Notas de Aula — MAC0316
Baseadas no livro texto

Gubi

27 de novembro de 2019

Sumário

1 Introdução

A variante do *racket* que usaremos é a `plai-typed`, coloque a linha abaixo no início de todos os programas.

```
#lang plai-typed
```

Além de ser tipada, permite a definição de tipos, o que facilita bastante o desenvolvimento.
A construção com ':' faz a verificação de tipo:
(define-type....)
(test val result) testa se *val* tem o resultado esperado (*result*).

type-case — verifica o subtipo com case.
A definição de um tipo é análoga à declaração de uma classe abstrata com diversas subclasses:

```
(define-type CLASSE_ABSTRATA
   [SUBCLASSE1 (CAMPO : tipo)]
   [SUBCLASSE2 (CAMPO : tipo)] ...
)
```

2 Primeira Linguagem

Começamos com uma linguagem simples, que apenas trata aritmética.

```
(define-type ArithC
  [numC  (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])
```

Esta linguagem inicialmente possui apenas soma e multiplicação, mas vamos crescê-la aos poucos.

2.1 Parser

O livro usa o *read* do scheme/racket: listas aninhadas foram a árvore.
Brincaremos com tradutores.
No plai-typed, o read lê um texto e retorna uma *s-expression*, uma lista "quoted" com os elementos.

É possível usar *casting*:

```
        s-exp->list
        symbol->string
```

read não produz tipos, apenas *s-expressions*, o *parser* não é essencial para trabalhar, é apenas o *front-end* do compilador. Apesar do livro fazer apenas um *parser*, eu construirei quase todos os necessários para testes. O restante fica como exercício.

Exemplo de um parser simples:

```
(define-type ArithC
  [numC  (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)])

(define (parse [s : s-expression]) : ArithC
  (cond
    [(s-exp-number? s) (numC (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusC (parse (second sl)) (parse (third sl)))]
         [(*) (multC (parse (second sl)) (parse (third sl)))]
         [else (error 'parse "invalid␣list␣input")]))]
    [else (error 'parse "invalid␣input")]))
```

2.2 Interpretador

Primeiro um template:

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) ... ]
    [multC (l r) ... ])
```

Para completar, troque o ... pela semântica correspondente.

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ l r)]
    [multC (l r) (* l r)])
```

Não funciona, porque l e r também não são números!

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]
```

2.3 Subtração

Para incluir a operação de subtração, trocamos o sinal e somamos. Construímos uma nova árvore, `ArithS` e depois transformamos na `ArithC`. `ArithC` contém nossas primitivas, operações de `ArithS` são composições de operações de `ArithC`.

```
(define-type ArithS
  [numS    (n : number)]
  [plusS   (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [multS   (l : ArithS) (r : ArithS)])
```

O `bminusS` será tratado com açúcar sintático, fará a seguinte transformação:

```
 (- a b) → (+ a (* -1 b))
 ArithS  → ArithC
```

Precisamos de uma função "desugar", que arranca o açúcar

```
(define (desugar [as : ArithS]) : ArithC  ; recebe ArithS e devolve
                                          ; ArithC
  (type-case ArithS as
    [numS    (n)   (numC n)]              ; conversão direta
    [plusS   (l r) (plusC (desugar l)     ; todas as subárvores devem
                          (desugar r))]   ; ter o açúcar retirado
    [multS   (l r) (multC (desugar l)
                          (desugar r))]
    [bminusS (l r) (plusC (desugar l)     ; aqui está a transformação
                     (multC (numC -1) (desugar r)))]
    ))
```

O interpretador continua o mesmo, pois no final temos `ArithC`.

O parser muda para gerar `ArithS`.

```
(define (parse [s : s-expression]) : ArithS
  (cond
    [(s-exp-number? s) (numS (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusS (parse (second sl)) (parse (third sl)))]
         [(*) (multS (parse (second sl)) (parse (third sl)))]
         ; agora temos o '-'
         [(-) (bminusS (parse (second sl)) (parse (third sl)))]
         [else (error 'parse "invalid␣list␣input")]))]
    [else (error 'parse "invalid␣input")]))
```

2.4 Menos unário (negação)

Colocar `uminusS` (- unário) é um pouquinho mais chato, mas a estrutura de `ArithS` é praticamente a mesma.

```
(define-type ArithS
  [numS    (n : number)]
  [plusS   (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [uminusS (e : ArithS)]
  [multS   (l : ArithS) (r : ArithS)])
```

`uminusS` tem apenas um argumento. O problema aparece no `desugar`.

Uma possibilidade é expandir `(uminusS e)` para `(bminus (numC 0) e)`. Isto faz parte do `desugar`, que pode ser recursivo, no entanto....

```
(define (desugar [as : ArithS]) : ArithC
  (type-case ArithS as
    [numS    (n)   (numC n)]
    [plusS   (l r) (plusC (desugar l)
                          (desugar r))]
    [multS   (l r) (multC (desugar l)
                          (desugar r))]
    [bminusS (l r) (plusC (desugar l)
                          (multC (numC -1) (desugar r)))]
    [uminusS (e)   (desugar (bminusS (numC 0) e)] ; <------ problema
    ))
```

Usar a entrada na expansão diretamente torna `uminusS` um macro. Isso não é problema. O problema é que a recursão se aplica em uma função do argumento, e não em uma parte dele. Esta é uma recursão generativa e perigosa: não há garantias de que o parâmetro seja modificado em momento algum e a recursão pode ser infinita.

Podemos usar o `desugar` diretamente no argumento, aí tudo bem.

```
    [uminusS (e)   (bminusS (numC 0) (desugar e))]
```

Quando desugar for tratar "e", ele será decomposto. Só que não funciona... `desugar` retorna `ArithC` e não `ArithS`!

Ainda ocorre um outro problema: `uminusS` depende da implementação de `bminusS`, seria melhor depender apenas das primitivas.

```
(define (desugar [as : ArithS]) : ArithC
  (type-case ArithS as
    [numS    (n)   (numC n)]
    [plusS   (l r) (plusC (desugar l)
                          (desugar r))]
    [multS   (l r) (multC (desugar l)
                          (desugar r))]
    [bminusS (l r) (plusC (desugar l)
                          (multC (numC -1) (desugar r)))]
    [uminusS (e)   (multC (numC -1) (desugar e))]
    ))
```

2.5 Condicionais (if)

Considerando falso como 0 e verdadeiro como todo o resto.

As transformações são triviais, já que estamos tratando como primitiva

```
(define-type ArithC
  [numC (n : number)]
  [plusC (l : ArithC) (r : ArithC)]
  [multC (l : ArithC) (r : ArithC)]
  [ifC   (condição : ArithC) (sim : ArithC) (não : ArithC)]
  )
```

O `ArithS` também muda.

```
; inclui ifS
(define-type ArithS
  [numS    (n : number)]
  [plusS   (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [uminusS (e : ArithS)]
  [multS   (l : ArithS) (r : ArithS)]
  [ifS     (c : ArithS) (s : ArithS) (n : ArithS)]
  )
```

Idem o `desugar`.

```
(define (desugar [as : ArithS]) : ArithC
  (type-case ArithS as
    [numS    (n)   (numC n)]
    [plusS   (l r) (plusC (desugar l) (desugar r))]
    [multS   (l r) (multC (desugar l) (desugar r))]
    [bminusS (l r) (plusC (desugar l) (multC (numC -1) (desugar r)))]
    [uminusS (e)   (multC (numC -1) (desugar e))]
    [ifS     (c s n) (ifC (desugar c) (desugar s) (desugar n))]
    ))
```

O interpretador precisa cuidar do `ifC`, mas é bem simples. A única curiosidade é inverter as condições, porque eu resolvi testar para zero...

```
(define (interp [a : ArithC]) : number
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]
    [ifC (c s n) (if (zero? (interp c))  ;<== aqui
                     (interp n) (interp s))]
    ))
```

O `parser` também segue direto.

```
(define (parse [s : s-expression]) : ArithS
  (cond
    [(s-exp-number? s) (numS (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusS (parse (second sl)) (parse (third sl)))]
         [(*) (multS (parse (second sl)) (parse (third sl)))]
         [(-) (bminusS (parse (second sl)) (parse (third sl)))]
         [(~) (uminusS (parse (second sl)))]
         [(if) (ifS (parse (second sl))
                    (parse (third sl)) (parse (fourth sl)))]
         [else (error 'parse "invalid␣list␣input")]))]
    [else (error 'parse "invalid␣input")]))

(define (interpS [a : ArithS]) (interp (desugar a)))

(parse '(if (- 3 2) 42 (+ 5 8)))
(interpS (parse '(if (- 3 2) 42 (+ 5 8))))
(interpS (parse '(if (- 3 3) 42 (+ 5 8))))
```

3 Segunda Linguagem

3.1 Funções

Existem alguns conceitos fundamentais na definição e uso de funções:

- Declaração — qual o contrato e o que ela faz.
- Parâmetros — não são variáveis, apenas associação de símbolos a valores.
- Aplicação — chamada da função, que é uma operação por si mesma.

Funções serão definidas separadamente, em um novo tipo:

```
(define-type FunDefC
[fdC (name : symbol) (arg : symbol) (body : ExprC)])
```

Alguns exemplos de definições:

```
(fdC 'double 'x (plusC (idC 'x) (idC 'x)))
(fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))
(fdC 'const5 '_ (numC 5))
```

As equivalentes em *racket*:

```
(define (double x) (+ x x ))
(define (quadruple x) (double (double x)))
(define (const5 _) 5)
```

3.2 Nova estrutura

Vamos usar um novo nome para a estrutura, mudamos qualitativamente de linguagem. `ExprC` inclui o tratamento de funções.

As funções ficarão definidas em uma estrutura a parte e, por enquanto, terão apenas um argumento. Para `ExprC` precisamos de duas novas entradas: identificadores (nomes de parâmetros) e aplicação (ou chamada) de função.

O tipo de definição de função é bem simples:

```
; definição de função com 1 argumento
(define-type FunDefC
  [fdC (name : symbol) (arg : symbol) (body : ExprC)]
  )
```

Veja que o nome e o argumento são símbolos. O corpo é, naturalmente, uma `ExprC`.

A `ExprC` fica:

```
; Novo tipo, com funções.
; Precisamos de duas novas entradas:
;     - identificador, para argumentos
;     - aplicação da função
(define-type ExprC
  [numC (n : number)]
  [idC  (s : symbol)]  ; identificador
  [appC (fun : symbol) (arg : ExprC)] ; aplicação, com o nome da função
                                      ; e o valor do argumento
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [ifC   (condição : ExprC) (sim : ExprC) (não : ExprC)]
  )
```

3.3 Açúcar sintático?

Se quisermos manter o açúcar sintático, precisamos do tipo `ExprS` (equivalente ao `ArithS`).

```
(define-type ExprS
  [numS    (n : number)]
  [idS     (s : symbol)]
  [appS    (fun : symbol) (arg : ExprS)]
  [plusS   (l : ExprS) (r : ExprS)]
  [bminusS (l : ExprS) (r : ExprS)]
  [uminusS (e : ExprS)]
  [multS   (l : ExprS) (r : ExprS)]
  [ifS     (c : ExprS) (s : ExprS) (n : ExprS)]
  )
```

A função `desugar` deve reconhecer os novos elementos.

```
(define (desugar [as : ExprS]) : ExprC
  (type-case ExprS as
    [numS    (n) (numC n)]
    [idS     (s) (idC s)] ; este é fácil
    [appS    (fun arg) (appC (fun (desugar arg)))] ; fun é um symbol,
                                                   ; não precisa de
                                                   ; desugar
    [plusS   (l r) (plusC (desugar l) (desugar r))]
    [multS   (l r) (multC (desugar l) (desugar r))]
    [bminusS (l r) (plusC (desugar l) (multC (numC -1) (desugar r)))]
    [uminusS (e)   (multC (numC -1) (desugar e))]
    [ifS     (c s n) (ifC (desugar c) (desugar s) (desugar n))]
    ))
```

3.4 Interpretador

O interpretador precisa tratar de substituir o parâmetro pelo valor!
Além disso, precisa receber uma lista com as definições de funções. Na aplicação, o nome da função deverá ser procurado na lista. Isto será feito por `get-fundef`:

```
get-fundef : symbol * (listof FunDefC) -> FunDefC
```

A substituição dos símbolos pelas expressões será feita por `subst`:

```
subst : ExprC * symbol * ExprC -> ExprC
```

3.4.1 Substituição

Os três parâmetros de `subst` significam o valor (primeiro argumento) que deverá substituir o símbolo (segundo argumento) na expressão (terceiro argumento): `(subst` *`VALOR ISSO EM`*`)`. Ou seja, substitui o símbolo *ISSO* por *VALOR* na expressão *EM*.

```
(define (subst [valor : ExprC] [isso : symbol] [em : ExprC]) : ExprC
  (type-case ExprC em
    [numC (n) em]    ; nada a substituir, repassa
    [idC (s) (cond   ; poderia ser 'if', mas existem coisas no futuro...
               [(symbol=? s isso) valor] ; símbolo, troque
               [else em])] ; deixa quieto
    [appC  (f a) (appC f (subst valor isso a))] ; chamada de função
                                                ; arruma o argumento
    [plusC (l r) (plusC (subst valor isso l) (subst valor isso r))]
    [multC (l r) (multC (subst valor isso l) (subst valor isso r))]
    [ifC (c s n) (ifC   (subst valor isso c)
                        (subst valor isso s) (subst valor isso n))]
  ))
```

3.4.2 Agora o interpretador!

```
(define (interp [a : ExprC] [fds : (listof FunDefC)]) : number
  (type-case ExprC a
    [numC (n) n]
    ; Aplicação de função é que precisa de subst
    [appC (f a)
          (local ([define fd (get-fundef f fds)]) ; pega a def. em fd
            (interp (subst a          ; interpreta o resultado de subst
                           (fdC-arg fd)
                           (fdC-body fd)
                           )
                    fds))]
    ; Não devem sobrar identificadores livres na expressão
    [idC (_) (error 'interp "não deveria encontrar isso!")]
    [plusC (l r) (+ (interp l fds) (interp r fds))]
    [multC (l r) (* (interp l fds) (interp r fds))]
    [ifC (c s n) (if (zero? (interp c fds)) (interp n fds) (interp s
       fds))]
    ))
```

Falta definir `get-fundef`. É apenas uma busca linear.

```
(define (get-fundef [n : symbol] [fds : (listof FunDefC)]) : FunDefC
  (cond
    [(empty? fds) (error 'get-fundef
                         "referência para função não definida")]
    [(cons? fds) (cond
                   [(equal? n (fdC-name (first fds))) ; achou!
                                          (first fds)]
                   [else (get-fundef n (rest fds))] ; procura no resto
                   )]
   ))
```

3.5 Parser

O *parser* tem uma modificação mínima, basta colocar algo para identificar a chamada.

```
; o parser precisa tratar de chamadas
(define (parse [s : s-expression]) : ExprS
  (cond
    [(s-exp-number? s) (numS (s-exp->number s))]
    [(s-exp-list? s)
     (let ([sl (s-exp->list s)])
       (case (s-exp->symbol (first sl))
         [(+) (plusS (parse (second sl)) (parse (third sl)))]
         [(*) (multS (parse (second sl)) (parse (third sl)))]
         [(-) (bminusS (parse (second sl)) (parse (third sl)))]
         [(~) (uminusS (parse (second sl)))]
         [(call) (appS (s-exp->symbol (second sl)) (parse (third sl)))]
         [(if) (ifS (parse (second sl)) (parse (third sl)) (parse (
           fourth sl)))]
         [else (error 'parse "invalid␣list␣input")]))]
    [else (error 'parse "invalid␣input")]))
```

3.6 Testes

Nesta linguagem, as funções são definidas à parte. Vejamos um exemplo com `dobro`, `quadrado` e `fatorial`:

```
(define biblioteca
   (list
      [fdC 'dobro 'x (plusC (idC 'x) (idC 'x))]
      [fdC 'quadrado 'y (multC (idC 'y) (idC 'y))]
      [fdC 'fatorial 'n
            (ifC  (idC 'n)
                  (multC (appC 'fatorial (plusC (idC 'n) (numC -1)))
                                 (idC 'n))
                  (numC 1))]
      [fdC 'narciso  'narciso (multC (idC 'narciso) (numC 1000))]))

(interp (desugar (parse '(+ -1400 (call fatorial 7)))) biblioteca)

(test
  (interp (desugar (parse '(call narciso (call fatorial 7))))
    biblioteca)
  5040000)
```

A função `narciso` é curiosa. O argumento tem o mesmo nome da função! Tente entender por que funciona, ou melhor, como acontece de estarem em *namespace*s diferentes.

O que acontece neste código?

