Programação Funcional
Cálculo Lambda - Aula Nº 3

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)

  •Se nós primeiro reduzirmos a aplicação de **(**λ**x.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 $E_1 -> E$ e $E_2 -> E$

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 $E_1 -> F$ e $E_2 -> F$.

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 $E_1 -> E_2$, 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**.