SASL : linguagens funcionais e avaliação por demanda

SASL

- Programação funcional utiliza fechamentos para obter maior poder de expressão e flexibilidade
- Vejamos um exemplo Pascal/Scheme
- Pascal

```
function search (n : integer) : integer;
var i: integer;
begin
      i := 1;
      while (i <= n) and not (pred (i))
         do i := i + 1;
    search := i
end;
```

- Scheme (direto, se tivéssemos while)

```
(define iter-search (lambda (n)
        (begin
              (set i 1)
              (while (and (<= i n)
                          (not (pred i)))
                     (set! i (+ 1 i)))
              i)))
```

SASL

- Programação funcional utiliza fechamentos para obter maior poder de expressão e flexibilidade
- Vejamos um exemplo Pascal/Scheme:

Pascal

```
function search (n : integer) : integer;
var i: integer;
begin
        i := 1;
        while (i <= n) and not (pred (i))
           do i := i + 1;
    search := i
end;
```

Scheme (versão recursiva)

```
(define interval (lambda (start end)
                 (if (< end start) '()
                    (cons start (interval (+ 1 start) end)))))
(define fun-search (lambda (n)
                (find-val pred (interval 1 n) (+ 1 n))))
(define find-val (lambda (pred lista valor-falha)
                 (if (null? lista)
                    valor-falha
                    (if (pred (car lista))
                           (car lista)
                           (find-val pred (cdr lista) valor-falha)
                           ))))
```

SASL

- Programação funcional utiliza fechamentos para obter maior poder de expressão e flexibilidade
- Vejamos um exemplo Pascal/Scheme:

Pascal

```
function search (n : integer) : integer;
var i: integer;
begin
        i := 1;
        while (i <= n) and not (pred (i))
           do i := i + 1;
    search := i
end;
```

Scheme (versão recursiva)

```
(define interval (lambda (start end)
                 (if (< end start) '()
                    (cons start (interval (+ 1 start) end)))))
(define fun-search (lambda (n)
                 (find-val pred (interval 1 n) (+ 1 n))))
(define find-val (lambda (pred lista valor-falha)
                 (if (null? lista)
                    valor-falha
                    (if (pred (car lista))
                           (car lista)
                           (find-val pred (cdr lista) valor-falha)
                           ))))
```

- ????
  - Versão em scheme é mais longa e menos clara!!!

Vantagens da versão funcional

- Porém:
  - Utilizando estilo funcional tornamos código mais REUTILIZÁVEL
  - Podemos gerar novas funções mais rapidamente e com MENOS ERROS
- Ex: achar 1º quadrado que satisfaz um predicado *pred*

```
(set fun-search-sqr (lambda(n)
      (find-val pred
                (mapcar sqr (interval 1 n))
                (sqr (+ 1 n))))
```

- Na versão pascal teríamos que reescrever toda a função

Versão funcional

- Problemas
  - Ineficiente → sempre geramos a lista toda
  - Não pode ser utilizado se não conhecemos limite de *i*
- Solução
  - só gerar o pedaço da lista que vamos utilizar
  - LAZY EVALUATION (“avaliação por demanda”)

Avaliação por demanda: princípio básico

- NÃO CALCULE O VALOR DE UMA EXPRESSÃO ATÉ QUE ELE SEJA NECESSÁRIO
- Scheme viola este princípio
  1. Calcula o valor dos argumentos de uma função antes que esta seja chamada pelo usuário
  2. Note porém que:
     - Ao aplicar *CONS* não é necessário calcular o valor dos argumentos até que *CAR* ou *CDR* sejam aplicados à célula

Avaliação por demanda no exemplo

- (Vamos supor que predicado *pred* é válido para n=2)

(fun-search 100)
(find-val pred (interval 1 100) (+ 1 100))

(

Avaliação por demanda no exemplo

- (Vamos supor que predicado *pred* é válido para n=2)

(fun-search 100)

(find-val pred (interval 1 100) (+ 1 100))

; precisamos testar lista e portanto precisamos iniciar o cálculo de
;(interval 1 100), mas SÓ O SUFICIENTE
(interval 1 100)
(if (> 1 100) '() (cons 1 (interval (+ 1 1) 100))
(cons 1 (interval (+ 1 1) 100)
(1 ...)