```
(interp (desugar (parse '(call dobro (+ 5 1)))))
```

Quantas vezes a operação `(+ 5 1)` é feita? Como e por quê mudar isso? Valeria a pena deixar como está?

4 Ambientes (*Environments*)

Esta forma de substituição faz com que as funções sejam percorridas duas vezes a cada chamada: uma para trocar os argumentos e outra para a interpretação propriamente dita.

A troca do código é ruim por vários motivos. O programa não deveria ser reescrito o tempo todo quando executado.

Para evitar a troca constante, precisamos de duas coisas:

- Uma "tabela de símbolos" onde cada identificador possa ser consultado e seu valor retornado.
- Troca preguiçosa, o identificador só é trocado quando necessário.

O único ponto do nosso sistema que precisa ser trocado é o interpretador! Ele passa a ter um novo argumento com a lista de associações: o *environment* (ambiente).

O interpretador fará a busca e troca em tempo de execução apenas. Cada associação é conhecida como *binding*.

Resta saber a forma da associação. Vamos ligar nomes a quê? Para interpretação imediata, precisamos ligar a respostas (*numbers*), não a expressões.

```
(define-type Binding
      [bind (name : symbol) (val : number)])
```

Uma lista de `Binding`s forma o *environment*. Precisaremos de uma lista vazia (*eMpTy*) e uma função para estender o *environment*. Estas últimas podem ser mapeadas diretamente no *racket*:

```
(define-type-alias Env (listof Binding))
(define mt-env empty)
(define extend-env cons)
```

Com o *environment*, não usaremos mais o `subst`. As trocas são feitas diretamente a partir das associações. `interp` ganha mais um argumento: o *environment corrente*.

```
(define (interp [a : ExprC]
                [env : Env]
                [fds : (listof FunDefC)]) : number
```

Todas as chamadas recursivas devem agora incluir este argumento. Um identificador passa a ser buscado e não provoca mais um erro imediato:

```
    ; um identificador deve ser trocado pela sua associação
    [idC (n) (lookup n env)]
```

Naturalmente precisaremos definir `lookup`.

A aplicação de funções fica muito mais simples. Não há substituição, tudo o que precisa ser feito é incluir uma nova associação em `env` (um `cons` da nova associação com a lista antiga):

```
    ; aplicação de função ainda precisa procurá-la,
    ; mas não faz mais de substituições
    ; o que precisa ser feito é apenas mais uma associação
    [appC (f a)
          (local ([define fd (get-fundef f fds)])
            (interp (fdC-body fd)               ; expressão
                    (extend-env
                        ; nova associação
                        (bind (fdC-arg fd) (interp a env fds))
                        env)
                    fds))]
```

Finalmente, `lookup`, muito parecida com `get-fundef`:

```
; lookup
(define (lookup [for : symbol] [env : Env]) : number
       (cond
            [(empty? env) (error 'lookup "name not found")]
            [else (cond
                  [(symbol=? for (bind-name (first env)))
                                 (bind-val (first env))]
                  [else (lookup for (rest env))])]))
```

4.1 Problemas....

O que acontece neste caso?

```
(interp (appC 'f1 (numC 3)) mt-env
        (list
            [fdC 'f1 'x (appC 'f2 (numC 4))]
            [fdC 'f2 'y (plusC (idC 'x) (idC 'y))]))
```

E em uma chamada com `fatorial`, definida na `biblioteca` acima?

O problema é que o *environment* mantém todas as associações desde o início da interpretação. Nenhuma associação é descartada, isto gera escopo dinâmico, um símbolo definido em algum ponto do programa é valido em todos os outros dali para frente. Lembre do `local` em *perl*.

Para saber o valor de um identificador com escopo dinâmico, é preciso não apenas olhar o programa inteiro, com o todo o histórico de execução!!!

Pense neste exemplo do livro (em *Scheme*):

```
(define y 1)
(define f (let ((z y)) (lambda (x) (+ x y z))))
(define y 2)
```

4.2 Vantagens e desvantagens de explicitar o ambiente

Explicitar o *environment* pode ser interessante se quisermos definir constantes da mesma forma que definimos funções. Quanto o programa começa a ser interpretado, algumas definições já estão disponíveis.

5 Funções como valores

Nossa linguagem trabalha com expressões (`ExprC`), mas os resultados são apenas números. Todos os operadores se aplicam a números ou expressões que representam números.

Do jeito em que está, não é possível definir funções dentro da linguagem. Precisamos criá-las separadamente e oferecer uma biblioteca (`(listof FunDefC)`). O modo de resolver isso é tornar funções valores válidos. É claro que algumas operações não fazem sentido, como somar ou multiplicar, mas é possível aplicar funções a expressões[1].

Uma primeira versão parece trivial, basta incluir a declaração de funções em `ExprC`, adicionando `fdC` e modificando `appC`.

```
(define-type ExprC
  [numC (n : number)]
  [idC  (s : symbol)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [fdC (name : symbol) (arg : symbol) (body : ExprC)]
            ; declaração faz parte da expressão
  [appC (fun : ExprC) (arg : ExprC)] ; a aplicação recebe uma função
  [ifC   (condição : ExprC) (sim : ExprC) (não : ExprC)]
  )
```

Não precisamos mais da lista de definições, pois `appC` agora recebe uma expressão e não mais um identificador. As funções são declaradas e usadas (funções imediatas, lembre de um λ).

Mas isso quebra o interpretador! Agora teremos função como valor e não apenas `number` como resultado!

5.1 Arrumando o interpretador

Vamos criar um tipo de retorno do interpretador.

```
(define-type Value
      [numV (n : number)]
      [funV (name : symbol) (arg : symbol) (body : ExprC)])
```

Agora ele poderá retornar tanto `numV` ou `funV`. As mudanças correspondentes devem ser feitas em `lookup` e `Binding`, que não podem mais retornar `number`.

[1] Não se preocupe, não trataremos de espaços de Hilbert nesta disciplina, hehe

Os resultados das operações aritméticas devem ser adaptados para devolver `numV`, também. Precisamos de novos operadores aritméticos:

```
(define (num+ [l : Value] [r : Value]) : Value
    (cond
        [(and (numV? l) (numV? r))
             (numV (+ (numV-n l) (numV-n r)))]
        [else
             (error 'num+ "Um␣dos␣argumentos␣não␣é␣número")]))

(define (num* [l : Value] [r : Value]) : Value
    (cond
        [(and (numV? l) (numV? r))
             (numV (* (numV-n l) (numV-n r)))]
        [else
             (error 'num* "Um␣dos␣argumentos␣não␣é␣número")]))
```

Para interpretar números e funções, basta convertê-los para o tipo `Value`:

```
; Não precisamos mais da lista de funções, pelo menos por enquanto....
(define (interp [a : ExprC] [env : Env]) : Value
  (type-case ExprC a
    [numC (n) (numV n)] ; garantir o retorno do tipo esperado
    [idC (n) (lookup n env)]
    [fdC (n a b) (funV n a b)] ; a função se autorrepresenta
```

A aplicação de função é aparentemente bem simples, pois não precisamos mais procurar:

```
[appC (f a) (local ([define fd f])
                (interp (fdC-body fd)
                     (extend-env (bind (fdC-arg fd)
                                       (interp a env))
                                 mt-env)))]
```

Mas, aqui temos um problema. E se `f` não for uma definição de função? Podemos testar se é, ou tentar avaliar e verificar se o resultado é uma definição.

A segunda opção é mais flexível, afinal podemos ter expressões cujo valor seja uma definição de função. A modificação é simples, basta interpretar `f` antes de usar e tomar cuidado com os tipos:

```
[appC (f a) (local ([define fd (interp f env)])
                (interp (funV-body fd)
                     (extend-env (bind (funV-arg fd)
                                       (interp a env))
                                 mt-env)))]
```

5.2 Desugar e parser

As modificações são simples, mas é preciso ter cuidado com as conversões.

No `desugar`, só precisamos converter o corpo da função.

```
; agora é preciso tomar cuidado com as modificações
(define (desugar [as : ExprS]) : ExprC
  (type-case ExprS as
    [numS    (n) (numC n)]
    [idS     (s) (idC s)]
    [fdS     (n a b)  (fdC n a (desugar b))] ; deve converter o corpo
    [appS    (fun arg) (appC (desugar fun) (desugar arg))]
    [plusS   (l r) (plusC (desugar l) (desugar r))]
    [multS   (l r) (multC (desugar l) (desugar r))]
    [bminusS (l r) (plusC (desugar l) (multC (numC -1) (desugar r)))]
    [uminusS (e)   (multC (numC -1) (desugar e))]
    [ifS     (c s n) (ifC (desugar c) (desugar s) (desugar n))]
    ))
```

O `parser` precisa incluir o tratamento para símbolos livres, que aparecem no corpo das funções e tratar a definição das mesmas. Usarei `func` para defini-las.

Basta incluir duas linhas nos lugares corretos:

```
   [(s-exp-symbol? s) (idS (s-exp->symbol s))] ;  símbolo livre
```

e

```
         [(func) (fdS (s-exp->symbol (second sl))
                      (s-exp->symbol (third sl))
                      (parse (fourth sl)))] ; corpo
```

Curiosamente, não usamos o nome da função em lugar nenhum. Ele é necessário?

5.3 Problemas....

Funções podem ser definidas dentro de funções, como fica o *environment*?

```
(interpS  '(func f1 x (func f2 x (+ x x))))
(interpS  '(call (func f1 x (func f2 x (+ x x))) 4))
(interpS  '(call (call (func f1 x (func f2 y (+ x y))) 4) 5))
```

Nos dois primeiros casos `f1` é solenemente ignorada e o terceiro resulta em um erro, pois `x` não está definido.

6 *Closures*

As associações devem se propagar para as funções "internas". Revendo os exemplos da seção 5.3, vemos que no primeiro o `x` de `f1` não tem associação em `f2`. Isto não é surpresa, pois o *environment* é criado a partir do zero a cada aplicação de função.

Por outro lado, não queremos um *environment* que cresça o tempo todo, senão caímos na situação anterior.

A solução é que cada função, ao ser definida, mantenha consigo todas as associações presentes. Detalhando um pouco mais: quando acontece uma aplicação, o *environment* é estendido com a associação do argumento com seu valor $\mathcal{V}$ (no caso de interesse, $\mathcal{V}$ é uma `ExprC`). Inicialmente temos o *environment* vazio, mas se uma nova função é definida dentro de $\mathcal{V}$, o *environment* onde aconteceu a aplicação deve ser mantido. No último exemplo acima, a definição de `x` em `f1` não foi passada para `f2`.

Em resumo, cada função deve ter o *environment* como um componente adicional. Este pacote de função + *environment* é chamado de *fechamento* ou *closure*.

6.1 Mudando a nomenclatura

Percebemos que funções não precisam de nome, elas são valores. Trocaremos `fdC` para `lamC` (de *lambda*):

```
(define-type ExprC
  [numC (n : number)]
  [idC  (s : symbol)]
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)] ; nomes não são mais necessários
  [appC (fun : ExprC) (arg : ExprC)]
  [ifC   (condição : ExprC) (sim : ExprC) (não : ExprC)]
  )
```

De modo similar, o valor de retorno não é mais função ou número, mas *closure* ou número:

```
(define-type Value
  [numV  (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)])
```

As modificações no `desugar` e no `parser` são triviais.

6.2 Interpretador

Em primeiro lugar, `lamC` deve retornar um valor válido: `closV`:

```
  [lamC (a b) (closV a b env)] ; def. de função captura o environment
```

E `appC` não usará mais o *environment* vazio, mas o *environment* da *closure* chamada:

```
    [appC (f a)
          (local ([define f-value (interp f env)])
            ; f-value fica mais claro do que usar fd
            (interp (closV-body f-value)
                    (extend-env
                        (bind (closV-arg f-value) (interp a env))
                        (closV-env f-value) ; environment da closure
                    )))]
```

Será que poderíamos usar o *environment* corrente na chamada, ao invés do embutido na *closure*?

6.3 Problema?

O que acontece se tivermos um identificador no valor do argumento, como neste caso:

```
(interpS '(call (func f (func x (call f 10))) ( + x y)))
```

O que deveria acontecer e o que acontece? Por quê?

6.4 Nomeando as funções

Como podemos dar nome para funções? Basta colocá-las no *environment*!

Como fazer isso? Com funções!

```
(interpS '(call (func f (call f 32)) (func x (* x x))))
```

Feio? É só recorrer ao `desugar`.

7 Mutação

Qual a diferença entre estas expressões em java?

- `f = 42;`
- `o.f = 42;`
- `f = 42;`

Dependem do contexto! `f` pode ser um campo na primeira ou na terceira expressão, quem sabe?

A semântica do programa também muda com a mutação de valores. É preciso olhar o histórico de execução para saber o valor de um símbolo.

Sem mutação, um trecho de código que recebe o mesmo conjunto de entrada, retorna sempre o mesmo valor. Com mutação, o resultado dependerá se a variável foi alterada.

Note que existe uma diferença entre mudar a associação e mudar o valor de uma variável. Uma variável tem uma associação apenas (por exemplo, nome e posição de memória), mas seu valor muda com o tempo. Daí o nome *variável*.

7.1 Construção minimalista de campo

Para manter a linguagem simples, vamos considerar uma estrutura com um único campo. Um contêiner que guarda apenas um valor.

O termo usado para este contêiner é `box` e possui 3 operações básicas: armazenar, recuperar e alterar. Em linguagens tipadas, é preciso preservar o tipo do valor armazenado.

Com mutação, aparece a possibilidade (motivada por uma necessidade real) de transferir valores de um contêiner para outro. Na nossa linguagem isso só poderá ser feito com sequenciamento de operações: *primeiro* pega o valor e *depois* armazena em outro lugar.

7.2 Sequenciamento

Sequenciamento (`begin`) pode ser construído com açúcar sintático (como?). Mas talvez valha a pena colocar na linguagem central (`ExprC`), depende do balanço entre minimalismo e praticidade. Uma solução intermediária é permitir duas operações em sequência, construindo o resto por *sugaring*, se necessário.

```
(define (beg l) (unless (empty? l) (let ([a (car l)]) (beg (cdr l)))))
```

Observe que o valor de `a` não é usado.

7.3 Colocando `boxes`

Colocar *boxes* e sequenciamento na `ExprC` é, como usual, bem simples, basta declarar os tipos:

```
   [boxC (arg : ExprC)]
   [unboxC (arg : ExprC)]
   [setboxC (b : ExprC) (v : ExprC)]
   [seqC (b1 : ExprC) (b2 : ExprC)])
```

- `boxC` define uma caixa ou pacote.
- `unboxC` desempacota uma valor.
- `setboxC` coloca um valor na caixa. Veja que a caixa também é determinada por uma expressão.
- `seqC` é apenas uma lista de duas expressões que devem ser executadas sequencialmente.

Como `setboxC` pode receber uma expressão para representar uma caixa, temos um novo tipo de valor: `boxV`

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)]
  [boxV (v : Value)])
```

7.4 Interpretador, primeira parte

Interpretar a criação de caixas e o desempacotamento é fácil, basta interpretar os valores:

```
  [boxC (a)   (boxV (interp a env))]   ; cria a caixa
  [unboxC (a) (boxV-v (interp a env))] ; pega o valor associado
```

`setboxC` é mais complicado, pois é preciso identificar a caixa, depois trocar o valor. Ainda bem que existe o `seqC`!

Pode ser implementado como acima

```
  [seqC (b1 b2) (let ([v  (interp b1 env)]) (interp b2 env))]
```

Ou usando o `begin` do *racket*.

```
  [seqC (b1 b2) (begin (interp b1 env) (interp b2 env))]
```

Nos dois casos, existe um problema. O resultado de `b1` é perdido, a menos que ele produza um efeito colateral ou seja *armazenado* em algum lugar. Em outras palavras, sequenciamento depende de mutação.

Qual o problema com mutação? *Poltergeist* ou, como está colocado no livro: *spukhafte Fernwirkung*. Veja este exemplo em *racket*:

```
(let ([b (box 0)])
     (begin (begin (set-box! b (+ 1 (unbox b)))
                   (set-box! b (+ 1 (unbox b))))
            (unbox b)))
```

O mesmo texto `(set-box! b (+ 1 (unbox b)))` produz resultados diferentes.

O resultado deve ficar em algum lugar, mas não pode ser no *environment*, pois ele cuida do escopo léxico!

7.5 Armazenamento do estado

A solução é colocar uma *segunda* tabela de valores: o armazém ou *store*. O *environment* guarda associações de símbolos e o *store* guarda os valores das associações.

- *environment* $\longrightarrow$ localização
- localização $\longrightarrow$ *store*

Parece com algo conhecido? A localização é a "posição de memória". É até mais prático usar números para identificar localizações.

Binding muda ligeiramente para associar símbolos a posições.

E o *store*, chamado de `Storage`, é bastante similar ao *Environment*.

```
(define-type-alias Location number)
(define-type Binding
        [bind (name : symbol) (val : Location)])

(define-type-alias Env (listof Binding))
(define mt-env empty)
(define extend-env cons)

(define-type Storage
      [cell (location : Location) (val : Value)])
(define-type-alias Store (listof Storage))

(define mt-store empty)
(define override-store cons)
```

E `boxV` passa a receber uma localização, ao invés de um valor.

`lookup` deve ser atualizado e precisamos da função equivalente para o `Storage`: `fecth`.

Env	*Storage*
`bind`	`cell`
símbolo → local	local → valor
`mt-env`	`mt-store`
`extend-env`	`override-store`
`lookup`	`fetch`

7.6 Interpretador, segunda parte

O interpretador precisa devolver o novo estado, pois ele pode ter sido alterado. O valor de retorno muda novamente:

```
(define-type Result
    [v*s (v : Value) (s : Store)])

(define (interp [expr : ExprC] [env : Env] [sto : Store]) : Result
```

Todas as entradas no `interp` precisam retornar algo do tipo `Result` via construtor `v*s`.

