Paradigmas de Programação – Exercícios (4)

26 de junho de 2010

Ex. 1 — Escreva um procedimento com *memoização*: quando é chamado pela primeira vez com argumentos `a1, a2, ...` ele calcula seu valor de retorno. Quando chamado pela segunda vez, se lembra do valor da função para aqueles argumentos e repete o valor, sem precisar calcular. (Use hashtables para guardar os valores de resultado para cada série de argumentos).

Ex. 2 — Faça uma macro `memoize`, que recebe um nome de procedimento e o transforma em outro que faça memoização.

Ex. 3 — No kernel do Linux é frequente o uso de `goto`. Os desenvolvedores argumentam que é a maneira mais legível de traduzir certos algritmos em C[1].
Por exemplo, na função `vfs_kern_mount` em `fs/super.c` há o seguinte[2]:

```
    mnt = alloc_vfsmnt(name);
    if (!mnt)
        goto out;
    if (data && !(type->fs_flags &
                  FS_BINARY_MOUNTDATA)) {
        secdata = alloc_secdata();
        if (!secdata)
            goto out_mnt;
        error =
           security_sb_copy_data(data,
                                 secdata);
        if (error)
            goto out_free_secdata;
    error = type->get_sb(type, flags,
                         name, data,
                         mnt);
    if (error < 0)
        goto out_free_secdata;

    BUG_ON(!mnt->mnt_sb);
    WARN_ON(!mnt->mnt_sb->s_bdi);

    error =
      security_sb_kern_mount(mnt->mnt_sb,
                             flags,
                             secdata);
    if (error)
        goto out_sb;

    /* FAZ ALGO AQUI */

    return mnt;

out_sb:
    dput(mnt->mnt_root);
    deactivate_locked_super(mnt->mnt_sb);
out_free_secdata:
    free_secdata(secdata);
out_mnt:
    free_vfsmnt(mnt);
out:
    return ERR_PTR(error);
```

Neste caso os `goto`s seguem um padrão: saltam "para a frente", e os rótulos ficam na ordem inversa dos `goto`s, porque identificam locais onde se faz limpeza de código. A idéia é que quando se precisa fazer A, B, C e D antes de um trecho de código, e B falhar, deve-se desfazer apenas B e A:

```
    do A
    if (error)
        goto out_a;
    do B
    if (error)
        goto out_b;
    do C
    if (error)
        goto out_c;

    goto out;

    out_c:
    undo C
    out_b:
    undo B:
    out_a:
    undo A

    out:
    return ret;
```

Implemente a mesma coisa em Scheme, mas de forma

[1] Há uma discussão online a respeito disso em `http://kerneltrap.org/node/553/2131`, e comentários sobre o assunto também no livro Linux Device Drives, de Rubini e Corbet.

[2] A formatação do código foi alterada para tomar menos espaço.

abstrata: deve ser possível ao usuário fazer algo como

```
(with-device a
  (with-device b
    (with-device c
      (do-something a b c))))
```

Diga de quantas outras maneiras você conseguiria implementar `with-device`, e comente sobre a legibilidade de programas que o usam, comparando com outros programas Scheme.

Ex. 4 — Explique detalhadamente o significado de cada forma abaixo, e o que cada uma retorna. Diga também qual é o contexto capturado por `call/cc`.

```
(+ 1
   (call/cc (lambda (k)
            (* 2 5))))

(+ 1
   (call/cc (lambda (k)
            (* 2 (k 5)))))

;;

(+ 1
   (call/cc
    (lambda (k)
      (* 2 (call/cc
            (lambda (kk)
              (k 5)))))))

(+ 1
   (call/cc
    (lambda (k)
      (* 2 (call/cc
            (lambda (kk)
              (set! k kk)
              (k 5)))))))
```

Ex. 5 — Diga qual é o contexto capturado por `call/cc`:

a)
```
(+ (* (call/cc
        (lambda (c)
          (+ (/ a b) (c d)))) e) f)
```

b)
```
(if a b (call/cc
          (lambda (c)
            (c d))))
```

Ex. 6 — Implemente uma pilha onde `push` e `pop` não avaliam seus argumentos; `peek` sim.