LISP

- Autor: John McCarthy
- Desenvolvido para computação simbólica
- Programas concisos e próximos das definições matemáticas, adequados para utilização em IA

Idéias Principais

- Programação aplicativa
  - definir uma função ao invés de escrever um programa
  - analisa uma expressão ao invés de executar um programa
- Sintaxe simples
- Recursão como estrutura principal de controle
- Diferença principal em relação à primeira linguagem é o conjunto básico de valores utilizado, chamado *s-expressions* (expressões simbólicas)
- funções **não** tem efeitos colaterais, i.é., não mudam valor dos args.

Expressões simbólicas (s-expressions)

- símbolo, número ou uma lista ,“(s1 s2 ... sn)”, com zero ou mais “s-expressions”
- lista com zero elementos, “()”, é chamada *nil,* ou *lista nil*
- *#f é o valor falso*
- *#t é o valor verdadeiro*

Operações Básicas - (op s_1 ... s_n)

- < , > → retornam #t se s_1 e s_2 forem números e obedecerem à comparação, *#f* caso contrário
- + → se s_1 e s_2 forem números, retorna sua soma, caso contrário retorna erro
- - , / , * → similares ao anterior
- remainder → retorna o resto da divisão do primeiro elemento pelo segundo [(remainder 10 5) → 0 (remainder 5 10) → 5]
- = → retorna símbolo #t se s_1 e s_2 são o mesmo número, o mesmo símbolo ou se ambos forem *null*, retorna *#f* caso contrário
- and, or, not → operações lógicas com #t e #f

Operações Básicas - (op s_1 ... s_n)

- **number?**, **symbol?**, **list?**, **null?** - predicados que testam tipo de s_1
- **cons** - se s_2 for a lista *(s_{21} ... s_{2n})*, (**cons** s_1 s_2) retorna a lista (s_1 s_{21} ... s_{2n}). Se s_2 não for uma lista ocorre um erro.
- **car** - Se s_1 for a lista *(s_{11} s_{12} ... s_{1n})*, retorna s_{11}. Se s_1 não for uma lista, erro.
- **cdr** - Se s_1 for a lista *(s_{11} s_{12} ... s_{1n})*, retorna a lista *(s_{12} ... s_{1n})*. Se s_1 não for uma lista, erro.

- input -> expression | fundef
- fundef -> (**define (**function arglist) expression)
- arglist -> (variable*)
- expression -> value
  - | variable
  - | (**if** expression expression expression)
  - | (**define** variable expression)
  - | (**begin** expression+)
  - | (**optr** expression*)
  - | (**display** expression)
- optr -> function | value-op

Sintaxe - 2

- value -> inteiro | quoted-const
- quoted-const -> ´S-expression
- S-expression -> inteiro | símbolo | **(** S-expresion* **)**
- value-op -> + | - | * | / | = | < | > | **print | cons | car | cdr | number? | list? | null?|eq? | and | or | not | remainder**
  - (nota ‘=' só funciona para números, 'eq?' para quaisquer números/símbolos)
- function -> nome
- variable -> nome
- inteiro -> sequência de dígitos, talvez precedida do sinal ´-´
- nome -> sequência de caracteres que não sejam um inteiro e não contenham brancos

Criando constantes simbólicas

- podemos utilizar o símbolo de quotação simples “ ´ ” :
  - (cons ´a ´())

  (a)
  - (cons ´a ´(b))

  (a b)
  - (cdr '(a (b (c d)))) obs: lista com 2 elmentos, ´a e ´((b (c d)))

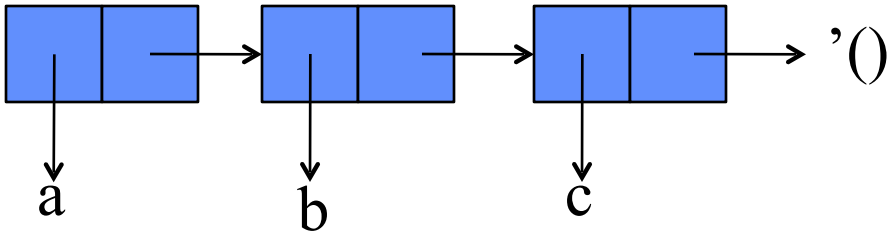
  ((b (c d)))
  - (null? ´())

  #t
  - (null? ´(()))