Os valores diretos (`numC`, `lamC` e `idC`) são fáceis, o único cuidado é fazer a busca dupla para `idC`: em `env` e em `sto`.

As outras entradas precisam ser mais cuidadosas, pois uma parte pode alterar `sto` e isso pode mudar o comportamento da outra. Por exemplo, o primeiro e o segundo termo de uma soma (pense neste trechinho de C: `a++ + (a*=2)`.

Isto ocorre em todos os lugares onde existe mais de uma subexpressão sendo avaliada. O caso mais direto é o `seqC`.

```
[seqC (b1 b2)
   (type-case Result (interp b1 env sto)
            [v*s (v-b1 s-b1)
                 (interp b2 env s-b1)])]
```

Esta forma é um tanto sutil, mas interessante:

1. Interpreta `b1` e olha o resultado
2. O resultado é composto por um valor e um *store*: `v-b1` e `s-b1`.
3. Retorna o valor de `b2` interpretado com o *store* devolvido pela interpretação de `b1`.
4. Note que o valor de `b1` é descartado.

O mesmo truque é usado para serializar as operações de soma e multiplicação.

Para definir localizações novas para `box`es, precisamos de um "alocador".

```
;; retorna a próxima localização disponível
(define new-loc
   (let ( [ n (box 0)])
        (lambda ()
           (begin
              (set-box! n (+ 1 (unbox n)))
              (unbox n)))))
```

A criação do `box` também é mais delicada

```
; cria uma caixa, precisa do valor e de um novo local
[boxC (a)
      (type-case Result (interp a env sto) ; resultado
        [v*s (v-a s-a) ; valor e store
             (let ([onde (new-loc)]) ; onde é uma nova posição
               (v*s (boxV onde)  ; cria, mas com store atualizado
                    (override-store (cell onde v-a) s-a)))])]
```

Com `sto` bem definido, `unboxC` fica trivial. O mesmo vale para `setoboxC`. A única questão é se temos mais de uma localização com a mesma identificação...

Falta `appC`. Os passos são estes:

- Calcula a localização da função
- Calcula a localização do argumento
- Calcula o corpo da *closure* no *environment* estendido

Onde entra o *store*?

```
[appC (f a)
   (type-case Result (interp f env sto) ; função
      [v*s (v-f s-f)
           (type-case Result (interp a env s-f) ; argumento
              [v*s (v-a s-a)
                   (let ([onde (new-loc)])
                        (interp (closV-body v-f) ; corpo
                                (extend-env (bind (closV-arg v-f) onde
                                                  )
                                            (closV-env v-f))
                                (override-store (cell onde v-a) s-a)))
               ])])]
```

7.7 Observações

1. A ordem de avaliação passa a ser *muito* importante. Há uma serialização das operações, que muda a semântica.
2. O *store* é dinâmico, pois muda com a execução do programa. Isto é chamado de persistência. Escopo é para o *environment*, a diferença é sutil, mas significativa. Associação a endereço é uma coisa, conteúdo é outra.
3. Há o problema de reutilização de associação de localizações e uso exagerado de locais no *store*. Além disso, há locais que não são mais usados e precisariam ser liberados por coleta de lixo.
4. É preciso muito cuidado na construção do interpretador, para saber qual `store` usar em cada momento. Estude variações.

5. Manter a lista de modificações no `store`, fazendo novas associações de locais, pode ser interessante. É possível fazer um *rollback*, como em bancos de dados. Isto permite fazer *software transational memory*, onde modificações são marcadas e tratadas apenas ao final, para evitar conflitos entre *threads*.

O que acontece se mudarmos a implementação para fazer a associação direta no *environment* entre identificadores e valores dentro de caixas? É preciso continuar passando o `Store`, ou não? Por quê?

8 Variáveis

Com *boxes*, os identificadores apontam para caixas, e estas tem o seu valor alterado. O exemplo é uma `class`, onde o identificador é o objeto e as caixas são os campos. O valor do identificador é a caixa, que não muda.

Com variáveis, o valor do identificador muda diretamente. O valor *varia*. Esta variação pode ocorrer de duas formas diferentes: entre chamadas de funções, ou *dentro da mesma chamada*.

8.1 Implementação

Precisamos de uma atribuição, que efetivamente altera o valor associado e não precisamos mais de *boxes*, pois usaremos o valor diretamente, mas ainda mantendo o `Store`.

```
(define-type ExprC
  [numC (n : number)]
  [varC  (s : symbol)] ; não é mais identificador
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)]
  [lamC (arg : symbol) (body : ExprC)]
  [appC (fun : ExprC) (arg : ExprC)]
  [ifC   (condição : ExprC) (sim : ExprC) (não : ExprC)]
  [setC (var : symbol) (arg : ExprC)] ; atribuição
  [seqC (b1 : ExprC) (b2 : ExprC)]  ; executa b1 depois b2
  )
```

Consequentemente, não existe mais o valor para caixa.

```
; Não precisamos mais da caixa
(define-type Value
  [numV  (n : number)]
  [closV (arg : symbol) (body : ExprC) (env : Env)])
```

Na atribuição (`setC`), precisamos primeiro calcular o argumento e depois fazer a troca.

A troca em si é a mesma coisa que é feita nas *boxes*. O que realmente muda é que precisamos buscar no *environment* qual a posição do `Store` que representa a variável.

O valor no *environment* é chamado de *l-value*, isto é algo que pode ficar à esquerda de uma atribuição, usualmente é um endereço de memória.

```
[setC (var val) (type-case Result (interp val env sto)
                      [v*s (v-val s-val)
                           (let ([onde (lookup var env)]) ; procura var
                             (v*s v-val
                                  (override-store ; atualiza
                                   (cell onde v-val) s-val)))])]
```

Veja que "`onde`" é o endereço da variável.

Um exemplo em *C*:

```
  x = 2; /* precisamos apenas do endereço de x */
  y = x; /* precisamos do valor de x */
```

8.2 Vantagens e desvantagens de mutação

Implementamos duas formas de mutação: com `boxes` e diretamente. Algumas linguagens implementam ambas, outras apenas uma delas e outras como *SML*, nenhuma.

As vantagens são estas:

- Se estado for necessário e não tivermos mutação, é necessário aumentar o número de parâmetros e valores de retorno. Isto deveria ser feito para todas as funções que precisam se comunicar. A introdução de estado permite uma forma de modularização.
- Estado permite a construção de estruturas de dados cíclicas (autorreferentes) de uma maneira simples.
- É possível criar estado interno de procedimentos.

Por outro lado, estado implica em ação à distância e em *aliasing*. *C* é um bom exemplo dos problemas que podem aparecer.

A chamada por referência é um exemplo de *aliasing* e não é realmente necessária. A passagem de *boxes* pode servir de indicação para a função chamada que ela pode mexer nos valores.

9 Estruturas recursivas

Recursão é um problema por causa da autorreferência. A autorreferência aparece como curiosa ou problemática em diversos lugares: renormalização, consciência, Goedel, etc.

9.1 Dados

A recursividade em estruturas de dados pode acontecer de duas formas: referência a objetos do mesmo tipo, ou ao mesmo objeto.

Para objetos do mesmo tipo é mais simples, pois não há autorreferência. Um exemplo são árvores e listas.

Para o próprio objeto, como listas circulares e grafos cíclicos, é preciso cuidado adicional. Por exemplo, para percorrer o grafo.

Para resolver, é preciso primeiro usar uma caixa com qualquer coisa dentro, para criar uma associação, *e depois* usar o valor. Precisamos da mutação. Com variáveis, é similar:

```
> (interpS '(def b b b))
. . lookup: b não foi encontrado
> (interpS '(def b 1 (seq (:= b 30) b)))
- Result
(v*s (numV 30) (list (cell 5 (numV 30)) (cell 5 (numV 1))))
```

9.2 Funções

O problema é parecido. O identificador da função não está definido ainda. Cria-se uma associação qualquer e depois troca-se com mutação. Compare com o uso de protótipos em C.

```
> (interpS '(def fat (func n (if n (* n (call fat (- n 1))) 1)) (call fat 10)))
- Result
. . lookup: fat não foi encontrado
> (interpS '(def fat 1729
                 (seq (:= fat (func n
                                 (if n (* n (call fat (- n 1))) 1)))
                      (call fat 10))))
- Result
(v*s
 (numV 3628800)
 (list
  (cell 21 (numV 0))
  (cell 20 (numV 1))
  (cell 19 (numV 2))
  (cell 18 (numV 3))
  (cell 17 (numV 4))
  (cell 16 (numV 5))
  (cell 15 (numV 6))
  (cell 14 (numV 7))
  (cell 13 (numV 8))
  (cell 12 (numV 9))
  (cell 11 (numV 10))
  (cell 10 (closV 'n (ifC (varC 'n) (multC (varC 'n) (appC (varC 'fat) (plusC (......
  (cell 10 (numV 1729))))
>
```

9.3 Atropelando o uso

O que acontece se a variável ou a caixa for usada antes da mutação? Há um *vazamento* do valor temporário (variável não inicializada).

Possíveis soluções:

- Usar um valor inválido ou especial (*undef* ou *NaN*).
- Fazer um teste explícito a cada referência — muito caro.
- Limitar sintaticamente o acesso. Por exemplo, permitir apenas para definição de funções, mas isto impede estruturas de dados recursivas.

9.4 Evitando a trapaça

Usar o `def` é apelar para uma recursão pré-existente. Como criar recursão sem usar recursão?

Usando uma notação simplificada, apenas para esta discussão:

```
fact ::
  (λ (n)
       (if (zero? n)
           1
           (* n (??? (sub1 n)))))
```

??? deveria ser substituído pela própria definição. Não serve...

Que tal se ??? for o resultado de uma chamada de função?

```
mk-fact ::=
  (λ (f)
      (λ (n)
         (if (zero? n)
             1
             (*  n (f (sub1 n))))))
```

Podemos tentar usar o **mk-fact** no lugar do `f`, forçando a recursão.

```
    (λ (n)
        (if (zero? n)
            1
            (*  n (mk-fact (sub1 n)))))
```

Não funciona, pois pois **mk-fact** espera um procedimento como argumento!

Outra tentativa:

```
mk-fact ::
  (λ (f)
      (λ (n)
           (if (zero? n)
               1
               (*  n ((f [*]) (sub1 n))))))
```

Aplicando a mk-fact:

```
  (λ (n)
     (if (zero? n)
          1
          (*  n ((mk-fact [*]) (sub1 n)))))
```

Testando valores:

- Com 0: 1 (ok)
- Com 1: Funciona se tivermos *lazyness*, verifique.
- Com 2: Não funciona por causa do [*]

Mas, e se [*]for o próprio `mk-fact`? cada vez que for necessário, um novo "corpo" é criado!

```
mk-fact ::
  (λ (f)
   (λ (n)
   (if (zero? n)
      1
      (*  n ((f f) (sub1 n))))))
```

O argumento é duplicado na chamada, sempre que for necessário
Para ser fatorial, o `f` deve ser `mk-fact`

```
fact ::
   (mk-fact mk-fact)
```

ou, de forma mais completa:

```
fact::
  (
   (λ (mk-fact) (mk-fact mk-fact))
    (λ  (f)
        (λ (n)
          (if (zero? n)
             1
             (*  n ((f f) (sub1 n))))))
   )
```

Funciona! mas muito complicado!
O ideal seria

```
(make-recursive-procedure
       (λ (fact)
                (λ (n) (if (zero? ...)))
```

Fatorial poderia ser escrito assim:

```
(
   (λ (mk-fact) (mk-fact mk-fact))
   (λ (f)
      (λ (g)
         (λ (n)
             (if  (zero? n)
                  1
                  (* n (g (sub1 n))))))
          (f f))
  )
```

Usando `(f f)` $\rightarrow$ `g`, a sacada é que o corpo do fatorial ficou isolado!

```
    (λ (g)....(g...))
```

Temos um construtor de recursão:

```
make-recursive-procedure ::
 (λ (p)
    (
        (λ (f) (f f))
        (λ (f) (p (f f)))
    )))
```

Só que `(f f)` será executada antes da hora, causando recursão infinita!
Solução — mais um λ! Isso é chamado combinador Y, o gerador de recursão

```
Y ::
 (λ (p)
  (
   (λ (f) (f f))
   (λ (f) (p (λ (a) ((f f) a))))
  ))
```

Na linguagem com *closure* apenas, sem estado, o código de fatorial fica assim:

```
(interpS '(call
             (call
              (func p
                    (call
                     (func m (call m m))
                     (func f (call p (func a (call (call f f) a))))
                     ))
              (func g (func n (if n (* n (call g (- n 1))) 1))))
             5)
)
```

Com Y, é possível construir toda a linguagem apenas com 3 primitivas:

1. definição de função (λs)
2. aplicação de funções ()
3. variáveis

Alonso Church inventou o *lambda calculus* e Alan Turing inventou a máquina universal. São equivalentes, mas *lambda calculus* é mais expressiva.

10 Orientação a objetos

Algumas vezes queremos parametrizar uma função com informação ativa, não apenas valores estáticos. Isso pode ser facilmente feito com funções, mas uma função não permite flexibilidade.

O que é um objeto? É um conjunto de valores tratados como uma unidade. O que estes valores representam depende da semântica. Do ponto de vista da linguagem, trata-se apenas de uma coleção, acompanhada de um meio de acesso para cada valor individualmente. Lembre que valor pode ser uma função, no nosso caso.

Queremos um conjunto de valores e funções representando alguma coisa. Como um *closure* de várias funções sobre o mesmo conjunto de dados.

No que segue, um **objeto** é definido por um agrupamento de funções, acompanhado de meios para escolher uma delas. Não é difícil, pelo menos conceitualmente, estender a definição para outros valores e inserir estado.

10.1 Sem herança

A forma mais simples de pensar em um objeto é um *valor* que mapeia *nomes* a outros *valores ou métodos*. Quase um `let` ou mesmo um `lambda`. A diferença está justamente no conjunto de nomes em cada objeto.

10.1.1 Incluindo objetos na linguagem central

O objeto é apenas uma associação de nomes a valores. Isto pode ser feito da mesma forma que fizemos o *environment*, mas manter os valores em uma lista separada simplifica o código[2].

[2]Será mesmo? Procure alterar a implementação abaixo para suar associações como no *environment*

Primeiro, um objeto agora é um valor possível.

```
(define-type Value
     [numV (n : number)]
     [closV (arg : symbol) (body : ExprC) (env : Env)]
     [objV (ns : (listof symbol)) (vs : (listof Value))])
```

Ele também deve fazer parte da linguagem.

```
[objC (ns : (listof symbol)) (es : (listof ExprC))]
```

Interpretar um objeto significa interpretar seus valores.

```
[objC (ns es) (objV ns (map (lambda (e)
                              (interp e env))  es))]
```

Precisamos de obter o valor associado a um nome, pense em algo como `o.n`.

```
[msgC (o : ExprC) (n : symbol)]
```

A interpretação de `mscC` consiste em buscar o valor, precisaremos de uma função `lookup-msg`.

```
[msgC (o n) (lookup-msg n (interp o env))]
```

`lookup-msg` é bem parecido com `lookup`, naturalmente.

```
(define (lookup-msg [n : symbol] [o : Value]) : Value
    (type-case Value o
          [objV (nomes valores)
                (cond
                  [(empty? nomes)
                         (error 'lookup-msg
                                (string-append (symbol->string n)
                                               "␣não␣foi␣encontrado"))]
                  [(symbol=? n (first nomes)) (first valores)] ; achou
                  [else (lookup-msg n
                                (objV (rest nomes)
                                      (rest valores)))])] ; olha no resto
          [else (error 'lookup-msg "Valor␣passador␣não␣é␣um␣objeto!")])
)
```

Esta é uma `ExprS` válida, qual o resultado da interpretação?

```
(letS 'o (objS (list 'add1 'sub1)
       (list (lamS 'x (plusS (idS 'x) (numS 1)))
             (lamS 'x (plusS (idS 'x) (numS -1)))))
             (msgS (idS 'o) 'add1 (numS 3)))
```

Se estivéssemos escrevendo em uma linguagem como *java* ou *C++*, o código correspondente seria este (o nome da classe é apenas para ilustração, assim como o tipo numérico[3]):

[3]Lembre que, no *racket*, um número pode ser inteiro, double, racional ou complexo. Aqui usei inteiro para simplificar.

```
class Elevador {
    int add1(int x) {return x+1;}
    int sub1(int x) {return x-1;}
}
```

e em algum lugar do código

```
   Elevador o;
   o.add1(3);
```

10.1.2 Entendo objetos como uma coleção de nomes

Para simplificar a discussão e se ater apenas nos conceitos fundamentais, vamos implementar objetos diretamente em *racket*, se limitando às construções usadas na linguagem central (*core*). Em outras palavras, as características apresentadas daqui para frente podem ser incluídas por açúcar e traduzidas para o *core*.

