



**Curso:** Engenharia Informática

**Disciplina:** Linguagens de Programação

**Professores:** Teresa Gonçalves e João Leitão

# Interpretador Para Cálculo Lambda

*Primeiro Trabalho Prático*

**Trabalho Elaborado Por:**

Luis Polha Nº 20464

Marlene Oliveira Nº 25999

# Introdução

No âmbito da disciplina de Linguagens de Programação foi-nos solicitada a elaboração de um interpretador de cálculo lambda, utilizando a linguagem de programação C ou Java. Para elaborar o nosso interpretador foi utilizada a linguagem C (em conjunto com os analisadores sintáticos Flex/Bison).

Formulado inicialmente por Alonzo Church em 1936, o cálculo lambda (também designado por lambda cálculo) consiste num sistema matemático que permite explicar alguns conceitos de linguagens de programação de um modo simples e puro. A primeira palavra que constitui o nome deste sistema formal provém do uso da letra grega  $\lambda$  e a palavra “cálculo” provém do modo como a redução pode ser utilizada para computar o resultado obtido após ser aplicada uma função a um ou mais argumentos. O cálculo efectuado com recurso à utilização deste sistema formal consiste numa forma de avaliação simbólica de expressões. Através de expressões funcionais, o cálculo lambda fornece formas fundamentais de parametrização e, através de declarações, de ligação (binding).

Tal como foi dito no enunciado deste trabalho, o cálculo lambda tradicional possui três partes principais:

- Uma notação para definir funções;
- Um sistema de prova para resolver equações entre funções;
- Um conjunto de regras de cálculo, a redução, que permite fazer uma avaliação simbólica das expressões.

As variáveis, abstracções e aplicações são consideradas termos do cálculo lambda.

Em lambda cálculo, uma abstracção define uma função. Por exemplo:

$\lambda x.F$

Se  $F$  for uma expressão, o termo anterior representa a função obtida ao tratar  $F$  como função da variável  $x$ . Note-se que, no âmbito de lambda, uma abstracção estende-se o mais para a direita possível.

No cálculo lambda, as aplicações permitem tratar a aplicação de uma função a um argumento. Por exemplo:

$$(1) \quad (\lambda x.M_1) M_2$$

Se  $M_2$  for uma expressão, é aplicada a função  $(\lambda x.M_1)$  a  $M_2$ . A aplicação associa à esquerda.

Considera-se ainda que abstracções e aplicações entre parêntesis também são termos lambda.

A gramática BNF que define a linguagem do cálculo lambda é semelhante à do enunciado, ou seja:

```
<termo> ::= <variavel>  
          | (!<variavel>.<termo>)  
          | (<termo> <termo>)
```

Note-se que todos os tipos de termos enunciados anteriormente estão representados nesta gramática.

Uma das partes fundamentais no cálculo lambda é a redução. A redução permite avaliar simbolicamente expressões lambda. Podem existir várias reduções alternativas para o mesmo termo e também podem existir termos onde é possível reduzir sempre sem que se chegue à forma normal (expressão que não pode ser reduzida).

Na disciplina de Linguagens de Programação utilizamos duas estratégias para efectuar a redução de termos lambda:

- Call-by-name;
- Call-by-value.

A estratégia call-by-name escolhe o redex (*reduction expression*) mais exterior, enquanto a estratégia call-by-value escolhe o redex mais interior. Apesar de ser garantida a chegada à forma normal de um termo quando é utilizada a estratégia call-by-name, tal estratégia é menos eficiente que a estratégia call-by-value. A estratégia call-by-value não garante a chegada à forma normal de um termo, mas é consideravelmente mais rápida que a estratégia call-by-name.

Todas as fases da implementação do nosso interpretador de cálculo lambda podem ser consultadas de seguida.

## Implementação do Interpretador

A primeira fase na implementação do nosso interpretador consistiu na criação de um analisador lexical para a linguagem do cálculo lambda. Tal implementação foi efectuada, como foi referido na introdução, através do uso do analisador lexical Flex.

