2.2. A Semântica Operacional do Cálculo Lambda

Até agora foi descrita a sintaxe do cálculo-λ. Para chamá-lo de "cálculo", devemos porém dizer como "calcular" com ele.

Basicamente isto é realizado através de três regras de conversão, que descrevem como converter uma expressão-λ em outra.

2.2.1. Introdução à conversão: Variáveis *atadas* e *livres (bound/free)*

Consideremos a expressão-λ: **(λx. + x y) 4**

Para avaliar esta expressão necessitamos:

- saber o valor "global" de **y**.
- não necessitamos saber o valor global de x, pois é o parâmetro formal da função.
- Assim vemos que: **x** e **y** possuem um status bastante diferente.

A razão é que **x** ocorre ***atado*** pelo λ**x**, é somente um encaixe dentro do qual o argumento **4** é colocado quando a abstração-λ for aplicada ao argumento.

Por outr lado, **y** não é atado por nenhum λ e assim ocorre ***livre*** na expressão.

A ocorrência de uma variável é atada se há uma expressão-λ envolvente que a amarra, senão é livre. No exemplo a seguir, x e y ocorrem atados, z porém, ocorre livre:

λx. + ((λy. + y z) 7) x

Observe que os termos atado e livre se referem a ocorrências específicas da variável em uma expressão.

Uma variável pode possuir tanto uma ocorrência atada como uma livre em uma expressão. Considere o exemplo:

+ x ((λx. + x 1) 4)

Aqui **x** ocorre livre (a primeira vez) e atada (a segunda). Cada ocorrência individual de uma variável deve ser ou atada ou livre.

As definições formais de livre a atado são dadas abaixo:

Definição de *ocorre livre:*

x ocorre livre em x (mas não em outra variável ou constante qualquer)

x ocorre livre em (**E F**) $\Leftrightarrow$	**x** ocorre livre em **E** ou **x** ocorre livre em **F**
x ocorre livre em λ**y.E** $\Leftrightarrow$	**x** e **y** são variáveis diferentes e **x** ocorre livre em **E**

Nota: nenhuma variável oorre atada em uma expressão consistindo e uma única constante.

Definição de *ocorre atada:*

x ocorre atada em (**E F**) $\Leftrightarrow$	**x** ocorre atada em **E** ou **x** ocorre atada em **F**
x ocorre atada em λ**y**.**E** $\Leftrightarrow$	(**x** e **y** são a mesma variável e **x** ocorre livre em **E**) ou x ocorre atada em **E**

2.2.2. Conversão-Beta

Uma abstração-λ denota uma função. Assim necessitamos descrever como aplicá-la a um argumento.

A função **(λx. + x 1) 4** é a justaposição do função **(λx. + x y)** e do argumento **4** e, por conseguinte, denota a aplicação de uma certa função, denotada pela abstração-λ, ao argumento **4**.

A regra para a aplicação de uma função é muito simples:

- O resultado da aplicação de uma abstração-λ um argumento é uma instância do corpo da abstração-λ no qual ocorrências (livres) do parâmetro formal no corpo são repostos pelo argumento.

O resultado de se aplicar a abstração ao argumento é: **+ 4 1** onde o **(+ 4 1)** é um instância do corpo **(+ x 1)** no qual ocorrências do parâmetro formal **x** foram repostas pelo argumento **4**.

Nós escrevemos as conversões utilizando o símbolo $\rightarrow$ como antes:

$$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow +\ 4\ 1$$

Esta operação é chamada de β-redução e a sua implementação eficiente é um aspecto importante na elaboração de linguagens de programação funcional.

Exemplos simpes de β-redução:

- O parâmetro formal pode ocorrer várias vezes no corpo:

$$(\lambda x.\ +\ x\ x) \rightarrow +\ 5\ 5 \rightarrow 10$$

Da mesma forma, poderá não haver ocorreêncuias do parâmetro formal no corpo: $(\lambda x.\ 3)\ 5 \rightarrow 3$

Nesse caso não há ocorrências do parâmetro formal **(x)**, pelo qual o argumento **5** poderia ser substituído. Assim o argumento é descartado sem uso.

O corpo de uma abstração λ poderá ser constituído por outra abstração λ:

$$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow (\lambda y.\ -\ y\ 4)\ 5 \rightarrow -\ 5\ 4 \rightarrow 1$$

Observe-se que, ao se construir a instância do corpo da abstração λ**x**, o corpo inteiro inclusive a abstração λ**y** envolvida por esta, substituindo-se **x**, é copiado.

