Composição de Funções

Fundamentos de Algoritmos

INF05008

Funções Compostas

- Um programa é composto por definições de **funções** e **variáveis**
- **Funções auxiliares** são definidas para processar **dependências**
- A partir de agora, funções auxiliares serão propostas com um **objetivo mais amplo**

Projeto de Programas Complexos

- Somente programas **muito simples** são compostos por **apenas uma função**
- Tratar as dependências usando funções auxiliares **facilita o entendimento do programa**
- Antes de iniciar um programa, os **dados de entrada e saída** devem ser **analisados**

Projeto de Programas Complexos (cont.)

- Possíveis situações:
  - Se a formulação de uma resposta **depende de uma avaliação de números**, usa-se `cond`
  - Se a função requer o processamento de dados de um **domínio particular**, usam-se **funções auxiliares**
  - Se for necessário processar um **número natural**, uma **lista** ou outra forma de **dados de tamanho arbitrário**, usam-se **funções auxiliares**
  - Se a **função principal apenas enumera os processamentos** que devem ser feitos, esta é composta por **chamadas a funções auxiliares**

- Daremos ênfase aos dois últimos itens

Projeto de Programas Complexos (cont.)

- Ao determinar quais funções auxiliares são necessárias, devemos criar:
  - Um **contrato**, um **objetivo** e um **cabeçalho** para cada uma delas;
  - Um **corpo** para cada uma delas, seguindo o **determinado pelo objetivo** particular.

- Se uma das funções auxiliares **já existe**, **não redefiní-la!**

- Cada **função auxiliar** deve ser **desenvolvida e testada individualmente**

- Finalmente, o **programa principal** (o que chama as funções auxiliares) deve também ser testado

Funções Auxiliares Recursivas

- *Ordenação* é uma função que comumente aparece em programas: dada uma lista de números, devolver a lista com os números segundo algum **critério de ordenação** (ordem ascendente, por exemplo)

- Pode-se usar uma função auxiliar para fazer a ordenação de dados

Ordenação de Números

```
;; ordena: lista-de-números -> lista-de-números
;; Gera uma lista ordenada (ordem descendente) com os
;; números fornecidos na lista de entrada
(define (ordena ldn)...)

;; Exemplos:
;; (ordena empty) produz empty
;; (ordena (cons 12 (cons 20 (cons -5 empty))))
;;         produz (cons 20 (cons 12 (cons -5 empty)))
```

Ordenação de Números (cont.)

```
(define (ordena ldn)
   (cond
      [(empty? ldn) ...]
      [else...(first ldn)...(ordena (rest ldn))...]))
```

1. (first ldn) extrai o primeiro número da lista
2. (ordena (rest ldn)) produz uma lista ordenada dos números da lista (rest ldn)

Ordenação de Números (cont.)

- `(first ldn)` produz 12
- `(rest ldn)` é `(cons 20 (cons -5 empty))`
- `(ordena (rest ldn))` produz `(cons 20 (cons -5 empty))`
- Deve-se inserir 12 entre 20 e -5
- Usa-se uma expressão `cond` que insere `(first ldn)` na posição correta de `(ordena (rest ldn))`
- Uma **função auxiliar** pode ser criada com esse intuito

Ordenação de Números (cont.)

```
;; insere: número lista-de-números -> lista-de-números
;; Dados um número n e uma lista ordenada, insere n
;; na posição correta da lista, de forma que a
;; lista final esteja ordenada
    (define (insere n ldn)...)

;;(insere -5 empty) produz (cons -5 empty)
;;(insere 12 (cons 20 (cons -5 empty)))
;;   produz (cons 20 (cons 12 (cons -5 empty)))
```

Ordenação de Números (cont.)

Usando `insere`, pode-se completar a definição de `ordena`

```
;; ordena: lista-de-números -> lista-de-números
;; Gera uma lista ordenada (ordem descendente) com os
;; números fornecidos na lista de entrada

(define (ordena ldn)
  (cond
    [(empty? ldn) empty]
    [else (insere (first ldn) (ordena (rest ldn)))]))
```

Ordenação de Números (cont.)

```
(define (insere n ldn)
    (cond
        [(empty? ldn)...]
        [else...(first ldn)...(insere n (rest ldn))...]))
```

Ordenação de Números (cont.)

