8 FUNÇÕES RECURSIVAS

8.1 Funções Recursivas de Kleene

8.1.1 Substituição Composicional
8.1.2 Recursão Primitiva
8.1.3 Minimização
8.1.4 Função Recursiva Parcial e Total

8.2 Cálculo Lambda

8.2.1 Aspectos Gerais do Cálculo Lambda
8.2.2 Linguagem Lambda
8.2.3 Variável Livre e Substituição
8.2.4 Cálculo Lambda
8.2.5 Tipos de Dados Básicos
8.2.6 Recursão e Ponto Fixo
8.2.7 Cálculo Lambda e Computabilidade

8.3 Funções Recursivas e Ciência da Computação

8.3.1 Importânca das Funções Recursivas
8.3.2 Linguagem de Progamação Funcional

8.4 Conclusões

8 FUNÇÕES RECURSIVAS

Formalismos para especificar algoritmos

- ***Operacional***

  Define-se uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado.

Exemplos: formalismos Máquina Norma e Máquina de Turing

- ***Axiomático.***

  Associam-se regras às componentes da linguagem. As regras permitem afirmar o que será verdadeiro após a ocorrência de cada cláusula, considerando o que era verdadeiro antes da ocorrência.

Exemplos: Gramática. Dependendo de restrições feitas na definição das gramáticas é possível estabelecer uma hierarquia, conhecida como ***Hierarquia de Chomsky***

Autômatos Finitos ⇔ Gramáticas Regulares
Autômatos Não-Determinísticos com Uma Pilha ⇔ Gramáticas Livres do Contexto;
Máquinas Universais ⇔ Gramáticas Irrestritas.

- ***Denotacional ou Funcional.***

  Trata-se de uma função construída a partir de funções elementares de forma composicional no sentido em que o algoritmo denotado pela função pode ser determinado em termos de suas funções componentes.

Exemplos: *Funções Recursivas Parciais* introduzidas por Kleene (1936), as quais são funções parciais definidas recursivamente.

➢ Foi provado que a Classe das Funções Turing-Computáveis era igual à Classe das Funções Recursivas Parciais.

➢ verifica-se que a substituição composicional de funções naturais simples: Função número zero; sucessor; projeção; juntamente com recursão primitiva e minimização, constituem uma forma compacta e natural para definir muitas funções e suficientemente poderosa para descrever toda função intuitivamente computável.

8.1 Funções Recursivas de Kleene

As funções recursivas parciais propostas por Kleene são funções construídas sobre funções básicas

- natural zero visto como uma função;
- sucessor (de um número natural);
- projeção (na realidade uma família de funções, pois depende do número de componentes, bem como de qual componente deseja-se projetar);

juntamente com as seguintes operações:

- substituição composicional (generaliza o conceito usual de composição de funções);
- recursão primitiva (definição de uma função em termos dela mesma);
- minimização (busca, em um tempo finito, o menor valor para o qual uma certa condição ocorre);

constituindo uma forma compacta e natural para definir muitas funções e suficientemente poderosa para descrever toda função intuitivamente computável.

8.1.1 Substituição Composicional

Definição 8.1
Substituição Composicional de Funções.

Sejam g, f_1, f_2, …, f_k funções parciais tais que:

$g = g(y_1, y_2, \ldots, y_k): N^k \to N$

$f_i = f_i(x_1, x_2, \ldots, x_n): N^n \to N$, para $i \in \{1, 2, \ldots, k\}$

A função parcial h tal que: $h = h(x_1, x_2, \ldots, x_n): N^n \to N$
é a *Substituição Composicional de Funções* definida a partir de g, f_1, f_2, …, f_k por:

$$h(x_1,x_2,\ldots, x_n)= g(f_1 (x_1, x_2,\ldots, x_n), f_2 (x_1, x_2,\ldots, x_n),\ldots, f_k(x_1,x_2,\ldots, x_n))$$

A função parcial h é dita *definida* para $(x_1,x_2,\ldots x_n)$ se, e somente se:

- $f_i(x_1, x_2,\ldots, x_n)$ é definida para todo $i \in \{1, 2, \ldots, k\}$
- $g(f_1(x_1, x_2,\ldots, x_n), f_2 (x_1, x_2,\ldots, x_n),\ldots, f_k(x_1, x_2,\ldots, x_n))$ é definida.