Aqui se observa também o efeito do Currying: a aplicação da abstração λ**x** retornou uma função (a abstração λ**y**) como seu resultado. Esta por sua vez, aplicada resultou em **(- 5 4)**.

Funções encapsuladas umas dentro das outras são comumente abreviadas:

$$(\lambda x.\ (\lambda y.\ E)) \quad \textit{para} \quad (\lambda x.\lambda y.\ E)$$

Funções podem ser argumentos também:

$$(\lambda f.\ f\ 3)(\lambda x.\ +\ x\ 1) \rightarrow (\lambda x.\ +\ x\ 1)\ 3 \rightarrow +\ 3\ 1$$

$\rightarrow$ 4

Uma instância da abstração λ**x** é substituída por **f** onde quer que **f** apareça no corpo da abstração de λ**f**.

Nomenclatura:

Nomes de parâmetros formais podem não ser únicos:

(λx.(λx. + (- x 1)) x 3) 9 $\rightarrow$ **(λx. + (- x 1)) 9 3**

$\rightarrow$ **+ (- 9 1) 3**

$\rightarrow$ **11**

Observe-se que o **x** interno não foi substituído na primeira redução, pois estava protegido pelo λ**x** envolvente, ou seja, a ocorrência interna de **x não** é livre no corpo da abstração λ**x** externa.

Dada uma asbtração-λ, **(λx.E)**, é possível identificar-se exatamente as ocorrências de **x** que deveriam ser substitídas atraves da identificação de todas as ocorrências de **x** que estão livres.

Dessa forma, para o exemplo acima, examinamos o corpo da abstração:

(λx. + (- x 1)) x 3

e vemos que a segunda ocorrência de x é livre, podendo ser substituída.

A idéia do aninhamento do escopo de variáveis em uma linguagem de programação estruturada é análoga a esta regra.

Outro exemplo:

(λx.λy.+ x ((λx.- x 3) y)) 5 6

$\rightarrow$ **(λy.+ 5 ((λx.- x 3) y)) 6**

$\rightarrow$ **+ 5 ((λx.- x 3) 6)**

$\rightarrow$ **+ 5 (- 6 3)**

$\rightarrow$ **8**

Novamente, o **X** mais interno não é substituído, uma vez que não é livre no corpo da abstração mais externa.

Exemplo maior: Modelagem de Construtores de Dados

Definimos CONS, HEAD e TAIL da seguinte forma:

$$\begin{aligned} \textbf{CONS} &= (\lambda a.\lambda b.\lambda f.f\ a\ b) \\ \textbf{HEAD} &= (\lambda c.c\ (\lambda a.\lambda b.a)) \\ \textbf{TAIL} &= (\lambda c.c\ (\lambda a.\lambda b.b)) \end{aligned}$$

as fórmulas acima obedecem às regras para CONS, HEAD e TAIL definidas anteriormente (Seção 2.1.3). Exemplo:

$$\begin{aligned} \textbf{HEAD (CONS p q)} &= (\lambda c.c\ (\lambda a.\lambda b.a))(\textbf{CONS p q}) \\ &\rightarrow \textbf{CONS p q}\ (\lambda a.\lambda b.a) \\ &= (\lambda a.\lambda b.\lambda f.f\ a\ b)\ p\ q\ (\lambda a.\lambda b.a) \\ &\rightarrow (\lambda b.\lambda f.f\ p\ b)q\ (\lambda a.\lambda b.a) \\ &\rightarrow (\lambda f.f\ p\ q)(\lambda a.\lambda b.a) \\ &\rightarrow (\lambda a.\lambda b.a)p\ q \\ &\rightarrow (\lambda b\ p)\ q \\ &\rightarrow p \end{aligned}$$

Isto significa:

- Não existe necessidade real para as funções embutidas HEAD e CONS ou TAIL.
- Todas as funções embutidas podem ser modeladas como abstrações-lambda.
- De um ponto de vista teórico, isto é satisfatório, para efeitos práticos porém, não é utilizado (eficiência).

2.2.3. Conversão-Alfa

Considere:

$$(\lambda x.\ +\ x\ 1) \quad e \quad (\lambda y.\ +\ y\ 1)$$

Evidentemente elas devem ser equivalentes. A conversão-α é o nome dado à operação de mudança de nome (consistente) de um parâmetro formal.

