1. Introdução

1.1. Histórico

A elaboração de modelos de computação (resolução de problemas por uma máquina) baseia-se em trabalhos de dois pesquisadores com enfoques bastante diferentes:

- Máquinas de Turing (Alan Turing, 1936)
- Cálculo Lambda (Alonzo Church, 1936)

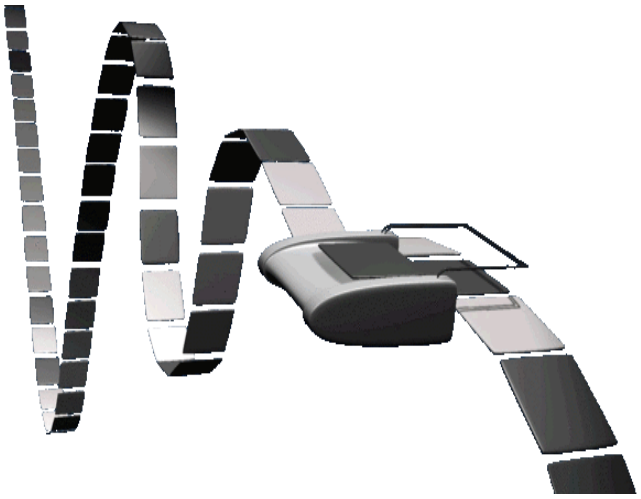
Alonzo Church

Alan Turing apresentou em um artigo em 1936 um predicado formalmente exato para o predicado informal e coloquial "*pode ser calculado através da utilização de um método efetivo*", as LCMs.

Este era o modelo matemático, descrito nas seguintes palavras pelo próprio Turing: '*LCMs (logical computing machines) podem realizar tudo o que puder ser descrito através de um conjunto de "regras de bom senso" ou de "forma puramente mecânica"* '.

LCMs foram mais tarde chamadas de Máquinas de Turing e assim são conhecidas até hoje.

Uma Máquina de Turing genérica como descrita originalmente consiste de uma um cabeçote de leitura/escrita que manipula dados contidos em uma fita com instruções. A posição seguinte da fita a ser lida depende do 'estado atual' registrado. Para nós esse "modelo computacional" parece uma coisa inefetiva, grosseira e possivelmente incapaz de resolver problemas mais complexos do que adicionar dois números, mas era revolucionário para a época e Turing provou que era de fato um modelo computacional universal.



Representação esquemática de uma Máquina de Turing


David Hilbert

Dessa forma, pergunta sobre a existência ou inexistência de um método automático formal para a resolução de algum determinado problema pôde acuradamente ser substituida pela pergunta se o problema em questão é calculável ou não com uma Máquina de Turing. Esta forma de descrever a relação entre problemas e seu modelo, na verdade a Tese de Turing, foi apresentada no decorrer de uma série de discusões sobre o **Problema da Decisão** (*Entscheidungsproblem*) de um sistema de lógica simbólica. A *Entscheidbarkeit* (Decidabilidade) de um sistema de lógica simbólica, colocada por David Hilbert em 1928, é dada pelo problema de encontrar-se um método *efetivo* através do qual, dada uma expressão Q qualquer na notação deste sistema lógico simbólico, pode-se determinar se Q é ou não é passível de se provada (como verdadeira ou como falsa, tanto faz) pelo sistema. Hilbert inicialmente acreditava que todo problema matematicamente formulável também fosse decidível e Turing provou que ele estava errado.

O teste de tabela verdade para prova de tautologias é um método desta categoria para o Cálculo Proposicional. Cálculo Proposicional logo representa uma classe de Problemas Decidíveis. Para um pequeno número de variáveis você resolve o teste facilmente e nada impede que se resolva para qualquer número de variáveis dado um estoque suficiente de papel e lápis e uma vida longa. Turing mostrou que, dada a sua tese, não existe um método assim para o Cálculo de Predicados.

Turing atacou este problema da seguinte forma: Suponha que você possua um algoritmo geral de decisão para Lógica de Primeira Ordem. A decisão se uma máquina de Turing pára ou não pode ser formulada como uma expressão de primeira ordem, a qual seria então suscetível ao algoritmo. Mas Turing havia provado pouco antes que não existe um algoritmo geral que pode decidir se uma dada Máquina de Turing pára ou não. Este é o Problema da Parada.