Avaliação por demanda no exemplo

- (Vamos supor que predicado *pred* é válido para n=2)

(fun-search 100)

(find-val pred (interval 1 100) (+ 1 100))

; precisamos testar lista e portanto prcisamos iniciar o cálculo de
;(interval 1 100), mas SÓ O SUFICIENTE
(interval 1 100)
(if (> 1 100) '() (cons 1 (interval (+ 1 1) 100))
(cons 1 (interval (+ 1 1) 100)
(1 ...)

NÃO CALCULAMOS SEGUNDA PARTE!

Avaliação por demanda no exemplo

- (Vamos supor que predicado *pred* é válido para n=2)

(fun-search 100)

(find-val pred (interval 1 100) (+ 1 100))

; precisamos testar lista e portanto prcisamos iniciar o cálculo de
;(interval 1 100), mas SÓ O SUFICIENTE
(interval 1 100)
(if (> 1 100) '() (cons 1 (interval (+ 1 1) 100))
(cons 1 (interval (+ 1 1) 100)
(1 ...)

(if (null? '(1 …)) (+ 1 100)
    (if (pred (car '(1 ...)) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100))))

PRIMEIRO TESTE É FALSO

Avaliação por demanda no exemplo

```
(if (null? '(1 …)) (+  1 100)
    (if (pred (car '(1 ...)) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car '(1 ...)) (car '(1 ...)
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
```

Avaliação por demanda no exemplo

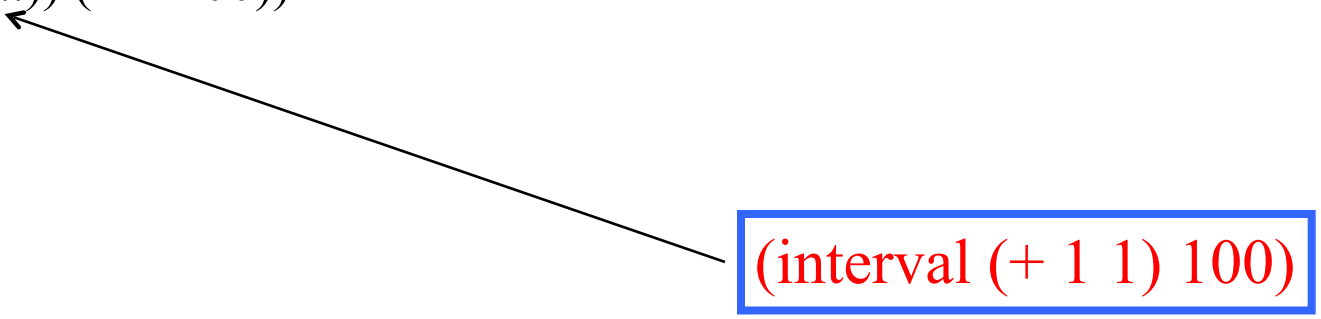
```
(if *null? ‘(1 ...) (+  1 100)
    (if (pred (car ‘(1 ...)) (car ‘(1 ...))
        (find-val pred (cdr ‘(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car ‘(1 ...)) (car ‘(1 ...)
        (find-val pred (cdr ‘(1 ...)) (+ 1 100)))
(if (pred 1) (car ‘(1 ...))
        (find-val pred (cdr ‘(1 ...)) (+ 1 100)))
```

Avaliação por demanda no exemplo

```
(if *null? '(1 ...) (+  1 100)
    (if (pred (car '(1 ...)) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car '(1 ...)) (car '(1 ...)
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
(if (pred 1) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
```