Notação:

$$(\lambda x.\ +\ x\ 1) \quad \underset{\alpha}{\leftrightarrow} \quad (\lambda y.\ +\ y\ 1)$$

α-congruência: duas expressões-λ **M** e **N** são α-congruentes (ou α-iguais), denotado por **M** $\cong$ **N** se ou **M** $\equiv$ **N** ou **M** $\underset{\alpha}{\leftrightarrow}$ **N**, ou **N** é obtido de **M** através da reposição de uma subx-pressão **S** de **M** por uma expressão-λ **T** tal que **S** $\underset{\alpha}{\leftrightarrow}$ **T**, ou existe alguma expressão-λ **R** tal que **M** $\cong$ **R** e **R** $\cong$ **N**.

2.2.4. Conversão-Eta

Sejam: **(λx. + 1 x)** e **(+ 1)**. Estas expressões se comportam exatamente da mesma maneira, quando aplicadas a um argumento: **ambas adicionam 1 ao argumento**.

Conversão-η é o nome dado à regra que expressa essa equivalência:

$$(\lambda x.\ +\ 1\ x) \quad \underset{\eta}{\leftrightarrow} \quad (+\ 1)$$

De forma mais geral, a regra da conversão-η pode ser expressa assim:

$$(\lambda x.\ F\ x) \quad \underset{\eta}{\leftrightarrow} \quad F$$

desde que **x** não ocorra livre em **F** e **F** denote uma função.

A condição de que **x** não deve ocorrer livre em em **F** previneconversões errôneas. Exemplo:

(λx. + x x) não é η-convertível para **(+ x)**

pois **x** ocorre livre em **(+ x)**.

A condição de que **F** deve denotar uma função previne outras conversões errôneas envovlendo constantes embutidas (predefinidas). EXemplo:

TRUE não é η-convertível para **(λx. TRUE x)**

Quando a conversão-η é utilizada da esquerda para a direita, é chamada de **redução-η**.

2.2.5. Provas de Interconvertibilidade

- Freqüentemente será necessário poder-se demonstrar a interconvertibilidade de duas expressões-λ.

- Quando as duas expressões denotam funções, esta prova pode se tornar extremamente longa.

Há porém, métodos para abreviar provas, que não sacrificam o seu rigor. **Exemplo**:

$$\textbf{IF TRUE } ((\lambda \textbf{ p.p) 3)} \quad \text{e} \quad \textbf{(}\lambda\textbf{x. 3)}$$

Ambas as expressões denotam a mesma função, que invariavelmente retorna o valor **3**, não importa o valor de seu argumento e seria de se esperar que ambas sejam interconvertíveis.

Isto realmente ocorre:

$$\begin{aligned} \textbf{IF TRUE } ((\lambda \textbf{ p.p) 3)} &\underset{\beta}{\leftrightarrow} \textbf{IF TRUE 3} \\ &\underset{\eta}{\leftrightarrow} \textbf{(}\lambda\textbf{x. IF TRUE 3 x)} \\ &\rightarrow \textbf{(}\lambda\textbf{x. 3)} \end{aligned}$$

onde o passo final é a regra de redução para **IF**.

Um método alternativo de se provar a interconvertibilidade de duas expressões, muito mais conveniente, é o de se aplicar as duas expressões a um argumento **w**, como:

$$\begin{aligned} &\textbf{IF TRUE } ((\lambda \textbf{ p.p) 3) w} \\ &\rightarrow \textbf{(}\lambda\textbf{ p.p) 3} \\ &\rightarrow \textbf{3} \end{aligned} \qquad \begin{aligned} &\textbf{(}\lambda\textbf{x. 3) w} \\ &\rightarrow \textbf{3} \end{aligned}$$

Portanto: $\textbf{IF TRUE } ((\lambda \textbf{ p.p) 3)} \leftrightarrow \textbf{(}\lambda\textbf{x. 3)}$

Esta prova tem a vantagem de somente utilizar a redução e de evitar o uso explícito da conversão-η.

O raciocínio por detrás da generalização da interconvertibilidade acima segue abaixo. Suponha que possamos demonstrar que:

$$\mathbf{F_1\ w} \rightarrow \mathbf{E}$$

e

$$\mathbf{F_2\ w} \rightarrow \mathbf{E}$$

onde **w** é uma variável que não ocorre livre tampouco em $\mathbf{F_1}$ como em $\mathbf{F_2}$, e **E** é alguma expressão, então podemos raciocinar da seguinte forma:

$$
\begin{aligned}
\mathbf{F_1} &\underset{\eta}{\leftrightarrow} && (\lambda \mathbf{w.\ F_1\ w}) \\
&\leftrightarrow && (\lambda \mathbf{w.\ E}) \\
&\leftrightarrow && (\lambda \mathbf{w.\ F_2\ w}) \\
&\underset{\eta}{\leftrightarrow} && \mathbf{F_2}
\end{aligned}
$$

e por conseguinte $\mathbf{F_1} \leftrightarrow \mathbf{F_2}$.