Portanto, a substituição composicional generaliza a composição usual de funções, construindo h a partir da substituição dos y_i em $g(y_1, y_2,\ldots, y_k)$ pelos correspondentes $f_i(x_1, x_2,\ldots, x_n)$, para todo $i \in \{1, 2,\ldots, k\}$.

Exemplo 8.1
Substituição de Funções - Funções Constantes.

Suponha as seguintes funções:

f_{zero}: N → N tal que, ∀x ∈ N, $f_{zero}(x) = 0$ função constante zero

sucessor: N → N tal que, ∀x ∈ N, sucessor(x) = x + 1 função sucessor

adição: N^2 → N tal que, ∀x,y ∈ N, adição(x, y) = x + y função adição

As seguintes funções são definidas, usando substituição de funções:

f_{um} = sucessor(f_{zero}): N → N função constante um

f_{dois} = sucessor(f_{um}): N → N função constante dois

$f_{três}$ = adição(f_{um}, f_{dois}): N → N função constante três

Exemplo 8.2
Substituição de Funções - Média de uma Turma de Alunos

Para melhor visualizar a idéia da substituição em um contexto aplicado, suponha uma turma com k alunos na qual foram realizadas n provas sendo a nota final de cada aluno i ∈ { 1, 2,…, k } dada pela função $nota_i(p_1, p_2,\ldots, p_n)$.

Assim, o cálculo da média de toda turma considerando as provas é dada pela função turma a qual é a media das notas dos alunos, ou seja:

$turma(p_1, p_2,\ldots, p_n) =$
$media(nota_1(p_1, p_2,\ldots, p_n), nota_2(p_1, p_2,\ldots, p_n),\ldots, nota_k(p_1, p_2,\ldots, p_n))$

8.1.2 Recursão Primitiva

Definição 8.2
Recursão Primitiva

Sejam f e g funções parciais tais que:

$f = f(x_1, x_2,\ldots, x_n): N^n \to N$

$g = g(x_1, x_2,\ldots, x_n, y, z): N^{n+2} \to N$

A função parcial h tal que: $h = h(x_1, x_2,\ldots, x_n, y): N^{n+1} \to N$
é definida por *Recursão Primitiva* a partir de f e g como segue:

$$h(x_1, x_2,\ldots, x_n, 0) = f(x_1, x_2,\ldots, x_n)$$

$$h(x_1, x_2,\ldots, x_n, y + 1) = g(x_1, x_2,\ldots, x_n, y, h(x_1, x_2,\ldots, x_n, y))$$

A função h é dita *definida* para $(x_1, x_2,\ldots, x_n, y)$ se, e somente se:

- $f(x_1, x_2,\ldots, x_n)$ é definida;
- $g(x_1, x_2,\ldots, x_n, i, h(x_1, x_2,\ldots, x_n, i))$ é definida para todo $i \in \{ 1,2,\ldots,y \}$

Exemplo 8.3
Recursão Primitiva - Adição.

Suponha as seguintes funções:

id(x) = x — função identidade $N \to N$

sucessor(x) = x + 1 — função sucessor $N \to N$

proj33 (x,y,z) = z função projeção da 3ª componente da tripla $N^3 \to N$

A função adição nos naturais tal que: $N^2 \to N$

adição(x,y) = x + y

é definida usando recursão como segue:

adição (x, 0) = id(x)

adição (x, y + 1) = proj33 (x, y, sucessor(adição (x, y)))

Por exemplo, adição (3, 2) é como segue:

adição (3, 2) =

= proj33 (3, 1, sucessor(**adição (3, 1)**)) =

= proj33 (3, 1, sucessor(proj33 (3, 0, sucessor (**adição (3,0)**)))) =

= proj33 (3, 1, sucessor(proj33 (3, 0, sucessor (id(3))))) =

= proj33 (3, 1, sucessor(proj33 (3, 0, sucessor (**3**)))) =

= proj33 (3, 1, sucessor(proj33 (3, 0, 4))) =

= proj33 (3, 1, sucessor(**4**)) =

= proj33 (3, 1, 5) =

= **5**

8.1.3 Minimização

O conceito de minimização que segue não é intuitivo na noção de recursão. Entretanto, é fundamental para garantir que a Classe de Funções Recursivas Parciais definida adiante possa conter qualquer função intuitivamente computável.

