Processando Múltiplos Dados Compostos

Fundamentos de Algoritmos

INF05008

Sintaxe Especial para Listas

- Usar `cons` para criar listas pode ser trabalhoso e complicado
- Scheme provê a operação **`list`** para facilitar a criação e uso de listas
- `list` consome um **número arbitrário de valores** e, automaticamente, os **organiza em um lista**

```
(list a1 a2 ... an) -> (cons a1(cons a2...(cons an empty)...))

(list )             -> empty
(list 'a)           -> (cons 'a empty)
(list 'a 'b)        -> (cons 'a (cons 'b empty))
```

Sintaxe Especial para Listas (cont.)

- Operação `list` pode também ser usada sobre **expressões**

```
(list (+ 2 1) (- 6 3) (/ 4 2))
=
(list 3 3 2)
```

- A construção da lista ocorre **após a avaliação de cada expressão**, a fim de verificar se não há erros

```
(list (+ 3 7) (/ 1 0))
=
/: division by zero
```

Sintaxe Especial para Listas (cont.)

- As operações `first` e `rest` **continuam válidas**

```
(first (list 1 2 3))
=
1

(rest (list 1 2 3))
=
(list 2 3)
```

Processamento de Listas

- Vimos funções que usam **recursão** para percorrer um **único dado composto**.
- Hoje, veremos recursão sobre **mais de um dado composto de entrada** (listas)
- Dadas **duas listas**, uma função pode processá-las de duas formas diferentes:
  1. **Processar apenas uma lista** e tratar a **outra como um dado atômico**
  2. Processar as **duas listas simultaneamente**
- Veremos três casos e como processar as listas em cada caso

Caso 1: Concatenação de Listas

- Dadas duas listas l1 e l2, de **mesma classe de dados**, retornar uma **lista única** com os elementos de l1 e l2

Caso 1: Concatenação de Listas

- Dadas duas listas l1 e l2, de **mesma classe de dados**, retornar uma **lista única** com os elementos de l1 e l2

- **Solução:** eliminar o `empty` de l1 e colocá-la como prefixo de l2

```
(concatena (list 12 4 7) (list 0 9))
  ==>
(list 12 4 7 0 9)
```

- Função **processa l1** , a fim de encontrar o `empty`, e **trata l2 como dado atômico**

Caso 1: Concatenação de Listas (cont.)

```
;; concatena : list list -> list
;; Construir uma nova lista trocando o 'empty' de l1 por l2
(define (concatena l1 l2)

;; Exemplos:
;; (concatena empty l2) produz l2
;; (concatena l1 empty) produz l1
;; (concatena (list 1 2 3) (list 4 5)) produz (list 1 2 3 4 5)
```

Note que `l1` e `l2` nos exemplos representam listas de uma classe de dados qualquer, contendo um número arbitrário de elementos

Caso 1: Concatenação de Listas (cont.)

```
;; concatena : list list -> list
;; Construir uma nova lista trocando o 'empty' de l1 por l2

(define (concatena l1 l2)
  (cond
    [(empty? l1) l2]
    [else (cons (first l1) (concatena (rest l1) l2))]))
```

Caso 2: Operações entre Elementos de Listas

- Já vimos a função `salário : number -> number`, que retornava o salário a ser pago, dada uma quantidade de horas trabalhadas. O valor por hora trabalhada era de $12.

- Consideremos agora um **conjunto de funcionários** (lista) para os quais devemos calcular o respectivo salário.

- Cada funcionário pode ter sido contratado com um **rendimento (valor por hora trabalhada) diferente**

- Como seria uma função que, dadas uma **lista de salários por hora trabalhada** e uma **lista de horas trabalhadas**, retornasse uma **lista de salários**?

Caso 2: Operações entre Elementos de Listas (cont.)

```
;; salário : number number  ->  number
;; Calcular o salário a partir do valor por hora (vph) e
;; o número de horas trabalhadas (h)

(define (salário vph h)
  (* vph h))
```

Caso 2: Operações entre Elementos de Listas (cont.)

- Novamente, listas contêm a **mesma classe de dados**
- No entanto, lista resultante requer que operemos **simultaneamente** sobre as listas de entrada
- Para este exemplo, deve-se assumir que as listas têm **mesmo comprimento**

Caso 2: Operações entre Elementos de Listas (cont.)

```
;; horas->salário : lista-de-números lista-de-números ->
;; lista-de-números
;; Construir uma nova lista através da multiplicação
;; de elementos correspondentes de l1 e l2
;; ASSUME-SE: as duas listas têm mesmo comprimento

(define (horas->salário l1 l2) ...)
```

Caso 2: Operações entre Elementos de Listas (cont.)