Alonzo Church, por sua vez, também em 1936, encontrou também uma resposta negativa para o *Entscheidungsproblem*, mas de uma forma completamente diferente: Ele utilizou o conceito existente na época de Lambda-definibilidade. Uma função de inteiros positivos é dita ser Lambda-definível se os valores da função podem ser calculados através de um processo de substituições repetidas. Ele provou que não existe um algoritmo definido através de funções recursivas capaz de decidir para duas Expressões-Lambda se elas são equivalentes ou não. Para isso ele teve de inventar o Cálculo-Lambda.

Alan Turing provou em 1937 a equivalência entre uma máquina de Turing e o Cálculo Lambda em termos de **computabilidade**.

O resultado final desta troca de interações entre Turing e Church resultou no que é chamado de *Tese de Church-Turing*, a qual trata da noção de um método *efetivo* ou *mecânico* em lógica e em matemática. Efetividade e mecanicidade não possuem nesse contexto o significado coloquial, mas sim um significado matemático. Pode-se dizer que `M` é um método *efetivo* e *mecânico* somente se:

1. `M` é descrito em termos de um conjunto finito de instruções exatas, cada qual por sua vez expressa sob a forma de um número finito de símbolos;

2. `M` produzirá, se executado sem erros, o resultado desejado em um número finito de passos;

3. M pode, na prática ou pelo menos em princípio, ser executado por um ser humano sem nenhuma outra ajuda que um lápis e papel;

4. M não exige do ser humano que o executa nenhum brilhantismo, originalidade ou genialidade.

No fundo, isto nada mais é que adefinição matemática formal do conceito de algoritmo, realizada primeiramente por Alonzo Church através da utilização de seu enfoque.

Detalhes sobre podem ser encontrados na Stanford Encyclopedia of Philosophy.

A conseqüência direta disso para nós mortais comuns, além de progressos na Teoria da Computação dos quais usufruímos, é a de que o Cálculo Lambda serviu de base teórica para as Lingugens de Programação Funcionais.

1.2. Linguagens Funcionais

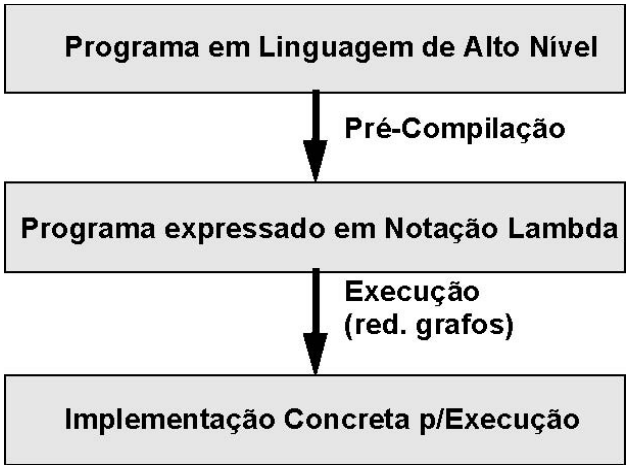
- Interessantes pela sua simplicidade sintática
- Facilidade de descrever-se problemas recursivos.
- Muitas das implementações são poucos aceitas devido à ineficiência em comparação com linguagens de programação „tradicionais“.
- Novas implementações de interpretadores/compiladores e novas linguagens mais modernas tem surgido.

Exemplos:

- LISP (LISt Processing - década de 60). Muito simples em muitos aspectos. Atualmente é ainda a mais utilizada.
- Miranda (Turner 1985)
- Haskell (1990)
- Orwell (Wadler 1985)
- Outras: ML, KRC, LML, SASL.

1.3. Utilidade do Cálculo Lambda em Programação Funcional:

- Ponto de Partida: Maioria da Linguagens de Programação Funcional são semelhantes e diferem somente em aspectos sintáticos (S.P.Jones).
- Uma linguagem funcional de alto nível pode ser compilada em um código intermediário com uma sintaxe e semântica extremamente simples. Essa linguagem é expressa em termos de Cálculo Lambda.
- O Cálculo Lambda é suficientemente expressivo para permitir a codificação de uma linguagem funcional de alto nível.
- Aloritmos para a resolução de problemas computacionais expressos em Cálculo Lambda podem ser facilmente implementados de diferentes formas.



Implementação de um Programa Funcional

O capítulo 2 desta disciplina vai se ocupar basicamente da notação-lambda e de problemas relacionados à compilação de um programa funcional em termos de Cálculo Lambda.

2. Aspectos Teóricos Básicos: Cálculo Lambda

O Cálculo Lambda (calculo – λ) é uma coleção de diversos sistemas formais baseados em uma notação para funções desenvolvida inicialmente por Alonzo Church em 1936.