Definição 8.3
Minimização.

Seja f uma função parcial tal que: $f(x_1, x_2,\ldots, x_n, y): N^{n+1} \to N$
A função parcial h tal que:

$$h(x_1, x_2,\ldots, x_n): N^n \to N$$

é dita definida por *Minimização* de f e é tal que:

$$h(x_1,x_2,..,x_n)=\min\{y \mid f(x_1,\ldots,x_n,y)=0 \text{ e, } (\forall z)\ z<y,\ f(x_1,\ldots, x_n,z) \text{ é definida } \}$$

Portanto, a função h, para o valor $(x_1,\ldots, x_n)$, é definida como o menor natural y tal que $f(x_1,\ldots, x_n, y) = 0$.

Adicionalmente, a condição:

$$\forall z \text{ tal que } z < y,\ f(x_1,\ldots, x_n, z) \text{ é definida}$$

garante que é possível determinar, em um tempo finito, se, para qualquer valor z menor do que y, $f(x_1,\ldots, x_n, z)$ é diferente de zero. Note que a função h é parcial .

Por simplicidade, no texto que segue, para uma função h definida por minimização de f, a seguinte notação é adotada (compare com a definição acima):

$$h(x_1, x_2,\ldots, x_n) = \min\{ y \mid f(x_1,\ldots, x_n, y) = 0 \}$$

Observação 8.4
Valores Constantes como Funções

Todo valor constante pode ser visto como uma função. Tal artifício é importante, pois respeita o princípio denotacional (funções construídas composicionalmente a partir de funções elementares e assim sucessivamente). Assim, uma constante a (vista como função) quando compostas com uma função f resultando em $f \circ a$, representa a aplicação da função f à constante a, ou seja, f(a).

A técnica usada para representar valores constantes como funções, baseia-se no fato de que o número de funções que existem a partir de um conjunto unitário 1 (com um único elemento) para outro conjunto A, é o cardinal de A.

Portanto, existe uma função para denotar cada elemento de A. Assim, por exemplo, para os conjuntos 1 = { * } e Booleano = { v, f }, existem as seguintes funções:

$const_v$: 1 → Booleano tal que $const_v$(*) = v denota a constante v
$const_f$: 1 → Booleano tal que $const_f$(*) = f denota a constante f

Seguindo o princípio denotacional, para a função negação: Booleano → Booleano, tem-se que:

$const_f$ = negação $\circ$ $const_v$
$const_v$= negação $\circ$ $const_f$

Outra técnica muito comum para denotar constantes com funções é usando funções sem domínio como, por exemplo:

$const_f$: → Booleano denota a constante f

Entretanto, essa técnica é incoerente com a definição de função usada neste livro.

Exemplo 8.4
Minimização – Função Número Zero.

Suponha a função constante $f_{zero}(x) = 0: N \to N$.

Seja $const_{zero}$ o natural 0 visto como função

$const_{zero}: 1 \to N$ função número zero: $const_{zero}(*) = 0$

A função número $const_{zero}$ pode ser definida, usando minimização, como segue:

$$const_{zero} = \min\{ y \mid f_{zero}(y) = 0 \}$$

Exemplo 8.5
Minimização, Recursão Primitiva - *antecessor*.