Se um objeto pode ser visto como uma coleção de valores com nomes, como nas duas listas acima, podemos implementá-los diretamente com um lambda e um `case`. O objeto do último exemplo fica assim:

```
(define o-1
     (lambda (m)
        (case m
            [(add1) (lambda (x) (+ x 1))]
            [(sub1) (lambda (x) (- x 1))])))
```

Olhando para este código, vemos que um objeto é um λ! A diferença é precisamos multiplexar a entrada.

Precisamos também fazer a invocação de métodos. Isto é simples, mas iremos um pouco além:

```
(define (msg o m . a)
        (apply (o m) a))
```

Temos um problema se tentarmos executar este código com o `plai-typed`. Como o `define` neste caso retorna uma função (`msg` é um λ), precisaríamos declarar o seu tipo de retorno. A solução mais rápida é deixar momentaneamente de lado a verificação de tipos, optando por usar `plai` e não `plai-typed`.

Aqui aparecem dois elementos novos do *racket*. O `apply` transforma a lista passada como argumento em uma lista de argumentos. Confuso? Um exemplo esclarece:

```
(apply + '(1 2 5 3))
; é o mesmo que
(+ 1 2 5 3)
```

O outro elemento é o `.`, um operador que associa o símbolo seguinte ao resto da lista passada como argumento. No exemplo `a` é a lista dos argumentos restantes (segundo em diante) passados na invocação de `msg`, o primeiro argumento fica associado a `m`. Ainda no exemplo, esta lista terá apenas um elemento, o segundo argumento.

Construtores Nada mais é que uma função chamada na criação do objeto. Não há novidade aqui.

```
(define (o-constr-1 x)
        (lambda (m)
             (case m
                 [(addX) (lambda (y) (+ x y))])))
```

Estado Basta usar as variáveis ou `boxes`.

```
(define (o-state-1 count)
        (lambda (m)
            (case m
                [(inc) (lambda () (set! count (+ count 1)))]
                [(dec) (lambda () (set! count (- count 1)))]
                [(get) (lambda () count)])))
```

É interessante notar que `count` é agora um atributo ligado a cada objeto, graças a *closures*.

Membros privados Está praticamente pronto, basta fazer uma associação interna. Privacidade tem a ver com escopo léxico.

```
(define (o-state-2 init)
    (let ([count init])
       (lambda (m)
           (case m
              [(inc) (lambda () (set! count (+ count 1)))]
              [(dec) (lambda () (set! count (- count 1)))]
              [(get) (lambda () count)]))))
```

Neste exemplo, `init` é passado para o construtor, que associa seu valor a `count`. `count` não é acessível de fora.

Algumas linguagens permitem funções e classes "amigas".

Membros estáticos Também chamados de "variável de classe", são mais simples do que parecem. Basta que seu escopo seja *externo ao construtor*.

Este é o exemplo clássico do contador de objetos construídos.

```
(define o-static-1
   (let ([counter 0])
      (lambda (amount)
         (begin
            (set! counter (+ 1 counter))
               (lambda (m)
                  (case m
                     [(inc) (lambda (n) (set! amount (+ amount n)))]
                     [(dec) (lambda (n) (set! amount (- amount n)))]
                     [(get) (lambda () amount)]
                     [(count) (lambda () counter)])))))))
```

Objetos com autorreferência Pode ser feito diretamente com mutação. A técnica é exatamente a mesma usada na seção 9.2, com uma leve adaptação.

```
(define o-self!
   (let ([self 'dummy])
      (begin
          (set! self
             (lambda (m)
                (case m
                   [(first) (lambda (x) (msg self 'second (+ x 1)))]
                   [(second) (lambda (x) (+ x 1))])))
           self)))
```

o funcionamento pode ser testado assim

```
(test (msg o-self! 'first 5) 7)
```

É possível evitar a mutação, usando o combinador Y.

```
(define o-self-no!
   (lambda (m)
      (case m
      [(first) (lambda (self x) (msg/self self 'second (+ x 1)))]
      [(second) (lambda (self x) (+ x 1))])))

(define (msg/self o m . a)
        (apply (o m) o a))
```

É interessante notar que neste caso, todo método precisa receber uma referência par si mesmo como argumento. Isto é feito tanto em *perl*, como em *python*, por exemplo. Seria melhor colocar com *desugar*?

Chamada definida em tempo de execução Ou *dynamic dispatch*. Objetos podem invocar funções em outros objetos, sem saber qual a função será de fato chamada. Esta característica permite uma flexibilidade enorme e é uma das mais importantes na orientação a objetos.

Exemplo com árvore binária

```
; folha
(define (mt)
    (let ([self 'dummy]) ; só para autorreferência
       (begin
         (set! self
           (lambda (m)
             (case m
                [(add) (lambda () 0)])))) ; retorna 0
          self)))

; nó interno
(define (node v l r)
    (let ([self 'dummy])
       (begin
          (set! self
            (lambda (m)
              (case m
                 [(add) (lambda () (+ v  ; soma os valores dos filhos
                                      (msg l 'add)
                                      (msg r 'add)))])))
            self)))
```

10.2 Com herança

A implementação de herança depende essencialmente da localização de membros em outras classes.

10.2.1 Nomes dos membros

O acesso a membros de um objeto é feito por nomes, como vimos. Existem duas propriedades independentes que definem o conjunto e a forma de acesso:

- Os nomes podem ser computados em tempo de execução (**dinâmicos**) ou pré-compilados (**estáticos**).
- O conjunto pode ter tamanho **fixo** ou **variável**.

Isto gera 4 possibilidades, mas apenas 3 fazem sentido[4]:

1. Nomes **estáticos** e tamanho de conjunto **fixo** — é o caso comum de linguagens compiladas, como Java.
2. Nomes **dinâmicos** e conjunto **fixo** — linguagens que usam reflexão sobre nomes fixos. Algumas variações de Java.
3. Nomes **dinâmicos** e conjunto **variável** — a maioria das linguagens de *script*, que usam tabelas dinâmicas para armazenar os nomes. Exemplos são Perl (*hashes* e Python (*dicionários*).

[4]Esta situação ocorre com uma frequência muito suspeita...

4. Nomes **estáticos** e conjunto **variável** — Se os nomes são fixos, o conjunto varia de que forma? Não faz muito sentido.

Para quem viu a implementação de objetos em perl, o terceiro caso é bastante familiar. Vamos nos concentrar no primeiro caso, que é suficiente para explorar os conceitos importantes.

10.2.2 Procura dos membros

Na nossa implementação em Racket, usamos um `case` para identificar qual função chamar (Se estivéssemos mexendo também com atributos, o procedimento seria o mesmo).

Veja a seção 10.1.2, por exemplo, lá são dois métodos `add1` e `sub1`. Neste exemplo e em todos os que seguiram, os `cond`s não incluem a cláusula `else`. Em princípio isto faz sentido, poiso conjunto de nomes é finito e os nomes são estáticos.

Mas existe uma utilidade para o `else`: buscar o método em *outra* classe. Isto é o mesmo que dizer que se um objeto não consegue fazer uma coisa (não possui o método definido), ele passa a responsabilidade para outro. Isto é a base para herança!

Tudo o que precisamos é uma função que faz a busca do método em outro objeto (não estamos falando em *classes* ainda).

```
(define o-e
     (lambda (m)
        (case m
            [(add1) (lambda (x) (+ x 1))]
            [(sub1) (lambda (x) (- x 1))]
            [else (parent-object m)]
     )))
```

Neste exemplo, `parent-object` é o objeto "pai" de `o-e`. Este código não funciona da forma em que está, pois `parent-object` não está definido. Precisamos estudar como estender um objeto.

Estendendo um objeto Para ser estendido, um objeto `obj` precisa saber quem é seu pai. Uma alternativa é que o construtor de `obj` receba o pai como argumento. Isto implica que o pai deve ter sido construído antes e que *os mesmos* parâmetros sejam passados para o construtor de `obj`, de modo a manter a consistência.

Uma solução é que `obj` construa seu pai, usando os parâmetros que recebeu. Tudo o que é necessário é que o construtor de `obj` receba um parâmetro adicional: o construtor de seu pai!

Suponha que queiramos incluir um método `size`, que retorna o tamanho das árvores do último exemplo. Para fazer isto de acordo com a orientação a objetos, vamos procurar não mexer nas definições de nós (`node`) e folhas (`mt`).

Precisamos apenas dos novos construtores `node/size` e `mt/size`.

```
(define (node/size parent-maker v l r)
  (let ([parent-object (parent-maker v l r)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(size) (lambda () (+ 1
                                      (msg l 'size)
                                      (msg r 'size)))]
                [else (parent-object m)])))
      self)))
```

```
(define (mt/size parent-maker)
  (let ([parent-object (parent-maker)]
        [self 'dummy])
    (begin
      (set! self
            (lambda (m)
              (case m
                [(size) (lambda () 0)]
                [else (parent-object m)])))
      self)))
```

Um objeto `tree/size` pode ser construído assim:

```
(define a-tree/size
  (node/size node
             10
             (node/size node 5 (mt/size mt) (mt/size mt))
             (node/size node 15
                        (node/size node 6 (mt/size mt) (mt/size mt))
                        (mt/size mt))))
```

Veja que os construtores dos pais são chamados a cada vez.

Escopo dos membros dos pais Quando um construtor e chamado, os construtores dos pais, avós, etc, são todos chamados em cadeia. Cada objeto ganha uma cópia particular de tudo o que está acima na hierarquia.

Um ponto que fica solto é saber quais versões dos métodos[5] estão disponíveis para cada objeto. Java disponibiliza apenas a versão mais próxima do método (o primeiro ancestral). Por outro lado, todos os atributos são visíveis. Linguagens de *script* permitem que todos os métodos sejam visíveis também.

[5] Aqui estamos considerando os "`public`" e "`protected`".

Quase classes Veja que estas construções de objetos, que criam e chamam os métodos dos objetos superiores (pais), já se comportam como classes, mas uma classe estendida não é uma classe completa, apenas definimos a extensão.

Esta definição parcial será chamada de *blob*: uma função que é parametrizada pelos pais.

Protótipos Ao invés de chamar os construtor do pai, o construtor de um objeto poderia receber uma referência para o pai já pronto. Esta é uma outra opção para orientação a objetos. Protótipos ao invés de classes.

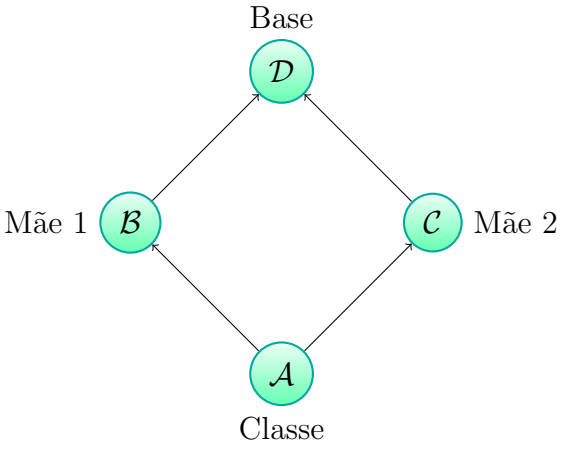
O protótipo é um objeto pai compartilhado. Com protótipos, pode-se construir o comportamento de classes, mas o contrário não é possível — pode-se argumentar que protótipos são mais fundamentais do que classes. Algumas linguagens, como *Self* e *Javascript*, seguem esta linha.

10.2.3 Herança múltipla

Quando um membro não é encontrado, o *else* procura no objeto pai (ou classe mãe). E princípio, pode-se colocar uma lista de objetos ou classes onde procurar. Ou seja, um objeto pode herdar membros de vários outros, não apenas de um só: herança múltipla.

Quando se tem uma hierarquia de classes com herança múltipla, a coisa complica. Em que ordem deve ser feita a busca? Olhando "para cima", existem diversas ramificações, temos um grafo orientado e acíclico, como percorrê-lo? Largura ou profundidade? Cada caso leva a um resultado diferente e igualmente válido[6].

Considere este caso particularmente terrível:



Quantas cópias de $\mathcal{D}$ devem existir em $\mathcal{A}$? Duas cópias é um desperdício de espaço, uma única cópia pode mudar o comportamento de $\mathcal{C}$ ou $\mathcal{D}$.

10.2.4 Subindo ou descendo?

Normalmente podemos usar `super` para chamar um método equivalente de uma superclasse. Isto tipicamente ocorre em construtores, mas também indica que a classe tem prioridade sobre a classe mãe, substituindo seus métodos.

Uma outra visão é chamar o método da classe mãe, sempre, e deixar que os métodos equivalentes das subclasses sejam chamados quando necessários: ao invés de subirmos na hierarquia, descemos. Ao

[6]Perl define a ordem de busca a priori e o programador precisa levar isto em conta.

invés de `super`, usamos `inner`. Esta característica não é comum, mas existe em algumas linguagens como *beta*.

10.2.5 *Mixins* e *traits*

Uma definição de classe contém apenas o código correspondente à classe, não à superclasse. Só colocamos a extensão da classe. Por exemplo, em java:

```
class C extends B { ... }
```

Esta definição pode ser pensada como uma operação separada: temos a classe base, `B` e uma função que gera uma extensão.

```
classext E  { ... }

class C = E(B);
```

Este "extensor" pode ser usado para criar várias classes:

```
class X = E(M);
class Y = E(N);
class Z = E(P);
```

O nome dado a um "extensor" deste tipo é *mixin*. Um *mixin* completa uma classe, não a estende, permitindo herança múltipla sem usar herança múltipla de verdade. Uma solução mais simples é o uso de *interfaces*, usada em Java. O *mixin* vai um pouco mais longe, permitindo o uso de estado.

O nome *mixin* vem de uma sorveteria, pois você podia escolher quais coisas você queria em cima do sorvete (*mix in*), como chocolate granulado, docinhos, etc.

No entanto, um problema persiste: *mixins* acumulados podem entrar em conflito, se tiverem métodos e assinaturas iguais. Suponha que `E` e `F` incluam métodos com mesmo nome:

```
mixin E  { ... }
mixin F  { ... }
class X = E(F(M));
```

Qual implementação deve prevalecer? Seja qual for a decisão, o resultado depende da ordem.

Para resolver estes problemas, existe o conceito de *traits*, que incluem restrições sobre *mixins*, suponha que `C` seja uma classe que usa um conjunto de *traits* $T_1, T_2, \ldots, T_n$:

- T_is não implementam estado. Estado deve ser implementado por `C`
- Conflitos de nomes devem ser resolvidos explicitamente por `C` — isto implica em especificações adicionais, indicando qual versão deve ser usada. No caso de `C` ter uma classe mãe `A`, os métodos de um *trait* têm precedência sobre os da classe `A`, mas são substituídos pelos da classe `C`.
- Funciona da mesma forma que código *inline*, portanto não depende da ordem de inclusão.

Veja estes exemplos em PHP, retirados de `www.php.net`:

```
<?php
class Base {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait SayWorld {
    public function sayHello() {
        parent::sayHello();
        echo 'World!';
    }
}

class MyHelloWorld extends Base {
    use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
?>
```

Neste exemplo `MyHelloWorld` usa o *trait* `SayWorld`, o código de `SayWorld` poderia simplesmente ser copiado para dentro de `MyHelloWorld`.

No próximo caso, a situação é mais sutil. Como são vários *traits* usados na mesma classe, é preciso eliminar a ambiguidade entre os métodos explicitamente. PHP usa as palavras reservadas `insteadof` para indicar qual método deve ser usado e `as` para renomear um método e evitar o conflito.

```php
<?php
trait A {
    public function smallTalk() {
        echo 'a';
    }
    public function bigTalk() {
        echo 'A';
    }
}

trait B {
    public function smallTalk() {
        echo 'b';
    }
    public function bigTalk() {
        echo 'B';
    }
}

class Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
    }
}

class Aliased_Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
        B::bigTalk as talk;
    }
}
?>
```

11 Gerenciamento de memória

A liberação da memória que não é mais utilizada é conhecida como "coleta de lixo".

Para variáveis locais, implementadas na pilha de execução ou em registradores, a coleta é muito simples e rápida, pois basta restaurar a pilha. Em C, estas variáveis são chamadas de "automáticas", pois são removidas ao final do escopo.

Outros tipos de variáveis, por outro lado, sobrevivem ao escopo, e precisam ser removidas em momento adequado. Existem dois problemas potenciais:

- falta de solidez (*soundness*) — a recuperação da memória é feita mais cedo do que o necessário.

- falta de completude(*completness*) — a recuperação é muito tardia.

No primeiro caso, uma ou mais posições de memória podem receber acesso depois de liberadas! Isto gera lixo de verdade. No segundo, o sistema pode ficar sem memória sem necessidade: um bloco pode continuar ocupando memória mesmo que não vá mais ser usado.

As estratégias para recuperação se dividem em manual e automática. Na recuperação manual, o programador deve colocar `mallocs` e `frees` (ou equivalentes) por conta própria e nos lugares corretos. Diz a experiência universal que isto nem sempre funciona.

É impossível construir um algoritmo que consiga solidez e completude ao mesmo tempo. Como falta de completude é menos nociva, os algoritmos procuram garantir solidez.

11.1 Recuperação manual

A recuperação manual, além de estar mais sujeita a erros, como liberar a memória duas vezes ou esquecer de liberá-la, introduz o problema de fragmentação: à medida em que blocos são alocados e liberados, a memória fica cheia de "buracos" pequenos. Mesmo que exista uma grande quantidade de memória livre no total, pode não acontecer de existir um bloco contíguo suficientemente grande para uma determinada necessidade.

