Programação Lógica e Funcional

Uma Introdução ao Cálculo Lambda

Introdução

Este material contém notas de aula sobre Cálculo Lambda. O objetivo é apresentar os conceitos desta notação formal, sua sintaxe, formas válidas para expressões, substituições e numerais de Church. Tais assuntos são importantes para avançar nos estudos sobre Haskell, uma vez que esta linguagem tem sua base teórica fortemente ligada ao Cálculo Lambda, tanto em sua modelagem de funções quanto em seu sistema de tipos.

Cálculo Lambda

Cálculo Lambda (ou λ-Cálculo, ou Cálculo-λ, onde λ é a letra lambda minúscula no alfabeto grego) é um sistema formal matemático criado na década de 1930 por Alonzo Church. Ramos (2012) afirma que ele pretendia formalizar a matemática através da noção de funções ao invés da teoria de conjuntos, e apesar de não conseguir sucesso, seu trabalho teve grande impacto em outras áreas, especialmente na computação.

Esse sistema é capaz de formalizar as noções de ***função*** e ***computação***, permitindo representar e manipular funções de maneira abstrata, estudar a execução de programas e investigar problemas de decidibilidade em matemática e lógica para expressar a noção de

computação. Este sistema é equivalente à Máquina de Turing e expressa toda computação possível, sendo ambos modelos abstratos de computador.

Há diversas aplicações que este modelo possibilita, tais como (RAMOS, 2012):

- representação de funções computáveis;
- teoria, especificação e implementação de linguagens de programação;
- verificação de programas;
- teoria das provas;
- estudo da computabilidade.

O Cálculo Lambda é a base para diversas linguagens funcionais como Haskell, LISP, ML, Cloujure e Elixir. Nele, funções são definidas de forma abstrata e são tratadas como valores de primeira classe, assim como caracteres, números e tuplas.

Exemplos de valores de primeira classe na linguagem Haskell:

- **Literais**:
    - `True`
    - `'x'`
    - `"Olá"`
    - `3.1415`
- **Variáveis**:
    - `nome = "Ricardo"`
    - `ano_nasc = 1989`
    - `sexo = 'M'`
- **Argumentos**:
    - `not True`
    - `length [1,2,4]`
    - `(+) 2 3`
    - `sqrt 9.9`
    - `(&&) True False`
    - `map chr [80,81] (exige import Data.Char)`
- **Resultados**:
    - `1`

    - `3.146426544510455`
    - `False`
    - `"PQ"`
- **Componentes**:
    - `("Ricardo", 1989)`
    - `[["Ricardo", "1989"], ["João", "1981"]]`
    - `[("oi", 2), ("tchau", 5)]`

O elemento central do Cálculo Lambda é a ***Expressão Lambda*** (Expressão-λ ou λ-Expressão), que representa funções anônimas e fornece a estrutura básica para a construção de programas e manipulação de dados. Em outras palavras, funções anônimas são aquelas em que não há identificador associado, ou seja, a função é descrita apenas por sua definição e pode ser usada diretamente no ponto em que é escrita. Esse caráter anônimo torna as expressões lambda especialmente úteis para representar transformações simples e para compor funções maiores a partir de blocos menores.

Sintaxe

Existem três formas válidas para expressões-λ ou termos-λ:

1. Variável
2. Aplicação de função
3. Abstração lambda (ou λ-abstração, ou abstração-λ)

Formalmente, as três formas citadas para expressões-λ são:

<expressão> = x (1)
| <expressão> <expressão> (2)
| λx . <expressão> (3)

Exemplos:

1. A variável pode ser a, b, c, x, y…
2. A aplicação de função pode ser, por exemplo, duas variáveis: a b.
3. λx . x.

- Neste terceiro caso, λx é o ***parâmetro formal*** e o que há depois do ponto (neste caso, 'x') é o ***corpo da função***.
- Ainda sobre este exemplo, diz-se que a função recebe o parâmetro formal x e retorna x. Esse exemplo representa a função identidade.