Sejam a função número $const_{zero}$, e a função de projeção:

$proj\ 2_1 = proj\ 2_1\ (x,y) = x: N^2 \to N$ função projeção da 1ª componente do par

A seguinte função antecessor nos naturais:

$$antecessor\ (x): N \to N$$

pode ser definida usando recursão primitiva (supondo que antecessor de 0 é 0)

$antecessor\ (0) = const_{zero}$

$antecessor\ (y + 1) = proj\ 2_1\ (\ y,\ antecessor(y)\ )$

antecessor(2) = **(1)**

$= proj\ 2_1\ (\ 1,\ \mathbf{antecessor\ (1)}\ ) =$

$= proj\ 2_1\ (\ 1,\ proj\ 2_1\ (\ 0,\ \mathbf{antecessor\ (0)}\ )\ ) =$

$= proj\ 2_1\ (\ 1,\ proj\ 2_1\ (\ 0,\ const_{zero})\ ) =$

$= proj\ 2_1\ (\ 1,\ proj2_1\ (\ 0,\ 0\ )\ ) =$

$= proj\ 2_1\ (\ 1,\ 0\ ) =$

$= 1$

Repare que foi necessário usar uma função de projeção. Observe que após **(1)**, a aplicação de projeção seria suficiente para obter o resultado desejado. As demais etapas são apenas para satisfazer a definição de recursão primitiva.

Exemplo 8.6
Minimização, Recursão Primitiva - Subtração.

Sejam a função número $const_{zero}$ (O número zero natural), a função antecessor e as seguintes funções:

Id(x) = x: N → N   função identidade

proj 33 (x,y,z)= z   função projeção da 3ª componente da tripla

sub(x, 0) = id(x)
sub(x, y + 1) = antecessor o proj33(x, y, sub(x, y))

Por exemplo, sub(3, 2) é como segue:

sub(3, 2) =
antecessor o proj33(3, 1, **sub(3, 1)**) =
antecessor o proj33(3, 1, antecessor o proj33(3, 0, **sub(3, 0)**)) =
antecessor o proj33(3, 1, antecessor o proj33(3, 0, id(3))) =
antecessor o proj33(3, 1, antecessor o proj33(3, 0, 3)) =
antecessor o proj33(3, 1, antecessor (3)) =
antecessor o proj33(3, 1, 2) =
antecessor (2) =
1

8.1.4 Função Recursiva Parcial e Total

- Funções recursivas parciais são definidas a partir de três funções básicas:
  - natural zero visto como uma função;
  - sucessor (de um número natural);
  - projeção (na realidade uma família de funções, pois depende do número de componentes, bem como de qual componente deseja-se projetar).

Definição 8.5
Função Recursiva Parcial.

Uma *Função Recursiva Parcial* é indutivamente definida como segue:

a) ***Funções Básicas***. As seguintes funções são recursivas parciais:

$\text{const}_{\text{zero}}(\,*\,) = 0: 1 \to N$ função número zero

$\text{sucessor}(x) = x + 1: N \to N$ função sucessor

$\text{proj}\ n_i\,(x_1,x_2,\ldots,x_n) = x_i: N^n \to N$ projeção: i-ésima componente da n-upla

b) ***Substituição Composicional de Funções.*** Se as seguintes funções são recursivas parciais :

$g(y_1, y_2,\ldots, y_k): N^k \to N$

$f_i(x_1,x_2,\ldots,x_n): N^n \to N$, para todo $i \in \{1, 2, \ldots, k\}$

então a seguinte função é recursiva parcial :

$h(x_1,x_2,\ldots,x_n): N^n \to N$

a qual é definida pela *substituição composicional de funções* a partir de $g,f_1,f_2,\ldots,f_k$ por:

$$\mathbf{h(x_1,x_2,\ldots,x_n) = g(\, f_1(x_1,x_2,\ldots,x_n),\, f_2(x_1,x_2,\ldots,x_n),\ldots,\, f_k(x_1,x_2,\ldots,x_n)\,)}$$

c) ***Recursão Primitiva.*** Se as seguintes funções são recursivas parciais :

$f(x_1,x_2,\ldots,x_n): N^n \to N$

$g(x_1,x_2,\ldots,x_n, y, z): N^{n+2} \to N$

então a seguinte função é recursiva parcial :

$h(x_1,x_2,\ldots,x_n, y): N^{n+1} \to N$

a qual é definida por *recursão primitiva* a partir de f e g como segue:

$$\mathbf{h(x_1,x_2,\ldots,x_n, 0) = f(x_1,x_2,\ldots,x_n)}$$
$$\mathbf{h(x_1,x_2,\ldots,x_n, y + 1) = g(\, x_1,x_2,\ldots,x_n, y, h(x_1,x_2,\ldots,x_n, y)\,)}$$

d) ***Minimização.*** Se a seguinte função é recursiva parcial:

$f(x_1,x_2,\ldots,x_n, y): N^{n+1} \to N$

então a seguinte função é recursiva parcial:

$h(x_1,x_2,\ldots,x_n): N^n \to N$

a qual é definida por *minimização* de f, como segue:

$$\mathbf{h(\, x_1,x_2,\ldots,x_n) = min\{\, y \mid f(x_1,x_2,\ldots,x_n, y) = 0\,\}}$$

Exemplo 8.7
Funções Recursivas Parciais.

a) *Função Identidade*. id (x)= x: N $\rightarrow$ N
é recursiva parcial, pois é uma função básica de projeção, ou seja:
id = proj 1_1

b) As seguintes funções são recursivas parciais:

adição: $N^2 \rightarrow N$	função adição
sub: $N^2 \rightarrow N$	função subtração
f_{zero}: N $\rightarrow$ N	função constante zero
f_{um}: N $\rightarrow$ N	função constante um
f_{dois}: N $\rightarrow$ N	função constante dois
$f_{três}$: N $\rightarrow$ N	função constante três
$const_{zero}$: 1 $\rightarrow$ N	função número zero
antecessor: N $\rightarrow$ N	função antecessor

Definição 8.6
Função Recursiva Total.

Uma Função Recursiva Total *é uma função recursiva parcial definida para todos os elementos do domínio.*

Teorema 8.7
Funções Recursivas $\times$ Funções Turing-Computáveis.

As seguintes classes de funções são equivalentes:
a) Funções Recursivas Parciais e Funções Turing-Computáveis;
b) Funções Recursivas Totais e Funções Turing-Computáveis Totais.

A relação entre classes também pode ser estabelecida:

- Funções Recursivas Parciais $\Leftrightarrow$ Linguagens Enumeráveis Recursivamente;
- Funções Recursivas Totais $\Leftrightarrow$ Linguagens Recursivas.

8.2 Cálculo Lambda

- transcende o estudo da computabilidade, tendo importantes aplicações no estudo da lógica e das linguagens de programação.
- foi a inspiração de muitas *linguagens de programação funcionais* como, por exemplo, *Lisp* e *Haskell*.

8.2.1 Aspectos Gerais do Cálculo Lambda

Considere a função $f: N \rightarrow N$ tal que: $f(x) = x + 1$

É importante reparar que o que efetivamente está sendo definido não é a função f propriamente dita, mas sim f(x), ou seja, o resultado da aplicação de f ao argumento x (o qual é suposto assumir valores nos números naturais).

- $f(x) = x + 1$ não é uma função,
- mas uma equação, pois $x + 1 - f(x) = 0$.

Linguagem Lambda

Uma forma de definir funções com mais rigor

termo lambda - elementos que definem ou representam funções

a) *Abstração Lambda*. Permite abstrair a definição da função.

$$\lambda x.(x + 1)$$

função tal que, para um argumento arbitrário x resulta em x + 1

b) *Aplicação Lambda*. Determina o valor da função aplicada a um dado argumento.

$$(\lambda x.(x + 1))\ 3$$

aplicação da função λx.(x + 1) ao valor 3

A notação λ permite representar e diferenciar regra de associação (função) da aplicação da mesma a um argumento.

A notação deixa claro qual variável é o dado de entrada.

Como primeiro exemplo, considere a função sucessor $\lambda x.(x + 1)$, a qual faz a associação da regra $x \rightarrow x + 1$.

Caso de funções de mais de uma variável (adição de números)

- pode parecer necessário definir uma abstração lambda para cada variável.
- uma função de duas variáveis pode ser vista como uma função de uma variável quando se fixa um dos parâmetros.
- Exemplo: a função de adição, fixando o segundo argumento no valor **1,** obtém-se a função sucessor **λx.(x + 1).**
- O mesmo raciocínio permite construir as funções correspondentes às regras **$x \rightarrow x + 2$, $x \rightarrow x + 3$**,…

Generalizando este raciocínio, a adição pode ser representada pelo seguinte termo lambda: **$\lambda y.(\lambda x.\ (x+y))$**

Dessa forma, somente com as operações de abstração lambda e de aplicação lambda, é possível representar funções com qualquer número de variáveis.