11.1.1 Contagem de referências

Uma solução para identificar quais blocos liberar é usar a contagem de referências. Cada vez que o bloco de memória recebe uma referência, um contador é incrementado. Toda vez que a referência deixar de existir, o contador é decrementado.

Quando o bloco é alocado, seu contador começa com o valor 1. Se o contador chegar a 0, é sinal que não há mais como ter acesso ao bloco e ele pode ser liberado.

Feito manualmente, é preciso ter cuidado para garantir que todas as referências sejam consideradas, mesmo as mais sutis. Outro problema são referências cíclicas, que levam, em princípio, a um valor infinito no contador. Mesmo que este "infinito seja renormalizado", ainda há o problema de "ilhas", blocos que se autorreferenciam, mas que não são referenciados por nenhum outro ponto do programa.

Perl, por exemplo, usa a contagem de referências automaticamente. Esta estratégia é bastante simples, mas traz uma série de custos adicionais:

- Os objetos precisam ter contadores, aumentando seu tamanho
- Acessos são mais caros
- Quando um objeto é liberado, todos os objetos referenciados por ele precisam ter seu contador decrementado. Com isto, novos objetos podem ser liberados, gerando uma reação em cadeia.
- Percorrer objetos para decrementar contadores pode alterar cache e paginação.

11.2 Recuperação automática

A coleta de lixo (**GC**) se baseia em percorrer a árvore de referências. Partindo de um conjunto raiz (*root set*), formado pelas variáveis globais e as do contexto corrente, o algoritmo marca todos os blocos referenciados como *vivos*, recursivamente. Ao final, os blocos não marcados podem ser liberados.

A definição de *vivo* e como a árvore é percorrida permite variações.

11.2.1 Verdade e demonstrabilidade

A "verdade" procurada é saber quais são exatamente os blocos que não serão mais atingidos. Mas isto é impossível no caso geral (Gödel), é preciso se contentar com aqueles que conseguimos demonstrar.

11.2.2 Suposições centrais

Existem duas suposições para garantir um algoritmo sólido:

1. O GC conhecer o tipo de cada objeto e sua representação na memória;
2. As referências construídas pelo programa devem satisfazer estas condições:
   (a) Não pode haver referências fora do *root set*.
   (b) As referências só podem se referir a pontos específicos de cada objeto.

11.2.3 Gerenciamento conservativo

Para a maioria das linguagens, o *root set* definido acima está perfeito. Mas em C e similares, é possível transformar qualquer número em ponteiro....

Para contornar este problema, a estratégia é invertida: o GC procura o que claramente não é uma referência e cresce a árvore a partir daí. Não conseguirá uma completude muito boa, mas pode ter solidez e eficiência.

12 Decisões de representação

Por que usamos a representação de números do Racket, mas criamos a nossa própria representação de função? Compare `numV` com `funV` ou `closV`.

12.1 Mudando a representação

A escolha não foi uniforme por questões didáticas: o centro da discussão foi a construção de representação de *closures*, não de números. Mas existe uma discussão importante por trás.

Até onde consideramos o material pré-existente suficiente. Se olharmos para um processador, já temos definidos os tipos fundamentais no nível da máquina: inteiros de 32 ou 64 bits, ponto flutuante com esta ou aquela precisão, etc.

Por outro lado podemos usar tipos sofisticados como base, pro exemplo MATLAB ou APL, computação simbólica, ou mesmo Racket, cuja representação de números é muito flexível.

Se quisermos, podemos tratar as *closures* como nativas:

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value -> Value))])

(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    [numC (n) (numV n)]
    [idC  (n) (lookup n env)]
    [appC (f a) (local ([define f-value (interp f env)]
                        [define a-value (interp a env)])
                  ((closV-f f-value) a-value))]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
    [ifC (c s n) (if (zero? (numV-n (interp c env)))
                             (interp n env) (interp s env))]
    [lamC  (a b) (closV (lambda (arg-val)
                          (interp b
                                  (extend-env (bind a arg-val)
                                              env))))]))
```

Implementar na mão permite entender o processo com mais detalhe, mas depois de entendê-lo, o interpretador fica mais conciso.

A interpretação da nossa linguagem ocorre em um interpretador "hospedeiro". Delegar tarefas para o hospedeiro simplifica o nosso trabalho, mas pode trazer uma série de problemas potenciais.

12.1.1 Tratamento de erros

Um erro que ocorre no *hospedeiro* pode não fazer sentido se olharmos apenas a linguagem interpretada. É preciso converter as condições de erro para condições equivalente na linguagem final (ou da "superfície").

Pode acontecer o contrário, uma situação que deveria causar erro na superfície executar sem alarme no hospedeiro. É preciso ter cuidado para que apenas uma parte do hospedeiro seja usada, exatamente a que pode ser mapeada diretamente na linguagem final.

Deixar a semântica para o hospedeiro é muito ruim, pois pode gerar resultados inesperados, além de destruir a portabilidade.

12.1.2 Mudança na semântica

Quando o mapeamento é natural, não existe muito problema. Mas, se as linguagens possuem semânticas diferentes em algumas situações, delegar funcionalidades para o hospedeiro é claramente complicado.

Exemplos incluem escopo dinâmico versus estático, execução postergada ou não.

12.2 Mudança de ambiente

Um caso interessante para implementação na linguagem hospedeira é o ambiente (*environment*). Ele já é interno à linguagem e não há manipulação direta (ocorre encapsulamento).

O *environment* nada mais é do que um mapeamento e, portanto, pode ser tratado como uma função que recebe um nome e retorna o valor associado.

```
(define-type-alias Env (symbol -> Value))
```

O *environment* vazio é uma função constante, que retorna erro.

```
(define (mt-env [name : symbol])
                (error 'lookup "name␣not␣found"))
```

Como estender o *environment*? Trocando por uma outra função!

```
(define (extend-env [b : Binding] [e : Env])
  (lambda ([name : symbol]) : Value
    (if (symbol=? name (bind-name b))  ; achou?
                       (bind-val b)    ; sim, é o próprio
                       (lookup name e) ; não, olha no anterior
                       )))
```

O `lookup` fica muito simples, basta aplicar o próprio *environment*:

```
(define (lookup [n : symbol] [e : Env]) : Value (e n))
```

13 A importância do açúcar

Açúcar sintático serve tanto para manter a linguagem central pequena (encolhendo), como para estender a linguagem como um todo, tornando-se uma ferramenta muito poderosa.

Algumas linguagens, como Racket, permitem a adição de açúcar na própria linguagem. Isto é, permitem construções que complementam a sintaxe dentro do próprio programa. Isto ocorre em algumas linguagens orientadas a objetos mais modernas, de forma mais sutil.

13.1 Um primeiro exemplo

Já vimos que *desugar* nada mais é do que uma transformação que leva a sintaxe "açucarada" na sintaxe central da linguagem. No fundo é um processamento de macros.

O exemplo do `let` é bem claro:

```
(let (var val) body) ⟶  ((lambda (var) body) val)
```

Para declarar esta regra em Racket (ou melhor, no plai), usamos `define-syntax` e `syntax-rules`:

```
(define-syntax my-let-1
  (syntax-rules ()
    [(my-let-1 (var val) body)      ; macro (açúcar)
     ((lambda (var) body) val)]))   ; expansão (na core language)
```

`syntax-rules` é uma definição de macros. Em cada conjunto `[...]` define-se a sintaxe no primeiro elemento e sua expansão no segundo. Veja que é realmente macro, se fizermos `(my-let-1 (1 2) 3)` aparecerá um erro correspondente a `((lambda (1) 3) 2)`: `1` não é identificador.

Existe uma outra abreviação, para permitir a aplicação dos macros em listas de valores. A notação '...' é usada em dois momentos:

1. Na definição do macro, recebe uma lista de identificadores `L` do seu lado esquerdo e a transforma em várias listas, uma para cada elemento de `L`

2. Na expansão, replica a expressão expandida para cada identificador, casando identificadores correspondentes.

Um exemplo torna seu funcionamento mais claro, `my-let-2` permite a declaração de várias associações:

```
(define-syntax my-let-2
  (syntax-rules ()
    [(my-let-2 ([var val] ...) body)
     ((lambda (var ...) body) val ...)]))
```

ainda de outra forma, `([var val] ...)` representa `([var`$_1$ `val`$_1$`] [var`$_2$ `val`$_2$`] ...)` e na expansão teríamos

```
     ((lambda (var1) body) val1)]))
     ((lambda (var2) body) val2)]))
                      .
                      .
```

Esta notação pode ser usada no sentido inverso também, juntando listas.

13.2 Transformadores de sintaxe

Os transformadores de sintaxe, como *desugar*, são funções que podem ser incorporadas na linguagem, como acima. No entanto, são funções que operam em *tempo-de-compilação* e não em *tempo-de-execução*.

Ao contrário do *desugar*, que faz todas as transformações, nas definições de macros, são criadas várias funções menores, uma para cada macro. Se olharmos bem, *desugar* é pouco mais do que uma coleção de casos.

`define-syntax` cria uma definição de macro, mas não especifica sua expansão deve ser feita. A forma mais simples de especificar a expansão é com `syntax-rules`, mas existe um modo bem mais flexível, com `syntax-case`.

Como o nome sugere, `syntax-case` faz uma seleção entre padrões, associando uma expressão para cada um. Com `syntax-rules`, a expressão é a própria expansão, em `syntax-rules` a expansão é o resultado da expressão.

Sintaxe é um tipo especial de dados, assim como `s-expression` e os outros. O símbolo especial `#'` indica que a `s-expression` seguinte é na verdade uma sintaxe.

A definição abaixo ilustra todos estes pontos:

```
1 (define-syntax (my-let-3 x)
2   (syntax-case x ()
3     [(my-let-3 (var val) body)
4      #'((lambda (var) body) val)]))
```

Vamos examinar este código linha a linha:

1. O `x` na primeira linha é um pedaço da sintaxe que segue `my-let-3` e `(my-let-3 x)` é o padrão que deve ser reconhecido.

2. Declara o início da seleção, destacando que `x` é o trecho que determina a escolha.

3. A única regra é a que inicia nesta linha. Como estamos ilustrando essencialmente o mesmo macro, sua forma é igual à do `my-let-1`.

4. Esta linha é mais sutil. Note a presença do construtor de sintaxes (`#'`). Sem ele, a expressão seria executada e seu resultado seria armazenado como a expansão.

Deve ser possível perceber que pode-se definir `syntax-rules` como açúcar, expandindo para `syntax-case`.

13.3 Proteção (guarda)

A principal razão para usar `syntax-case` no lugar de `syntax-rules`, além da maior flexibilidade, é que existe uma forma adicional de escrever a regra de expansão, usando 3 termos ao invés de 2.

Com 2 expressões, a primeira é o padrão e a segunda a função que retorna a expansão. Com 3, a expressão do meio é um *predicado*: uma expressão que retorna verdadeiro se tudo está correto. Isto permite fazer uma verificação no padrão para saber se ele está condizente com o que se procura.

No nosso exemplo, `var` precisa ser um identificador, caso contrário a definição não faz sentido. No entanto, `identifier?` verifica trechos de sintaxe e devemos escrever `#'var` no lugar.

```
(define-syntax (my-let-4 x)
  (syntax-case x ()
    [(my-let-3 (var val) body)
     (identifier? #'var)
     #'((lambda (var) body) val)]))
```

13.4 Macros com várias opções

Em Racket e outras linguagens derivadas de Lisp é possível ter funções flexíveis. O `or`, por exemplo, pode ter dois *ou mais* argumentos. As operações lógicas poliádicas são essencialmente condicionais encaixados e podem ser implementadas por macros.

Entretanto, existem algumas sutilezas e é preciso ter bastante cuidado ao escrever os macros. Uma implementação ingênua de `or` pode ser esta:

```
(define-syntax (my-or-1 x)
  (syntax-case x ()
    [(my-or-1 e0 e1 ...)
     #'(if e0
           e0
           (my-or-1 e1 ...))]))
```

Se `e0` for verdadeiro, o resultado é ele mesmo, caso contrário, o resultado é a aplicação do macro no resto.

Esta definição não funciona, pois como sempre chamamos o macro com um parâmetro a menos, falta uma definição para `(my-or-1)`, sem argumentos. Isto é fácil de corrigir:

```
(define-syntax (my-or-2 x)
  (syntax-case x ()
    [(my-or-2) #'#f]
    [(my-or-2 e0 e1 ...)
     #'(if e0
           e0
           (my-or-2 e1 ...))]))
```

É importante colocar os casos mais específicos primeiro, pois a avaliação é sequencial.

Também é interessante cobrir o caso com apenas um argumento, poupando expansões e simplificando para o que vem a frente.

```
(define-syntax (my-or-3 x)
  (syntax-case x ()
    [(my-or-3) #'#f]
    [(my-or-3 e0) #'e0]
    [(my-or-3 e0 e1 ...)
     #'(if e0
           e0
           (my-or-3 e1 ...))]))
```

13.5 Protegendo a avaliação

Outro ponto importantíssimo é que macros são basicamente manipulação de código. Os problemas tradicionais que existem no preprocessador da linguagem C aparecem aqui também. Se uma expressão aparecer duplicada na expansão, sua avaliação também o será!

Verifique o resultado desta expressão:

```
(let ([init #f])
  (my-or-3 (begin (set! init (not init))
                  init)
           #f))
```

Para contornar o problema, em princípio basta avaliar `e0` e associar o valor a um identificador.

```
(define-syntax (my-or-4 x)
  (syntax-case x ()
    [(my-or-4)
     #'#f]
    [(my-or-4 e)
     #'e]
    [(my-or-4 e0 e1 ...)
     #'(let ([v e0])   ; salva a avaliação em v
         (if v
             v
             (my-or-4 e1 ...)))]))
```

Pode acontecer da avaliação ser feita no contexto errado ou não ser necessária. Note que fazemos a avaliação fora da expressão. Como diria Vinícius, é preciso ter cuidado.

13.6 Higiene

O que acontece neste caso?

```
(let ([v #t]) (my-or-4 #f v))
```

Usamos a mesmo identificador (`v`) que é usado na expansão! Veja como seria a expansão:

```
(let ([v #t]) (my-or-4 #f v))
                   ↓
(let ([v #t]) (let ([v #f]) (if v v (my-or-4 v))))
                                       ↓
(let ([v #t]) (let ([v #f]) (if v v     v)))
```

No entanto, o resultado gerado é o correto. Por se tratar de funções, o escopo dos identificadores é protegido naturalmente, evitando a promiscuidade.

13.7 Usando um identificador externo

Em algumas situações, macros higiênicos podem atrapalhar. Se por um lado um identificador definido dentro do macro não é confundido com o mesmo identificador definido fora, por outro ele mascara um nome que talvez seja necessário.

Um bom exemplo são as estruturas com autorreferência usando estado, como na definição de objetos:

```
(define os-1
  (object/self-1
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

Definindo `object/self-1` como um macro:

```
(define-syntax object/self-1
  (syntax-rules ()
    [(object [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name) ; self é uma associação vazia
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self                  ; arrumada aqui
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))                   ; pronta para usar
```

O `self` do `os-1` é mascarado pelo `self` do `object/self-1` quando não deveria.

Uma solução é "arrastar" o primeiro `self` para dentro da definição, pré-associando o `self`:

```
(define os-2
  (object/self-2 self
                 [first (x) (msg self 'second (+ x 1))]
                 [second (x) (+ x 1)]))

(define-syntax object/self-2
  (syntax-rules ()
    [(object self [mtd-name (var) val] ...)
     (let ([self (lambda (msg-name)
                   (lambda (v) (error 'object "nothing here")))])
       (begin
         (set! self
               (lambda (msg)
                 (case msg
                   [(mtd-name) (lambda (var) val)]
                   ...)))
         self))]))
```

O chato disso é que o usuário do macro precisa passar o nome na definição.

Uma forma de contornar o problema é usar o `with-syntax`, que é essencialmente um `let` para variáveis de sintaxe. Precisamos também poder transformar uma componente da expressão em sintaxe, o que é feito por `datum->syntax`.

```
(define-syntax (object/self-3 x)
  (syntax-case x ()
    [(object [mtd-name (var) val] ...)
     ; associa sintaticamente self com o self passado no argumento
     (with-syntax ([self (datum->syntax x 'self)])
       #'(let ([self (lambda (msg-name)
                       (lambda (v) (error 'object "nothing here")))])
           (begin
             (set! self
                   (lambda (msg-name)
                     (case msg-name
                       [(mtd-name) (lambda (var) val)]
                       ...)))
             self)))]))


(define os-3
  (object/self-3
   [first (x) (msg self 'second (+ x 1))]
   [second (x) (+ x 1)]))
```

13.8 Custos

Enquanto macros facilitam bastante a construção de linguagens e têm inúmeras aplicações em outras ferramentas, a inserção direta na linguagem central é em princípio muito mais eficiente.

Acontece que os compiladores conseguem reconhecer expressões e otimizá-las. No caso do `let`, a presença de um `lambda` que devolve um `lambda` que por sua vez deve ser imediatamente aplicado, pode ser facilmente identificado e trocado por uma extensão do ambiente.

