

**Universidade Federal de Alagoas**  
Instituto de Computação  
Curso de Ciência da Computação

**Curry**  
**Paradigmas de Linguagens de Programação**

João Victor Ribeiro Ferro  
Lucas Albuquerque Lisboa

Maceió  
2020.1

<b>Introdução</b>	<b>3</b>
<b>Programa</b>	<b>3</b>
Igualdade e Restrições	3
Alta ordem	4
<b>Tipos de Dados</b>	<b>4</b>
Boolean	4
Função	5
Int	5
Float	5
List	5
Characters	6
Tuples	6
<b>Expressões</b>	<b>6</b>
Sequências Aritméticas	6
Variáveis Livres	7
<b>Exemplo de Programa</b>	<b>7</b>
<b>Referências Bibliográficas</b>	<b>8</b>

# Introdução

Curry é uma linguagem declarativa de propósito geral que mescla programação funcional com programação lógica, com forte influência de Haskell e Prolog. Seu nome é em homenagem ao matemático Haskell Curry, em especial pela sua contribuição em lógica combinatória. Foi implementada por Michael Hanus e Sergio Antoy.

## Programa

Um programa em Curry especifica a semântica das expressões e ao executar significa simplificar uma expressão até que um único valor (junto com ligações de variáveis livres) seja calculado, ou seja, consistem em um conjunto de declarações de tipo e função. As declarações de tipo definem os domínios computacionais (construtores) e as declarações de função as operações nesses domínios. Predicados lógicos podem ser representados como funções com tipo de retorno Bool.

Curry é fortemente tipada. Caso a variável seja declarada sem especificar o tipo, o sistema de inferência de tipos determinará o tipo da variável a partir do sistema de tipos de Hindley/Milner, o qual implementa o polimorfismo paramétrico, em que realiza o lambda calculus para determinar o melhor tipo de dado na expressão e atribuir às variáveis com tipos omissos (STAGNI, 2011). Como o nome sugere, esse sistema decorre do trabalho de Roger Hindley e, posteriormente, de Robin Milner e é essencial para grande parte das linguagens funcionais estaticamente tipadas.

A linguagem utiliza a técnica de avaliação preguiçosa, isto é, a avaliação de uma expressão é retardada até o momento em que o valor é necessário. Isso evita que o programa faça avaliações desnecessárias.

## Igualdade e Restrições

Para comparar duas expressões, existe um predicado pré-definido `'=='` a qual testa o seu tipo de dado. Exemplo: a expressão  $e_1 == e_2$  é reduzida para *True* se  $e_1$  e  $e_2$  tem sua redução de dados iguais, caso contrário é retornado *False*, esse tipo de comparação é denominada igualdade estrita .

O operador `'<math>:=</math>'` é utilizado para fazer exigência que duas coisas são iguais, ou seja, uma restrição de equação, como o exemplo abaixo mostra.

Ex.:

```
last xs | ys++[e] == xs = e where ys, e free
```

## Alta ordem

Curry suporta algumas técnicas de programação funcional como função parcial e abstrações lambda.

Em situações em que uma função é usada apenas uma vez não é preciso atribuir um nome a ela. Para tais casos, funções anônimas (abstrações -  $\lambda$ ).

Ex.:

$$\lambda x_1 \dots x_n \rightarrow e$$

## Tipos de Dados

### Boolean

Os valores booleanos são definidos pela declaração do tipo de dados

Ex.:

```
data Bool = True | False
```

A conjunção operador `&&` é associativo a esquerda:

EX.:

```
True && y = y
False && x = False
```

Da mesma forma, a disjunção (sequencial) `||` e a negação **not** são definidos como de usual. Sendo mais utilizados em condicionais (`if_then_else`).

Ex.:

```
if_then_else :: Bool → a → a → a
if_then_else b t f = case b of True → t
                          False → f
```

Uma função com tipo de resultado `Bool` é frequentemente chamado de predicado, tem como padrão pré-definidos `"=="` e `"<="`. Além disso, há também predicados integrados para comparar objetos, como `"<"`. Os dados definidos pelo usuário são comparados na ordem de sua definição nas declarações de tipo de dados e recursivamente nos argumentos.

EX.:

```
data Coin = Head | Tail
```

Pode também comparar expressões contendo variáveis livres. Por exemplo a avaliação `"x < [Tail]"` retornando `True` se `{x = []}` e `{x = (Head:_)}`. Para números ou caracteres, pode haver suspensão ao comparar valores desconhecidos.