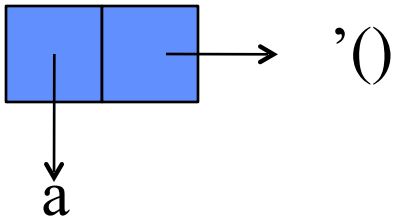
  #f

Notação Gráfica

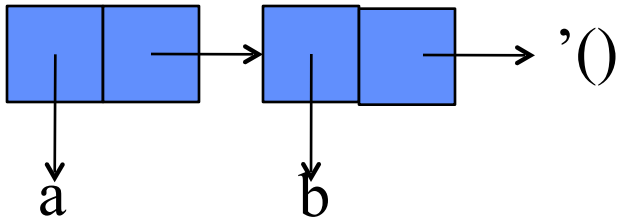
- '(a b c)



- (cons 'a '()) ou ' (a)



- (cons 'a '(b)) ou ' (a b)



  - Assim, “car” indica a seta da esquerda e “cdr” a seta da direita

Exemplo:

```
(define (tamanho l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l)))))
```

Exemplo:

```
(define (tamanho (l )
     (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l)))))
```

vamos ver como fica “(tamanho '(a b))”?

(tamanho '(a b))

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

(tamanho '(a b))

(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica "(tamanho '(a b))"?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr '(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
(+ 1 (tamanho (cdr '(a b)))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l)))))
```

vamos ver como fica "(tamanho '(a b))"?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr '(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b)))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr '(b)))))
(+ 1 (+ 1 (tamanho '())))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
(+ 1 (+ 1 (tamanho ‘())))
(+ 1 (+ 1 (if (null? '()) 0 (+ 1 (tamanho (cdr '()))))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica "(tamanho '(a b))"?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
(+ 1 (+ 1 (tamanho ‘())))
(+ 1 (+ 1 (if (null? '()) 0 (+ 1 (tamanho (cdr '()))))))
(+ 1 (+ 1 (if #t 0 (+ 1 (tamanho (cdr ()))))))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
(+ 1 (+ 1 (tamanho ‘())))
(+ 1 (+ 1 (if (null? '()) 0 (+ 1 (tamanho (cdr '()))))))
(+ 1 (+ 1 (if #t 0 (+ 1 (tamanho (cdr ()))))))
(+ 1 (+ 1 0))
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
(+ 1 (+ 1 (tamanho ‘())))
(+ 1 (+ 1 (if (null? '()) 0 (+ 1 (tamanho (cdr '()))))))
(+ 1 (+ 1 (if #t 0 (+ 1 (tamanho (cdr ()))))))
(+ 1 (+ 1 0))
(+ 1 1)
```

Exemplo:

```
(define (tamanho (l )
    (if (null? l ) 0 (+ 1 (tamanho (cdr l))))))
```

vamos ver como fica “(tamanho '(a b))”?

```
(tamanho '( a b))
(if (null? ' (a b)) 0 (+ 1 (tamanho (cdr ‘(a b))))
(if  #f  0 (+ 1 (tamanho (cdr '(a b))))
(+ 1 (tamanho (cdr '(a b)))
(+ 1 (tamanho '(b))
(+ 1 (if (null? '(b)) 0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (if  #f  0 (+ 1 (tamanho (cdr '(b))))))
(+ 1 (+ 1 (tamanho (cdr ‘(b)))))
(+ 1 (+ 1 (tamanho ‘())))
(+ 1 (+ 1 (if (null? '()) 0 (+ 1 (tamanho (cdr '()))))))
(+ 1 (+ 1 (if #t 0 (+ 1 (tamanho (cdr ()))))))
(+ 1 (+ 1 0))
(+ 1 1)
2
```

Algumas funções auxiliares

- Segundo elemento de uma lista:

```
(define (cadr  l ) (car (cdr l)))
```

- Primeiro elemento do primeiro elemento de lista:

```
(define (caar  l ) (car (car l)))
```

- Terceiro elemento de lista:

```
(define (caddar l) (car (cdr (cdr l))))
```

- Pergunta de um elemento é atômico (não pode ser dividido):

```
(define (atom? x)
        (if (null? x)  # lista nula não pode ser dividida
            #t
            (not (list? x))))
```

Exemplo - números primos <= n

- Um bom exemplo para entendermos o funcionamento de listas e o uso de recursão é tentar solucionar o problema de, dado N, produzir a lista de todos os números primos menores que N.
- Intuitivamente podemos fazer isso primeiro criando uma lista de todos os números menores que N e depois removermos os números não primos.