Exemplos “enriquecidos”

A versão original do cálculo lambda só possui funções, sem números e operações. No entanto, a notação enriquecida aceita esses recursos e pode facilitar a aprendizagem.
Se $\lambda x.x^2 + 7$ é a definição de uma função, λx indica que x é o parâmetro formal dessa função, isto é, um nome temporário para o valor a ser recebido pela função (MACHADO, 2013).

Se a função for aplicada ao valor 8, por exemplo, tem-se: $(\lambda x.x^2 + 7)\ 8 = 8^2 + 7 = 71$.
Se a função for aplicada a y, por exemplo, tem-se: $(\lambda x.x^2 + 7)\ y = y^2 + 7$. Neste caso, diferentemente do anterior, o resultado não é um número, mas uma expressão simbólica.

Já em λx.x + y, se aplicada ao argumento 9, resulta em (λx.x + y) 9 = 9 + y.
Analogamente, se a função for λy.x + y, o mesmo argumento 9 resulta em (λy.x + y) 9 = x + 9.

Se, para o mesmo exemplo, a função pudesse receber múltiplos parâmetros, sua definição seria λxλy.x + y, ou, em uma notação resumida, adotada por muitos autores, seria λxy.x + y. A aplicação da função poderia ser (λxλy.x + y) (8) (9), o que resultaria em:
(λxλy.x + y) (8) (9) = (λy.8 + y) (9) = 8 + 9 = 17.

PARA PENSAR

Se o cálculo lambda original não possui operações e números, como essa versão pura consegue ser tão expressiva quanto qualquer máquina?

Em Haskell

Em Haskell, a função identidade (λx . x) seria implementada da seguinte maneira:

```
identidade x = x.
```

Porém, dessa forma a função tem um identificador e, como visto, expressões-λ são funções anônimas. Para utilizar função anônima em Haskell, faz-se da seguinte maneira:

```
> ghci
prelude> (\x -> x) (-9)
-9
prelude> (\x -> x) 9
9
```

Outra forma:

```
prelude> identidade = \x -> x
prelude> identidade (-9)
-9
```

Generalizando, a sintaxe de uma expressão lambda em Haskell é:

`\padrão`$_1$ `padrão`$_2$ `... padrão`$_n$ `-> expressão`

A \ (barra invertida) representa o lambda, pois o caractere λ não é aceito no código). Cada padrão representa um parâmetro, assim como os parâmetros formais definidos na sintaxe do Cálculo Lambda. Esses parâmetros podem ser um identificador simples ou até um padrão de tupla, lista, etc.). Já a `expressão` representa o corpo da função.

Como a expressão é gerada pelas regras?

A árvore sintática genérica da Figura 1 demonstra a aplicação das regras de reescrita, no que pode-se chamar de derivação.

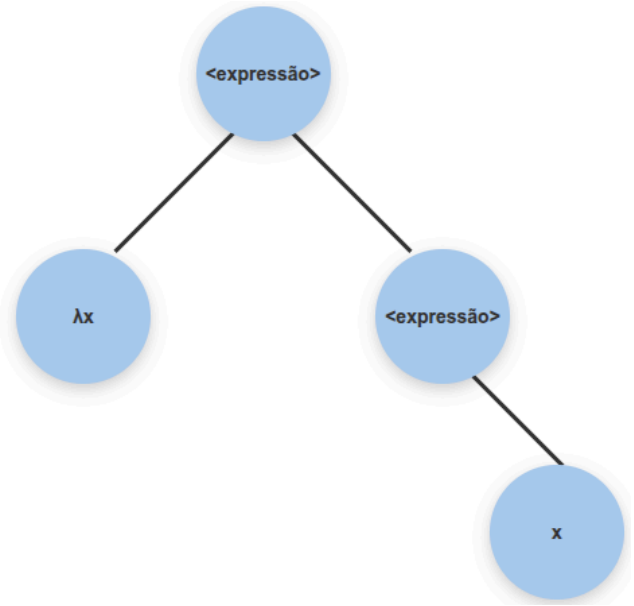
Essa árvore foi gerada da seguinte forma:

1. Começa-se com <expressão>, que será a raíz da árvore.
2. Aplica-se a regra 3: <expressão> → λx . <expressão>.
3. Em <expressão>, aplica-se a regra 1 <expressão> → x.