Resumidamente,

Cálculo Lambda é a Linguagem Lambda juntamente com um conjunto de axiomas e regras que permitem inferir quando dois termos lambda denotam a mesma função.

8.2.2 Linguagem Lambda

Linguagem lambda é o conjunto dos termos lambdas sobre um conjunto de variáveis.

Definição 8.8
Termo Lambda.

- Seja V um conjunto contável (infinito e enumerável) de *variáveis*
- Então um *λ–termo*, *Termo Lambda*, *Expressão Lambda* ou *Palavra Lambda* sobre V é indutivamente definido por:

a) Toda variável $x \in V$ é um termo lambda;

b) *Abstração Lambda* ou λ-*Abstração*. Se M é um termo lambda e $x \in V$, então

$(\lambda x.M)$ é um termo lambda.

c) *Aplicação Lambda* ou λ-*Aplicação*. Se M e N são termos lambda, então:

$(M\ N)$ é um termo lambda;

Sintaxe de Expressões Lambda

```
<expression> :=    <variable>
                 | <constant>
                 | ( <expression> <expression> )
                 | ( λ <variable> . <expression> )
```

Simplificações de notação adotadas:

a) *Parênteses Externos*. Parênteses externos podem ser eliminados, Sejam M e N termos. M N é uma simplificação de (M N);

b) *Associatividade à Esquerda*. Parênteses podem ser eliminados, respeitando a propriedade associatividade. Sejam M, N e P termos. M N P é uma simplificação de ((M N) P);

c) *Escopo de uma Variável*. Parênteses podem ser eliminados, respeitando o escopo de uma variável em uma abstração. Sejam M, N e P termos, e x a variável. λx•M N P é uma simplificação de λx•(M N P);

Observe que:

-
- não existem constantes na definição de um λ-termo. No Cálculo Lambda dito puro, não existem constantes. Constantes usadas são introduzidas por razões puramente didáticas;
- os λ-termos são anônimos, ou seja, não são explicitamente nomeados.

Exemplo 8.8

Sejam x, y e z variáveis.
As seguintes expressões são termos lambda:

x	i (uma variável por si só é um termo lambda)
(x y)	i, iii (aplicação de i duas vezes e de iii uma vez)
λx.x	i,ii
λx.y	i,ii
(λx.x λx.x)	i, ii, iii
(λz.z (x y))	i, ii, iii

Definição 8. 9
Linguagem Lambda.

Seja V um conjunto contável (infinito e enumerável) de variáveis. Uma *Linguagem Lambda* sobre V é o conjunto de todos os termos lambda sobre esse conjunto.

8.2.3 Variável Livre e Substituição

Definição 8.10
Variável Livre.

Sejam x e y variáveis tal que $x \neq y$ e N, P λ-termos.

A variável x é dita *Variável Livre* em um λ-termo,nos seguintes casos:

i) ***Variável*. ** x é **livre** em y. Em particular, x é livre em x
ii) ***Abstração Lambda*.**
x é **livre** em λy.N
iii) ***Aplicação Lambda*.**
Se x é **livre** em N e em P, então x é **livre** em N P

Definição 8.11
Variável Ligada

Uma variável em um λ-termo é dita *Variável Ligada* se ocorre no termo e **não é livre** nesse termo.