Exemplo - números primos <= n

- Para isso vamos inicialmente usar uma função que cria uma lista de números consecutivos para um intervalo definido

```
(define (intervalo m n)
        (if (> m n) '()
            (cons m (intervalo (+ m 1) n))))
```

Exemplo - números primos <= n

- Para isso vamos inicialmente usar uma função que cria uma lista de números consecutivos para um intervalo definido

```
(define (intervalo m n)
        (if (> m n) '()
            (cons m (intervalo (+ m 1) n))))
```

- Em seguida vamos fazer uma função que, dada uma lista e um número, retorna uma nova lista que é a lista original sem os múltiplos do número

```
(define (remove-multiplos num lista)
      (if (null? lista) lista
          (if (divide? num (car lista))
              (remove-multiplos num (cdr lista))
              (cons (car lista) (remove-multiplos num (cdr lista))))))
(define (divide? a b) (= (remainder b a) 0))
```

Exemplo - números primos <= n

- Agora, podemos usar remove multiples de maneira interativa usando uma função recursiva, que remove os múltiplos do primeiro elemento de uma lista e é aplicada recursivamente no restante da lista (cuidado! é sutil)

```
(define (filtro lista)
    (if (null? lista)
        lista
        (cons (car lista)
              (filtro (remove-multiplos (car lista) (cdr lista))))))
```

Exemplo - números primos <= n

- Agora, podemos usar remove multiples de maneira interativa usando uma função recursiva, que remove os múltiplos do primeiro elemento de uma lista e é aplicada recursivamente no restante da lista (cuidado! é sutil)

```
(define (filtro lista)
    (if (null? lista)
        lista
        (cons (car lista)
              (filtro (remove-multiplos (car lista) (cdr lista))))))
```

- Basta finalmente usarmos *intervalo* e *filtro*:

```
(define (primos<=n n)
          (filtro (intervalo  2 n)))
```

Exercícios-1

- Encontrar o último elemento de uma lista:

  (define (ultimo lista) ….

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista))
        (car lista)
```

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista0))
        (car lista)
        (ultimo (cdr lista))))
```

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista0))
        (car lista)
        (ultimo (cdr lista))))
```

- Ver se duas expressões simbólicas arbitrárias são iguais

```
(define  (equal? e1 e2) …
```

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista0))
        (car lista)
        (ultimo (cdr lista))))
```

- Ver se duas expressões simbólicas arbitrárias são iguais

```
(define  (equal? e1 e2) …
    (if (and (atom? e1) (atom? e2))
        (eq? e1 e2)
```

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista0))
        (car lista)
        (ultimo (cdr lista))))
```

- Ver se duas expressões simbólicas arbitrárias são iguais

```
(define  (equal? e1 e2) …
    (if (and (atom? e1) (atom? e2))
        (eq? e1 e2)
        (if (or  (atom? e1) (atom? e2))
            #f
```

Exercícios-1

- Encontrar o último elemento de uma lista:

```
(define (ultimo lista)
    (if (null? (cdr lista0))
        (car lista)
        (ultimo (cdr lista))))
```

- Ver se duas expressões simbólicas arbitrárias são iguais

```
(define  (equal? e1 e2) …
    (if (and (atom? e1) (atom? e2))
        (eq? e1 e2)
        (if (or  (atom? e1) (atom? e2))
            #f
            (and (equal? (car e1) (car e2))
                 (equal? (cdr e1) (cdr e2))))))
```

Exercícios - 2 => Insertion Sort

- Como estamos trabalhando com listas, um problem interessante é modelar algoritmos de ordenação
- Vamos tentar o velho “insertion sort”.
- Neste algoritmo vamos acrescentando elementos de uma lista desordenada em uma nova lista.
- A cada passo inserimos um novo elemento colocando ele na posição correta na nova lista
- Assim, se tivermos a lista de números (8, 7, 1, 2) construiríamos a lista ordenada passo a passo:
  - ()
  - (8)
  - (7,8)
  - (1, 7, 8)
  - (1, 2, 7, 8)
- Vejamos como fica isso em Lisp

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenada, com o novo elemento

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenada, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
    (if (nulll? lista)
        (cons novo '())
        (if  (< novo (car lista))
             (cons novo lista)
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< novo (car lista))
           (cons novo lista)
           (cons (car lista) (insert novo (cdr lista))))))
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< novo (car lista))
           (cons novo lista)
           (cons (car lista) (insert novo (cdr lista))))))
```

- Agora podemos definir o insertion sort, recursivamente, ordenamos o final da lista, depois inserimos o primeiro elemento

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< novo (car lista))
           (cons novo lista)
           (cons (car lista) (insert novo (cdr lista))))))
```

- Agora podemos definir o insertion sort, recursivamente, ordenamos o final da lista, depois inserimos o primeiro elemento

```
(define (insertion-sort  lista )
    (if  (null? lista) lista
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< novo (car lista))
           (cons novo lista)
           (cons (car lista) (insert novo (cdr lista))))))
```

- Agora podemos definir o insertion sort, recursivamente, ordenamos o final da lista, depois inserimos o primeiro elemento