A geração citada foi feita na forma *top-down*, ou seja, a partir da expressão genérica <expressão>, chegou-se até as folhas da árvore que são os símbolos finais da expressão-λ.

Mas ela também pode ser feita no sentido oposto, ou seja, chegar até a raíz (<expressão>) a partir dos nós folha (a expressão-λ), na derivação chamada de *bottom-up*.

Figura 1. Árvore sintática de derivação da função identidade em Cálculo-λ.



Formas de Identificação da Validade Sintática da Expressão-λ

- Verificando se ela segue uma das três formas aceitas e permite realizar substituições até obter <expressão> de forma *bottom-up*; ou
- Realizando o teste na forma *top-down*, partindo de <expressão> até chegar na expressão a ser validada (a expressão-λ).

Como demonstrado anteriormente:

<expressão>	(substituir pela regra 3)
λx . <expressão>	(substituir pela regra 1)
λx . x	(fim)

Como foi possível partir de <expressão> e chegar até a expressão-λ, esta é uma expressão-λ sintaticamente válida.

EXERCÍCIO 1

Verifique se λa . λb . ab é uma expressão-λ sintaticamente válida:

EXERCÍCIO 2

Suponha que a expressão λa . λb . a + b deva ser utilizada em Haskell. Como fazer isso?

RESPOSTAS

1]

<expressão>	(substituir pela regra 3)
λa . <expressão>	(substituir pela regra 3)
λa . λb . <expressão>	(substituir pela regra 2)
λa . λb . <expressão> <expressão>	(substituir pela regra 1)
λa . λb . a <expressão>	(substituir pela regra 1)
λa . λb . a b	(fim)

2]

```
prelude> (\a b -> a + b) 1 3
4
```

Observação: A notação com operadores aritméticos não está presente no cálculo lambda original; como dito, essa notação é conhecida como ***cálculo-λ enriquecido***.

Associatividade e Precedência (RAMOS, 2012)

As seguintes convenções são adotadas na notação do cálculo lambda:

- Aplicações de função têm prioridade sobre abstrações;
- Aplicações são associativas à esquerda;
- Abstrações são associativas à direita.

Exemplos:

- λx.AB é uma abstração. Portanto, se escrita desta forma, tem o mesmo significado que λx.(AB) e (λx.(AB)). No entanto, essa abstração é diferente de (λx.A)B.
- ABCD é o mesmo que (((AB)C)D), pois são termos quaisquer e estes seguem a aplicação com associatividade à esquerda;

- λxyz.A é o mesmo que (λx.(λy.(λz.A))). Como dito anteriormente, essa equivalência é muito utilizada para simplificar a expressão, deixando-a com menos caracteres de uma forma geral (caracteres lambda e parênteses). Neste caso, nota-se que a lambda abstração é associativa à direita.

Escopo

Na utilização de termos-λ, parênteses fazem diferença e podem diferenciar duas expressões em termos de escopo. Para exemplificar, considere que (1) e (2) são as expressões abaixo definidas:

1. (λa . λb . b) (λc . λd . c)
2. λa . λb . b (λc . λd . c)

Essas expressões são diferentes por possuírem escopos diferentes. O escopo de λa na expressão (1) é λb . b. O escopo de λa na expressão (2) é λb . b (λc . λd . c). A expressão (1) consiste em uma aplicação de função, onde (λa . λb . b) é uma abstração-λ e (λc . λd . c) é outra, mas que funciona como argumento para a primeira. Já a expressão (2) é uma única abstração-λ.