14 Controle

Controle se refere a controle de fluxo, basicamente desvios e desvios condicionais. Nos casos extremos, existem coisas parecidas como `longjmp` e *exceptions*. Transferência de controle de uma instrução para seguinte sempre ocorre, o caso interessante é quando a transferência não é local.

Controle, neste sentido, não aumenta o poder computacional, apenas a expressividade.

14.1 Web

O controle em aplicativos distribuídos é particularmente interessante, pois existe uma dependência grande no protocolo. O `http` merece um estudo à parte.

Considere um servidor *web* que precisa interagir com o usuário. Normalmente um serviço *web* envia conteúdo estático (páginas ou arquivos), não há como receber informação. Isto faz sentido, pois o número de usuários é potencialmente infinito e nem todos completam as tarefas. Como não é possível determinar se a computação acabou, o protocolo não mantém estado.

Um programa que pergunta 2 números e apresenta sua soma é bastante simples se tivermos todo o contexto (e estado) disponível:

```
(define (read-number [prompt : string]) : number
  (begin
    (display prompt)
    (let ([v (read)])
      (if (s-exp-number? v)
          (s-exp->number v)
          (read-number prompt)))))

(display
 (+ (read-number "First number")
    (read-number "Second number")))
```

mas o servidor não tem como ficar esperando. A segunda parte do programa deve ser independente, ainda que ligada à primeira.

É fácil escrever a segunda parte como um λ:

```
(lambda (v1)
  (display
   (+ v1
      (read-number "Second number"))))
```

mas o resultado da primeira parte deve ser guardada em algum lugar.

14.1.1 Primeira solução

A comunicação entre cliente e servidor é feita por *CGI*[7], os famosos formulários (*forms*) presentes no HTML. Um formulário permite a entrada de diversos tipos de dados e seu envio por meio de uma *action*, associada a um botão especial (*submit*). Ainda assim, o servidor não tem como relacionar a solicitação recebida com a anterior.

Uma forma de resolver é usar uma tabela para armazenar resultados, associando uma entrada para cada iteração. Assim, cada resposta a uma requisição inclui a posição da tabela que contém os dados anteriores. A tabela pode ser um *hash* e as iterações podem ser numeradas automaticamente, com *labels* sucessivos:

```
;; retorna o próximo label disponível
(define new-label
  (let ([n (box 0)])
    (lambda ()
      (begin
        (set-box! n (+ 1 (unbox n)))
        (unbox n)))))

(define-type-alias label number)
(define table (make-hash empty))
```

Precisamos de uma versão especial do `read-number`, que armazena na tabela o que deve ser feito quanto (e se) o resultado chegar. Como ela irá parar na primeira parte do programa, chamaremos de `read-number/suspend`. Uma segunda função, `resume`, retoma a execução de onde parou.

```
(define (read-number/suspend [prompt : string] rest)
  (let ([g (new-label)])
    (begin
      (hash-set! table g rest)
      (display prompt)
      (display "␣To␣enter␣it,␣use␣the␣action␣field␣label␣")
      (display g))))

(define (resume [g : label] [n : number])
  ((some-v (hash-ref table g)) n))
```

Temos os dois passos da chamada *web*, `read-number/suspend` faz a primeira e `resume` é o tratamento da ação.

14.1.2 Sem estado

Uma solução é não manter a tabela indefinida, mas ter as funções prontas e pré-definidas. A requisição pode escolher apenas uma delas.

Ainda assim, temos o problema do argumento, mas este pode ficar armazenado no cliente (*cookies* e campos ocultos (*hidden*)).

[7] Common Gateway Interface

14.1.3 Interagindo com o estado

A diferença entre *cookies* e campos ocultos é que todas as páginas tem acesso aos *cookies* e os campos ocultos são locais.

Se usarmos o programa acima e retomarmos uma operação antiga, não há problema. Vejamos uma implementação com *cookies*.

```
(define cookie '-100)
(read-number/suspend "\nFirst␣number␣(cookie)"
                     (lambda (v1)
                       (begin
                         (set! cookie v1)
                         (read-number/suspend "\nSecond␣number␣(cookie)"
                                              (lambda (v2)
                                                (display
                                                 (+ cookie v2)))))))
```

Agora, tente as seguintes expressões e analise a resposta, com e sem estado:

```
(resume 1 3)
(resume 2 10)
(resume 2 15)
(resume 1 5)
(resume 3 10)
(resume 2 10)
```

14.2 Continuações (um estilo)

Esta estratégia de passar a informação para que a computação possa ser completada (ou *continuada*) é chamado de estilo de passagem de continuação, ou simplesmente "continuações".

Para tornar mais intuitivo, pense que qualquer computação é um (*lambda*) que recebe como argumento o procedimento que deve ser aplicado ao resultado da computação que é feita em seu corpo. Reveja o λ na página 54. Este argumento é a continuação.

O método é transformar funções de um argumento (não todas, apenas as que envolvem interação remota) em outra função com **dois** argumentos. O segundo argumento é uma função que executa o resto do processamento: a continuação. Um exemplo interessante é um jogo de xadrez por correio convencional, o jogador manda uma carta com sua jogada e o desenho atual do tabuleiro.

Esta transformação é geral, pode ser aplicada em qualquer programa e não altera a semântica, por isso é chamada de estilo. Por ser uma transformação da linguagem na própria linguagem, pode ser vista como uma espécie de açúcar (ou desaçúcar). Como consequência, o interpretador da linguagem pode também interpretar as continuações.

14.2.1 Implementação com açúcar

Para evitar confusão com Racket, vamos usar nomes diferentes para algumas coisas:

Nosso	Racket
with	let
rec	letrec
lam	lambda
cnd	if
set	set!
seq	begin

Para transformar em macro, cada expressão de interesse deve ser transformada em uma função com um argumento (a continuação). Os casos que trataremos são estes:

- `with`
- `rec`
- `lam`
- `cnd`
- `display`
- `read-number`
- `seq`
- `set`
- `quote`
- `app1`
- `app2`
- `atomic`

Os casos que envolvem constantes ou valores simples (`atomic` e `quote`) são os mais fáceis:

```
[(_ atomic)
  #'(lambda (k)
      (k atomic))]
```

esta é na verdade a forma geral. O valor `atomic` já é o próprio resultado. `k` é o que deve ser feito com este valor de forma a prosseguir com a computação.

Os casos `with` e `rec` já são macros e a tradução é direta, basta garantir que a expansão também seja transformada para o estilo de continuações:

```
[(_ (with (v e) b))
 #'(cps ((lam (v) b) e))]
[(_ (rec (v f) b))
 #'(cps (with (v (lam (arg) (error 'dummy "nothing")))
              (seq
               (set v f)
               b)))]
```

Para aplicação de funções, a coisa complica um pouco, pois elas podem ser criadas na própria linguagem. Assim, para este estudo, temos dois casos: funções com um argumento são criadas, as com 2 argumentos são pré-definidas e não envolvem interação.

Precisamos converter tanto a função quanto seu argumento, de modo encadeado, mas se fizermos a tradução direta de "`(f a)`", usando os passos:

1. calcular `f` usando continuações, o resultado será passado para o argumento `fv` da continuação
2. montar a continuação para a aplicação
3. nesta primeira continuação, calcular `a` usando continuações, o valor será passado no argumento `av`.
4. nesta segunda continuação, realizar a aplicação `(fv av)`
5. aplicar a continuação no resultado da aplicação: `(k (fv av))`

```
[(_ (f a))
   #'(lambda (k)
       ((cps f) (lambda (fv)
           ((cps a) (lambda (av)
                  (k (fv av)))))))]
```

teremos problemas se `fv` for uma *closure* que potencialmente dispara uma interação remota. Neste caso, como o que será chamado é a continuação de `fv` definida pelo cliente, o `k` se perdeu. A resposta correta é passar `k` para a aplicação:

```
[(_ (f a))
   #'(lambda (k)
       ((cps f) (lambda (fv)
           ((cps a) (lambda (av)
                  (fv av k)))))))]
```

Este problema não aparece com as funções internas de dois argumentos. Internas que são, não possuem interação remota[8]. A tradução pode ser feita da maneira direta:

```
    [(_ (f a b))
     #'(lambda (k)
         ((cps a) (lambda (av)
                    ((cps b) (lambda (bv)
                               (k (f av bv)))))))]
```

A definição de funções (declaração de *lambdas*) também tem uma sutileza:

```
  [(_ (lam (a) b))
     (identifier? #'a)
     #'(lambda (k)
         (k (lambda (a dyn-k)
              ((cps b) dyn-k))))] ; por que dyn-k e não k?
```

[8] É claro que podemos pensar em funções com mais de um argumento que precisam ser transformadas também, mas estamos nos limitando aos casos simples.

é importante que a continuação passada para o corpo `(cps b)` seja a passada no momento da execução, e não na hora da definição do *lambda*. Afinal de contas, queremos que a computação prossiga do ponto em que a função foi chamada, não do ponto em que foi criada.

Os outros casos estão escritos nas listagens e são relativamente diretos.

14.2.2 Implementação no núcleo

Para implementar no *core*, basta fazer `interp` trabalhar com continuações, isto é compatível com o fato de continuações ser um estilo de execução. É só construir um `interp/k` aplicando as transformações da seção anterior no `interp`. O único cuidado é que precisamos mudar o `Value`, já que agora `closV` passa a ter dois argumentos:

```
; Value muda, pois closV tem um lambda como segundo argumento
(define-type Value
  [numV  (n : number)]
  [closV (f : (Value (Value -> Value) -> Value))])
```

O novo interpretador fica assim:

```
; interp com continuações
(define (interp/k [expr : ExprC] [env : Env] [k : (Value -> Value)]) :
   Value
  (type-case ExprC expr
    [numC (n) (k (numV n))]
    [idC  (n) (k (lookup n env))]
    [appC (f a) (interp/k f env (lambda (fv)
                      (interp/k a env
                          (lambda (av) ((closV-f fv) av k)))))]
    [plusC (l r) (interp/k l env (lambda (lv)
                     (interp/k r env (lambda (rv) (k (num+ lv rv)))))))]
    [multC (l r) (interp/k l env
                     (lambda (lv) (interp/k r env
                                       (lambda (rv) (k (num* lv rv)))))))]
    [ifC (c s n) (interp/k c env
                      (lambda (cv)
                          (if (zero? (numV-n cv))
                              (interp/k n env (lambda (nv)  (k nv)))
                              (interp/k s env (lambda (sv)  (k sv)))))))
                                 ]
    [lamC  (a b) (k (closV (lambda (arg-val dyn-k)
                             (interp/k b
                                    (extend-env (bind a arg-val)
                                                env) dyn-k)))))]
    ))
```

Para mantermos um `interp`, podemos definir um padrão, que fornece um *environment* vazio e uma continuação identidade:

```
; Disparador
(define (interp [expr : ExprC]) : Value
  (interp/k expr mt-env (lambda (id) id)))

; Facilitador
(define (interpS [s : s-expression]) (interp (desugar (parse s))))

; Testes
(test (interp (plusC (numC 10) (appC (lamC '_ (numC 5)) (numC 10))))
              (numV 15))
(interpS '(+ 10 (call (func x (+ x x)) 16)))
```

14.3 Geradores

A noção de gerador discutida aqui é mais abrangente do que normalmente se encontra nas descrições de linguagens. Normalmente um *gerador* é uma função que retorna uma parte do resultado por vez, por exemplo, gerando uma lista elemento por elemento. Pense em um iterador do Java.

Em uma formulação mais geral, um gerador (*generator*) é uma função que quando é chamada novamente, recomeça da última posição onde parou. Isto naturalmente significa que a função deve poder sair antes do final, um "`return`" temporário. Em inglês isso é chamado de *yield* que não possui tradução direta em português, significa algo como "repassar", "dar a preferência" ou "abrir mão" ou "passar a vez": a função retorna o controle do processamento para quem chamou. Na próxima chamada, ou reentrada, o controle volta para o ponto onde o *yield* foi feito.

14.3.1 Variações de projetos

Em algumas linguagens um gerador é um objeto: iteradores do Java, já citados, são um exemplo. Em Python, um gerador possui o comando `yield` no seu corpo. Em Racket e outras linguagens similares, `yield` não é um comando, mas um procedimento passado como argumento[9]

Existem algumas coisas curiosas. Em algumas linguagens o nome do *yielder* é forçosamente *yield*. Em Python está pré-definido. Em Racket, o nome associado (*bound*) ao procedimento tem que ser `yield`. Outro ponto é se `yield` deve ser um comando ou um procedimento, a segunda opção é naturalmente mais flexível.

Uma última questão é como o gerador avisa que terminou seu processamento. Pode retornar um valor especial ou gerar um sinal (exceção).

14.3.2 Implementação

Com as diversas opções de projeto, devemos partir de algumas decisões:

1. Usaremos a linguagem com macros `CPS` (alguma dúvida que apareceriam continuações?)
2. Um gerador retoma sua execução com uma chamada direta, com os parâmetros necessários.

[9]Este tipo de procedimento é chamado de *applicable*, ou aplicável.

3. A operação de *yield* é feita por um procedimento, mas que terá um nome explícito, para simplificar os macros.

4. O final do processamento será indicado por uma exceção.

Um gerador possui dois comportamentos: na sua chamada e quando retorna o controle. As situações são estas e o gerador deve (note a dualidade):

Chamada	**Yield**
lembrar onde está quem chamou	saber para onde retornar
saber onde deve prosseguir	lembrar onde parou

Como o livro bem destaca, as ocorrências das palavras "onde" correspondem a continuações.

A entrada, no processador de macros, correspondente o gerador, deve ter esta cara:

```
[(_ (generator (yield) (v) b))
   (and (identifier? #'v) (identifier? #'yield))
     #'(lambda (k)
                (k valor do gerador))
```

Para preencher o "valor do gerador", é preciso entender o que ele deve fazer. Em primeiro lugar, deve receber dois argumentos, o *yielder* e a continuação.

Vejamos a tabela acima, na chamada é preciso lembrar para onde voltar na hora do *yield*: isto é fácil, o "onde voltar" nada mais é do que a continuação de quem chamou. Para lembrar, basta colocar no estado. Da mesma forma, o gerador deve armazenar a ponto de onde continuar no estado também.

O núcleo do gerador deve armazenar estes estados:

```
(lambda (v dyn-k)
        (begin
            (set! where-to-go dyn-k)
            (resumer v)))
```

O valor inicial de `where-to-go` não está definido, pois ele é a continuação de quem chamou.

O `resumer` é o ponto de onde o processamento deve prosseguir, em cada chamada ao gerador. Inicialmente é o próprio corpo do gerador, pois deve começar do começo! Quando não houver continuação para o `resumer`, ativamos um erro indicando o final do processamento.

Assim, o "valor do gerador" fica:

```
    (let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))])
         (letrec ([resumer (lambda (v)
                             ((cps b) ; faz b e depois gera erro
                                 (lambda (k)
                                    (error 'generator "ACABOU"))))]
                  [yield (lambda (v gen-k)
                           (begin
                            (set! resumer gen-k) ; resumer é a cont.
                            (where-to-go v)))])  ; chama resto com v
                 (lambda (v dyn-k)
                     (begin
                        (set! where-to-go dyn-k)
                          (resumer v)))
           ))
```

Este é o código final, a ordem das definições das expansões é muito importante:

```
(define-syntax (cps e)
  (syntax-case e (generator with rec lam cnd seq set quote)
    ; generator
    [(_ (generator (yield) (v) b)) ; yielder, valor, corpo
     (and (identifier? #'v) (identifier? #'yield))
     #'(lambda (k)
         (k (letrec (
                     [where-to-go (lambda (v) (error 'where-to-go
                                                "nothing"))]
                     [resumer (lambda (v)
                                ((cps b) (lambda (v)
                                           (error  'generator
                                             "passou␣dos␣limites"))))]
                     [yield (lambda (v gen-k)
                              (begin
                                (set! resumer gen-k)
                                (where-to-go v)))])
                (lambda (v dyn-k)
                  (begin
                    (set! where-to-go dyn-k)
                    (resumer v)))
                )))]
                          .
                          .
                          .
```

Um exemplo do uso de *generator* é este:

```
(run (cps (with (ns (generator (yield) (from)
                          (rec (f (lam (n) (seq
                                            (yield n)
                                            (f (+ n 1))))
                                  )
                               (f from))))
           (seq (ns 0) (ns 42)))))
```

14.4 Continuações e a pilha de execução

O formato com CPS ilustra de fato como é a execução com pilha. Vejamos o que significa a pilha de execução:

- marca o que falta ser feito na computação
- a pilha é um encadeamento de *frames*, cada continuação encabeça um encadeamento das continuações anteriores...

Uma aplicação de função escrita no estilo de continuações é assim (veja o código do macro, seção 14.2.1):

```
(lambda (k)
       ((cps f) (lambda (fv)
                        ((cps a) (lambda (av)
                                         (fv av k))))))]
```

cuja interpretação é esta:

1. `k` representa o estado da pilha antes da função ser chamada. `k` é a continuação
2. `lambda (fv)` é o início da avaliação de `f` e corresponde à criação de um *frame*, que referencia o restante da pilha (`k`). Lembre-se do `%ebp`.
3. de modo similar, a avaliação de `a` cria uma nova entrada na pilha, com referências para `k` e `fv`. Neste caso pode ser otimizado, pois basta ter o endereço de `fv` calculado e colocado em alguma posição temporária, talvez na própria pilha.