É projetado para capturar os aspectos mais básicos da maneira pela qual operadores ou funções podem ser combinados para formar outros operadores.

O Cálculo Lambda serve como uma ponte entre linguagens funcionais de alto nível e suas implementações de baixo nível. Razões para a apresentação do Cálculo Lambda como uma linguagem intermediária:

- É uma **linguagem extremamente simples**, consistindo de somente algumas poucas construções sintáticas e de uma semântica simples.

  Uma implementação do Cálculo Lambda necessita somente de suportar algumas construções simples.

  A sua semântica simples nos permite analisar facilmente a correção de sua implementação.
- É uma linguagem **expressiva**, a qual é suficientemente poderosa para expressar todos os programas funcionais e, por conseguinte, todas as funções computáveis.

  Isto significa que, se nós possuímos uma boa implementação do Cálculo Lambda, nós podemos implementar qualquer linguagem funcional através da implementação de um compilador desta para o Cálculo Lambda.

2.1. A Sintaxe do Cálculo Lambda

Um exemplo simples de uma expressão em Cálculo Lambda é:

```
(+ 4 5)
```

Todas as aplicações de funções no Cálculo Lambda são escritas em notação prefixada. Assim a função **+** precede os argumentos 4 e 5. Um exemplo um pouco mais complexo é:

```
(+ (* 5 6) (* 8 3))
```

Do ponto de vista de implementação, um programa funcional pode ser visto como uma expressão, que é "executada" através da sua ***avaliação***.

A avaliação ocorre através da **seleção repetida de uma *expressão redutível* (redex) e de sua *redução***. Expressão redutível é aquela que pode ser avaliada imediatamente.

- No exemplo temos dois redexes: `(* 5 6)` e `(* 8 3)`.
- A expressão inteira `(+ (* 5 6) (* 8 3))` não é um redex, uma vez que a função **+** necessita de ser aplicada a dois números para poder ser redutível.
- Através da escolha arbitrária de do primeiro redex para redução, podemos escrever:

  ```
  (+ (* 5 6) (* 8 3)) -> (+ 30 (* 8 3))
  ```

  onde `->` é pronunciado „reduz para".
- Agora há somente um redex `(* 8 3)`, do qual resulta: `(+ 30 24)`
- Esta redução gera um novo redex, que reduz: `(+ 30 24) -> 54`

2.1.1. Aplicação de Funções e „Currying"

Em Cálculo Lambda, a aplicação de uma função tem um papel tão central, que pode ser denotada por simples justaposição.

Assim escrevemos `f x` para denotar „a função **f aplicada ao argumento x".**

Para expressar a aplicação de uma função a vários argumentos não utilizamos a notação intuitiva `(f (x,y))` e sim a altenativa de escrever:

```
((+ 3) 4)
```

- A expressão `(+ 3)` denota a "função + aplicada ao argumento 3", cujo resultado é uma "função aplicada ao valor 4".
- Como em todas as linguagens funcionais, o Cálculo Lambda permite que uma função retorne um função como resultado.
- Este dispositivo nos permite imaginar todas as funções como possuindo somente um argumento. Esta notação foi introduzida por Schonfinkel em 1924 e utilizada amplamente nos trabalhos de Curry, de onde provém a expressão „currying".

2.1.2. Uso dos Parênteses

Convecionalmente em matemática omite-se parênteses redundantes para evitar tornar expressões pouco legíveis. Ex.:

`(ab) + ((2c)/d)` pode ser escrita `ab + 2c/d`

A segunda forma é mais fácil de ser lida e a semântica não se altera. O perigo é que a segunda expressão pode ser ambígua e resultados diferentes podem ser obtidos, dependendo de como se

executa.

Isto pode ser evitado através do estabelecimento de convenções sobre a precedência de operadores e de funções (exemplo: multiplicação liga de forma mais forte do que adição). As vezes, parênteses não podem ser omitidos, como: (b + c)/a

Convenções similares são úteis no Cálculo Lambda.

Considere:

((+ 3) 2)

- Através do estabelecimento da convenção de que *aplicação de funções associa à esquerda*, podemos escrever:

  (+ 3 2)

  ou até:

  + 3 2

  Abreviações deste tipo foram feitas no exemplos anteriores.

- Um exemplo mais complexo é a expressão:

  ```
  ((f ((+ 4) 3)) (g x))
  ```

  que pode ser escrita:

  ```
  f (+ 4 3)
  ```

2.1.3. Funções Embutidas e Constantes

Funções embutidas como + não existem no Cálculo Lambda na sua forma mais pura. Para fins práticos, uma extensão que as suporte é útil.