- Podemos definir `l1` como sendo a lista de salários por hora trabalhada e `l2`, a lista de horas trabalhadas

- Dessa forma, podemos ter os seguintes exemplos de uso:

```
;; Exemplos:
;; (horas->salário empty empty) produz empty
;; (horas->salário (list 5.65) (list 40)) produz (list 226)
;; (horas->salário (list 5.65 8.75) (list 40 30))
;; produz (list 226 262.5)
```

Caso 2: Operações entre Elementos de Listas (cont.)

```
(define (horas->salário l1 l2)
  (cond
    [(empty? l1) ... ]
    [else ... ]))
```

Caso 2: Operações entre Elementos de Listas (cont.)

```
(define (horas->salário l1 l2)
  (cond
    [(empty? l1) empty]
    [else ... ]))
```

Lembre-se que, assumindo-se que ambas as listas têm mesmo comprimento, **`(empty? l1)` só será verdadeira se `(empty? l2)` também for verdadeira**

Caso 2: Operações entre Elementos de Listas (cont.)

```
;; horas->salário : lista-de-números lista-de-números ->
;; lista-de-números
;; Construir uma nova lista através da multiplicação
;; de elementos correspondentes de l1 e l2
;; ASSUME-SE: as duas listas tem mesmo comprimento

(define (horas->salário l1 l2)
  (cond
    [(empty? l1) empty]
    [else (cons (salário (first l1) (first l2))
                (horas->salário (rest l1) (rest l2)))]))
```

Caso 3: Acesso a Elemento Específico de uma Lista

- **Arranjos** são estruturas de dados **contíguos**, cujos elementos são acessados por meio de um **índice**
- Apesar de **listas** não serem contíguas, **também podem ser acessadas através de índices**
- Deve-se criar uma função que, dada uma lista `l` e um número `n`, retorna o `n`-ésimo elemento de `l`

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Exemplos:

```
(recupera (list 'a 'b 'c) 1) ==> 'a

(recupera (list 'a 'b 'c) 3) ==> 'c

(recupera empty            3) ==> ???

(recupera (list 'a)       14) ==> ???
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

```
;; recupera : lista-de-símbolos n[>= 1]  ->  symbol
;; Recupera o n-ésimo elemento de uma lista de símbolos l,
;; contando a partir da posição 1; sinaliza um erro caso
;; não haja elemento algum na posição requisitada

(define (recupera l n) ...)
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Como definimos as clásulas da função `recupera`?

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Como definimos as clásulas da função `recupera`?
- Devemos considerar **todos os possíveis casos**

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Como definimos as clásulas da função `recupera`?
- Devemos considerar **todos os possíveis casos**

```
(recupera empty 1) ==> error      [(= n 1) and (empty? l)]

(recupera (list 'a) 1) ==> 'a     [(= n 1) and (cons? l)]

(recupera empty 3) ==> error      [(> n 1) and (empty? l)]

(recupera (list 'a) 3) ==> error  [(> n 1) and (cons? l)]
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

```
(define (recupera l n)
  (cond
    [(and (= n 1) (empty? l)) ...]
    [(and (> n 1) (empty? l)) ...]
    [(and (= n 1) (cons? l)) ...]
    [(and (> n 1) (cons? l)) ...]))
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Lembrando que, em situações em que a **lista é vazia, não é possível acessar-se um elemento**, qualquer que seja a posição requisitada:

```
(define (recupera l n)
  (cond
    [(and (= n 1) (empty?  l)) (error 'recupera "posição não existe")]
    [(and (> n 1) (empty?  l)) (error 'recupera "posição não existe")]
    [(and (= n 1) (cons? l)) ...]
    [(and (> n 1) (cons? l)) ...]))
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Se a **lista não é vazia**, acessar a posição 1 significa obter o **primeiro elemento da lista**:

```
(define (recupera l n)
  (cond
    [(and (= n 1) (empty? l)) (error 'recupera "posição não existe")]
    [(and (> n 1) (empty? l)) (error 'recupera "posição não existe")]

    [(and (= n 1) (cons?  l)) (first l)]

    [(and (> n 1) (cons? l)) ...]))}}
```

Caso 3: Acesso a Elemento Específico de uma Lista (cont.)

- Para a quarta cláusula, temos de usar **recursão** para percorrer a lista, i.e., `(recupera ...)`
- Usaremos o valor de `n` para **contar** o número de posições
  - Como a contagem **parte da posição que queremos acessar**, devemos usar um **contador regressivo**
  - A cada posição acessada, reduzimos em 1 o **número de avanços que ainda restam**
  - Usaremos a operação **`sub1`** , a qual consome um número e retorna este **número subtraído de 1**, i.e., `(sub1 n) = n - 1`
- Outro valor será a lista de elementos **nas posições ainda não acessadas**, i.e., `(rest l)`

```
(define (recupera l n)
  (cond
    [(and (= n 1) (empty? l)) (error 'recupera "posição não existe")]
    [(and (> n 1) (empty? l)) (error 'recupera "posição não existe")]
    [(and (= n 1) (cons? l)) (first l)]
    [(and (> n 1) (cons?  l)) (recupera (rest l) (sub1 n))]))
```