

Listas

Fundamentos de Algoritmos

INF05008

Listas

- Estruturas são apenas uma forma de representar informação **composta**
- Elas são úteis quando sabemos exatamente **como e por quantas partes** um determinado dado é composto
- Quando não sabemos ao certo quantos elementos pertencem a um determinado dado, podemos formar uma **lista**
- Uma lista pode ter um tamanho **arbitrário** (ou seja, um número **finito**, mas **indeterminado**, de itens).

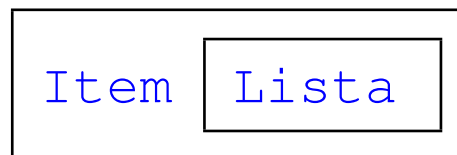
cons **truindo listas...**

Para construir uma lista, podemos partir de uma **lista vazia** e incluir itens nesta lista. Em Scheme, usamos as seguintes funções:

- `empty` : Representa a lista vazia;



- `(cons Item Lista)` : Função que inclui o item `Item` na lista `Lista`

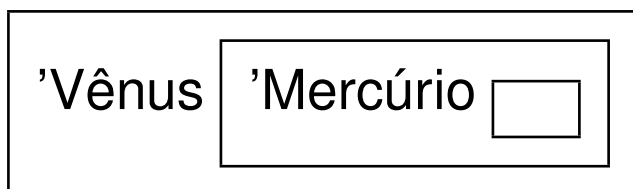


Exemplos...

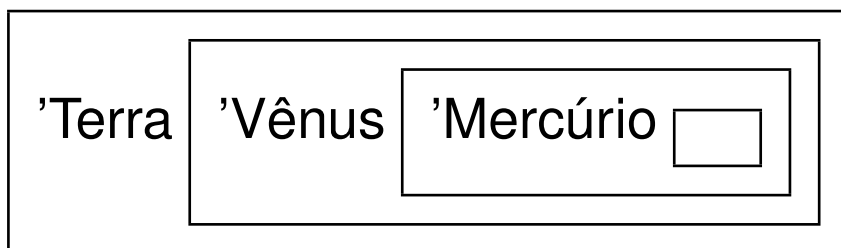
```
(cons 'Mercúrio empty)
```



```
(cons 'Vênus (cons 'Mercúrio empty))
```



```
(cons 'Terra (cons 'Vênus (cons 'Mercúrio empty)))
```



Mais Exemplos...

- Lista com 10 números:

```
(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9 empty))))))))))
```

Mais Exemplos...

- Lista com elementos de **tipos diferentes**:

```
(cons 'RobbyRound  
  (cons 3  
    (cons true  
      empty)))
```

Função que Soma os 3 Valores de Uma Lista

Uma lista-de-3-números é

```
(cons x (cons y (cons z empty)))
```

onde 'x', 'y' e 'z' são números

```
:: soma-3 : lista-de-3-números -> número
```

```
:: Somar os 3 números de uma lista de 3 números
```

```
:: Exemplos e Testes:
```

```
:: (= (soma-3 (cons 2 (cons 1 (cons 3 empty)))) 6)
```

```
:: (= (soma-3 (cons 0 (cons 1 (cons 0 empty)))) 1)
```

```
(define (soma-3 uma-lista-de-3-números) ...)
```

Função que Soma os 3 Valores de Uma Lista

Uma lista-de-3-números é

```
(cons x (cons y (cons z empty)))
```

onde 'x', 'y' e 'z' são números

```
:: soma-3 : lista-de-3-números -> número
```

```
:: Somar os 3 números de uma lista de 3 números
```

```
:: Exemplos e Testes:
```

```
:: (= (soma-3 (cons 2 (cons 1 (cons 3 empty)))) 6)
```

```
:: (= (soma-3 (cons 0 (cons 1 (cons 0 empty)))) 1)
```

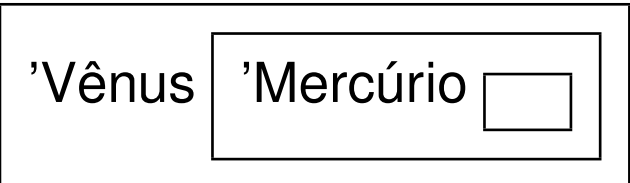
```
(define (soma-3 uma-lista-de-3-números) ...)
```

Mas, como acessar os elementos de uma lista?