Um gerador usa uma pilha própria, pois tem que administrar a sua continuação além da continuação do programa principal. As manipulações de `where-to-go` e `resumer` indicam a troca das pilhas de execução.

14.5 Recursão de cauda

Nem sempre é realmente necessário criar uma nova entrada completa na pilha, como no exemplo de aplicação de função acima. Muitas vezes precisamos armazenar apenas um resultado temporário.

Quando a continuação de uma chamada é a mesma da que quem chamou, a pilha não precisa ser alterada, proporcionando a chance de uma otimização importante. Em linguagens capazes de reconhecer e fazer esta otimização, é possível criar comportamento iterativo apenas com recursão.

14.6 Recursão como uma característica da linguagem

Continuações são um estilo de execução, intimamente ligado a pilhas. Mas será possível usar continuações na própria linguagem e ganhar alguma coisa com isso?

Um dos pontos críticos é o uso da continuação na aplicação da função. Definimos (corretamente) que deve ser usada a continuação dinâmica,

```
[(_ (lam (a) b))
   (identifier? #'a)
   #'(lambda (k)
       (k (lambda (a dyn-k)
            ((cps b) dyn-k))))] ; dyn-k -> cont. da chamada
```

isto é, a continuação da *chamada* e não da *definição*. Se quisermos explorar a situação contrária, isto é, retornarmos a chamada ao ponto de definição, basta mudar o macro ou criar um novo.

```
[(_ (let/cc kont b))
  (identifier? #'kont)
  #'(lambda (k)
      (let ([kont (lambda (v dyn-k)
                    (k v))])
        ((cps b) k)))]
```

Neste exemplo, `kont` simplesmente aplica a sua continuação no seu argumento. `kont` pode ser usada na expressão descrita em `b`, cujo resultado será passado para a continuação *no momento da definição*. Note que `dyn-k` é descartado em favor de `k`[10].

Faça as seguintes expansões "na mão" e compare com o resultado no *DrRacket*:

```
(run (cps (let/cc esc 3)))
(run (cps (let/cc esc (esc 3))))
(run (cps (+ 1 (let/cc esc (esc 3)))))
(run (cps (let/cc esc (+ 2 (esc 3)))))
```

O último caso é o mais interessante, pois a continuação dinâmica é

```
(lambda (v k) (k (+ 2 v)))
```

ao passo que a continuação da definição é apenas `identity`.

Porque alguém faria uma coisa dessas? Pense, por exemplo, no tratamento de *exceções* em Java ou C++. Uma chamada a `throw` faz com que a computação atual seja cancelada e o controle passe para algum bloco *try-catch* mais acima na pilha.

[10] Se estas coisas ainda estiverem confusas, lembre que a aplicação de `(cps x)` a uma continuação `k`, faz com que `k` receba o resultado de `x`. Toda continuação é um *lambda*.

14.6.1 Continuações em Racket

Racket permite chamar um procedimento com a continuação atual (ou corrente), usando `call/cc` (*call with current continuation*). O argumento de `call/cc` é um procedimento que recebe um argumento, que é passado para a continuação, iniciando (ou continuando) com a chamada em cascata na pilha. Definir `let/cc` fica muito mais fácil:

```
(define-syntax let/cc
  (syntax-rules ()
    [(let/cc k b)
     (call/cc (lambda (k) b))]))
```

Com *let/cc* definido em termos de *call/cc*, fica mais simples definir um gerador, não há mais a necessidade de aplicar `cps` para passar a continuação, isto passa a ser feito automaticamente.

```
(define-syntax (generator e)
   (syntax-case e ()
    [(generator (yield) (v) b)
      #'(let ([where-to-go (lambda (v) (error 'where-to-go "nothing"))
         ])
           (letrec ([resumer (lambda (v)
                               (begin b
                                  (error 'generator "fell through")))]
                    [yield (lambda (v)
                             (let/cc gen-k
                               (begin
                                 (set! resumer gen-k)
                                 (where-to-go v))))])
          (lambda (v)
             (let/cc dyn-k
               (begin
                  (set! where-to-go dyn-k)
                  (resumer v))))))]))
```

Um exemplo de uso simples é, novamente, um contador:

```
(define g1 (generator (yield) (v)
               (letrec ([loop (lambda (n)
                                (begin
                                  (yield n)
                                  (loop (+ n 1))))])
                (loop v))))
```

e o que acontece neste outro exemplo?

```
(define g2 (generator (yield) (v)
               (letrec ([loop (lambda (n)
                                (loop (+ (yield n) n)))])
                (loop v))))
```

14.6.2 *Threads*

Uma *thread* é simplesmente uma linha de execução. A novidade é que podemos ter várias *threads* ativas e queremos passar o controle de uma para outra, com alguma forma de intercalação.

A troca do controle é feita por um escalonador, que podemos definir como um procedimento que recebe uma lista de funções (as *threads*), cada uma delas recebendo continuações.

Para a implementação de referência, usaremos *threads* cooperativas, isto é, cada uma passa o controle espontaneamente (não preemptivo), e o escalonador será do tipo *round-robin*: uma por vez e em ordem cíclica.

Um exemplo de uso seria este, suponha que "`d`" seja o "`display`" e "`y`" o "*yield*":

```
(scheduler-loop-0
(list
  (thread-0 (y) (d "t1-1␣") (y) (d "t1-2") (y) (d "t1-3␣"))
  (thread-0 (y) (d "t2-1␣") (y) (d "t2-2") (y) (d "t2-3␣"))
  (thread-0 (y) (d "t3-1␣") (y) (d "t3-2") (y) (d "t3-3␣"))))
```

Primeira tentativa O escalonador deve chamar cada *thread* com uma continuação que é um retorno para si. Deve iterar sobre as *threads* enquanto existirem *threads* sobrando. Note a construção da lista circular com `rest` e `append`.

```
(define (scheduler-loop-0 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
     (begin
       (let/cc after-thread ((first threads) after-thread))
       (scheduler-loop-0 (append (rest threads)
                                 (list (first threads)))))]))
```

A definição de *thread* tem alguns pontos em comum com os geradores, pois precisamos do *yielder* e do *resumer*. O *yielder* retorna para o escalonador (em sched-k) e a retomada deve ocorrer na continuação *dentro* da própria *thread* (thread-k).

O *yielder* precisa salvar a continuação da *thread* em algum ligar, que chamaremos de `thread-resumer` e retornar para a continuação mais recente do escalonador. Isto significa que cada chamada ao *yielder* precisa capturar a continuação do escalonador. Pode-se brincar com o estado, mas a forma mais direta é construir um *yielder* a cada retorno para a *thread*.

Da mesma forma dos geradores, a continuação original da *thread* é seu próprio corpo. O argumento da *thread* é a continuação do escalonador, sched-k, já que foi chamada por ele em modo de continuações.

Não existe valor de retorno para as *threads*, assim não importa o que devolvemos para o escalonador.

A implementação é essa:

```
(define-syntax thread-0
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...))]
              [yielder (lambda () (error 'yielder "vazio..."))])
       (lambda (sched-k)
         (begin
           (set! yielder
                 (lambda ()
                   (let/cc thread-k
                     (begin
                       (set! thread-resumer thread-k)
                       (sched-k 'qualquer-coisa)))))
           (thread-resumer 'tres))))]))
```

Veja que o corpo da *thread* é uma sequência de expressões — `(begin e ...)` — para permitir que sejam intercaladas chamadas do *yielder* com o prosseguimento da computação.

O *yielder* simplesmente salva a continuação dinâmica da *thread*, thread-k, em `thread-resumer` e chama a continuação do escalonador, sched-k, com um argumento qualquer.

No entanto, o código tem um problema sério! Onde está o controle de parada? Olhando o escalonador, percebe-se que cada *thread* é recolocada no final da fila sempre que retornar o controle.

Segunda solução É preciso notificar o escalonador sobre o estado da *thread*. É preciso diferenciar entre suspensa e terminada:

```
(define-type ThreadStatus
   [Tsuspended]
   [Tdone])
```

agora basta verificar `ThreadStatus` e, se a *thread* terminar, basta retirá-la da lista:

```
(define (scheduler-loop-1 threads)
  (cond
    [(empty? threads) 'done]
    [(cons? threads)
      (type-case ThreadStatus
                 (let/cc after-thread ((first threads) after-thread))
           [Tsuspended ()  (scheduler-loop-1 (append (rest threads)
                                                     (list (first threads))))]
           [Tdone () (scheduler-loop-1 (rest threads))])]))
```

Naturalmente, as *threads* devem retornar seu estado para o escalonador, já não podemos usar qualquer coisa. Para sinalizar o término, colocamos um *yielder* especial no final da *thread* (`finisher`).

```
(define-syntax thread-1
  (syntax-rules ()
    [(thread (yielder) b ...)
     (letrec ([thread-resumer (lambda (_)
                                (begin b ...
                                       (finisher)))]
              [finisher (lambda () (error 'finisher "nada por aqui"))]
              [yielder (lambda () (error 'yielder "vazio ..."))])
       (lambda (sched-k)
         (begin
           (set! finisher
                 (lambda ()
                   (let/cc thread-k
                     (sched-k (Tdone)))))
           (set! yielder
                 (lambda ()
                   (let/cc thread-k
                     (begin
                       (set! thread-resumer thread-k)
                       (sched-k (Tsuspended))))))
           (thread-resumer 'tres))))]))
```

15 Tipos

Na ciência em geral, a noção de invariantes é fundamental. Exemplos de invariantes são as leis de conservação da física e da química, teoremas e propriedades.

Em computação, expressões invariantes servem para demonstrar e verificar o funcionamento de programas. Todo laço, por exemplo, implica na existência de um invariante: a expressão condicional é sempre válida no início do bloco iterado.

Tipagem garante uma outra forma de invariante: a garantia de que um símbolo vai estar sempre associado a um tipo de dado.

15.1 Verificação de tipos estática

Em uma verificação de tipos estática, é possível capturar o erro em tempo de compilação, ou seja, não é necessário executar o programa para detectar o erro. Nesse sentido, o tratamento de tipos pode ser feito, em princípio, por uma análise puramente sintática.

Um tipo é uma representação de um conjunto de valores, um símbolo de um certo tipo só pode receber (ser associado) a valores deste conjunto.

Os símbolos podem ser anotados e verificados se podem ser usados em determinadas condições.

15.2 Uma visão clássica dos tipos

Começamos com uma versão tradicional de uma linguagem tipada.

15.2.1 Um verificador simples de tipos

A nossa linguagem de partida será a linguagem de *closures*, com funções de primeira classe, mas sem mutações, estado ou condicionais. As anotações de tipo serão feitas na entrada e saída de procedimentos. Os operadores "nativos" possuem tratamento de tipos implícito, ou podem ser substituídos por *lambdas*.

Precisamos definir um tipo para guardar os tipos da linguagem. No nosso caso tempos números e funções:

```
(define-type Type
   [numT]
   [funT (arg : Type) (ret : Type)])
```

A linguagem básica tem anotações para entrada e saída de funções:

```
(define-type TyExprC
  [numC  (n : number)]
  [idC   (s : symbol)]
  [plusC (l : TyExprC)   (r : TyExprC)]
  [multC (l : TyExprC)   (r : TyExprC)]
  [lamC  (arg : symbol)  (argT : Type) (retT : Type) (body : TyExprC)]
  [appC  (fun : TyExprC) (arg : TyExprC) ]
)
```

Precisamos verificar os diversos casos onde a tipagem é importante:

- Expressões que exigem número, como no caso dos operadores aritméticos
- O tipo de uma função deve ser `funT`
- O tipo do argumento da função aplicada deve ser o mesmo do declarado na definição da função

Com isso, poderíamos construir um verificador para responder se um programa está correto ou não, no que diz respeito à tipagem. Seria portanto uma função booleana, que precisará de um *environment* próprio, que associa identificadores a tipos. No entanto, um oráculo não é suficiente, pois em alguns casos precisamos deduzir o tipo de uma expressão. Por exemplo, para saber se a aplicação de uma função produz o tipo correto.

Com isso, nosso *type-checker* deve ter a seguinte forma:

```
(define (tc [expr : TyExprC] [tenv : TyEnv]) : Type
  (type-case TyExprC expr . . .
```

e precisamos analisar cada um dos possíveis casos para `TyExprC`:

- `numC`
- `idC`
- `plusC`
- `multC`
- `appC`

- `lamC`

O caso mais simples é `numC`, cujo tipo é naturalmente `numT`.
`idC` tem o tipo descrito no `tenv`. Se `idC` não estiver presente em `tenv`, temos um erro.
Adição e multiplicação devem verificar seus argumentos antes de garantir que o resultado é `numT`:

```
[plusC (l r) (let ([lt (tc l tenv)] [rt (tc r tenv)])
                  (if (and (equal? lt (numT))
                           (equal? rt (numT)))
                      (numT)
                      (error 'tc "+ not both numbers")))]
```

isso vale para qualquer função com dois argumentos numéricos. A avaliação de tipos não executa a expressão.

A aplicação precisa verificar se o tipo do argumento que a função recebe é o esperado. O tipo final é o tipo de retorno da função.

```
[appC (f a) (let ([ft (tc f tenv)]
                  [at (tc a tenv)])
               (cond
                 [(not (funT? ft))
                  (error 'tc "not a function")]
                 [(not (equal? (funT-arg ft) at))
                  (error 'tc "app arg mismatch")]
                 [else (funT-ret ft)]))]
```

A definição de função precisa garantir que o tipo do valor calculado pelo corpo é o mesmo declarado como retorno, mas antes é preciso colocar a associação do tipo do argumento em `tenv`.

```
[lamC (a argT retT b)
      (if (equal ? (tc b (extend-ty-env (Tbind a argT) tenv)) retT)
          (funT argT retT)
          (error 'tc "lam type mismatch"))]
```

E curioso notar a diferença entre a extensão do *environment* na avaliação e na verificação de tipos. No primeiro caso, o *environment* é estendido na aplicação, mas neste é na definição.

15.2.2 Verificando o tipo de condicionais

Condicionais implicam em diversas decisões sobre o projeto da linguagem, como já vimos. No que diz respeito a tipos as escolhas se tornam ainda mais importantes. Vejamos duas das mais significativas:

- Qual o tipo da expressão de teste? Algumas linguagens usam *booleano*, enquanto outras definem valores como representantes de verdade e outros de falso. Algumas linguagens podem ainda escolher "indefinido". Exemplos interessantes são *Perl*, *Python*, *C* e *Java*.

- Qual o tipo das expressões em ***then*** e ***else***?
  Algumas linguagens forçam que sejam do mesmo tipo, outras deixam os tipos independentes. Isto muda radicalmente tanto a tipagem como a linguagem.

```
[ifC (c s n) (let ([ct (tc c tenv)]
                   [st (tc s tenv)]
                   [nt (tc n tenv)])
               (cond
                [(not (numT? ct))
                 (error 'tc "Condition must be numeric")]
                [(not (equal? st nt))
                 (error 'tc "Both branches must be of the same type")]
                [else st]))]
```

15.2.3 Recursão no código

Não é tão simples como parece, se é que parece simples.

Primeira tentativa A recursão mais simples é o laço infinito. Para escrever apenas com funções é fácil

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

São dois termos idênticos, a notação histórica para cada um deles é ω e a aplicação é Ω. Neste caso em particular ω é `(lambda (x)(x x))` e Ω é a expressão acima.

Será que sempre acontece dos tipos das duas ocorrências de ω serem iguais? Neste caso a resposta é sim, pois ω é duplicado diretamente.

Seja γ o tipo de ω, vamos tentar especificá-lo:

- ω é uma função, portanto é algo da forma $\phi \to \psi$
- Como o argumento é ω, ϕ é o mesmo tipo: $(\phi \to \psi) \to \psi$
- Expandindo: $((\phi \to \psi) \to \psi) \to \psi$
- ..., não tem fim!

Isto significa que γ não pode ser descrito por uma cadeia finita e o nosso sistema de tipagem não consegue determinar o tipo de Ω. Na verdade, não consegue determinar o tipo de nenhuma expressão potencialmente recursiva.

Quando o sistema de tipos garante que todo tipo é determinado depois de uma série finita de passos, dizemos que ele possui **normalização forte**. Com ela, não podemos escrever programas infinitos: a semântica da linguagem muda!

Colocando tipo na recursão Quando implementamos continuações com açúcar sintático (14.2.1), usamos o `rec` (nossa versão do `letrec`). Agora, vamos deixar o `rec` como parte da linguagem central.

Para simplificar a exposição, vamos usar uma gramática concreta. Um programa possível é este, onde `num` é uma declaração de tipo.

```
(rec (Σ num (n num)
                    (if0 n
                         0
                         (n + (Σ (n - 1)))))
     (Σ 10))
```

Aqui vemos que pela aplicação de Σ seu tipo deve ser `num→num`. Se olharmos a definição, fica claro que o tipo é o mesmo. Quebramos a maldição da definição recursiva.

15.2.4 Recursão de dados

Dados recursivos podem ser criados dentro do programa. Por enquanto só temos os tipos fixos.