## Função

Uma função do tipo  $t_1 \rightarrow t_2$  é uma função que produz um valor do tipo  $t_2$  para cada argumento do tipo  $t_1$ . Caso a função tenha múltiplos parâmetros, é utilizada a técnica de *currying*, a qual, ao invés de uma única função receber todos estes parâmetros, é feita uma cadeia de função em que cada função recebe um único parâmetro (ELLIOTT, 2018). Assim, dado uma função:

$f: (X \times Y) \rightarrow Z$  com a técnica de currying ela é tratada da seguinte forma:

$f: X \rightarrow (Y \rightarrow Z)$

Assim, caso temos uma função com  $t_n$  parâmetros, o parâmetro  $t_1$  retorna uma nova função que recebe como parâmetro  $t_2$ , que gera uma nova função e recebe o próximo parâmetro, a chegar em  $t_n$  que retorna o resultado do último tipo ( $t_{n+1}$ )

**EX.:**

$t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t_{n+1}$

## Int

Os valores inteiros comuns, como “14” ou “-14”, são considerados construtores (constantes) do tipo *int*. Os operadores usuais, como + ou \*, são funções pré-definidas que são avaliadas apenas se ambos os argumentos forem valores inteiros.

## Float

Assim como para inteiros, valores como “3.1423” ou “5.0e-4” são considerados construtores do tipo *float*.

## List

O tipo [t] denota todas a lista cujos elementos são valores do tipo t. O tipo de listas pode ser considerado como pré-definido pela declaração:

**data [a] = [ ] | a : [a]**

Logo, [ ] representa uma lista vazia e  $x : xs$  é uma lista não vazia, na qual consiste o primeiro elemento x e restante xs. Também é comum representar a lista por colchetes, como mostra no exemplo a seguir:

**[e<sub>1</sub>, e<sub>2</sub>, ..., e<sub>n</sub>]**

Observa-se que a lista  $e_1 : e_2 : \dots : e_n : [ ]$  (é equivalente  $e_1 : (e_2 : (\dots : (e_n : [ ]) \dots))$ ) sendo ‘:’ é associativo à direita). O símbolo : serve para separar a construção, sendo uma desconstrução de lista similar à Haskell

## Characters

Valores como 'a' ou '0' são constantes do tipo char. Caracteres especiais podem ser denotados com uma barra invertida, por exemplo, '\n' ou '\228' para o caractere correspondente na tabela ASCII de posição 228.

## Tuples

Se  $t_1, t_2, \dots, t_n$  são tipos e  $n \geq 2$ , então  $(t_1, t_2, \dots, t_n)$  denotam os tipos de n-tuplas. O tipo unitário  $()$  tem um único elemento  $()$  e pode ser definido por,

$\text{data } () = ()$

que também pode ser interpretado como uma 0-tupla.

## Expressões

As expressões são parte essencial do funcionamento do Curry. Há dois princípios que regem o funcionamento do Curry (HANUS, 2000):

1. Se existe uma solução, ela (ou uma mais genérica) será computada;
2. Se a expressão pode ser reduzida a um valor, Curry computa esse valor;

Para garantir o primeiro ponto, Curry utiliza a estratégia de estreitamento preguiçoso (*lazy narrowing strategy*). Ela é implementada com o uso de *backtracking* para busca de uma solução, similar ao Prolog, o que pode gerar o problema da incompletude da expressão. Por outro lado, isso possibilita que, caso não haja um valor possível para atribuir a uma expressão, será retornada a expressão em uma forma genérica. No entanto, caso o programador prefira outras estratégias para avaliação das expressões, ele pode especificar em seu programa restrições a serem seguidas pelo compilador na execução do programa.