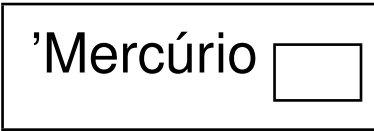
Operações Sobre Listas

Em Scheme, podemos usar as seguintes funções para acessar os elementos de uma lista:

- `first` : Devolve o **primeiro elemento** de uma lista:

`(first`  `) = 'Vênus`

- `rest` : Ignora o primeiro elemento e devolve o **resto da lista** :

`(rest`  `) =` 

Exemplos de Operações Sobre Listas

```
(first (cons 10 empty))
```

```
(rest (cons 10 empty))
```

```
(first (rest (cons 10 (cons 22 empty))))
```

Exemplos de Operações Sobre Listas

```
(first (cons 10 empty))  
= 10
```

```
(rest (cons 10 empty))
```

```
(first (rest (cons 10 (cons 22 empty))))
```

Exemplos de Operações Sobre Listas

```
(first (cons 10 empty))  
= 10
```

```
(rest (cons 10 empty))  
= empty
```

```
(first (rest (cons 10 (cons 22 empty))))
```

Exemplos de Operações Sobre Listas

```
(first (cons 10 empty))  
= 10
```

```
(rest (cons 10 empty))  
= empty
```

```
(first (rest (cons 10 (cons 22 empty))))  
= (first (cons 22 empty))  
= 22
```

Template da Função `add-up-3`

```
;; soma-3 : lista-de-3-números -> número
;; Soma os 3 números em uma lista de 3 números

(define (soma-3 uma-lista-de-3-números)
  ... (first uma-lista-de-3-números) ...
  ... (first (rest uma-lista-de-3-números)) ...
  ... (first (rest (rest uma-lista-de-3-números))) ... )
```

Exercício: Complete a definição desta função.

Definição de Dados para Listas de Tamanho Indeterminado

- Até agora, todas as definições de dados que fizemos foram de tamanho **fixo**
- Em muitas situações, não sabemos quantos elementos uma lista terá
- Nestes casos, precisamos de uma definição **genérica**, ou seja, definir a **classe de todas as listas finitas**

Definição Genérica de Lista

Uma lista-de-símbolos é ou

1. a lista vazia `empty` , ou
2. `(cons s lds)` , onde
 - 's' é um símbolo e
 - 'lds' é uma lista-de-símbolos.

Definição Genérica de Lista

Uma `lista-de-símbolos` é ou

1. a lista vazia `empty` , ou
2. `(cons s lds)` , onde
 - '`s`' é um símbolo e
 - '`lds`' é uma `lista-de-símbolos` .

*Esta é uma definição **recursiva** !!!*

Processando Listas de Tamanho Indeterminado

Suponha que queiramos construir um programa para verificar se uma lista de itens de uma loja de brinquedos contém 'boneca'.

Como já temos a definição do tipo de dados necessário ao problema (`lista-de-símbolos`), passamos à fase de descrever o contrato, o objetivo e o cabeçalho:

```
;; contém-boneca? : lista-de-símbolos -> boolean
;; Determinar se a palavra 'boneca ocorre em uma lista de
;; símbolos
```

```
(define (contém-boneca? uma-lista-de-símbolos) ...)
```

Exemplos para a função contém-boneca?

```
(boolean=? (contém-boneca? empty)
            false)
```

```
(boolean=? (contém-boneca? (cons 'bola empty))
            false)
```

```
(boolean=? (contém-boneca? (cons 'boneca empty))
            true)
```

```
(boolean=? (contém-boneca? (cons 'pote (cons 'jogo (cons 'bola empty))))
            false)
```

```
(boolean=? (contém-boneca? (cons 'flecha
                                (cons 'boneca (cons 'bola empty))))
            true)
```

Template para a função contém-boneca?

Como a definição dos dados de entrada da função (`lista-de-símbolos`) tem 2 cláusulas, o corpo da função deve ser uma expressão `cond`:

```
(define (contém-boneca? uma-lista-de-símbolos)
  (cond
    [(empty? uma-lista-de-símbolos) ...]
    [(cons? uma-lista-de-símbolos) ...]))
```

Note que, em vez de `(cons uma-lista-de-símbolos)`, podemos usar `else` na segunda cláusula.

Template para a função contém-boneca? ...