Definições de tipos de dados Definir um tipo de dado significa fornecer três operações:

- Criar um novo tipo.
- Permitir que as instâncias deste tipo tenham campos.
- Permitir que alguns campos possam se referir a instâncias do mesmo tipo.
  - Consequentemente, precisamos permitir bases não recursivas para o tipo.

Estes critérios definem *tipos algébricos*. Sua aplicação fica clara no seguinte exemplo, de uma árvore binária, retirada do livro:

```
(define-type BTnum
  [BTmt]
  [BTnd (n : number) (l : BTnum) (r : BTnum)])
```

Foi preciso criar o tipo `BTnum`, com referências a subtipos `BTmt` e `BTnd`. Para que haja utilidade, o tipo deve permitir campos (`n`, `l` e `r`).

Mas, de onde vem `BTnum`? Já deve estar definido na linguagem.

Os novos tipos O exemplo acima adiciona novos tipos a partir do `BTnum`:

```
; construtores
BTmt : -> BTnum
BTnd :  number * BTnum * BTnum -> BTnum
; predicados
BTmt? : BTnum -> boolean
BTnd? : BTnum -> boolean
; seletores
BTnd-n : BTnum -> number
BTnd-l : BTnum -> BTnum
BTnd-r : BTnum -> BTnum
```

Há alguns fatos importantes:

- Os construtores geram `BTnum`.

- Os predicados consomem `BTnum`
- Os seletores só trabalham com a variante correta, isto é, não tratam `BTmt`. Não dá para escrever isso em um sistema de tipos estático, por falta de um tipo estático adequado. A avaliação deve ser dinâmica.

Reconhecimento de padrões e "desaçucarando" Se olharmos o que foi feito com código, podemos usar um tratamento parecido. Isto é, para tratar o tipo `BTnum`, ao invés de usar um `type-case` como este:

```
(type-case BTnum t
   [BTmt () e1]
   [BTnd (nv lt rt) e2])
```

podemos simular a associação com `let`, onde `t` é o tipo:

```
(cond
   [(BTmt? t) e1]
   [(BTnd? t) (let ([nv (BTnd-n t)]
                    [lt (BTnd-l t)]
                    [rt (BTnd-r t)])
                e2)])
```

Ou seja, podemos usar um macro e portanto, *desugar*.

Porém o macro precisa da definição dos seletores de `BTnd`, que estão na linguagem central. Além disso, *e1* e *e2* só podem ser verificados após a expansão do macro. A expansão depende das definições de tipos e a definição de tipos dependem da expansão.

15.2.5 Tipos, tempo e espaço

Um sistema de tipos evidentemente melhora o desempenho de programas, pois muitos testes são estáticos, isto é, feitos em tempo de compilação e não de execução. No entanto, o programador deve convencer o sistema de tipos estático que o programa está correto.

Quanto a espaço, acontece o mesmo. Uma vez determinado estaticamente qual o tipo, não há necessidade de armazenar identificadores de tipos.

Exceto para os tipos com variantes, como `BTmt?` e `BTnd?`, que são usados em tempo de execução. Mesmo assim, alguma otimização é possível:

```
(cond
   [(BTmt? t) e1]
   [(BTnd? t) (let ([nv (BTnd-n/no-check t)]
                    [lt (BTnd-l/no-check t)]
                    [rt (BTnd-r/no-check t)])
                e2)])
```

Muito menos informação precisa ser armazenada para os seletores (1 bit neste caso, pois só há duas possibilidades).

15.2.6 Mutação

O exemplo abaixo mostra uma situação em que a tipagem é mais sutil:

```
(let ([x 10])
     (begin
       (set! x 5)
       (set! x "Ai␣caramba")))
```

Qual o tipo de `x`? Veja que a tipagem pode mudar em tempo de execução, basta trocar o `begin` por um condicional...

Para poder prosseguir com a tipagem, os verificadores exigem que o tipo da variável nunca mude. Ou trate de uma forma mais especial (como?).

15.2.7 Robustez de tipos

Um sistema de tipos é normalmente a combinação de três componentes:

1. Uma linguagem de tipos
2. Um conjunto de regras para os tipos
3. Um algoritmo de verificação e aplicação

O sistema de tipos deve garantir alguma propriedade interessante para todos os programas. Um verificador estático ocorre (obviamente) antes da execução, portanto faz uma *predição*.

Para garantir esta predição é preciso haver um teorema que a prove. Um sistema de tipos é naturalmente bastante suspeito, pois difere em muitos aspectos de um avaliador, que é a parte que realmente executa o programa:

Verificador	Avaliador
Vê apenas o texto	Trabalha com *stores*
Associa *ids* a tipos	Associa *ids* a valores ou locais
Transforma conjuntos de valores em tipos	Trata os valores individualmente
A verificação sempre termina	A execução pode não terminar
Passa pelo programa uma única vez	Execução passa de zero a infinitas

Dada a expressão e, seu tipo é t e seu valor calculado é v. Precisamos provar que o tipo de v é t. Isto é robustez.

A forma padrão de demonstrar é por duas partes: *progresso* e *preservação*, que são repetidas até a demonstração completa:

Progresso Se um termo passa pela verificação de tipo, então pode ser avaliado.

Preservação O resultado final é do tipo previsto.

Porém, ai porém, há dois casos diferentes:

- Um programa pode não terminar. O teorema não se aplica. *Mas* podemos demonstrar que cada passo é válido, mesmo não terminando, o tempo todo estará produzindo passos com significado correto.
- Uma linguagem suficientemente rica tem propriedades que não podem ser verificadas estaticamente. Exemplos são ponteiros perdidos ou acesso além dos limites de um vetor. Assim, está implícito nos teoremas de verificação de robustez de tipos um conjunto pré-definido de erros que podem ocorrer.

16 Material extra: Prolog

Durante a disciplina vimos a implementação de linguagens com base funcional, construímos orientação a objetos e até mesmo demos suporte para linguagens imperativas.

Uma forma diferente de escrever programas e que se adapta muito bem a problemas de lógica é **Prolog**. Nesta parte final vamos ver rapidamente como funciona esta linguagem.

16.1 Estrutura básica

Todos os programas em Prolog são formados por *cláusulas*, que por sua vez são *termos* seguidos de um ponto.

As cláusulas podem ser de três tipos:

fatos são declarações sempre verdadeiras.

regras são verdadeiras se certas condições são atingidas.

perguntas usadas para verificar se um certo objetivo é verdadeiro.

Vejamos um exemplo de programa em Prolog:

```
filho(tadeu,roberto).     /* fato */
filho(roberto,bruno).     /* fato */
filha(carla, maria).      /* fato */
avo(X,Y) :- filho(Z,Y),filho(X,Z). /* regra */
?- avo(tadeu, bruno).     /* pergunta */
```

16.1.1 Fatos

Os fatos são triplas de valores, com a seguinte forma:

```
predicado(argumento 1, argumento 2).
```

no primeiro exemplo acima, `filho` é o predicado, `tadeu` é o primeiro argumento e `roberto` é o segundo. `filho`, `tadeu` e `roberto`, enquanto símbolos, são *átomos*.

16.1.2 Regras

Uma regra é composta de uma *cabeça* e um *corpo*.

A cabeça é separada do corpo pelo sinal `:-` e essencialmente é um fato onde pelo menos um dos argumentos é uma *variável livre*. Variáveis livres começam com letras maiúsculas e não possuem valor pré-definido.

O corpo consiste em um conjunto de *subobjetivos* separados por '`,`' ou '`;`'. Vírgulas atuam como conjunção '*e*' e ponto e vírgula como '*ou*'.

No exemplo acima, a regra "avo" é válida se existe alguma tripla (x_1, y_1, z_1) de valores para (X, Y, Z) tal que valha:

- `filho(`z_1`,`y_1`)`
- `filho(`x_1`,`z_1`)`

note que a tripla não precisa ser única.

As regras podem ser recursivas e o predicado pode aparecer mais de uma vez:

```
ancestral(A,B) :- pai(A,B).
ancestral(A,B) :- pai(P,B), ancestral(A,P).
```

Para indicar negação em uma regra, usa-se o operador `\+`, por exemplo

```
alto(P) :- \+ baixo(P).
```

16.1.3 Perguntas

As perguntas verificam se a regra é satisfeita ou se um fato está declarado. As perguntas começam com `?-` .

No caso da verificação de regras, há uma busca (varredura) recursiva, usando *backtrack*.

16.1.4 Exemplo completo

Fatorial em prolog. Veja que a associação numérica é feita com o operador `is`.

```
fatorial(0,1).

fatorial(A,B) :-
            A > 0,
            C is A-1,
            fatorial(C,D),
            B is A*D.
```

16.2 Composições

Como esta é uma introdução rápida, só vou apresentar os elementos necessários para prosseguirmos e para ficar fácil acompanhar os diversos tutoriais disponíveis *online*.

16.2.1 Functors

É possível a criação de termos compostos por meio de *functors*, na forma de uma árvore, que representam estruturas mais complexas. O formato de um *functor* é parecido com um *fato*, mas pode ter qualquer número de argumentos, onde cada um deles é um também *functor*.

Vejamos um exemplo:

```
sentença(nome(josé), pred(verb(joga), obj(truco))).
```

O número de argumentos é chamado de *aridade*. Existe um predicado especial `functor(Termo,F,A)` que é bem sucedido se `Termo` for o functor `F` e possui aridade `A`:

```
?- functor(sentença(nome(josé), pred(verb(joga), obj(X))),F,A).
F = sentença,
A = 2.
```

Normalmente, functors são identificados com a aridade após um "/", por exemplo `sentença/2`.

16.2.2 Listas

Naturalmente iriam aparecer listas, alguma dúvida?

A lista vazia é representada por `[]`. O operador de criação, equivalente ao `cons` do *Racket*, possui representações diferentes dependendo da variante de prolog; enquanto o formato tradicional seja um ponto, o mais usado é o functor `[|]` com aridade 2. Tal como no `cons`, para que seja uma lista, o segundo argumento deve ser uma lista.

```
?- functor([a,b,c],F,A).
F = '[|]',
A = 2.
```

Tente entender o resultado desta consulta.

16.3 Unificação

A operação fundamental do Prolog é a *unificação*. Em uma pergunta, a unificação tenta instanciar cada variável livre com valores de forma a validar a regra. O algoritmo (simplificado) de unificação é o seguinte:

Entrada: dois termos T_1 e T_2.

1. Se T_1 e T_2 forem átomos ou números, retorne sucesso se forem iguais e falso caso contrário.
2. Se um dos dois for uma variável, instancie seu valor com o outro e retorne sucesso.
3. Se ambos forem termos compostos (functors) com a mesma aridade:
   (a) Se o functor principal (mais externo) de T_1 não for igual ao functor principal de T_2, retorne falso.
   (b) Sendo o mesmo, tente unificar os argumentos de cada um na ordem em que aparecem. Retorne falso se alguma unificação falhar e sucesso caso contrário.
4. Retorne falso.

O operador `=` executa uma unificação explicitamente.

16.4 Podas

Prolog se baseia em *backtracking*, o que pode ser muito caro em termos de processamento. No entanto, é possível determinar podas para encerrar o processamento prematuramente, por exemplo, quando se quer apenas a primeira unificação ou quando ficar claro que não vale a pena prosseguir. A poda é um subobjetivo especial identific ado por um ponto de exclamação.

```
minimo(X,Y,X) :-
    X < Y,
    !.
% Se X já for menor do que Y, não faz sentido usar a regra abaixo,
% pois sabemos que não existe outra solução possível
minimo(X,Y,Y) :-
    Y < X.
```

16.5 Exemplos completos

Este exemplo permite calcular todas as triplas pitagóricas com valores menores do que um N dado. Usa duas regras pré-definidas: *length* serve para garantir que N é um número e *between* verifica se o terceiro argumento está entre o primeiro e o segundo.

```
/* Triplas pitagóricas com valores menores do que N */
pythag(X,Y,Z,N) :-
   length(_, N),
   between(1,N,X),
   between(1,N,Y),
   between(1,N,Z),
   Z*Z =:= X*X + Y*Y.
```

Este é um exemplo bem mais complexo, colocado aqui para diversão.

```
/**************************************************
Problema atribuído a Einstein :

Em uma rua há cinco casas, uma de cada cor.

Em cada casa mora uma pessoa de uma nacionalidade diferente.  Cada um
dos moradores toma uma bebida distinta, fuma cigarros de uma marca
diferente e possui um animal de estimação único.

Queremos saber quem possui o peixe.

Existem 15 dicas:

 1. O inglês vive na casa vermelha.
 2. O sueco ama cachorros.
 3. O dinamarquês bebe chá.
 4. A casa verde é vizinha à esquerda da casa branca.
 5. O dono da casa verde bebe café.
 6. A pessoa que fuma  Pall Mall cria pássaros.
 7. O dono da casa amarela fuma Dunhill.
 8. O homem que mora da casa central bebe leite.
 9. O norueguês mora na primeira casa.
10. O fumante de Blends é vizinho do criador de gatos.
11. O homem que cuida de cavalos pé vizinho do fumante de Dunhill.
12. O fumante de  Blue Master bebe cerveja.
13. O alemão fuma Prince.
14. O norueguês mora ao lado da casa azul.
15. O fumante de  Blends tem um vizinho que bebe água.
**************************************************/
```

segue na próxima página...

```
/* Lista vazia */
persons(0, []) :- !.

/* Lista de N elementos */
persons(N,[(_Dono,_Cor,_Bebe,_Fuma,_Pet)|T]):-N1 is N-1,persons(N1,T).

/* Pega o N-ésimo elemento  (H é head, T é tail) */
person(1, [H|_], H) :- !.
person(N, [_|T], R) :- N1 is N-1, person(N1, T, R).

/* Dicas */
%  1. O inglês vive na casa vermelha.
hint1([(inglês, vermelha,_, _, _)|_]).
hint1([_|T]) :- hint1(T).

%  2. O sueco ama cachorros.
hint2([(sueco,_,_,_,_,cachorro)|_]).
hint2([_|T]) :- hint2(T).

%  3. O dinamarquês bebe chá.
hint3([(dinamarquês,_,chá,_,_)|_]).
hint3([_|T]) :- hint3(T).

%  4. A casa verde é vizinha à esquerda da casa branca.
hint4([(_,verde,_,_,_,_),(_,branca,_,_,_,_)|_]).
hint4([_|T]) :- hint4(T).

%  5. O dono da casa verde bebe café.
hint5([(_,verde,café,_,_)|_]).
hint5([_|T]) :- hint5(T).

%  6. A pessoa que fuma  Pall Mall cria pássaros.
hint6([(_,_,_,pallmall,pássaro)|_]).
hint6([_|T]) :- hint6(T).

%  7. O dono da casa amarela fuma Dunhill.
hint7([(_,amarela,_,dunhill,_)|_]).
hint7([_|T]) :- hint7(T).

%  8. O homem que mora da casa central bebe leite.
hint8(Persons) :- person(3, Persons, (_,_,leite,_,_)).

%  9. O norueguês mora na primeira casa.
hint9(Persons) :- person(1, Persons, (norueguês,_,_,_,_,_)).
```

segue...

```
% 10. O fumante de Blends é vizinho do criador de gatos.
hint10([(_,_,_,blends,_),(_,_,_,_,gato)|_]).
hint10([(_,_,_,_,gato),(_,_,_,blends,_)|_]).
hint10([_|T]) :- hint10(T).

% 11. O homem que cuida de cavalos pé vizinho do fumante de Dunhill.
hint11([(_,_,_,dunhill,_),(_,_,_,_,cavalo)|_]).
hint11([(_,_,_,_,cavalo),(_,_,_,dunhill,_)|_]).
hint11([_|T]) :- hint11(T).

% 12. O fumante de  Blue Master bebe cerveja.
hint12([(_,_,cerveja,bluemaster,_)|_]).
hint12([_|T]) :- hint12(T).

% 13. O alemão fuma Prince.
hint13([(alemão,_,_,prince,_)|_]).
hint13([_|T]) :- hint13(T).

% 14. O norueguês mora ao lado da casa azul.
hint14([(norueguês,_,_,_,_),(_,azul,_,_,_)|_]).
hint14([(_,azul,_,_,_),(norueguês,_,_,_,_)|_]).
hint14([_|T]) :- hint14(T).

% 15. O fumante de  Blends tem um vizinho que bebe água.
hint15([(_,_,_,blends,_),(_,_,água,_,_)|_]).
hint15([(_,_,água,_,_),(_,_,_,blends,_)|_]).
hint15([_|T]) :- hint15(T).

% quem tem o peixe?
question([(_,_,_,_,peixe)|_]).
question([_|T]) :- question(T).
```

o final está na página seguinte.

```
/* O conjunto de soluções é dada por: */

solution(Persons) :-
  persons(5, Persons),          % cria 5 pessoas
  hint1(Persons),               % verifica as dicas
  hint2(Persons),
  hint3(Persons),
  hint4(Persons),
  hint5(Persons),
  hint6(Persons),
  hint7(Persons),
  hint8(Persons),
  hint9(Persons),
  hint10(Persons),
  hint11(Persons),
  hint12(Persons),
  hint13(Persons),
  hint14(Persons),
  hint15(Persons),
  question(Persons).            % verifica a pergunta

%% é só chamar solution(P)
```