Os terminais da linguagem do cálculo lambda são os seguintes:

- As variáveis, que podem ser letras minúsculas ou maiúsculas. Note-se que no enunciado nada é dito sobre as variáveis poderem ser ou não letras maiúsculas. Neste caso, estamos a assumir que as variáveis podem ser também identificadas com letras maiúsculas. Quando é encontrada uma variável, a variável `yyval` do flex adquire o valor do primeiro elemento de `yytext` (variável pretendida) e retorna um *token* denominado VARIÁVEL. A regra para a operação descrita anteriormente é a seguinte:

```
[a-zA-Z] {yyval = yytext[0]; return VARIABEL;}
```

- ! – Símbolo definido no enunciado como equivalente à letra grega  $\lambda$ . Quando é encontrada uma ocorrência deste símbolo, é retornado o caractere '!'. A regra para a operação descrita anteriormente é a seguinte:

```
"!" {return '!';}
```

- ( – Parêntesis esquerdo. Quando é encontrada uma ocorrência deste símbolo, é retornado o caractere '(' . A regra para a operação descrita anteriormente é a seguinte:

```
"(" {return '(';}
```

- ) – Parêntesis direito. Quando é encontrada uma ocorrência deste símbolo, é retornado o caractere ')'. A regra para a operação descrita anteriormente é a seguinte:

```
")" {return ')';}
```

- . – Ponto. Quando é encontrada uma ocorrência deste símbolo, é retornado o caractere '.'. A regra para a operação descrita anteriormente é a seguinte:

```
"." {return '.';}
```

- " " – Espaço. Quando é encontrada uma ocorrência deste símbolo, é retornado o caractere ' '. A regra para a operação descrita anteriormente é a seguinte:

```
" " {return ' ';
```

**Nota:** O espaço foi considerado terminal para que, aquando da utilização da gramática, pudesse ser aceites aplicações com o formato *termo* espaço *termo*.

- “\n” – *Newline*. Este símbolo serve para identificar o final de uma linha. Quando é encontrada uma ocorrência deste caractere, é retornado um token denominado FIM\_LINHA. A regra para a operação descrita anteriormente é a seguinte:

```
"\n"           return FIM_LINHA;
```

- Todos os *tabs* (\t) que forem encontrados são ignorados. A regra para a operação descrita anteriormente é a seguinte:

```
[\t] ;
```

Finalmente existe uma função auxiliar que é chamada quando é encontrado o fim-de-ficheiro. A função utilizada para esta tarefa é a seguinte:

```
int yywrap() {  
    return 1;  
}
```

A segunda fase na elaboração do nosso interpretador consistiu na criação de um analisador sintáctico para a linguagem do cálculo lambda. Tal implementação foi efectuada, como foi referido na introdução, através do uso do Bison.

Nesta fase, começamos pelo prólogo, onde são efectuados os *includes*, são criadas as variáveis globais e são declarados os *headers* das funções auxiliares criadas.

Seguidamente são declarados os *tokens* e o símbolo inicial da gramática do código lambda. O código utilizado é o seguinte:

```
/* Tokens */  
  
%token VARIAVEL  
  
%token FIM_LINHA  
  
/*Símbolo Inicial */  
  
%start linha
```

Após terem sido efectuadas as declarações indicadas anteriormente, procedemos à elaboração das regras da gramática. Para tal, alterámos a gramática fornecida no enunciado deste trabalho e alterámos a mesma.

A gramática base do cálculo lambda, depois de efectuadas as alterações pretendidas é a seguinte:

```
linha: termo FIM_LINHA ;  
  
termo: VARIABEL  
  
| abstracao  
  
| termo_par  
  
| aplicacao  
  
;  
  
aplicacao: termo '' termo ;  
  
abstracao:  
  
'!'VARIABEL '.' termo ;  
  
termo_par: '(' termo ')';
```

A regra *linha* da nossa gramática permite aceitar linhas completas no nosso interpretador, ou seja, considera que uma linha é constituída por um termo lambda seguido de um caractere *newline*.

A regra *termo* da nossa gramática é, basicamente, a mesma que é fornecida no enunciado, considerando que uma variável, uma abstracção e uma aplicação são termos. Porém, na nossa gramática, considera-se que um termo que se encontre entre parêntesis é também um termo lambda.

A regra *aplicação* da nossa gramática permite aceitar expressões que consistam em dois termos separados por um espaço. Apesar de na gramática fornecida no enunciado não ser considerado o espaço entre os termos, nós decidimos considerar o mesmo de modo a que expressões semelhantes a “!x.x y” possam ser impressas correctamente ao invés de se verificar a seguinte impressão da mesma expressão do seguinte modo “!x.xy”.

A regra *abstracção* da nossa gramática permite aceitar expressões que contenham um lambda, seguido de uma variável, seguida de um ponto antes de um termo, por exemplo: !x.x.