```
(define (insertion-sort  lista )
  (if  (null? lista) lista

                (insertion-sort (cdr lista)))))
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< novo (car lista))
           (cons novo lista)
           (cons (car lista) (insert novo (cdr lista))))))
```

- Agora podemos definir o insertion sort, recursivamente, ordenamos o final da lista, depois inserimos o primeiro elemento

```
(define (insertion-sort  lista )
    (if  (null? lista) lista
         (insert (car lista)
                 (insertion-sort (cdr lista)))))
```

Exercícios - 2 => Insertion Sort

- Primeiro definimos uma função para inserir um elemento em uma lista ordenada, produzindo uma nova lista ordenda, com o novo elemento

```
(define (insert novo lista)
  (if (null? lista)
      (cons novo '())
      (if  (< x (car l))
           (cons x l)
           (cons (car l) (insert x (cdr l))))))
```

- Agora podemos definir o insertion sort, recursivamente, ordenamos o final da lista, depois inserimos o primeiro elemento

```
(define (insertion-sort  l )
  (if  (null? l) l
       (insert (car l)
               (insertion-sort (cdr l)))))
```

- Agora tentemos (insertion-sort ´(4 3 1 2))

Exercícos - 3 => Listas de Associação

- Uma lista de associação é um dicionário onde associamos chaves arbitrárias (qualquer elemento da linguagem), a valores arbitrários.
- Vamos usar uma nova função do Racket para nos ajudar a criar listas sem usar tantos ‘cons’:
  - (list arg1 arg2 …argn)
- Esta função cria uma lista com o numero de elementos passados como argumentos. Ela aceita numero arbitrário de argumentos
- Assim
  - (list ‘a ‘b 'c) = (cons ‘a (cons ‘b (cons ‘c ‘())))
- Outra vantagem é que aceita expressões arbitrárias como argumento
  - (list (+ 2 3) a ‘a) →(cons (+ 2 3) (cons a (cons ‘a ‘())))
  - Em outras palavras, uma lista de 3 elementos, o primeiro é o resultado de (+ 2 3), o segundo o valor da variável “a" e o terceiro o símbolo ‘a.

Exercícos - 3 => Listas de Associação

- Primeiramente devemos pensar em como vamos representar este dicionário. Como só temos listas, vamos usar uma lista de pares <chave, valor>
  - Ex: ‘((alan verde) (pedro amarelo) (wander roxo))