Avaliação por demanda no exemplo

```
(if *null? '(1 ...) (+  1 100)
    (if (pred (car '(1 ...)) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car '(1 ...)) (car '(1 ...)
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
(if (pred 1) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
(find-val pred (cdr '(1 ...)) (+ 1 100))
```



Avaliação por demanda no exemplo

```
(if *null? ‘(1 ...) (+  1 100)
    (if (pred (car ‘(1 ...)) (car ‘(1 ...))
        (find-val pred (cdr ‘(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car ‘(1 ...)) (car ‘(1 ...)
        (find-val pred (cdr ‘(1 ...)) (+ 1 100)))
(if (pred 1) (car ‘(1 ...))
        (find-val pred (cdr ‘(1 ...)) (+ 1 100)))
(find-val pred (cdr ‘(1 ...)) (+ 1 100))
(if (null? (cdr ‘( 1 ...)) (+ 1 100)
    (if (pred (car (cdr ‘(1 ...))) (car (cdr ‘( 1 ...))
        (find-val  pred (cdr (cdr ‘( 1 ...)) (+ 1 100))))
```

Avaliação por demanda no exemplo

```
(if *null? '(1 ...) (+  1 100)
    (if (pred (car '(1 ...)) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100))))
; primeiro teste é falso!!!!
(if (pred (car '(1 ...)) (car '(1 ...)
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
(if (pred 1) (car '(1 ...))
        (find-val pred (cdr '(1 ...)) (+ 1 100)))
(find-val pred (cdr '(1 ...)) (+ 1 100))
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...)))
        (find-val  pred (cdr (cdr '( 1 ...))) (+ 1 100))))
```

Neste momento precisamos calcular o valor de *(cdr '(1 ...))*
Lembrando que:
… ~ (interval (+ 1 1) 100)

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
```

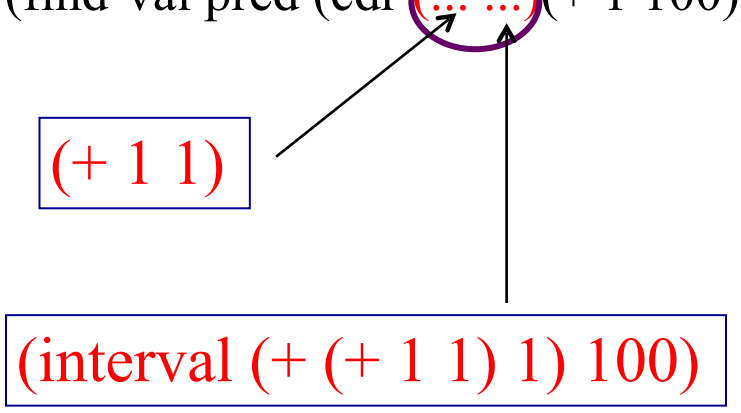
Avaliação por demanda no exemplo

(if (null? (cdr '(1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '(1 ...))
        (find-val pred (cdr (cdr '(1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (cons (+ 1 1) (interval (+ (+ 1 1) 1) 100)) (+1 100)
    (if (pred (car (cons (+ 1 1) (interval ...))) (car (cons (+ 1 1) (interval ...))
        (find-val pred (cdr (cons (+ 1 1) (interval ...)) (+ 1 100))))

(interval (+ 1 1) 100) ==> (cons (+ 1 1) (interval …))

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (... ...)) (+1 100)
    (if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...) (+ 1 100))))
```



Criada a célula, valores adiados (ambos precisam de processamento)

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (... ...)) (+1 100)
    (if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100))))
(if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100)))
(if (pred (2) (car (2 ...))
        (find-val pred (cdr (2 ...)(+ 1 100)))
```

Preciso acessar o *car* da célula, acho o valor do conteúdo (calculando (+ 1 1) e coloco o valor na célula

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (... ...)) (+1 100)
    (if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100))))
(if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100)))
(if (pred (2) (car (2 ...))
        (find-val pred (cdr (2 ...)(+ 1 100)))
```