A regra *termo\_par* permite aceitar expressões que consistam num termo (variável, abstracção, aplicação, termo entre parêntesis) que se encontrem entre parêntesis.

O *input* do utilizador é lido pelo através de código adicional que é executado sempre que a gramática aceita uma expressão.

A gramática final do nosso trabalho é a seguinte:

**termo**: VARIABEL {

```
    temp2[0] = $1;  
  
    temp2[1] = '\0';  
  
    strcat(temp,temp2);  
  
}  
  
| abstracao  
  
| termo_par  
  
| aplicacao  
  
;
```

**aplicacao**: termo '{strcat(temp," ");} termo

;

**abstracao**:

'!'VARIABEL '.'

```
{  
  
    strcat(temp,"!");  
  
    temp2[0] = $2;  
  
    temp2[1] = '\0';  
  
    strcat(temp,temp2);  
  
    strcat(temp,".");  
  
}
```

termo

;

**termo\_par**: '('{strcat(temp,"(");} termo ')'{strcat(temp,")");}'

;

A informação inserida pelo utilizador é armazenada numa String global que depois será o argumento da função que efectua a redução.

A terceira fase da construção do nosso interpretador consistiu fundamentalmente na criação do código C que efectua a redução dos termos lambda utilizando a técnica *call-by-name*.

A função que efectua a redução utilizando a estratégia call-by-name começa por fazer o print da expressão original prefixada por “<–” tal como acontece no enunciado. De seguida, procedemos à redução da expressão, utilizando a função auxiliar *redux*.

A função *redux* verifica o seguinte:

- Se a expressão não contém o caractere ‘!’, é retornada a string original. O termo reduzido é igual ao termo inicial.
- Se a expressão contém um lambda mas não contém espaços, é retornada a expressão. Isto verifica-se para expressões semelhantes a  $\lambda x.x$ . O termo reduzido é igual ao termo inicial.
- Se a expressão contém termos lambda mas a aplicação for igual a alguma das variáveis lambda, é retornada a expressão original. Isto significa que expressões do tipo  $\lambda x.\lambda y.x y$  não sofrem alterações. O termo reduzido é igual ao termo inicial.

Caso a expressão não seja semelhante a nenhuma das anteriores, o programa copia a informação que se encontra a seguir ao último espaço e substitui-a em todas as ocorrências da variável que se encontra a seguir ao lambda. São previstas situações em que passos intermédios na redução retornam expressões cujo termo à esquerda na aplicação não possui um lambda, sendo utilizada uma função auxiliar que troca os termos da aplicação de modo a que a redução possa ser concluída. Assim, se num passo intermédio se verificar a expressão:  $x \lambda y.y$ , a função auxiliar altera a mesma de modo a que se obtenha  $\lambda y.y x$  e efectua a redução desta última expressão.

Existem ainda funções auxiliares que permitem calcular o tamanho de uma string, contar o número de parêntesis, contar o número de espaços, contar o número de lambdas, remover parêntesis, contar o número de ocorrências de uma dada variável, inicializar uma string e trocar os termos de uma aplicação.

O nosso interpretador foi testado com todos os exemplos fornecidos no enunciado. À excepção do exemplo 8 (que retornava o valor  $\lambda y.y$  ao invés de  $\lambda x.x$ ), todos os exemplos foram reduzidos correctamente.

Para além disto, foram ainda elaborados os seguintes exemplos:

Exemplo	Retorno do Programa
$\lambda x.x \lambda y.y \lambda z.z$	$\lambda y.y$
$\lambda a.\lambda b.a b z$	$\lambda b.z b$
$\lambda x.\lambda y.x y \lambda a.a$	$\lambda y.\lambda a.a y$
$(\lambda f.f x) (\lambda y.y)$	$x$

## Bibliografia

### Livros:

- Mitchell, John C. 2003. *Concepts in Programming Languages*. Cambridge University Press.

### Fontes da Internet:

- A *Tutorial Introduction to the Lambda Calculus* (Online). (Acedido em 17-05-2012). Disponível em: <http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf>
- *Lambda Calculus* (Online). (Acedido em 19-05-2012). Disponível em: <http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Ch/>
- *Lambda Calculus* (Online). (Acedido em 21-05-2012). Disponível em: <http://www.mactech.com/articles/mactech/Vol.07/07.05/LambdaCalculus/index.html>

### Outros:

- Slides das aulas teóricas