  - Lista com tres associações
    - chaves:{‘alan, ‘pedro, ‘wander}
    - Valores { ‘verde , ‘amarelo, ‘roxo}

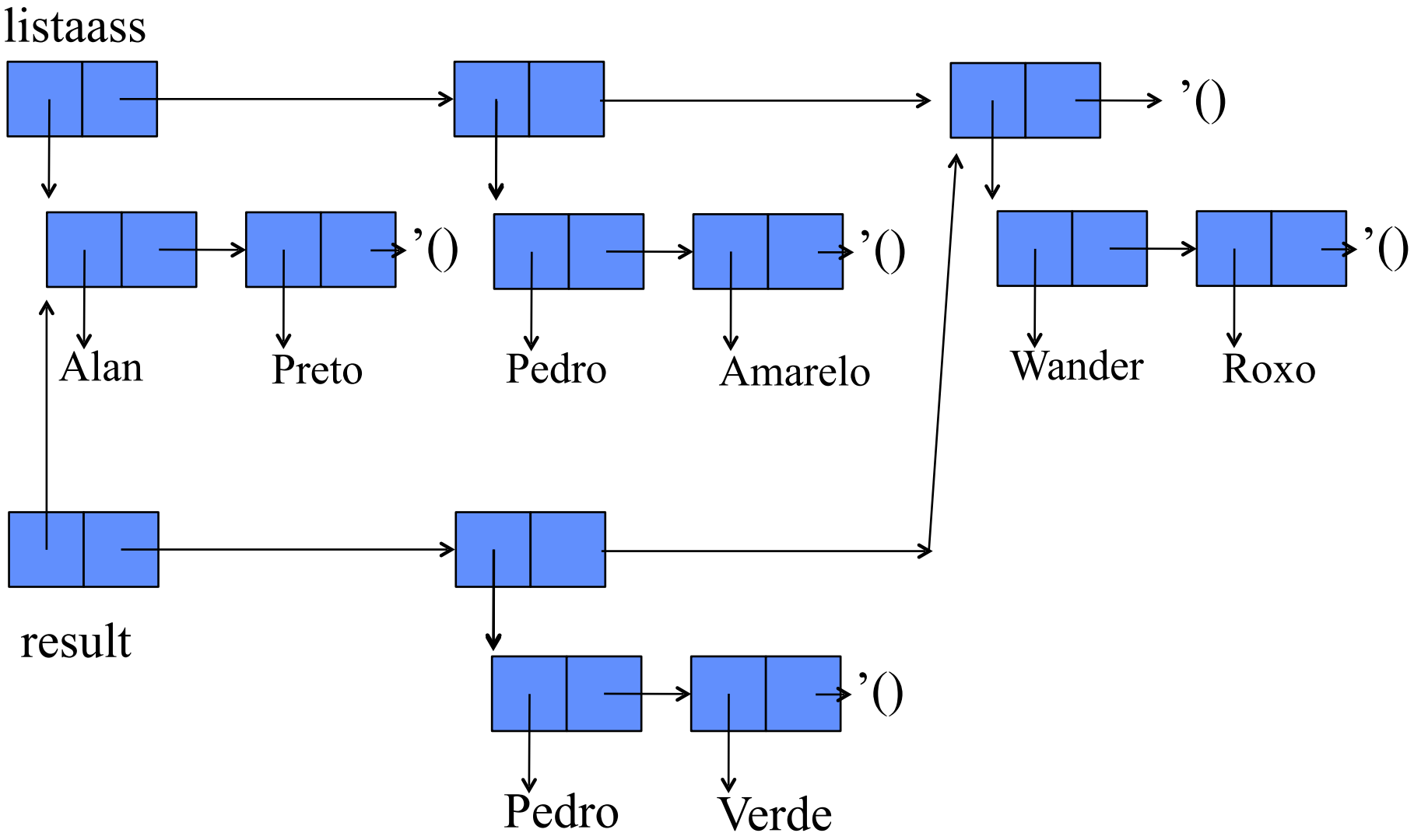
- Agora precisamos decidir o nome das funções para criar e manipular:
  - (*mkassoc chave valor lista)* - recebe uma lista de associação, e um novo par <chave, valor> para acrescentar nesta lista
  - (*assoc chave lista)*: recebe uma chave e uma lista de associação, e retorna o valos associado àquela chave, ou ‘() se não existir
    - (assoc ‘wander ‘((alan verde) (wander roxo))) -> ‘roxo
- Vejamos agora a definição das funções

Exercícos - 3 => Listas de Associação

- primeiro representação: lista de pares <chave, valor>
- funções para criar e manipular: *mkassoc* e *assoc:*
- (define (**assoc** chave lista)
  (if (null? lista) '()
    (if (equal? chave (caar lista))
      (cadar lista)
      (assoc chave (cdr lista)))))
- (define (**mkassoc** chave valor lista)
  (if (null? lista) (list (list chave valor))
    (if (= chave (caar lista))
      (cons (list chave valor) (cdr lista))
      (cons (car lista) (mkassoc chave valor (cdr lista))))))
- Note que não modificamos a lista, criamos uma nova lista onde compartilhamos partes da lista anterior

Notação Gráfica

(define listaass '((Alan Preto) (Pedro Amarelo) (Wander Roxo))
(define result (mkassoc Pedro Verde listas))



Exercícios - 4 => Conjuntos

- Agora vamos representar conjuntos
- Precisaremos, claro usar listas para isso
  - Com ou sem repetição?
    - Eficiência em adição vs eficiência de espaço e de membresia
- Quais são as funções principais?
  - conj-vazio: Criação de novo conjunto
  - Membro? - testa se um elemento pertence a um conjunto
  - Addiciona - dado um elemento e um conjunto, retorna novo conjunto com o elemento adicionado ao conjunto original
  - Operações entre conjuntos:
    - união, intersecção, contido?, etc'

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
  (if (membro? elem conj) conj (cons elem conj)))
(define (uniao conj1 conj2)
  (if (null? conj1) conj2
      (uniao (cdr conj1) (adiciona (car conj1) conj2))))
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ;
```