Note que o valor do *cdr* continua suspenso

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (... ...)) (+1 100)
    (if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100))))
(if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100)))
(if (pred (2) (car (2 ...))
        (find-val pred (cdr (2 ...)(+ 1 100)))
(car (2 ..))
2
```

Avaliação por demanda no exemplo

```
(if (null? (cdr '( 1 ...)) (+ 1 100)
    (if (pred (car (cdr '(1 ...))) (car (cdr '( 1 ...))
        (find-val  pred (cdr (cdr '( 1 ...)) (+ 1 100))))
(if (null? (interval (+ 1 1) 100)) (+1 100)
    (if (pred (car (interval (+ 1 1) 100))) (car (interval (+ 1 1) 100))
        (find-val pred (cdr (interval (+ 1 1) 100)) (+ 1 100))))
(if (null? (... ...)) (+1 100)
    (if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100))))
(if (pred (car (... ...)) (car (... ...))
        (find-val pred (cdr (... ...)(+ 1 100)))
(if (pred (2) (car (2 ...))
        (find-val pred (cdr (2 ...)(+ 1 100)))
(car (2 ..))
2
```

NOTE QUE AGORA O TAMANHO DA LISTA É IRRELEVANTE, POIS SÓ CALCULAMOS O QUE ACESSAMOS

Versão final para avaliação por demanda

- Agora podemos fazer um “loop infinito”

```
(set ints-from (lambda (i)
      (cons i (ints-from (+ i 1)))))
(set fun-search-while (lambda ()   ;não precisamos do limite mais
     (find-val pred (ints-from 1) '())) ;valor de falha irrelevante
```

- com avaliação por demanda funciona pois apenas a parte utilizada do (ints-from 1) é efetivamente calculada.

Implementação: *Thunk* (“suspensão”)

- Como podemos ter expressões não calcuadas?
  - Quando uma expressão é encontrada em um argumento criamos uma estrutura espcial, ao invés de calcular a expressão
- Esta estrutura (*Thunk* ou “suspensão”) é semelhante a um fechamento sem argumentos
  - Expressão + ambiente
- Diferença está no uso
- NOTAÇÃO:
  - “...” na impressão
  - Para descrever <| expressão, ambiente |>
  - Ex: <| (interval 1 n), {n →100} |>

IMPORTANTE

- EFEITOS COLATERAIS SÃO INCONSISTENTES COM AVALIAÇÃO POR DEMANDA
  - Termporalidade: como valores só são calculados quando utilizados, se tivermos avaliação por demanda, restultados podem ser afetados pelo fluxo e uso → muito difícil de depurar código
- Eficiência:
  - Thunks são subsituídos pelos valor resultante assim que executados pela primeira vez.
- Únicas funções estritas (i.é. cujos argumentos são calculados imediatamente) são aquelas *inerentemente* estritas (e.g.: + * / - …)

Avaliando expressões

- Seja (e_0 e_1 .... e_n) uma expressão a ser avaliada no ambiente Env

1. Calcula e_0 → v_0
2. Constrói suspensões <|e_1 , Env|> ... <|e_n , Env |>
3. Depende do operador:
   - V_0 ~ *if* → calcular e_1 em Env
     - caso verdadeiro calcular <| e_2, Env |> e retornar resultado
     - Caso falso calcular <| e_3 , Env |> e retornar resultado
   - v_0 é operador estrito → calcular todos os e_i em Env e retornar resultado
   - v_0 ~ *cons* → retornar par (<|e_1 ,Env|> , <|e_2 ,Env|>)
   - v_0 ~ << (lambda (x_1 ... x_n) e), Env'>> → calcular e no ambiente {x_1 →<|e_1 , Env|>,...., x_n → <| e_n , Env |>}Env'
     - Nota: quando suspensão é calculada, ela é substituída pelo seu resultado no ambiente
   - e_0 ~ *car* ou *cdr* → se elemento for uma suspensão, calcula a suspensão primeiro e substitui por valor calculado, em seguida retornar o valor.

Como ficam os nossos programas scheme?

- Todos os programas sem efeitos colaterais vão retornara exatamente o mesmo valor, com tempo de execução potencialmente menor.
- Duas diferenças
  - Listas serão calculadas apenas na medida que foram acessadas
  - Suspensões serão impressas diferentemente

Exemplos

(<<(lambda(x) (+ x x), {}>>, (+ 1 1))

→ calcula (+ x x) em Env={x → <| (+ 1 1), {}|>}

→ como + é estrito, calcula (+ 1 1) em {} e substitui no ambiente, resultando no valor 2

→ agora temos (+ x x) calculado em {x → 2}

4

Exemplos

→ (set x (mapcar +1 ‘(2 3)))

Exemplos

→ (set x (mapcar +1 ‘(2 3)))

Lembrando:

```
(set mapcar (lambda (f l)
    (if (null? l) ‘()
        (cons (f (car l))
              (mapcar f (cdr l))))))
```

Exemplos

→ (set x (mapcar +1 ‘(2 3)))

(... ...)

Lembrando:

```
(set mapcar (lambda (f l)
    (if (null? l) ‘()
        (cons (f (car l))
              (mapcar f (cdr l))))))
```

Exemplos

→ (set x (mapcar +1 ‘(2 3)))

(... ...)

→ (cadr x)

4

→ x

(… 4 …)

→ (car x)

3

→ x

(3 4 ...)

→ (cddr x)

‘()

→ X

→ ‘(3 4)

Exemplos: truque imprimir listas

- Finitas

```
(set force (lambda (x)
      (if (list? x)
          (if (force (car x))
              (force (cdr x))
              '())
           true)
)
```

- Infintas:

```
(set (first-n (lambda (l n)
           (if (null? l) '()
               (if (equal? (car l) 0)
                   (first-n (- n 1) (cdr l))
                   (first-n (- n 1) (cdr l))))))
```

Exemplos: truque imprimir listas

- Finitas

```
(set force (lambda (x)
      (if (list? x)
          (if (force (car x))
              (force (cdr x))
              '())
           true)
)
```

- Infintas:

```
(set (first-n (lambda (l n)
           (if (null? l) '()
               (if (equal? (car l) 0)
                   (first-n (- n 1) (cdr l))
                   (first-n (- n 1) (cdr l))))))
```

Esta opção NUNCA é executada (porque?)

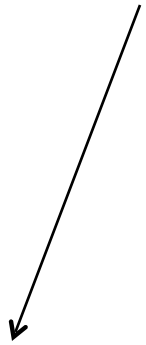
Exemplos: truque imprimir listas

- Finitas

```
(set force (lambda (x)
      (if (list? x)
          (if (force (car x))
              (force (cdr x))
              '())
          true)
)
```

- Infintas:

```
(set (first-n (lambda (n l)
          (if (= n 0)
              '()
              (if (null? l) '()
               (if (equal? (car l) 0)
                     (first-n (- n 1) (cdr l))
                    (first-n (- n 1) (cdr l)))))))
```



Retorna sempre nulo e ambas as alternativas para o *if* são iguais, porque?

Exemplos

→ (set inteiros (ints-from1))

(… …)

→ (set x (mapcar +1 inteiros))

(… …)

→ (caddr x)

4

→ x

(… … 4 …)

→ (first x 4)

‘()

→ x

(2 3 4 5 …)

Exemplos

→ (set alguns inteiros (interval 1 6))

(… …)

→ (set x (mapcar +1 alguns-inteiros)))

(… …)

→ (caddr x)

4

→ x

(… … 4 …)

→ (force x)

true

→ x

(2 3 4 5 6 7)

Exemplo: satisfatibilidade de expr. bool

- dada uma representação de uma expressão booleana, verificar se é possível encontrar conjunto de valores para as variáveis que torne a expressão verdadeira
- ex: not (p or ((not p or q) and (not p or not q)))
- i.e.´(**not** (**or** p
  (**and** (**or** (**not** p) q)
  (**or** (**not** p) (**not** q)))))
- para sabermos, basta tentarmos todas as possibilidades, isto é geramos todos os possíveis conjuntos de valores, e testarmos um a um
- (set satifatível? (lambda (e)
  (if (acha-valor-verdade e (lista-de-valores e))
  ´Satisfatível
  ´Não-satisfatível)))

Exemplo: parte2

- para simplificarmos, para cada conjunto de valores, vamos apenas listar as variáveis com valor “verdadeiro”. Assim, para acharmos todos os conjuntos de valores das variáveis de uma expressao, primeiro extraímos as variáveis, e então calculamos todos os sub-conjuntos desta lista:

```
(set lista-de-valores (lambda (e)
   (gera-sub-conjuntos (variaveis e))
(set variáveis (lambda (e)
    (if (symbol? e) (cons e ´())
        (if (= (car e) ´not) (variáveis (cadr e))
            (uniao (variáveis (cadr e))
                   (variáveis (caddr e)))))))
```

Exemplo: parte3

- Gerando sub-conjuntos:

```
(set gera-sub-conjuntos (lambda (l)
    (if (null? (cdr l)) (list (car l) ´()); conjunto c/ 1 elem.
        (mapaddx (car l) (gera-sub-conjuntos (cdr l))))))
(set mapaddx (lambda x lista)
    ;adiciona x a cada valor de lista e adiciona ao final de lista
    (append lista
            (mapcar (lambda (y) (cons x y)) lista)))
```

Exemplo: parte 4

- agora a função que verifica se expressão e satisfatível por algum elemento de uma lista de atribuições às suas variáveis

```
(set acha-valor-verdade (lambda (e listas-variáveis-verdadeiras)
  (if (null? listas-variáveis-verdadeiras) false; se acabaram as listas...
    (if (eval-expr-booleana (car lista-variáveis-verdadeiras) e) true
      (acha-valor-verdade e (cdr listas-variáveis-verdadeiras))))))
```

Exemplo: final

```
(set eval-expr-booleana (lambda (e variáveis-verdadeiras)
  (if (symbol? e)
     (member? e variáveis-verdadeiras)
     (if (equal? (car e) ´not)
        (not (eval-expr-booleana (cadr e)
                                 variáveis-verdadeiras))
        (if  (equal?  (car e) ´and)
           (and (eval-expr-booleana (car e) variáveis-verdadeiras)
                (eval-expr-booleana (cadr e) variáveis-verdadeiras))
           (or (eval-expr-booleana (car e) variáveis-verdadeiras)
                (eval-expr-booleana (cadr e) variáveis-verdadeiras))
        )))))
```

Exemplo: alcançabilidade

- dado um conjunto de vetores indicando os deslocamentos possíveis e dado um ponto no espaço, queremos descobrir de o ponto é alcançável utlizando os deslocamentos listados.
- se pudermos separar geração do espaço de busca da procura, basta
  1. gerarmos uma lista (infinita) de todos os pontos alcançáveis a partir da origem que não ultrapassem o ponto desejado,
  2. procurarmos neste espaço.

```
(set gera-caminhos (lambda (ponto-inicial vetores)
      (cons ponto-inicial
            (mapcar (lambda (r) (gera-caminhos r vetores))
                  (mapcar (lambda (q) (soma-pontos  ponto-inicial q))
                        vetores)))))
(set soma-pontos (lambda(p1 p2)
   (list2 (+ (car p1) (car p2)) (+ (cadr p1)(cadr p2)))))
```

Exemplo: parte2

- como separamos a gerção do espaço de busca da procura, podemos agora fácilmente implementaro diferentes estratégias de busca neste espaço
- busca por aprofundamento