Exemplo 8.9 Variável Ligada e Variável Livre.

a) No termo **λx.x^{k}**, as variáveis **x** e **k** são ditas **ligada** e **livre**, respectivamente.
b) No termo **λk.λx.x^{k}**, *as variáveis* x e k são ditas **ligadas**
c) No termo **λx.(x y):** x é variável **ligada**; y é variável **livre**;
d) No termo **λx.λy.(x y):** x e y são variáveis **ligadas**
e) No termo **x λx.λy.(x y):** a primeira ocorrência da variável x é **livre**; a segunda ocorrência da variável x é **ligada**;
f) No termo **(λx.ax) x:** no **sub-termo ax**, as variáveis a e x são **livres**; no **sub-termo λx.ax**, a variável a é **livre** e a variável x é **ligada**; no **termo (λx.ax) x,** a variável x é ambos, **ligada** (primeira ocorrência) e **livre** (segunda ocorrência)

Definição 8.12
Substituição

A *Substituição* de todas as ocorrências (se existe alguma) da variável livre x por X em um λ-termo M é denotada como segue:

$$M\ [x \leftarrow X]$$

e é tal que (suponha M, M_1, M_2 λ-termos e x, y variáveis tais que $x \neq y$):

i) ***Variável*.**

i.1) $\mathbf{x}\ [x \leftarrow X] = X$

i.2) $y\ [x \leftarrow X] = y$

ii) ***Aplicação Lambda*.**

Suponha M_1, M_2 λ-termos. Então:

$$\mathbf{(}M_1\ M_2\mathbf{)}\ [x \leftarrow X] = (\ (M_1\ [x \leftarrow X])\ (M_2\ [x \leftarrow X])\ )$$

iii) ***Abstração Lambda*.**

iii.1) $\lambda x.M\ [x \leftarrow X] = \lambda x.M$ (x não é livre - não há substituição)

iii.2) $(\lambda y.M)\ [x \leftarrow X] = \lambda y.(\ M\ [x \leftarrow X]\ )$ se y não ocorre livre no sub-termo X ou x não ocorre livre no sub-termo M

iii.3) $(\lambda y.M)\ [x \leftarrow X] = \lambda z.(\ (\ M\ [y \leftarrow z]\ )\ [x \leftarrow X]\ )$ se y ocorre livre no sub-termo Xe x ocorre livre no sub-termo M; nesse caso, z não ocorre livre nos sub-termos M e X

O item **iii.3)** da definição acima previne substituições que mudem a característica livre ou ligada de uma variável no termo.

Por exemplo, $\lambda u.v\ [v \leftarrow u]$ resulta em $\lambda u.u$ fazendo com que uma substituição supostamente livre resulte em ligada.

Por esta razão, a variável z é introduzida, resultando em $\lambda z.u$,
ou seja:

$$\lambda z.(\ (v\ [u \leftarrow z]\ )\ [v \leftarrow u]\ ) = \lambda z.u$$

Exemplo 8.10 – Substituição

Para cada termo, na coluna da direita é apresentada a correspondente substituição, conforme a definição de substituição:

a) $x\,[x \leftarrow z] = z$ i.1

b) $y\,[x \leftarrow z] = y$ i.2

c) $(x\ y)\,[x \leftarrow z] =$ ii
$(x\,[x \leftarrow z])\ (y\,[x \leftarrow z]) =$ i.1
$z\ (y\,[x \leftarrow z]) =$ i.2
$z\ y$

d) $\lambda x.(x\ y)\,[x \leftarrow z] = \lambda x.(x\ y)$ iii.1

e) $(\lambda u.v)\,[v \leftarrow u] =$ iii.3 (u ocorre livre em u e v ocorre livre em v)
$\lambda z.(\,(v\,[u \leftarrow z])\,[v \leftarrow u]\,) =$ i.2
$\lambda z.(\,v\,[v \leftarrow u]\,) =$ i.1
$\lambda z.u$