Exercícios - 4 => Conjuntos

ou (define (conj-vazio) ´())

Exercícios - 4 => Conjuntos

(define conj-vazio '()) ; obs: ou (define (conj-vazio) ´())

(define (membro? elem conj)

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
  (if (membro? elem conj) conj (cons elem conj)))
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
  (if (membro? elem conj) conj (cons elem conj)))
(define (uniao conj1 conj2)
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
  (if (membro? elem conj) conj (cons elem conj)))
(define (uniao conj1 conj2)
  (if (null? conj1) conj2
```

Exercícios - 4 => Conjuntos

```
(define conj-vazio '()) ; obs: ou (define (conj-vazio ) ´())
(define (membro? elem conj)
  (if (null? conj) #f
      (if (equal? elem (car conj)) #t
          (membro? elem (cdr conj)))))
(define (adiciona elem conj)
  (if (membro? elem conj) conj (cons elem conj)))
(define (uniao conj1 conj2)
  (if (null? conj1) conj2
      (uniao (cdr conj1) (adiciona (car conj1) conj2))))
```

Escopo de variáveis - cuidado!

- (define (soma-sexpr l)
  - (if (null? l) 0
  - (f (number? l) l
  - (+ (soma-sexpr (car l)) (soma-sexpr (cdr l))))))
- e se usássemos variáveis globais?
- (define (soma-errada l)

```
(if (null? l) 0
    (if (number? l) l
        (begin (define tmp (soma-errada (car l)))
               (+ (soma-errada (cdr l)) tmp))))))
```

Atenção isso não funciona

No nosso Dr. Racket atual, defines não são permitidos dentro de expressões, apenas em 'begin'

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

(define (soma-certa l) (soma-certa-aux l 0))

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

(define (soma-certa l) (soma-certa-aux l 0))

(define (soma-certa-aux l tmp)

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

```
(define (soma-certa  l ) (soma-certa-aux l 0))
(define (soma-certa-aux  l  tmp)
      (if (null? l) tmp
```

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

```
(define (soma-certa  l ) (soma-certa-aux l 0))
(define (soma-certa-aux  l  tmp)
      (if (null? l) tmp
        (if (number? l) (+ tmp l)
```

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

```
(define (soma-certa  l ) (soma-certa-aux l 0))
(define (soma-certa-aux  l  tmp)
      (if (null? l) tmp
        (if (number? l) (+ tmp l)
            (begin (set! tmp (soma-certa-aux (car l) tmp))
```

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

```
(define (soma-certa  l ) (soma-certa-aux l 0))
(define (soma-certa-aux  l  tmp)
      (if (null? l) tmp
        (if (number? l) (+ tmp l)
            (begin (set! tmp (soma-certa-aux (car l) tmp))
                   (soma-certa-aux (cdr l) tmp)))))
```

Criando variáveis locais

- utilizamos argumentos para criar variáveis locais

```
(define (soma-certa  l ) (soma-certa-aux l 0))
(define (soma-certa-aux  l  tmp)
      (if (null? l) tmp
        (if (number? l) (+ tmp l)
            (begin (set! tmp (soma-certa-aux (car l) tmp))
                   (soma-certa-aux (cdr l) tmp)))))
```

- **Recursão eficiente: recursão de cauda**

Exemplo: “Eval” em Lisp - 1

- programas em Lisp podem ser representados facilmente como expressões simbólicas
- => extensibilidade
- interpretador “meta-circular” => interpretador de Lisp escrito em Lisp
- representação: (+ x (* y 4)) como:

  ´(+ x (* y 4))
- Ou

```
(list ‘+
      ‘x
      (list ‘* 'y '4))
```

- Ou

```
(cons ´+
      (cons ´x
            (cons (cons ‘*
                        (cons  ´y
                               (cons 4 ´()))) ‘
                  '())))
```

“Eval” em Lisp - 2

- problema: como representar constantes simbólicas, i.e. ´x, ´(a (b c)), etc
- solução: novo operador *quote:*
  - ´x = (quote x)
  - ´(a (b c)) = (quote (a (b c))

“Eval” em Lisp - versão 1

- obs: só ‘numeros ou operações com dois argumentos

```
(define (eval exp)
    (if (number? exp) exp
        ; notem que vou usar uma função auxiliar para
        ; tratar cada operador, mas antes preciso avaliar
        ; os argumentos
        (apply-op (car exp)
            (eval (cadr exp))
            (eval (caddr exp)))))
```

“Eval” em Lisp - versão 1

- obs: só ‘numeros ou operações com dois argumentos

```
(define (eval exp)
        (if (number? exp) exp
            ; notem que vou usar uma função auxiliar para
            ; tratar cada operador, mas antes preciso avaliar
            ; os argumentos
            (apply-op (car exp)
               (eval (cadr exp))
               (eval (caddr exp)))))
;Agora tratamos as operações aritméticas
(define (apply-op f x y)
        (if (equal? f  '+)
             (+ x y)
            (if (equal? f  '-) (- x y)
                (if (equal? f  '/)
                    (/ x y)
                    (if (equal? f  '*)
                        (* x y)
                        ‘erro)))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(if (number? ’(+ 2 (* 3 4) )
    ’(+ 2 (* 3 4)
   (apply-op ’(car (+ 2 (* 3 4))
                   (eval ’(cadr '(+ 2 (* 3 4))
                   (eval (caddr '(+ 2 (* 3 4)))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(if (number? ’(+ 2 (* 3 4) )
    ’(+ 2 (* 3 4)
  (apply-op ’(car (+ 2 (* 3 4))
                  (eval ’(cadr '(+ 2 (* 3 4))
                  (eval (caddr '(+ 2 (* 3 4)))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          (eval ’2)
          (eval '(* 3 4)))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          (eval ’2)
          (eval '(* 3 4)))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (eval '(* 3 4)))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (if (number? '(* 3 4))
              ‘(* 3 4)
              (apply-op (car '(* 3 4))
                        (eval (cadr  '(* 3 4)))
                        (eval (caddr '(* 3 4))))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (if (number? '(* 3 4))
              ‘(* 3 4)
              (apply-op (car '(* 3 4))
                        (eval (cadr  '(* 3 4)))
                        (eval (caddr '(* 3 4))))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4)

```
(apply-op ’+
          2
          (apply-op (car '(* 3 4))
                    (eval (cadr  '(* 3 4)))
                    (eval (caddr '(* 3 4))))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4)