Se este exemplo fosse implementado em uma linguagem de programação, seria como utilizar um parâmetro dentro do escopo em que ele foi definido. Caso se tente utilizá-lo fora do escopo, obtém-se um erro.

Equivalência

Duas expressões-λ são ditas α-equivalentes (alfa equivalentes) se diferem apenas no nome das variáveis, ou seja,
λa . λb . b é α-equivalente a λx . λy . y;
λa . λb . b não é α-equivalente a λx . λy . x.

Variáveis Livres e Ligadas

- A variável livre não está ligada a uma abstração-λ.

- A variável ligada está ligada a uma abstração-λ.
- O conjunto de variáveis livres de uma expressão M é
    1. FV(x) = {x}
    2. FV(MN) = FV(M) U FV(N)
    3. FV(λx . M) = FV(M) - {x}

EXERCÍCIO 1

Quais são as variáveis livres em λa . fga?

EXERCÍCIO 2

Mostre que FV(λx . λf . g (fxy)) = {g, y}.

RESPOSTAS

1]

FV(λa . fga) = FV(fga) - {a} (de (3))
= FV(fg) U FV(a) - {a} (de (2))
= FV(f) U FV(g) U FV(a) - {a} (de (2))
= {f} U {g} U {a} - {a} (de (1))
= {f, g, a} - {a} (operação de união - conjuntos)
= {f, g} (operação de diferença - conjuntos)

2]

FV(λx . λf . g (fxy)) = FV(λf . g (fxy)) - {x} (de (3))
= FV(g (fxy)) - {f} - {x} (de (3))
= FV(g) U FV(fxy)) - {f} - {x} (de (2))
= FV(g) U FV(fx) U FV(y) - {f} - {x} (de (2))
= FV(g) U FV(f) U FV(x) U FV(y) - {f} - {x} (de (2))
= {g} U {f} U {x} U {y} - {f} - {x} (operação de união - conjuntos)
= {g, f, x, y} - {f} - {x} (operação de diferença - conjuntos)
= {g, y}

Substituição (Redução-β ou β-Redução)

A beta redução é o axioma[1] central do cálculo-λ. Ela nada mais é que a aplicação de uma função (abstração-λ) em um argumento. Em outras palavras, ocorre a substituição de todas as ocorrências do parâmetro formal por M no corpo (N) da λ-abstração.

Quando não é mais possível reduzir uma expressão-λ, diz-se que a expressão está em sua ***forma normal***.

Segundo Damiani (s.d.), a substituição de todas ocorrências de uma variável x por uma expressão M é representada por [M/x]:
(λx . N) M -> [M/x] N

Exemplo:
(λf . fx) (λy . y) -> aplicando a função (λy . y) em (λf . fx):
-> (λy . y) x
-> aplicando a função x em (λy . y)
-> x

Todas as expressões acima são ditas ***β-equivalentes***, pois por meio de substituições é possível alcançar qualquer uma das outras.

Exercício: aplique a β-redução na expressão (λa . aa) (λb . b) (λc . λd . d).

Resposta:
(λa . aa) (λb . b) (λc . λd . d) → aplicando a função (λb . b) em (λa . aa):
→ (λb . b) (λb . b) (λc . λd . d)
→ aplicando a função (λb . b) em (λb . b):
→ (λb . b) (λc . λd . d)
→ aplicando a função (λc . λd . d) em (λb . b):
→ (λc . λd . d)

[1]Axioma é uma premissa aceita como verdadeira.

Numerais de Church

Numerais de Church são representações de números baseadas na Codificação de Church. Esta codificação é a forma utilizada originalmente por Alonzo Church para trabalhar com números utilizando a notação do cálculo lambda. Para isso, a codificação trabalha com um mapeamento entre o número e uma função,

Suponha que s é a função sucessor de um número e que z = 0. Assim,

s(z) = 1

s(s(z)) = 2

s(s(s(z))) = 3

(...)