Como uma lista construída com `cons` é composta de um símbolo e uma lista de símbolos, na segunda cláusula provavelmente usaremos as funções `first` e `rest` para definir o algoritmo:

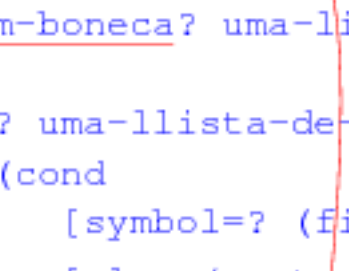
```
(define (contém-boneca? uma-lista-de-símbolos)
  (cond
    [(empty? uma-lista-de-símbolos) ...]
    [else ... (first uma-lista-de-símbolos)
               ... (rest uma-lista-de-símbolos) ...])))
```

Definindo a função contém-boneca? ...

```
(define (contém-boneca? uma-lista-de-símbolos)
  (cond
    [(empty? uma-lista-de-símbolos) false]
    [else (cond
              [(symbol=? (first uma-lista-de-símbolos) 'boneca)
               true]
              [else
               ... (rest uma-lista-de-símbolos) ...]]))])
```

Se a lista passada como argumento não for vazia e o primeiro elemento não for a palavra 'boneca, o quê devemos fazer?

Definindo a função contém-boneca? ...



```
(define contém-boneca? uma-lista-de-simbolos)
  (cond
    [(empty? uma-lista-de-simbolos) false]
    [else (cond
      [symbol=? (first uma-lista-de-simbolos) 'boneca) true]
      [else contém-boneca? (rest uma-lista-de-simbolos))])])
```

*Essa é uma função **recursiva** !!!*

Exercícios

Teste a definição da função `contém-boneca?` sobre os seguintes argumentos:

`empty`

`(cons 'bola empty)`

`(cons 'flecha (cons 'boneca empty))`

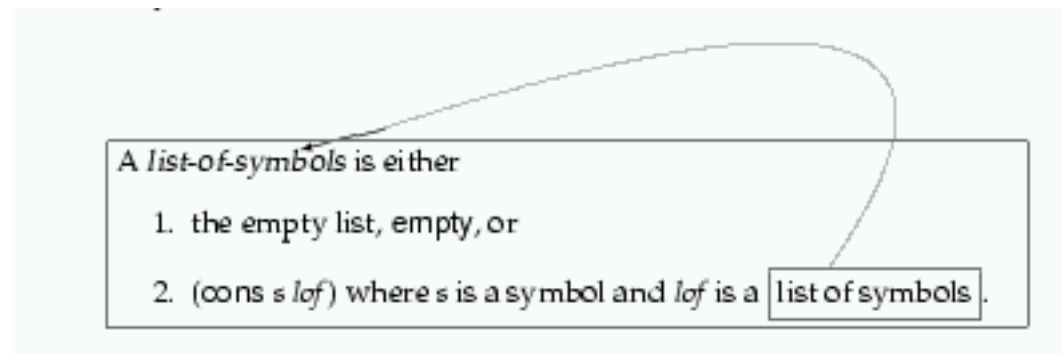
`(cons 'pote (cons 'flecha (cons 'bola empty)))`

Projeto de Funções Recursivas

Análise e Projeto de Dados: Se o problema envolver dados de **tamanho arbitrário**, precisamos usar **definições recursivas**. Para fazer definições recursivas válidas, duas **condições** devem ser satisfeitas:

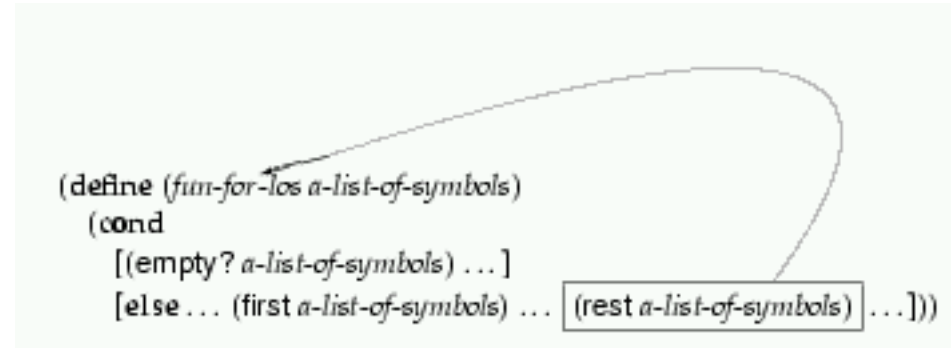
- 1. A definição deve ter no mínimo duas cláusulas;*
- 2. Pelo menos uma das cláusulas não deve referenciar o dado a ser definido.*

Para identificarmos melhor quais cláusulas são recursivas, é uma boa prática apontá-las, por exemplo, através de setas:



Projeto de Funções Recursivas...