- Para `(insere 7 (cons 6 (cons 5 (cons 4 empty))))`, o novo elemento é apenas inserido na **frente da lista**

- Para `(insere 3 (cons 6 (cons 2 (cons 1 (cons -1 empty)))))`, o 3 deve ser inserido **entre números da lista** (entre o 6 e o 2). Neste caso,
  - `(first ldn)` é `6`
  - `(insere n (rest ldn))` deve resultar em `(cons 6 (cons 3 (cons 2 (cons 1 (cons -1 empty)))))`

Ordenação de Números (cont.)

```
(cond
   [(>= n (first ldn)) ...]
   [(<  n (first ldn)) ...])
```

Ordenação de Números (cont.)

```
;; ordena : lista-de-números -> lista-de-números
;; Gera uma lista ordenada (ordem descendente) com os números
;; fornecidos na lista de entrada
(define (ordena ldn)
  (cond
    [(empty? ldn) empty]
    [(cons? ldn)(insere (first ldn) (ordena (rest ldn)))]))

;; insere : número lista-de-números -> lista-de-números
;; Dados um número n e uma lista ordenada, insere o número n
;; na posição correta da lista, de forma que a lista final esteja ordenada
(define (insere n ldn)
  (cond
    [(empty? ldn) (cons n empty)]
    [else
      (cond
        [(>= n (first ldn)) (cons n ldn)]
        [(< n (first ldn))(cons (first ldn)(insere n (rest ldn)))])]))
```

Generalização de Problemas e Funções

- Considere o problema de desenhar um polígono
- Um polígono é uma figura geométrica formada por um número arbitrário de ângulos
- Pode ser representado por uma lista de estruturas `posn`

Desenhando Polígonos

- Uma lista-de-posn é
  - vazia ou
  - `(cons p lp)`, onde `p` é uma estrutura `posn` e `lp` é uma lista de `posn`.

Desenhando Polígonos

- Cada `posn` representa um ângulo do polígono

```
(cons (make-posn 10 10)
  (cons (make-posn 60 60)
    (cons (make-posn 10 60)
      empty)))
```

Desenhando Polígonos (cont.)

Um polígono é

- `(cons p empty)`, onde `p` é uma estrutura `posn`
- `(cons p lp)`, onde `p` é uma estrutura `posn` e `lp` é um polígono

Desenhando Polígonos (cont.)

```
;; desenha-poli: polígono -> boolean
;; Desenha um polígono especificado por poli

(define (desenha-poli poli)
   (cond
     [(empty? (rest poli))...(first poli)...]
     [else ...(first poli)...
           ...(second poli)...
           (desenha-poli (rest poli))...]))
```

Desenhando Polígonos (cont.)

```
;; desenha-poli: polígono -> boolean
;; Desenha um polígono especificado por poli

(define (desenha-poli poli)
 (cond
   [(empty? (rest poli)) true]
   [else (and (draw-solid-line (first poli)(second poli))
              (desenha-poli (rest poli)))]))
```

Desenhando Polígonos (cont.)

- Para adicionar a última aresta pode-se

  1. Adicionar a primeira estrutura `posn` também no final da lista
  2. Adicionar a última `posn` também no ínicio da lista
  3. Modificar esta função de tal forma que uma linha também é desenhada conectando a primeira e a última estruturas `posn`

- Vamos desenvolver a **segunda opção**

Desenhando Polígonos (cont.)

```
;; último: polígono -> posn
;; Extrai o último posn de um polígono

(define (último poli)...)
```

Desenhando Polígonos (cont.)

```
;; último: polígono -> posn
;; Extrai o último posn de um polígono

(define (último poli)
   (cond
      [(empty? (rest poli))...(first poli)...]
      [else ... (first poli)...
            ... (second poli)...
            (último (rest poli))...]))
```

Desenhando Polígonos (cont.)

```
;; último: polígono -> posn
;; Extrai o último posn de um polígono

(define (último poli)
   (cond
      [(empty? (rest poli))(first poli)]
      [else (último (rest poli))]))
```

Desenhando Polígonos (cont.)

```
;; desenha-poli: polígono -> boolean
;; Desenha um polígono especificado por poli

(define (desenha-poli poli)
 (conecta-pontos (cons (último poli) poli)))

;; conecta-pontos: polígono -> boolean
;; Desenha as arestas entre ângulos de um polígono 'poli'

(define (conecta-pontos poli)
 (cond
  [(empty? (rest poli)) true]
  [else (and (draw-solid-line (first poli)(second poli) RED)
             (conecta-pontos (rest poli)))]))

;; último: polígono -> posn
;; Extrai o ultimo posn de um polígono

(define (último poli)
 (cond
  [(empty? (rest poli))(first poli)]
  [else (último (rest poli))]))
```