2.4.4. Resolução de alguns problemas de redução:

1. Redução direta:

(λx.x(xy))N-> N(Ny)

aqui N é substituído nos dois x, pois x está livre na subespressão x(xy).

2. Redução direta também:

(λx.y)N -> y

3. Menos trivial:

(λx.(λy.xy)N)M -> (λy.My)N

pois x está livre em (λy.xy)N

-> MN

4. Exemplo simples que não termina:

(λx.xx)(λx.xx) -> (λx.xx)(λx.xx)

-> (λx.xx)(λx.xx) ...etc

5. Exemplo catastrofal:

```
(λx.xxy)(λx.xxy)-> (λx.xxy)(λx.xxy)y
                -> (λx.xxy)(λx.xxy)yy
                -> (λx.xxy)(λx.xxy)yyy
                -> (λx.xxy)(λx.xxy)yyyy
```

6. Exemplo que pode ser simples

```
(λx.z)((λx.xxy)((λx.xxy)) -> z
```

7. O mesmo exemplo, quando feito de forma **errada**:

```
(λx.z)((λx.xxy)(λx.xxy)) ->(λx.z)((λx.xxy)(λx.xxy)y)
                         ->(λx.z)((λx.xxy)(λx.xxy)yy)
```

Aqui, ao invés de reduzir o redex mais à esquerda, foi aplicado $(\lambda x.xxy)$ sobre $(\lambda x.xxy)$

2.4.5. Resolução do problema recursivo com a série de Fibonacci:

Relembrando: Recursão é um conceio fundamental em matemática e ciência da computação. Uma definição simples de recursão é a de que um programa recursivo é um que chama a si mesmo. UMa função recursiva é uma que é definida em função de si mesma, daí o termo recursivo.

Os números de Fibonacci são uma relação de recorrência muito conhecida e utilizada na matemática. Os números de Fibonacci são definidos da seguinte forma:

$$F_N = F_{N-1} + F_{n-2} \text{ para } N >= 2, \text{ com } F_0 = F_1 = 1$$

definindo a série:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233..

para os números:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12..

e o algoritmo é bem simples:

```
fibonacci ( n )
(
   se n <= 1
      retorne 1
   senão
      retorne ( fibonacci (n-1) + fibonacci (n-2) )
)
```

isto, expresso em uma função recursiva simples, fica:

```
FIBO = (λn.IF (<= n 1) 1 (+ (FIBO(- n 1))(FIBO(- n 2))))
```

- Da mesma forma como fizemos com o fatorial FAC, aplicando uma abstração-β sobre FIBO, podemos transformar esta definição em:

FIBO = (λfibo.(λn. (...fibo...fibo...)))FIBO

- Esta definição podemos escrever então também da forma:

FIBO = H FIBO

- Usando Y:

FIBO = Y H

H=(λfibo.λn.IF (<= n 1) 1 (+ (fibo(- n 1))(fibo(- n 2))))

2.5. A Semântica Denotacional do Cálculo Lambda

- ☞ **Denotar** (Aurélio): (do latim *denotare*)
  Significar, exprimir, simbolizar

Há duas maneiras de se olhar para uma função:

- como um algoritmo que irá produzir um valor, dado um argumento, ou
- como um conjunto de pares ordenados argumento-valor.

☞ O primeiro enfoque é dinâmico ou **operacional**, já que vê uma função como uma seqüência de operações no tempo.

☞ O segundo enfoque é estático ou **denotacional**: a função é encarada como um conjunto fixo de associações entre argumentos e seus valores de função correspondentes.

Nos capitulos anteriores vimos como uma expressão ser avaliada pel aaplicação repetida de regras de redução.

Essas regras descrevem somente transformações ***sintáticas*** em expressões permitidas, sem fazer referência a o que essas funções ***significam***. O cálculo λ pode ser considerado como um **sistema formal para a manipulação de símbolos sintáticos**.

O desenvolvimento das regras de conversão foi baseado em nossas intuições sobre funções abstratas e nos proveu uma semântica operacional para o cálculo λ.

Ficou em aberto, uma definição de significado.

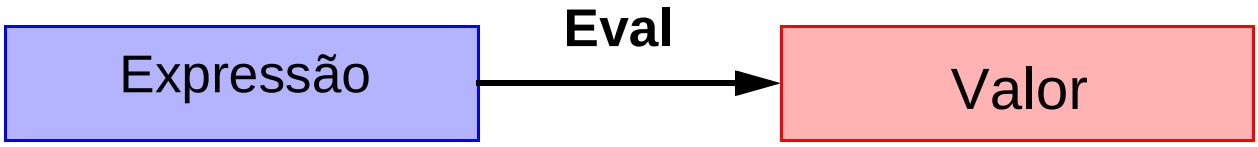
2.5.1. A Função `Eval`

O propósito da semântica denotacional de uma linguagem qualquer é o de ***atribuir um valor a toda expressão nesta linguagem***.

