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.

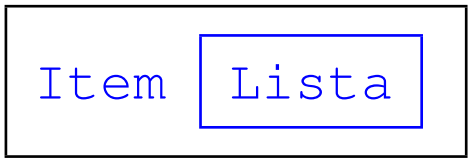
- Quando não sabemos ao certo quantos elementos pertencem a um determinado dado, podemos formar uma **lista** , que pode ter um tamanho arbitrário (ou seja, um número finito, mas indeterminado de ítens).

`cons` truindo listas...

Para construir uma lista, podemos partir de uma lista vazia e incluir ítens nessa lista. Em Scheme, usamos as seguintes funções:

- `empty` : representa a lista vazia;

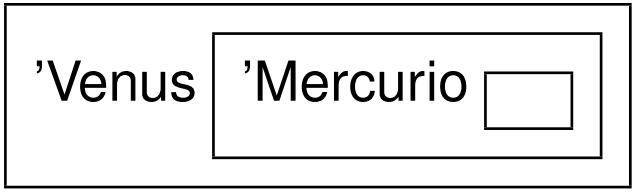
- `(cons Item Lista)` : é a função que inclui o item `Item` na lista `Lista`



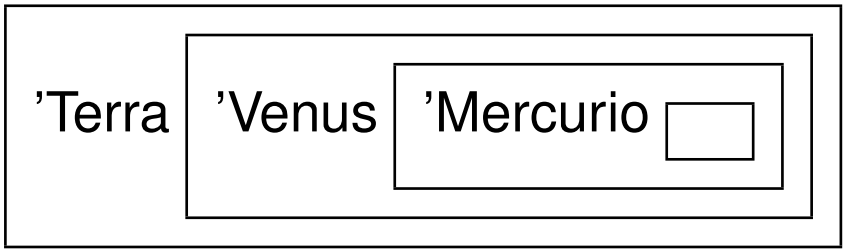
Exemplos...

```
cons 'Mercurio empty)
```

```
(cons 'Venus (cons 'Mercurio empty))
```




```
(cons 'Terra (cons 'Venus (cons 'Mercurio 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)))))))))
```

- lista com elementos de tipos diferentes:

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

Outro exemplo: 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.

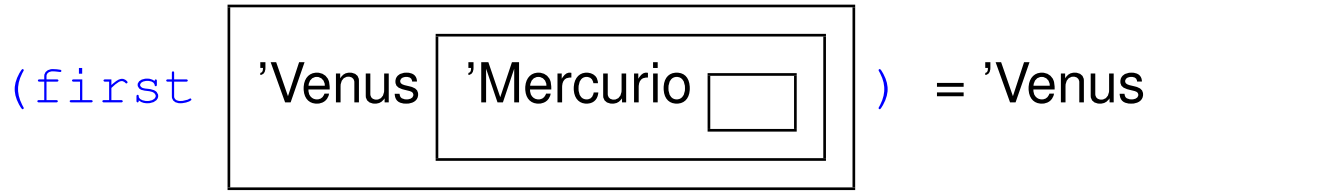
```
;; add-up-3 : lista-de-3-números  ->  number
;; obj.: somar os 3 números de uma lista de 3 números
;; exemplos e testes:
;;    (= (add-up-3 (cons 2 (cons 1 (cons 3 empty)))) 6)
;;    (= (add-up-3 (cons 0 (cons 1 (cons 0 empty)))) 1)
(define (add-up-3 a-list-of-3-numbers) ...)
```

Mas, como acessar os elementos de uma lista?

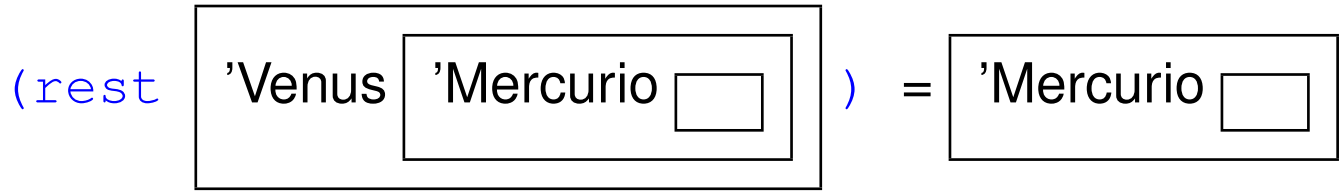
Operações sobre Listas

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

- `first` : aplicada sobre uma lista, devolve seu primeiro elemento;



- `rest` : aplicada sobre uma lista, retira seu primeiro elemento e devolve o resto da lista.