Estas incluem funções aritméticas (como `+, -, *, /`), constantes (como `0, 1`,...), funções lógicas (como `AND, NOT, OR`,...) e constantes lógicas (`TRUE, FALSE`).

Exemplos:

```
-5 4 -> 1

AND TRUE FALSE -> FALSE
```

Também incluímos uma função condicional, cujo valor é descrito pelas regras de redução abaixo:

$$\text{IF TRUE } E_t\ E_f \rightarrow E_t$$

$$\text{IF FALSE } E_t\ E_f \rightarrow E_f$$

Construtores de dados em Cálculo Lambda serão introduzidos inicialmente através da definição de três funções embutidas: CONS, HEAD e TAIL (as quais se comportam exatamente como as funções LISP: CONS, CAR e CDR). CONS constroe um objeto composto, o qual pode ser desmantelado com HEAD e TAIL. A operação é descrita pelas seguites regras de redução:

```
HEAD (CONS a b) -> a

TAIL (CONS a b) -> b
```

Além dessas funções, definiremos a constante NIL, o valor nulo.

A escolha exata de funções embutidas é arbitrária.

2.1.4. Abstrações Lambda

Abstrações lambda são construções em Cálculo Lambda que denotam funções novas, não embutidas em um cálculo específico.

Exemplo:

```
(λ x . +x 1)
```

- O λ indica o início de uma função e é imediatamente seguido de uma variável.
- Depois segue um ponto "." seguido pelo *corpo* da função. A variável é chamada de *parâmetro formal* e nós dizemos que o λ a *liga*.

O exemplo acima pode ser lido: „Aquela (λ) função de (x) a qual (.) adiciona x a 1 (+ x 1).

Uma abstração lambda sempre consiste dessas partes: o λ , o parâmetro formal, o . e o corpo. Uma abstração lambda pode ser considerada similar a uma definição de função em uma linguagem de programação convencional, como „C":

```
inc ( x ) {
      int x;
      ( return( x + 1); }
```

- O parâmetro formal da abstração lambda corresponde ao parâmetro formal da função.
- O corpo da abstração é antes uma expressão do que uma sequência de comandos. Diferença: funções em linguagens convencionais necessitam de um nome como inc, enquanto que em Cálculo Lambda funções são anônimas.

Sintaxe: O corpo de uma abstração se extende o tanto para a direita quanto for possível.

- Assim, na expressão:

```
(λ x . + x 1) 4
```

  o corpo da abstração é (+ x 1) e não somente +.

Usualmente são incluídos parênteses a mais para clarificar a sintaxe:

```
(λ x . (+ x 1)) 4.
```

Uma abstração lambda sozinha dispensa quaisquer parênteses:

```
λ x . + x 1
```

2.1.5. Sumário

Definimos uma *expressão lambda* **(**expressão-λ**)** como uma expressão no cálculo lambda.

A sintaxe de uma expressão lambda em notação BNF é dada abaixo.

<exp>	**::= <constante>**	*constantes embutidas*
	| <variável>	*nomes de variáveis*
	| <exp> <exp>	*aplicações*
	| λ <variável> . <exp>	*abstrações lambda*

ou, de forma mais estruturada (G.Revesz):

<exp>::= <constante>|<variável>|<aplicação>|<abstração>

<aplicação>	**::= (<exp>) <exp>**
<abstração>	**::= λ <variável> . <exp>**

Esta sintaxe então nos permite formar expressões-λ como:

$\lambda_{x.x}$	$\lambda_{x.}\lambda_{y.(y)x}$	$\lambda_{x.(f)x}$
(f)3	$\lambda_{f.(f)2}$	$(\lambda_{y.(x)y})\lambda_{x.(u)x}$

Resumindo a nomenclatura

Uma **aplicação-** λ é simplesmente a aplicação de uma expressão-λ sobre outra. A primeira dessas duas expressões-λ é chamada de *operador*, a segunda de *operando*.

- Note que qualquer expressão-λ pode ser utilizada tanto como operador como como operando.

Uma **abstração-** λ é formada com o símbolo especial λ seguido por uma variável, seguida por um ponto, seguido por uma expressão-λ arbitrária.

- O propósito da operação de abstração é o de formar uma expressão unária a partir de uma expressão-λ dada. A variável que ocorre próxima ao λ inicial dá nome ao argumento.
- Funções com mais de um argumento são formadas por abstrações repetidas.

Veja as nossas Referências Bibliográficas e Links em Cálculo Lambda.