. 3ª ed. Chapman & Hall, 2005. ISBN: 1584885084.

[Tan10] Andrew S. Tanenbaum. *Sistemas Operacionais Modernos*. 3a. Prentice Hall, 2010. ISBN: 8576052377.

Versão Preliminar

[Tur37] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". Em: *Proc. London Math. Soc* 2.42 (1937), pp. 230–65.

[Tur38] Alan M. Turing. "Correction to: On Computable Numbers, with an Application to the Entscheidungsproblem". Em: *Proc. London Math. Soc* 2.43 (1938), pp. 544–6.

[VG04] Luiz Velho e Jonas Gomes. *Fundamentos da Computação Gráfica*. IMPA, 2004.

Versão Preliminar

Versão Preliminar

ÍNDICE REMISSIVO

[15 de novembro de 2018 at 14:30]

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar

Versão Preliminar