Não é sempre que expressões-λ que "deveriam" dizer a mesma coisa são interconvertíveis. Isto será tratado mais tarde.

2.2.6. Sumário: Regras de Conversão

Há três regras de conversão que possibilitam a interconversão de expressões envolvendo abstrações-λ:

1. ***Mudança de nome***: a conversão-α permite que se troque o nome de parâmetros formais de uma abstração-λ , penquanto isto for feito de forma consistente.

2. ***Aplicação de funções***: A redução-β permite a aplicação de abstrações-λ a um argumentoatravés de geração de uma nova instância do corpo da abstração, substituindo o argumento por ocorrências livres do parâmetro formal.
   Cuidado especial deve ser tomado quando o argumento contém variáveis livres.

3. ***Eliminação de abstrações-λ redundantes***: A redução-η pode as vezes eliminar uma abstração-λ.

Dentro deste contexto, podemos considerar as funções embutidas (predefinidas) como mais uma forma de conversão. Esta forma de conversão recebe o nome de **conversão-δ**.

2.3. Ordem de Redução

Se uma expressão não contém mais redexes, então a avaliação está completa. Um expressão nesta forma é dita estar na forma normal.
Assim, a avaliação de uma expressão consiste na na redução sucessiva de redexes, até que a expressão esteja na f**orma normal.**
Definição (forma normal): uma expressão-l é dita estar na forma normal, se nenhum redex-b, isto é, nenhuma subexpressão da forma **(λx.P)Q** ocorre nela.
Uma expressão pode conter mais do que um redex, assim ***a redução pode acontecer por caminhos diferentes***.

- Exemplo:

```
(+ (* 3 4) (* 7 8))
```

pode ser reduzido à forma normal pelas seguintes seqüências:

```
(+ (* 3 4) (* 7 8))
 -> (+ 12 (* 7 8))
    -> (+ 12 56)
       -> 68
```

ou

```
(+ (* 3 4) (* 7 8))
 -> (+ (* 3 4) 56)
    -> ( 12 56)
       -> 68
```

- Nem toda expressão possue uma forma normal, considere:

```
(D D)
```

- onde D é (λx.x x).
  A avaliação desta expressão jamais terminaria, uma vez que (D D) reduz para (D D):

```
(λx.x x)(λx.x x) -> (λx.x x)(λx.x x)
                 -> (λx.x x)(λx.x x)
```

- Por conseguinte, ***algumas*** seqüências de redução poderão atingir a forma normal, outras não.
  Considere:

```
(λx.3) (D D)
```

²Se nós primeiro reduzirmos a aplicação de `(λx.3)` a `(D D)` (sem avaliar `(D D)`), obteremos o resultado 3; porém se primeiro reduzimos a aplicação de D sobre D, nós simplemente obtemos `(D D)` novamente.

- Se nós continuamos insistindo em escolher a aplicação de D sobre D, a avaliação vai continuar indefinidamente, não terminando.
- Esta situação corresponde à de um programa imperativo que entra em um laço infinito. Dizemos que a avaliação ou execução **não termina**.
- Para contornar esta situação, existe a **ordem normal de redução**, que garante que uma forma normal, caso exista, será encontrada.

2.3.1. Ordem Normal de Redução

- Uma questão que se coloca, além da questão "poderei encontrar uma forma normal ?", é a questão de se existe mais de uma forma normal para para uma expressão, ou:

  Poderá uma seqüência de redução diferente levar a uma forma normal diferente?

  Ambas as questões estão interligadas.

- A resposta para a última pergunta é: **não**.
  Isto é uma conseqüência dos teoremas de Church-Rosser CRT1 e CRT2.

Teorema de Church-Rosser 1 (CRT1):

Se $E_1 <-> E_2$, então existe uma expressão **E**, tal que $E_1 \rightarrow E$ e $E_2 \rightarrow E$

Corolário: Nenhuma expressão pode ser convertida em duas formas normais distintas. Isto significa que não existem duas formas normais para uma expressão que não sejam a-convertíveis entre si.