```
(set busca-aprof (lambda (termo sucesso? para?)
     (if (sucesso? (car termo)) true
        (if (para? (car termo)) false ; para para este ramo
            (busca-aprof* (cdr l) sucesso? para?))))))
(set busca-aprof* (lambda (lista sucesso? para?)
   (if (null? lista) false
      (if (busca-aprof (car lista) sucesso? para?) true
          (busca-aprof* (cdr lista) sucesso? para?)))))
```

Exemplo: parte3

- busca por espalhamento:

```
(set busca-espalha (lambda (termo sucesso? para?)
  (busca-espalha-c-fila(enfileira termo (fila-vazia))
                                  sucesso? para?)))
(set busca-espalha-c-fila(lambda (fila sucesso? para?)
  (if (empty? fila) false
     (if (sucesso? (car (frente fila ))) true
        (if (para? (car (frente fila)))
           (busca-espalha-c-fila(tira-frente fila) sucesso? para?)
           (busca-espalha-c-fila (enfileira* (tira-frente pilha) (topo pilha))
                                          sucesso? para?))))))
(set empilha* (lambda (filalista)
 (if (null? lista) fila(enfileira* (enfileira (car lista) pilha) (cdr lista))))))
```

Exemplo: final

- agora basta escolher a estratégia:

```
(set alcanca-ponto-aprof (lambda (ponto pontos-atingiveis)
  (busca-aprof  pontos-atingiveis
                (lambda (p1) (=ponto p1 ponto))
                (lambda (p1 (passou p1 ponto)))))
(set alcanca-ponto-espalha (lambda (ponto pontos-atingiveis)
  (busca-espalha  pontos-atingiveis
                 (lambda (p1) (=ponto p1 ponto))
                 (lambda (p1 (passou p1 ponto)))))
```

- para gerar os caminhos a partir de *ponto-inicial* e *vetores*:

```
(set caminhos (gera-caminhos ponto-inicial vetores))
```

Exemplo: raiz quadrada (método de Newton-Raphson)

- Algoritmo:

- SASL

```
(define criterio (lambda (epsilon)
        (lambda (l) (< (absolute-value (- (cadr l) (car l)) epsison))))
(define next-val (lambda (n xi)
         (/ (+ xi (/ n xi)) 2)))
(define x-list (lambda (n xi)
      (cons xi  (xlist (next-val n xi)))))
(define mk-xlist (lambda (n) (x-list n 1)))
```

Exemplo: raiz quadrada (método de Newton-Raphson)

- Algoritmo:

- SASL(cont)

```
(define find-list (lambda (pred extract l)
        (if (null? l) false
            (if (pred l) (extract l)
                (find-list (pred extract (cdr l)))))))
(set raiz-nr (lambda(n)
      (find-list (criterio 0.000001) cadr (mk-xlist n))))
```

Exemplo: raiz quadrada (método de Newton-Raphson)

- Algoritmo:

- SASL(cont)

```
(define find-list (lambda (pred extract l)
        (if (null? l) false
            (if (pred l) (extract l)
                (find-list (pred extract (cdr l)))))))
(set raiz-nr (lambda(n)
        (find-list (criterio 0.000001) cadr (mk-xlist n))))
```

Aspecto interessante: Separação da geração da lista e da condição de parada

Exemplo: geração de números primos

- (primos <= N) → gera todos os primos menores ou iguais a um inteiro n
  - Idéia básica: a)gerar lista com inteiros de 2 a N, b) percorrer lista removendo os múltiplos