Utilizando a notação do cálculo lambda, os Numerais de Church são definidos da seguinte forma:

0 = λf.λx. x

1 = λf.λx. f x

2 = λf.λx. f (f x)

3 = λf.λx. f (f (f x))

(...)

n = λf.λx. f^n x

Através da definição acima, percebe-se que um número n é mapeado em uma função que aplica a função f exatamente n vezes a um parâmetro x (que pode ser modelado como o valor zero). Através dessa abordagem, a quantidade de chamadas à função f determina a contagem de valores, permitindo obter o número natural desejado.

Exercício: Suponha que a função sucessor seja definida genericamente em cálculo-λ como (λw . λy . λx . y (wyx)). Mostre através de β-reduções que (λw . λy . λx . y (wyx)) (λs . λz . s(s(z))) é β-equivalente a λs . λz . s(s(s(z))).

Resposta:

(λw . λy . λx . y (wyx)) = Função sucessor

(λs . λz . s(s(z))) = Argumento 2

λs . λz . s(s(s(z))) = Valor 3

Substitui λw por toda expressão do argumento 2:

λy . λx . y ((λs . λz . s(s(z)))) yx

Substitui λs por y:

λy . λx . y ((λz . y(y(z)))) x

Substitui λz por x:

λy . λx . y (y(y(x)))

Trocando y por s e x por z, temos:

λs . λz . s(s(s(z)))

Portanto, (λw . λy . λx . y (wyx)) (λs . λz . s(s(z))) é β-equivalente a λs . λz . s(s(s(z))).

Implementação dos Numerais em Haskell

O arquivo `numerais-de-church.hs` traz exemplos de funções em Haskell que implementam os Numerais de Church, bem como a função `sucessor` e funções de conversão entre valores `Integer` e valores do tipo `Church` (definido no arquivo).

Combinadores e Recursão

Em desenvolvimento.

O arquivo `fatoriais-lambda.hs` traz um exemplo que faz uso do combinador de ponto fixo por meio da função `fix` (disponível em `Data.Function`) para criar a recursão na função `fatorial`.

Vantagens do Cálculo-λ

- Escrever facilmente funções em tempo real

- Notação simples
- Redução do código
- Pode resultar em código mais eficiente

Saiba Mais

Há outros tópicos de cálculo-λ não cobertos neste material, entre eles:

- **Operadores booleanos**: formas de implementação de valores (verdadeiro e falso) e operadores lógicos.
- **Cálculo-λ enriquecido**: notação estendida, com estruturas condicionais, operadores aritméticos etc. Utilizada por ser ainda mais simples, não por necessidade (cálculo-λ puro é capaz de resolver as mesmas questões).
- **Operador de ponto fixo**
- **Combinador**
- **Currificação** (*currying*)
- **Cálculos com Numerais de Church**: definição de operações de soma, multiplicação etc.
- **Tipos de avaliação** (por valor e preguiçosa)

Sugere-se que o leitor interessado em expandir seus conhecimentos em Cálculo Lambda prossiga suas leituras partindo desses tópicos. Uma boa fonte para estudar esses e outros tópicos é a página do Grupo de Estudos em Haskell da UFABC.

Conclusão

Este material apresentou os fundamentos do Cálculo Lambda, abrangendo sua sintaxe, as formas válidas de expressão, os conceitos de escopo, variáveis livres e ligadas, bem como a relevância da β-redução como mecanismo central de avaliação. Foram discutidos os Numerais de Church, que possibilitam representar números naturais exclusivamente por meio de funções. Esses elementos constituem a base teórica necessária para a compreensão de linguagens funcionais, como Haskell (a ser vista na sequência), cuja modelagem de funções, sistema de tipos e uso de funções de ordem superior derivam diretamente do Cálculo Lambda.

