Recursão e iteração

INF05008 - Fundamentos de Algoritmos

Iteração vs. Recursão

- Muitas introduções a programação baseadas em linguagens imperativas caracterizam recursão como ineficientes.
- Entretanto determinados tipos de recursão podem ser tão eficientes quanto construções de loop imperativas (while, for).

O problema

- Comecemos apresentando o problema de desempenho com o exemplo recursivo clássico:

```
;; factorial: number -> number
(define (factorial n)
  (cond [(= n 0) 1]
        [else (* n (factorial (- n 1)))]))
```

Passos de execução

```
(factorial 5)
(* 5 (factorial 4))
(* 5 (* 4 (factorial 3)))
(* 5 (* 4 (* 3 (factorial 2))))
(* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

Variante imperativa

- Em relação a codificação em Pascal:

```
function fatorial(n:integer):integer;
var i:integer := 0;
    fat:integer := 1;
begin
  for i := 1 to n do
    fat := fat * 1;
  return(fat);
end
```

- A recursiva apresenta dois custos extras

Custos de recursão

- **Memória**. É o custo mais grave.
  - A área vermelha precisa ser armazenada em memória (normalmente em uma estrutura de dados chamada pilha)
  - No pior momento usa N células de memória, enquanto o caso imperativo o uso de memória é constante (apenas duas células, uma para cada variável)
- **Processamento**. Passos a mais para empilhar e desempilhar dados.

Recursão eficiente

- Entretanto, dependendo do formato da chamada de recursão, um bom compilador pode executar de forma tão eficiente como a versão imperativa
- No caso de Scheme, qualquer compilador que atenda a especificação do padrão da linguagem têm essa otimização

Fatorial recursivo eficiente

```
;; factorial: number -> number
(define (factorial n)
  (local
    ((define (fac n acum)
      (cond [(= n 0) acum]
            [else (fac (- n 1) (* n acum))])))
    (fac n 1)))
```

Passos de execução

- `(factorial 5)`
- `(fac 5 1)`
- `(fac (- 5 1) (* 5 1))`
- `(fac 4 5)`
- `(fac (- 4 1) (* 4 5))`
- `(fac 3 20)`
- `(fac (- 3 1) (* 3 20))`
- `(fac 2 60)`
- `(fac (- 2 1) (* 2 60))`
- `(fac 1 120)`
- `(fac (- 1 1) (* 1 120))`
- `(fac 0 120)`
- `120`

Recursão de cauda

- A última versão de fatorial é tão eficiente quanto a imperativa.
- Esse tipo de recursão é chamada de recursão de cauda (*tail call*) (alguns livros chamam esta de “iteração”).
- A razão para o nome “de cauda” é o formato onde a chamada a recursão é a última instrução chamada.
- O valor retornado não é usado para mais nenhuma operação.

Formato:

- Recursão de cauda

```
(define (fname x)
 ...
  (fname x'))
```

- Recursão comum (não de cauda)

```
(define (fname x)
  ...
   (op (fname x')))
```

Recursão mútua

```
(define (fn1 x)
   ..
     (fn2 x'))
```

```
(define (fn2 y)
   ..
     (fn1 y'))
```

- Recursão em cauda também funciona com chamadas mutualmente recursivas (desde que o formato em cauda seja respeitado)

Recursão em cauda é sempre melhor?

Resposta: Não necessariamente.

Uma função em cauda

```
;; remove-all: number list-of-number -> list-of-number
(define (remove-all e lis)
  (local
    ((define (rem lis acum)
       (cond[(empty? lis) acum]
            [(= e (first lis)) (rem (rest lis) acum)]
            [else (rem (rest lis)
                       (cons (first lis) acum))])))
    (rem lis empty)))
```

Execução

```
(remove-all 3 (list 1 2 3 4))
(remove-all 3 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
(rem (cons 1 (cons 2 (cons 3 (cons 4 empty)))) empty)
(rem (cons 2 (cons 3 (cons 4 empty))) (cons 1 empty))
(rem (cons 3 (cons 4 empty)) (cons 2 (cons 1 empty)))
(rem (cons 4 empty) (cons 2 (cons 1 empty)))
(rem empty (cons 4 (cons 2 (cons 1 empty))))
(cons 4 (cons 2 (cons 1 empty)))
(list 4 2 1)
```

Uma função tradicional

```
;; remove-all: number list-of-number -> list-of-number
(define (remove-all e lis)
  (local
    ((define (rem lis)
       (cond[(empty? lis) empty]
            [(= e (first lis)) (rem (rest lis))]
            [else (cons (first lis) (rem (rest lis)))])))
    (rem lis)))
```

Execução

```
(remove-all 3 (list 1 2 3 4))
(rem (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
(cons 1 (rem (cons 2 (cons 3 (cons 4 empty)))))
(cons 1 (cons 2 (rem (cons 3 (cons 4 empty)))))
(cons 1 (cons 2 (rem (cons 4 empty))))
(cons 1 (cons 2 (cons 4 (rem empty))))
(cons 1 (cons 2 (cons 4 (empty))))
(cons 1 (cons 2 (list 4)))
(cons 1 (list 2 4))
(list 1 2 4)
```

Comparação

- O consumo de memória das duas versões é praticamente igual (constante em relação ao tamanho da lista).
- A versão com recursão em cauda pode ser um pouco melhor em termos de processamento, mas se a ordem for importante a necessidade de inverter a lista final certamente elimina qualquer vantagem.