Uma expressão é um ***objeto sintático***, formado de acordo com as *regras de sintaxe desta linguagem*, o que não diz **nada** sobre o seu significado.

Um valor, em contraste, é um ***objeto matemático abstrato*** , como "o número 5" ou "a função que quadra o seu argumento".

Nós podemos, por conseguinte, expressar a semântica de uma linguagem como uma função matemática Eval, de expressões para valores:



Podemos agora escrever equações da seguinte forma:

$$\mathbf{Eval[[}\ +\ 3\ 4\ \mathbf{]]} = 7$$

Isto diz "o significado (i.é. o valor) da expressão (+ 3 4) é o valor numérico abstrato 7".

Usamos colchetes duplos em negrito para fechar o argumento a Eval para enfatizar que é um objeto sintático. Esta convenção é muito utilizada na descrição de outras semânticas denotacionais.

Consideraremos a expressão **(+ 3 4)** como uma ***representação*** ou ***denotação*** do valor **7**.

- Daí o termo ***semântica denotacional***.

Agora, um breve desenvovlimento informal da função Eval:

- O objetivo da função é o de provêr um valor para **Eval[[** E **]]** para toda e qualquer expressão lambda E.
- Para atingir este objetivo podemos utilizar todos os recursos da sintaxe do cálculo lambda já vistos até agora.

Supponhamos, primeiramente, que E seja uma variável *x*.
Que valor deveria **Eval[[** x **]]** possuir ?

Como o valor de uma variável é dado pelo seu contexto circundante, não podemos dizer o seu valor isoladamente.

Podemos solucionar este problema, dando a **Eval[[ ]]** um parâmetro extra ρ, o qual representa a informação contextual de uma variável. O argumento ρ é chamado de ***ambiente*** (*environment*) e é uma função que mapeia nomes de variáveis para seus valores.

Assim: **Eval[[** x **]]** ρ = ρ x

A notação (ρ x), no lado direito significa “a função ρ aplicada ao argumento x”.

Para tratar aplicações, usamos um raciocínio semelhante: É razoável dizer-se que o o valor de (E_1 E_2) deveri ser o valor de E_1 aplicado ao valor de E_2.

Eval[[E_1 E_2 **]]** ρ = (**Eval[[** E_1 **]]** ρ) (**Eval[[** E_2 **]]** ρ)

O caso final é o de uma abstração lambda **Eval[[** λ**x . E]]** ρ. Certamente será uma função, de forma que nós o podemos definir completamente dando o seu valor quando aplicada a um argumento arbitrário a:

(**Eval[[** λ**x . E]]** ρ) a

Resumindo:

- O valor de uma abstração lambda, aplicada a um argumento, é o valor do corpo da abstração, em um contexto onde o parâmetro formal está atado ao argumento.
- Formalmente: **Eval[[** λ**x . E]]** ρ a = **Eval[[E]]** ρ **[**x=a**]**
  onde a notação x=a significa "a função ρ extendida com a informação de que a variável x está atada ao valor a".
- De forma mais precisa:
  ρ **[**x=a**]x = a**
  ρ **[**x=a**]y =** ρ **y**

  caso **y** seja uma variável diferente de **x.**

A parte de constantes e funções embutidas, as quais necessitam de um tratamento individual que veremos na seção 2.5.3, foi provida nesta seção uma semântica denotacional simples para o cálculo lambda.

Resumo geral:

Eval[[k **]]** ρ = **vide seção 2.5.3**
Eval[[x **]]** ρ = ρ x
Eval[[E_1 E_2 **]]** ρ = (**Eval[[** E_1 **]]** ρ) (**Eval[[** E_2 **]]** ρ)
Eval[[λ**x.E]]** ρ a **= Eval[[E]]** ρ **[**x=a**]**
onde:
k é uma constante ou função embutida**,**
x é uma variável**,**
E, E_1, E_2 são expressões-lambda**.**

Nota sobre a notação:

Como vimos, o ambiente r é um argumento essencial para Eval. Em todas a sutilizações de Eval, ρ tem um papel crucial e é sempre necessário. Por um aquestão de simplicidade, costuma-se omitir o ambiente ρ ao escrever Eval.

Portanto:

Eval[[E_1 **]]** ρ = **Eval[[** E_2 **]]** ρ

significa a mesma coisa que:

Eval[[E_1 **]]** = **Eval[[** E_2 **]]**

2.5.2. O Símbolo ⊥

Uma das características mais úteis da teoria descrita nesta seção, é que ela nos permite raciocinar a respeito da terminação ou não terminação de programas.