```
(apply-op ’+
          2
          (apply-op '*
                    (eval (cadr  '(* 3 4)))
                    (eval (caddr '(* 3 4))))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (apply-op '*
                    (eval 3)
                    (eval (caddr '(* 3 4))))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (apply-op '*
                    (eval 3)
                    (eval 4)))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (if (equal? '* '+)
              (+ 3 4)
              (if (equal? '*  '-)
                  (- 3 4)
                  (if (equal? '*  '/)
                      (/ 3 4)
                      (if (equal? '*  '*)
                          (* 3 4)
                          ‘erro))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4)))

```
(apply-op '+
          2
          (if (equal? '*  '*)
              (* 3 4)
              'erro))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op ’+
          2
          (* 3 4))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(apply-op '+
          2
          12)
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

```
(if (equal? '+ ‘+)
    (+ 2 12)
    (if (equal? '+  '-)
        (- 2 12)
        (if (equal? '+  ‘/)
            (/ 2 12)
             (if (equal? '+  ‘*
                 (* 2 12)
                  ‘erro))))
```

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

  (+ 2 12)

“Eval” em Lisp - versão 1

- Exemplo: (eval ‘(+ 2 (* 3 4))

  14

“Eval” em Lisp - acrescentando variáveis

- para acrescentar variáveis precisamos ser capazes de manter valores para variáveis
- Podemos usar uma estrutura já desenvolvida anteriormente: lista de associações
  - Associa “nome” de uma variável a um valor
- chamamos esta lista de ***Ambiente*** (“environment”)
- Para que isso funcione, agora o Ambiente precisa ser parâmetro de eval
- Para representar o nome das variáveis usaremos símbolos

“Eval” em Lisp - versão 2

```
(define (eval exp ambiente)
    (if (number? exp) exp
```

“Eval” em Lisp - versão 2

```
(define (eval exp ambiente)
    (if (number? exp) exp
        (if (symbol? exp)
            (assoc exp ambiente)
```

“Eval” em Lisp - versão 2

```
(define (eval exp ambiente)
    (if (number? exp) exp
        (if (symbol? exp)
            (assoc exp ambiente)
            (apply-op (car exp)
                      (eval (cadr exp) ambiente)
                      (eval (caddr exp) ambiente)))))
```

“Eval” em Lisp - versão 2

```
(define (eval exp ambiente)
    (if (number? exp) exp
        if (symbol? exp)
            (assoc exp ambiente)
            (apply-op (car exp)
                      (eval (cadr exp) ambiente)
                      (eval (caddr exp) ambiente)))))
```

exemplo: (eval ´(+ i (/ 9 i)) (mk-assoc ´i 3 ´()))

“Eval” em Lisp - versão 2

```
(define (eval exp ambiente)
    (if (number? exp) exp
        if (symbol? exp)
            (assoc exp ambiente)
            (apply-op (car exp)
                      (eval (cadr exp) ambiente)
                      (eval (caddr exp) ambiente)))))
```

exemplo: (eval ´(+ i (/ 9 i)) (mk-assoc ´i 3 ´()))

EXPRESSÃO AMBIENTE

“Eval” em Lisp - acrescentando expr. simb.

- Lembrando, vamos precisar indicar ao interpretador o que é o nome de uma variável e o que é um símbolo

  (eval ‘(car (cdr ′(a b c))) ′()) ==> não funciona

“Eval” em Lisp - acrescentando expr. simb.

- Lembrando, vamos precisar indicar ao interpretador o que é o nome de uma variável e o que é um símbolo

  (eval ‘(car (cdr ´(a b c))) ´()) ==> não funciona

- a solução é criar um novo operador como ', mas que nosso interpretador possa "ver":

  (eval ´(car (cdr (quote (a b c))))) ´()

"Eval" em Lisp - acrescentando expr. simb.

- Precisamos assim atualizar nossa função *eval:*