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 para a função `add-up-3`

```
;; add-up-3 : list-of-3-numbers  ->  number
;; to add up the three numbers in a-list-of-3-numbers
(define (add-up-3 a-list-of-3-numbers)
  ... (first a-list-of-3-numbers) ...
  ... (first (rest a-list-of-3-numbers)) ...
  ... (first (rest (rest a-list-of-3-numbers))) ... )
```

Exercício: Complete a definição desta funçao.

Definição de Dados para Listas de Tamanho Indeterminado

Até agora, todas as definições de dados que fizemos foram de tamanho fixo: estruturas ou listas com número fixo de elementos.

Em muitas situações, não sabemos quantos elementos as listas usadas em um determinado programa terão. Nesses casos, precisamo fazer uma definição genérica, ou seja, definir a classe de todas as listas finitas.

Uma `lista-de-símbolos` é ou

1. a lista vazia `empty` , ou

2. `(cons s los)` , onde `s` é um símbolo e `los` é uma lista de símbolos.

*Essa é uma definição **recursiva** !!!*

Processando Listas de Tamanho Indeterminado

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

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

```
;; contem-boneca? : lista-de-símbolos  ->  boolean
;; obj: determinar se a palavra 'boneca ocorre em uma lista de símbolos
(define (contem-boneca? uma-lista-de-simbolos) ...)
```

Exemplos para a função `contem-boneca`

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

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

(boolean=? (contem-boneca? (cons 'boneca empty))
           true)

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

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

Template para a função `contem-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 (contem-boneca? uma-lista-de-simbolos)
  (cond
    [(empty? uma-lista-de-simbolos) ...]
    [(cons? uma-lista-de-simbolos) ...]))
```

Ao invés de `(cons uma-lista-de-simbolos)` , podemos usar `else` na segunda cláusula.

Template para a função `contem-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:

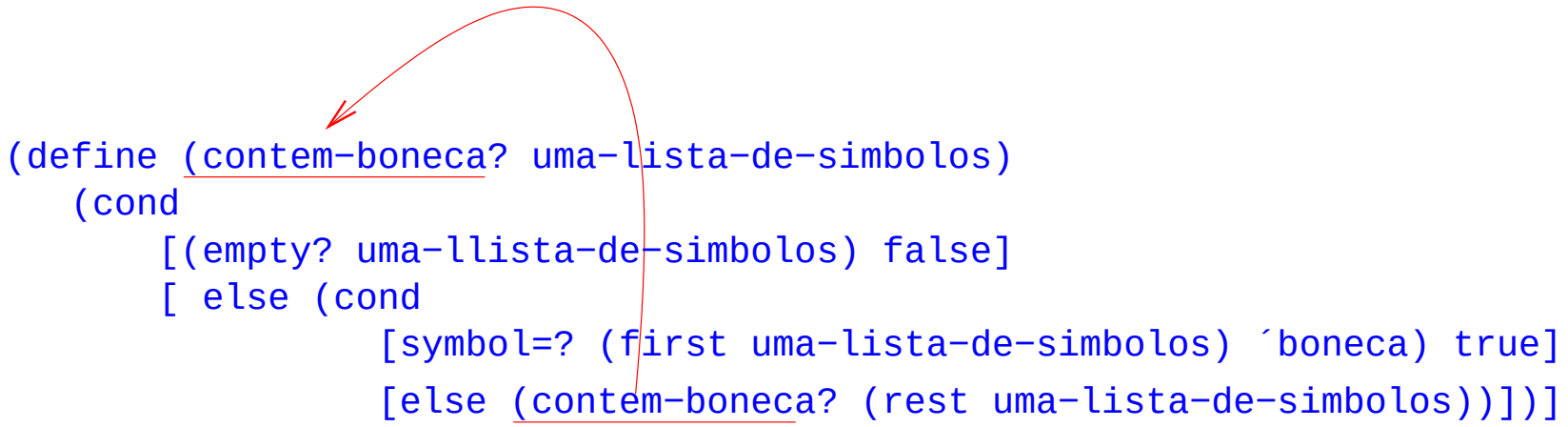
```
(define (contem-boneca? uma-lista-de-simbolos)
  (cond
    [(empty? uma-lista-de-simbolos) ...]
    [else ... (first uma-lista-de-simbolos)
          ... (rest uma-lista-de-simbolos) ...]))
```

Definindo a função `contem-boneca` ...

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

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

Definindo a função `contem-boneca` ...



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

*Essa é uma função **recursiva** !!!*

Exercícios

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

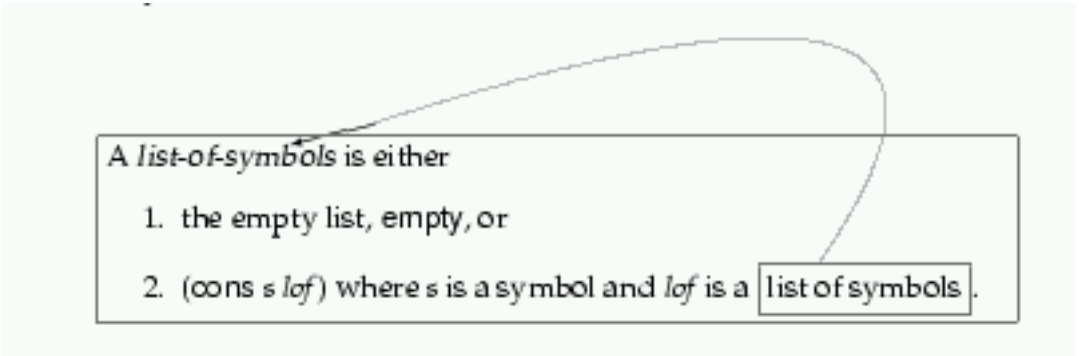
```
empty
(cons 'bola empty)
(cons 'flecha (cons 'boneca empty))
(cons 'pote (cons 'flecha (cons 'bola empty)))
```

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:

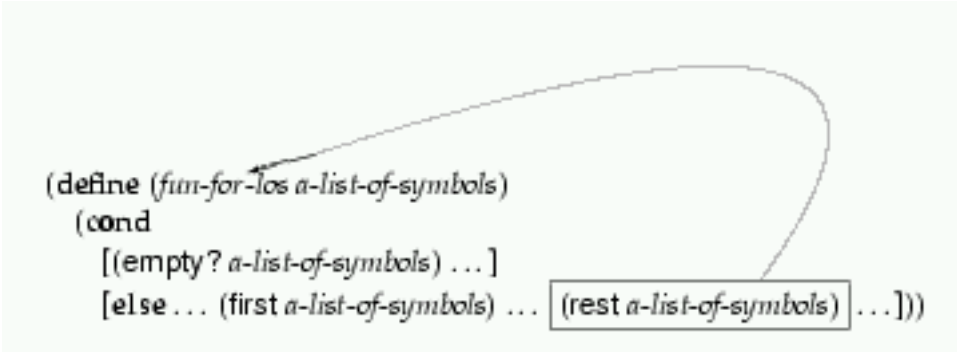
1. *A definição deve ter no mínimo duas cláusulas;*
2. *Pelo menos uma das cláusulas não deve referenciar o dado a ser definido.*

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 que 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 aos quais fazemos a chamada recursiva.

Exemplo de Definição do Corpo

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