```
(set remove-multiplos (lambda (n l)
        (if (null? l) l
            (if (divide n (car l))
                (remove-multiplos n (cdr l))
                (cons (car l) (remove-multiplos n (cdr l)))))))
(set peneira (lambda (l)
      (if (null? l) l
          (cons (car l) (peneira (remove-multiplos (car l) (cdr l)))))))
(set (primos<=  (lambda (n) (peneira (interval 2 n))))
```

Exemplo: geração de números primos

- Como fazer um código para os primeiros n primos?
- Em scheme precisaríamos reescrever algoritmos (não sabemos quantos inteiros precisamos na lista.
- Em SASL

(set primos (peneira (ints-from 2))) ;lista infinita com **todos** os primos

(set primeiros-n-primos (lambda (n)

(first-n n primos))

➔ ESTILO FUNCIONAL "PURO" (com avaliação por demanda) PROPICIA MAIOR REUTILIZAÇÃO.

Implementação de definições recursivas

- Uma técnica de codificação muito útil a ser usada em linguagens tipo SASL é a de “esquemas de recursão”
- Suponha uma sequência infinita $x_1, x_2, \ldots$ Definida por uma recursão do tipo, para uma função *f* qualquer:

$x_0 = <$ algum valor$>$

$x_{i+1} = f(x_i)$

- Podemos gerar esta lista muito fácilmente com a expressão

(set x (cons x_o (mapcar f x)))

Podemos generalizar este princípio de várias maneiras

- Podemos estender a idéia acima para facilitar geração de sequencias de valores do tipo

x_0 = < algum valor>

$x_{i+1} = f(x_i, i+1)$

- Vamos ver como usar esta idéia

```
(set mapcar2 (lambda (f lista1 lista2)
      (cons (f (car lista1) (car lista2))
            (mapcar2 f (cdr lista1)(cdr lista2)))))
```

Listas infinitas são úteis

- Podemos estender a idéia acima para facilitar geração de sequencias de valores do tipo

x_0 = < algum valor>

$X_{i+1} = f(x_i, i+1)$

- Vamos ver como usar esta idéia

```
(set mapcar2 (lambda (f lista1 lista2)
      (cons (f (car lista1) (car lista2))
            (mapcar2 f (cdr lista1)(cdr lista2)))))
```

Para facilitar supus listas infinitas

Listas infinitas são úteis

- Podemos estender a idéia acima para facilitar geração de sequencias de valores do tipo

x_0 = < algum valor>

$X_{i+1} = f(x_i, i+1)$

- Vamos ver como usar esta idéia

```
(set mapcar2 (lambda (f lista1 lista2)
      (cons (f (car lista1) (car lista2))
            (mapcar2 f (cdr lista1)(cdr lista2)))))
(inteiros (cons 1 (mapcar +1 inteiros)))
(set fatoriais (cons 1 (mapcar2 * fatoriais inteiros)))
```

Implementando listas infinitas em Scheme: Streams

- Em Scheme podemos implementar listas infintas com *Streams* estruturas como listas mas que o Scheme avalia por demanda.
- Sua operação é exatamente como com listas

```
(require racket/stream)
(define ints-from (lambda (x) (stream-cons x (ints-from (+ x 1)))))

(set find-val (lambda (pred my-stream valor-falha)
      (if (stream-empty? my-stream)
          valor-falha
          (if (pred (stream-first my-stream) ) (stream-first my-stream)
              (find-val pred (stream-rest my-stream) valor-falha)))))
```

Implementando listas infinitas em Scheme: fechamentos

- Podemos também simular suspensões com o uso de fechamentos
- Porém o código fica muito mais explícito e precisa ser executado todas as vezes que acessamos o valor.

```
>(set ints-de (lambda (x)
    (cons x (lambda () (ints-de (+ x 1))))))
>(set inteiros (ints-de 0))
(0 . #<procedure>)
```

- Vamos precisar de funções auxiliares

```
(define rest (lambda (lista-infita) ((cdr lista-infinta))))
        ;o cdr é um fechamento assim precisamos "aplicá-lo"
```

Implementando listas infinitas em Scheme: fechamentos

```
(set find-val (lambda (pred my-inf-list valor-falha)
    (if (empty? my-inf-list)
        valor-falha
        (if (pred (stream-first my-stream) ) (stream-first my-stream)
            (find-val pred (rest my-stream) valor-falha)))))
```