Destaca-se, ainda, que o Cálculo Lambda é um modelo formal de computação equivalente à Máquina de Turing, o que demonstra sua relevância histórica, conceitual e prática, especialmente nos campos da Teoria da Computação e da Programação Funcional.

Exercícios

1) Considere a expressão-λ a seguir para responder às questões.

   λa . λb . λc . a (λd . λe . e (d b)) (λf . c) (λf . f)

   a) A expressão é sintaticamente válida?
   b) Gere a árvore sintática derivando com a estratégia *bottom-up* a expressão até onde for possível, se ela for inválida sintaticamente.
   c) Qual(is) é(são) a(s) variável(is) livre(s) da expressão?
2) Utilizando o arquivo `numerais-de-church.hs`, defina as funções `seis`, `sete`, `oito` e `nove`.
3) Utilizando o arquivo `numerais-de-church.hs`, defina as funções `doisChurch`, `tresChurch`, `quatroChurch`, `cincoChurch`, `seisChurch`, `seteChurch` e `nove-Church`.
4) Considere a expressão abaixo na sintaxe da linguagem Haskell e faça o que se pede.

   ```
   (\x -> (\y -> y)) 3
   ```

   a) Converta para a notação do Cálculo Lambda.
   b) Realize beta-reduções passo a passo até chegar à forma normal da expressão.
5) Assumindo que λa . λb . λc . b (a b c) é a função `sucessor`, quem é a função abaixo?

   λd . λa . λb . λc . d b (a b c)
6) Elabore um texto de 4 a 8 parágrafos explicando o tópico em vermelho (*Combinadores e Recursão*). Seu texto deve conter, no mínimo:
   a) Definições
   b) Exemplos
   c) Referências

Respostas

1) Abaixo:
    a) Sim.
    b) Livre.
    c) Não há variáveis livres na expressão. Todas são ligadas, algumas no próprio escopo e outras no escopo externo.
2) Livre.
3) Livre.
4) Abaixo:
    a) (λx . (λy . y)) 3
    b) Primeiro, substitui-se 3 por todas as ocorrências de x no corpo da função. No entanto, não há ocorrências de x, o que nos permite apenas descartar o valor. Sendo assim, ficamos com (λy . y). Não há mais funções a serem aplicadas, portanto, esta é a forma normal da expressão.
5) A função soma, pois aplica-se a função sucessor *d* vezes.

Referências

DAMIANI, Cristiano. Notas de Aula. Material didático. s.d.

DAMIANI, Cristiano. Uma Brevíssima Introdução ao Cálculo Lambda (Aula 8). Disponível em: https://www.youtube.com/watch?v=jvAkNij65W4. Acesso em: 28 ago. 2025.

FRANÇA, Fabrício Olivetti de; FRANCESQUINI, Emilio. CÁLCULO λ. Disponível em: https://haskell.pesquisa.ufabc.edu.br/haskell/02.lambda/. Acesso em: 29 ago. 2025.

MACHADO, Rodrigo. Uma introdução ao cálculo lambda e a linguagens de programação funcionais. Minicurso 1. WEIT 2013. Disponível em: https://www.inf.ufrgs.br/~rma/documentos/minicurso-weit2013.pdf. Acesso em: 04 set. 2025.

MALAQUIAS, José Romildo. Capítulo 7: Expressões Lambda. Disponível em: http://www.decom.ufop.br/romildo/2012-1/bcc222/slides/07-lambda.pdf. Acesso em: 28 ago. 2025.

MALAQUIAS, José Romildo. Capítulo 8: Expressão Lambda. Disponível em: http://www.decom.ufop.br/romildo/2013-1/bcc222/slides/08-lambda.pdf. Acesso em: 28 ago. 2025.

RAMOS, Marcus Vinícius Midena. Cálculo Lambda. Disponível em: https://www.marcusramos.com.br/univasf/tc-2012-1/slides-5.pdf. Acesso em: 08 set. 2025.