```
;; how-many : list-of-symbols  ->  number
;; to determine how many symbols are on a-list-of-symbols
(define (how-many a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) ...]
    [else ... (first a-list-of-symbols) ...
      ... (how-many (rest a-list-of-symbols)) ...]))
```

```
(define (how-many a-list-of-symbols)
  (cond
    [(empty? a-list-of-symbols) 0]
    [else (+ (how-many (rest a-list-of-symbols)) 1)]))
```

Fases do Projeto de Algoritmos usando Recursão

Fase	Objetivo	Atividade
Projeto e Análise de Dados	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.

Contrato, Objetivo e Cabeçalho	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:
		`;; name : in1 in2 ...--> out}`
		`;; to compute ... from x1 ...}`
		`(define (name x1 x2 ...) ...)`

Fases do Projeto de Algoritmos usando Recursão...

Fase	Objetivo	Atividade
Exemplos	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 subclasse de dados aos quais a função pode ser aplicada
Template	formular um esboço para a função	colocar uma estrutura `cond` 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 autoreferências neste template e na definição de dados devem casar.

Fases do Projeto de Algoritmos usando Recursão...

Fase	Objetivo	Atividade
Corpo	completar a definição da função	construir expressões Scheme para cada uma das cláusulas do `cond`
Testes	encontrar erros	aplicar a função aos exemplos e checar 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 `dollar-store?` que, dada uma lista de preços, checa se todos os preços estão abaixo de 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 `convert` , que, dada uma lista de dígitos, produz o número decimal correspondente. O primeiro dígito da lista deve ser o menos significante.