```
(define (eval exp ambiente)
    (if (number? exp) exp
     (if (symbol? exp) (assoc exp ambiente)
         (if (equal?  (car exp)  'quote)
             (cadr  expr)
             (apply-op (car exp)
                       (eval (cadr exp) ambiente)
                       (eval (caddr exp) ambiente))))))
```

O Lisp como ele é -1

- A linguagem Lisp não é limitada como nossa linguagem exemplo
- Vejamos algumas diferenças importantes
- (cond (e11 e12) (e21 e22)... (eN1 eN2))
- expressões simbólicas: “cons” junta quaisquer dois elementos
- outros tipos de dados
  - números: precisão infinita para inteiros, complexos, ponto flutuante
  - vetores

O Lisp como ele é - 2

- Operador *cond:*
  - (cond (e_{11} e_{12}) (e_{21} e_{22})... (e_{N1} e_{N2}))
- Expressões simbólicas: “cons” junta quaisquer dois elementos
  - (cons 3 (cons 4 ‘()) ==> (3 4)
  - (cons 3 4) ==> (3 . 4)
  - Na verdade nosso racket já faz isso!
- outros tipos de dados
  - números: precisão infinita para inteiros, complexos, ponto flutuante
  - vetores (acesso indexado rápido)

O Lisp como ele é 3
efeitos colaterais

- efeito colateral = qualquer mudança nos valores do ambiente causada por uma função
- nosso interpretador altera apenas variáveis globais
- perigoso: chamadas de uma função podem alterar valor retornado por outra função
- rplaca, rplacd:

O Lisp como ele é - 4: rplaca, rplacd

=>(set L1 ´(a b c))
(a b c)
=>(set L2 L1)
(a b c)
=>(rplaca L2 ´d)
(d b c)
=> L2
(d b c)
=> L1
(d b c)

O Lisp como ele é - 5

- código mais eficiente (menos cons, menos car, menos cdr)
- listas circulares (com rplacd)
- macros
  - aplicada a expressões não a valores
  - permite estender linguagens (if, for)
  - “economiza” chamada
- coleta de lixo

O Lisp como ele é - 6
"Coleta de Lixo”

- Vocês devem ter notado que manipulação de listas sem uso de efeitos colaterais gera muitas células “mortas"
  - Por exemplo quando queremos mudar uma lista, reaproveitamos os elementos iguais e as células finais da lista mas criamos novas células para o início da lista.
  - Se a lista anterior não for mais usada, temos células que não tem utilidade: isso pode ocasionar preenchimento total da memória para um programa que roda muito tempo
  - Em C++ e C podemos descartadas estruturas explicitamente, quando sabemos que elas não serão mais usadas (“delete" e "free" vs “new” e “malloc”).
  - Em Lisp, não precisamos fazer esta deslocação explicitamente, o sistema faz uma operação chamada coleta de lixo que detecta automaticamente as células que não podem ser usadas e as “descarta"

Scheme

- Scheme é uma linguagem derivada do Lisp, com o mesmo princípio de funcionamento, mas que trata funções de maneira mais elegante e genérica
-

Racket

- Em nosso curso vamos usar a linguagem Racket.
- Em sua versão “crua” ela funciona como a linguagem Scheme
- Porém Racket contém várias extensões que devem ajudar no nosso curso como facilidades para
  - Tipar a linguagem
  - Criar tipos e mecanismos de teste
  - Funções para nos ajudar a criar interpretadores
- Faça download e instalação do Racket
  - Foi este o sistema que utilizamos nas demonstrações desta aula
- Leia a introdução à linguagem (o link está no PACA)

Usando Racket

- Racket é um sistema interativo que você pode usar diretamente da linha de comando, digitando os programas na medida que vai prosseguindo
- Porém ele tem um ambiente que facilita muito o uso, e é o que vamos utilizar
- Ao baixar o pacote, você pode utilizar a linguagem racker chamando-a da linha de comando
  - Neste caso você deve ver uma interface semelhante ao que usamos nesta aula.

Um interpretador em Racket

- Neste curso vamos implementar interpretadores das linguagens que estamos atendendo.
- Para isso usaremos um dialeto deRacket tipado
- Para isso devemos indicar “#lang plai-typed” no início de nossos programas
- Uma das diferenças de plai-typed para scheme é o suporte para tipos
  - Este pacote precisa ser instalado (veja no e-disciplinas)
- Tipos são definidos colocando-se “:” e o tipo após especificação de argumentos e de perfil da função
- Ex (obs: cuidado com os espaços antes de depois do “:”)

```
(define (soma-tipada [x : number] [y : number ]) : number
        (+ x y))
```

- Veja a documentação para mais detalhes.