Prova: Suponha que $E_1 <-> E$ e $E_2 <-> E$, onde E_1 e E_2 estão na forma normal.

Então $E_1 <-> E_2$ e, pelo CRT1, deve haver uma expressão `F` tal que $E_1 \rightarrow F$ e $E_2 \rightarrow F$.

Como tanto E_1 como E_2 não possuem redexes, logo $E_1 = F = E_2$

- Informalmente, o teorema CRT1 diz que todas as seqüências de redução que terminam, haverão de atingir o mesmo resultao.

  O segundo teorema de Church-Rosser, CRT2, diz respeito a uma ordem particular de redução, chamada **ordem normal de redução**.

Teorema de Church-Rosser 2 (CRT2):

Se $E_1 \rightarrow E_2$, e E_2 está na forma normal, então existe uma ***ordem normal*** de seqüência de redução de E_1 para E_2.

Conseqüências:

- Existe no mínimo um resultado possível e
- a ordem normal de redução encontrará este resultado, caso ele exista.
- Observe que nenhuma seqüência de redução poderá levar a um resultado incorreto, o máximo que poderá acontecer é a não-terminação.

A Ordem Normal de Redução especifica que o redex ***mais à esquerda mais externo*** deverá ser reduzido primeiro.

- No exemplo anterior `(λx.3) (D D)`, escolheríamos o redex-λx primeiro, não o (D D).

- Esta regra encorpa a intuição de que argumentos para funções podem ser descartados, de forma que deveríamos aplicar a função `(λx.3)` primeiro, ao invés de primeiro avaliarmos o argumento `(D D)`.

2.3.2. Ordens de redução ótimas

- Enquanto a ordem normal de redução garante que ou uma solução seja encontrada ou o não-término ocorra, ela não garante que isto ocorra no número mínimo de passos de redução, o que é um fato relevante na utilização do cálculo-l para a implementação de linguagens de programação interpretadas.
- No caso da implementação da resolução através de redução de grafos - que estudaremos mais tarde - porém, aparentemente a ordem normal de redução é "geralmente ótima" em questão de tempo de execução.

  Analisar uma expressão para encontrar o redex ótimo para ser reduzido é aparentemente muito mais trabalhoso e gasta mais tempo do que arriscar uma redução com mais passos seguindo "cegamente" a ordem normal.
- Alguns autores como (*Levy, J.J.; Optimal Reductions in the Lambda Calculus in Essays on Combinatory Logic, Academic Press, 1980*) se ocuparam de algoritmos para encontrar ordens ótimas ou quase-ótimas de redução que preservassem as qualidades da ordem normal de redução. Isto é assunto de um tópico avançado que não cabe nesta disciplina.

2.4. Funções Recursivas

- Se o propósito da utilização de cálculo lambda é a conversão de programas funcionais neste para que possamos resolvê-los, devemos ser capazes de representar uma das características mais marcantes das linguagens funcionais, a **recursão**.
- O cálculo lambda suporta a representação de recursividade sem maiores extensões através da utilização de alguns pequenos truques.

2.4.1. Funções Recursivas e o Y

- Considere a definição da função fatorial abaixo:

  FAC = (λn.IF (= n 0) 1 (* n (FAC (- n 1))))
- Esta definição baseia-se na capacididade de se dar um nome a uma abstração lambda e de fazer referência a esta dentro dela mesma.
- Nenhuma construção deste tipo é provida pelo cálculo lambda.
- O maior problema aqui é que funções lambda são ***anônimas***.
- Dessa forma elas não podem nomear-se e assim não podem se referenciar a si mesmas.

Para resolver este problema iniciamos com um caso onde encontramos a recursão na sua forma mais pura:

FAC = (λn. ... FAC ...)

- Aplicando uma abstração-b sobre FAC, podemos transformar esta definição em:

  FAC = (λfac.(λn. (...fac...)))FAC
- Esta definição podemos escrever da forma:

  FAC = H FAC (2.1)
- onde:

```
H = (λfac.(λn. (...fac...)))
```

- A definição de H é trivial.
  É uma abstração lambda ordinária e não usa recursão.
- A recursão é expressa somente pela definição 2.1.

A definição 2.1 pode ser encarada mais como uma equação matemática. Por exemplo, para resolver a equação matemática

$$x^2 - 2 = x$$

nós procuramos por valores de x que satisfazem a equação (neste caso x = -1 e x = 2) .

De forma similar, para resolver 2.1, nós procuramos uma expressão lambda para FAC que satisfaça 2.1.