## Sequências Aritméticas

Existem quatro notações para sequências aritméticas, sendo elas:

- $[e \dots]$  que equivale a **enumFrom**  $e$ , a qual denota uma lista infinita iniciada por  $e$  com passo de 1:  $[e, e + 1, e + 2, \dots]$ ;
- $[e_1, e_2, \dots]$  que equivale a **enumFromThen**  $e_1 e_2$ , a qual denota uma lista infinita iniciada por  $e_1$  com passo de  $i$ , com  $i = e_2 - e_1$ :  $[e_1, e_1 + i, e_1 + 2 * i, \dots]$ ;
- $[e_1 \dots e_2]$  que equivale a **enumFromTo**  $e_1 e_2$ , a qual denota uma lista finita iniciada por  $e_1$  até  $e_2$  com passo de 1:  $[e_1, e_1 + 1, e_1 + 2, \dots, e_2]$ ;

- $[e_1, e_2 .. e_3]$  que equivale a **enumFromThenTo**  $e_1 e_2 e_3$ , a qual denota uma lista finita iniciada por  $e_1$  até  $e_2$  com passo de  $i$ , com  $i = e_2 - e_1$ :  $[e, e + i, e + 2 * i, \dots, e_3]$ ;

## Variáveis Livres

Como Curry pretende cobrir o paradigma funcional e o paradigma lógico, expressões precisam conter variáveis livres, isto é, variáveis não ligadas e nem instanciadas. A ideia é reduzir a expressão a até um dado previamente declarado ou um valor computável. Dessa forma, **toda variável livre x deve ser explicitamente declarada como livre**, seguindo a notação:  $x \text{ free}$  (HANUS, 2016).

## Exemplo de Programa

```
data Person = Christine | Maria | Monica | Alice | Susan |
Antony | Bill | John | Frank | Peter | Andrew
```

```
married :: Person -> Person -> Bool
married Christine Antony = True
married Maria Bill = True
married Monica John = True
married Alice Frank = True
```

```
mother :: Person -> Person -> Bool
mother Christine John = True
mother Christine Alice = True
mother Maria Frank = True
mother Monica Susan = True
mother Monica Peter = True
mother Alice Andrew = True
```

```
father :: Person -> Person -> Bool
father f c | married m f & mother m c = True where m free
```

No topo, tem-se *Person* como a base de dados, similar aos fatos em Prolog. Posteriormente, há a função *married* que tem como parâmetro inicial *Person*, pelo mecanismo de currying recebe-se o segundo parâmetro *Person* posteriormente e retorna ao final do processo um *Bool*. Abaixo, há os casos em que é retornado *True*. De maneira análoga, ocorre com a função *mother*.

Já a função *Father* é feita por uma relação dedutiva. Os elementos *f* e *c* são variáveis ligadas, enquanto *m* é uma variável livre. Dessa forma, a expressão:

father John child

Será avaliada da seguinte forma: as duas condições para *father* deverão ser atendidas, a de que *John* seja casado com *m* e que *m* seja mãe de *child*. Para a primeira condição, a função *married* recebe como primeiro parâmetro *m*, que, por ser uma variável livre, será substituída por todos os dados de *Person* até que seja encontrado o dado que retorne *True* para o segundo parâmetro *John*. Já o segunda condição parte da resposta para a variável *m*, sendo verificado qual dado é retornado *True* para *child* na função *mother* tendo como parâmetro inicial *Monica*.

Assim, tem-se como resposta para a expressão inicial:

{child=Susan} | {child=Peter}

## Referências Bibliográficas

ELLIOTT, Eric. **Curry and Function Composition**. [S. l.: s. n.], 2018. Disponível em:

<https://medium.com/javascript-scene/curry-and-function-composition-2c208d774983>.

Acesso em: 31 mar. 2021.

HANUS, Michael. **Curry: An Integrated Functional Logic Language**. [S. l.: s. n.], 2016. Disponível em:

<https://www-ps.informatik.uni-kiel.de/currywiki/media/documentation/report.pdf>.

Acesso em: 31 mar. 2021.

HANUS, Michael; KUCHEN, Herbert; AACHEN, Rwth; LI, Informatik. **Curry: A Truly Functional Logic Language**. [S. l.: s. n.], 2000. Disponível em:

[https://www.researchgate.net/publication/2463383\\_Curry\\_A\\_Truly\\_Functional\\_Logic\\_Language](https://www.researchgate.net/publication/2463383_Curry_A_Truly_Functional_Logic_Language). Acesso em: 31 mar. 2021.

STAGNI, Henrique. **XBA: Uma linguagem orientada a objetos com tipagem estrutural**. 2011. 27 f. Monografia (Graduação) - Curso de Ciência da Computação, Universidade de São Paulo (USP), São Paulo, 2011. Disponível em:

[https://bcc.ime.usp.br/tccs/2011/henrique/monografia\\_pre.pdf](https://bcc.ime.usp.br/tccs/2011/henrique/monografia_pre.pdf). Acesso em: 31 mar. 2021.