Template:



```
(define (fun-for-los a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [else ... (first a-list-of-symbols) ...
      ... (fun-for-los (rest a-list-of-symbols)) ...]))
```

Chamamos as auto-aplicações de **RECURSÃO NATURAL**.

Projeto de Funções Recursivas...

Corpo:

Para montar o corpo da função, iniciamos definindo o quê fazer nos casos que não envolvem recursão natural. Esses são os chamados **CASOS-BASE**, e, normalmente, a solução do problema para esses casos é bastante simples.

Então, passamos aos casos que envolvem **RECURSÃO**, lembrando que, para as chamadas recursivas da função, **assumimos que a função já funciona como esperado** para os argumentos com os quais fazemos a chamada recursiva.

Exemplo de Definição do Corpo

Queremos definir a função `quantos?`, que determina quantos símbolos existem em uma lista de símbolos. Seguindo os passos descritos, temos os seguintes template e corpo:

```
;; quantos? : lista-de-símbolos -> number
;; obj.: Determinar qtos símbolos estão em uma lista-de-símbolos
(define (quantos? uma-lista-de-símbolos)
  (cond
    [(empty? uma-lista-de-símbolos) ...]
    [else ... (first uma-lista-de-símbolos) ...
              ... (quantos? (rest uma-lista-de-símbolos)) ...]))
```

```
(define (quantos? uma-lista-de-símbolos)
  (cond
    [(empty? uma-lista-de-símbolos) 0]
    [else (+ (quantos? (rest uma-lista-de-símbolos)) 1)]))
```

Fases do Projeto de Algoritmos usando Recursão

Fase	Objetivo	Atividade
<i>Projeto e Análise de Dados</i>	Formular uma definição de dados	Construir uma definição de dados com no mínimo 2 cláusulas, onde pelo menos uma delas não deve ser recursiva . Identificar explicitamente o ponto onde ocorre recursão na definição.
<i>Contrato, Objetivo e Cabeçalho</i>	Dar um nome à função, especificar as classes de entrada e saída, descrever o objetivo e formular um cabeçalho	Nomear a função, as classes de entrada e saída e especificar um objetivo: <pre>;; name : in1 in2 ...--> out} ;; to compute ... from x1 ...} (define (name x1 x2 ...) ...)</pre>

Fases do Projeto de Algoritmos usando Recursão...

Fase	Objetivo	Atividade
<i>Exemplos</i>	Caracterizar a relação entrada-saída através de exemplos	Criar exemplos da relação entrada-saída, levando em consideração que deve existir pelo menos um exemplo para cada sub-classe de dados à qual a função pode ser aplicada
<i>Template</i>	Formular um esboço para a função	Colocar uma estrutura <code>cond</code> com uma linha para cada cláusula da definição dos dados do problema. Usar os seletores apropriados para cada cláusula. Indicar onde ocorre recursão natural . TESTE: as auto-referências neste template e na definição de dados devem casar.

Fases do Projeto de Algoritmos usando Recursão...

Fase	Objetivo	Atividade
<i>Corpo</i>	Completar a definição da função	Construir expressões Scheme para cada uma das cláusulas do <code>cond</code>
<i>Testes</i>	Encontrar erros	Aplicar a função aos exemplos e verificar se os resultados são os esperados.

Exercícios

Usar o método de desenvolvimento de algoritmos para construir os programas a seguir:

1. Construa uma função que, dada uma lista de valores de objetos de uma loja, calcula o valor total do estoque da loja.
2. Construa a função `um-real?` que, dada uma lista de preços, checa se todos os preços estão abaixo de R\$1. Depois, generalize essa função, para obter como entrada a lista de preços e um valor, e verificar se todos os preços da lista estão abaixo deste valor.
3. Construa a função `converte` , que, dada uma lista de dígitos, produz o número decimal correspondente. O primeiro dígito da lista deve ser o menos significativo.