A equação 2.1 **FAC = H FAC** postula que, quando a função H é aplicada a **FAC**, o resultado é **FAC**.

Nós dizemos que **FAC** é um ***ponto fixo*** de H.

Uma função pode ter mais de um ponto fixo. Por exemplo, na função abaixo, tanto 0 quanto 1 são pontos fixos da função:

```
λx. * x x
```

a qual calcula o quadrado de seu argumento.

Em resumo, estamos procurando por um ponto fixo para H. Claramente isto depende somente de H.

Para isto, invente-se, a título provisório, uma função Y, a qual toma uma função como argumento e devolve o seu ponto fixo como resultado.

Assim Y tem o comportamento de:

```
Y H = H (Y H)
```

e Y é chamado de ***combinador de ponto fixo***.

Assim, caso possamos produzir um Y, temos uma solução para o problema da recursividade. Para 2.1 podemos prover a seguinte solução:

```
FAC = Y H
```

a qual é uma definição não-recursiva de FAC.

Como teste para esta estratégia, podemos computar o valor de **(FAC 1).** Tomemos as definições de **FAC** e **H**:

```
FAC = Y H

H = (λfac.λn. IF (= n 0) 1 (* n (fac (- n 1))))
```

Assim:
```
FAC 1
-> Y H 1
-> H (Y H) 1
->(lfac.ln.IF (= n 0) 1 (* n (fac (- n 1))))(Y H) 1
->(ln.IF (= n 0) 1 (* n (Y H (- n 1)))) 1
-> IF (= 1 0) 1 (* 1 (Y H (- 1 1)))
-> * 1 (Y H 0)
= * 1 (H (Y H) 0)
= * 1 ((lfac.ln.IF (= n 0) 1 (* n (fac (- n 1))))(Y H) 0)
-> * 1 ((ln.IF (= n 0) 1 (* n (Y H(- n 1)))) 0)
```

```
-> * 1 (IF (= 0 0) 1 (* n (Y H(- 0 1))))
-> * 1 1
-> 1
```

2.4.2. Y pode ser definido como uma Abstração Lambda

- Para a transformação de uma expressão recursiva em uma não-recursiva, utilizamos o combinador de ponto fixo, uma função que chamamos de Y.
- A propriedade que Y necessita ter é: **Y H = H (Y H)**
- e isto representa evidentemente a recursão da forma mais pura, uma vez que esta fórmula pode ser utilizada para representar, de forma abstrata, qualquer fórmul arecursiva que queiramos.
- O truque é que podemos representar Y como uma abstração lambda sem utilizar a recursão:

Y = (λh. (λx.h (x x)) (λx. h (x x)))

Para demonstrar que Y possui a qualidade desejada, evaluemos:

```
Y H
 = (λh. (λx.h (x x)) (λx. h (x x))) H
 <-> (λx.H (x x)) (λx. H (x x))
 <-> H ((λx.H (x x)) (λx. H (x x)))
 <-> H (Y H)
```

- O fato de Y poder ser definido como uma abstração lambda é, do ponto de vista matemático, realmente digno de nota.
- Do ponto de vista da implementação, é relativamente ineficiente implementar Y como uma abstração lambda. Para a construção de linguagens funcionais, utiliza-se geralmente uma função embutida com a regra de redução:

Y H -> H (Y H)

2.4.3. Exercícios (entregar próxima aula)

I. **Redução**:

Reduza as seguintes expressões-lambda às suas respectivas formas normais:

(((λf.λx.λy.(x)(f)y)p)q)r

(((λx.λy.λz.(y)x)(x)y)(u)z)y

(λx.(λy.(x)(y)y) λz.(x)(z)(λu.λv.u)w

(((λx.λy.λz.((x)z)(y)z)(λu.λv.u)w)λs.s)t

(((λx.(λz.(x)(y)y)ly.(x)(y)y)λz.λu.λv.(u)(z)v)(λr.λs.r)t)w

(λx.x(xy))N

(λx.y)N

(λx.(λy.xy)N)M

(λx.xx)(λx.xx)

(λx.xxy)(λx.xxy)

(λx.z)((λx.xxy)((λx.xxy))

II. **Recursividade**:

1. Dê uma definição recursiva para o maior divisor comum de dois inteiros e calcule o valor de **((mdc) 10) 14)** utilizando o combinador Y.
2. Tente o mesmo para os números de Fibonacci e use Y para computar **(Fibo) 5**.