- **Descrição da semântica de expressões que não atingem a forma normal**: Como foi descrito antes, a redução de uma expressão pode não atingir uma forma normal.

- Para descrever o valor de uma expressão dessas, incluímos o elemento ⊥, pronunciado “fundo”, no domínio dos valores de expressões, o qual é o **valor de uma expressão sem uma forma normal**:

$$\textbf{Eval[[}\langle\text{expressão sem forma normal}\rangle\textbf{]]} = \bot$$

- ⊥ possui um significado matemático perfeitamente respeitável na Teoria dos Domínios.
- À semelhança do símbolo 0 (que também está para “nada”), seu uso muitas vezes nos permite escrever equações sucintas ao invés de montes de palavras obtusas.
- Ao invés de dizermos “a evaluação da expressao E falha em terminar”, dizemos: $\textbf{Eval[[} E \textbf{]]} = \bot$

2.5.3. Definindo a Semântica de Funções Embutidas e Constantes

Nesta seção veremos como definir o valor de **Eval[[** k **]]** onde k é uma constante ou função embutida.

Exemplo: Qual é o valor de **Eval[[** * **]]** ?

Certamente é uma função de dois argumentos e nós podemos definí-la dando o valor desta função quando aplicada a argumentos arbitrários:

Eval[[$*$ **]]** a b = a x b

que dá o valor do cálculo lambda * em termos da operação matemática de multiplicação **x.**

A distinção entre $*$ e **x** é crucial:

- $*$ é uma construção sintática do cálculo lambda.
- x é uma operação matemática abstrata.

No caso da multiplicação a notação matemática x difere da notação de programa $*$. No caso da adição, por exemplo, o símbolo + é usado em ambas. É importante notar-se a diferença.

A equação anterior é, porém, uma especificação incompleta para $*$. Temos de definir o que $*$ faz para todo argumento possível, inclusive $\perp$. O conjunto completo de equações ficaria então:

Eval[[$*$ **]]** a b = a x b, caso a $\neq \perp$ e b $\neq \perp$

Eval[[$*$ **]]** $\perp$ b = $\perp$

Eval[[$*$ **]]** a $\perp$ = $\perp$

As duas novas equações complementam a definição de $*$, especificando que, se algum dos argumentos de $*$ falha em terminar, então da mesma forma o fará a aplicação de $*$.

Uma outra forma, mais "inteligente" de se definir a multiplicação seria através de um operador de multiplicação #:

Eval[[#]] a b = a x b, caso a $\neq \perp$ e b $\neq \perp$ e a $\neq 0$

Eval[[#]] 0 b = 0

Eval[[#]] $\perp$ b = $\perp$

Eval[[#]] a $\perp$ = $\perp$

- Estas equações implicam que # deveria primeira avaliar o seu primeiro argumento e, caso este retorne 0, retornar o valor zero, sem examinar o segundo argumento de todo.
- O uso de # ao invés de $*$ poderia levar à avaliação de algumas expressões que de outra forma não terminariam.

O ponto a ser observado nos exemplos acima é o de que o uso de equações "semânticas" para funções embutidas nos permite expressar variações sutis no seu comportamento, o que é muito difícil de se realizar somente através de regras de redução.

☞ As equações semânticas para uma função ao mesmo tempo especificam o significado da função e implicam seu comportamento operacional (regras de redução).

2.5.4. Estriticidade

Dizemos que uma função é estrita se temos certeza de que vamos necessitar do valor de seu argumento.

- Caso uma função **f** com certeza necessite do valor de seu argumento e a avaliação do argumento não termina, então a aplicação de **f** ao argumento vai com certeza não terminar.

Esta descrição leva a uma definição concisa de estriticidade:

> Uma função f é estrita se e somente se f ⊥ = ⊥.

Esta definição pode ser generalizada da mesma forma para funções de muitos argumentos.

Por exemplo: se g é uma função de três argumentos, então g é estrita em seu segundo argumento se e somente se:

g a ⊥ **c** = ⊥

2.5.5. Fim:

Nestas aulas foi dada uma visão, apenas superficial, do cálculo lambda. O objetivo foi só mostrar o “que está por trás” da implementação de liguagens funcionais, já que não se pretende nesta disciplina ensinar o uso do cálculo lambda para a criação de novas linguagens funcionais.

2.5.6. LISP:

- Início próxima aula. Uma sugestão para leitura é:
  Oakey, Steve; ***LISP para Micros***. Editora Campus, 1986
  ISBN 85-7001-326-4