f) $(\,(\,\lambda p.p\ (p\ q)\,)\ (\,\lambda r.(p\ r)\,)\,)\,[q \leftarrow p\ a] =$ ii
$(\,\lambda p.p\ (p\ q)\,)\,[q \leftarrow p\ a]\ \ (\,\lambda r.(p\ r)\,)\,[q \leftarrow p\ a] =$
iii.2 (r não ocorre livre em $p\ a$ e q não corre livre em $p\ r$)
$(\,\lambda p.p\ (p\ q)\,)\,[q \leftarrow p\ a]\ \ \lambda r.(\,(p\ r)\,[q \leftarrow p\ a]\,) =$ ii
$(\,\lambda p.p\ (p\ q)\,)\,[q \leftarrow p\ a]\ \ \lambda r.(\,p\,[q \leftarrow p\ a]\ r\,[q \leftarrow p\ a]\,) =$ i.2
$(\,\lambda p.p\ (p\ q)\,)\,[q \leftarrow p\ a]\ \ \lambda r.(\,p\ r\,[q \leftarrow p\ a]\,) =$ i.2
$(\,\lambda p.p\ (p\ q)\,)\,[q \leftarrow p\ a]\ \ \lambda r.(p\ r) =$ iii.3 (p ocorre livre em $p\ a$ e q corre livre em $p\ q$)
$\lambda z.(\,(\,p\ (p\ q)\,[p \leftarrow z]\,)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ ii
$\lambda z.(\,p\,[p \leftarrow z]\ (\,(p\ q)\,[p \leftarrow z])\,)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ i.1
$\lambda z.(\,z\ (\,(p\ q)\,[p \leftarrow z])\,)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ ii
$\lambda z.(\,z\ (\,p\,[p \leftarrow z]\ q\,[p \leftarrow z]\,)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ i.1
$\lambda z.(\,z\ (\,z\ q\,[p \leftarrow z]\,)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ i.2
$\lambda z.(\,z\ (z\ q)\,[q \leftarrow p\ a]\,)\ \ \lambda r.(p\ r) =$ ii
$\lambda z.(\,z\ (z\,[q \leftarrow p\ a]\ q\,[q \leftarrow p\ a])\,)\ \ \lambda r.(p\ r) =$ i.2
$\lambda z.(\,z\ (z\ q\,[q \leftarrow p\ a])\,)\ \ \lambda r.(p\ r) =$ i.1
$\lambda z.(\,z\ (z\ p\ a)\,)\ \ \lambda r.(p\ r)$ □

8.2.4 Cálculo Lambda

O Cálculo Lambda busca uma noção de igualdade, que tem por objetivo estabelecer quando dois λ-termos denotam a mesma função.

Definição 8.13
Cálculo Lambda
Cálculo Lambda *ou* λ-Cálculo *consiste na Linguagem Lambda munida dos axiomas e as regras de inferência abaixo.*

Sejam M, N e K λ-termos:

i) Principal axioma: $(\lambda x.M)\ N = M\ [x \leftarrow N]$

ii) Axiomas e regras lógicas:

ii.1) Igualdade

M = M (reflexiva)

se M = N, então N = M (simetrica)

se M = N e N = K, então M = K (transitiva)

ii.2) Regras de Compatibilidade

se M = N, então K N = K M (distribuitiva a esquerda)

se M = N, então N K = M K (distribuitiva a direita)

se M = N, então $\lambda x.N = \lambda x.M$

iii) Os termos M e N são _Iguais_ (M = N), se e somente se existe uma dedução de M = N no λ-Cálculo.

A partir da notação Lambda, tem-se uma linguagem e um cálculo.

- O cálculo objetiva verificar a igualdade de termos da linguagem.
- A noção de cálculo é explorada pelo conceito de *redução*, o qual pode ser interpretado como um "passo computacional" na busca de um "valor" representativo para um λ-termo qualquer.

Redução pode ser vista como uma operacionalização do λ-Cálculo.

- *Redução Alfa*, renomeação de variáveis ligadas;
- *Redução Beta*, aplicação de uma função a um argumento, via substituição.
- *Redução Iterada*, sucessiva aplicação de qualquer das reduções acima.

Definição 8.14
Redução Beta

Uma Redução Beta*, ou simplesmente* β-Redução*, denotada pelo símbolo* $\triangleright$*, é a seguinte transformação entre* λ*-termos:*

$$(\lambda x.M)N \triangleright M[x \leftarrow N] \qquad \beta\text{-redução}$$

o termo $(\lambda x.M)N$ é dito ser um *redex*,
$M[x \leftarrow N]$ é dito ser o seu *contractum*.

Definição 8.15
Redução Alfa

Uma *Redução Alfa*, ou simplesmente α-*Redução*, é a seguinte transformação entre λ-termos:

$$\lambda x.M \triangleright \lambda y.M[x \leftarrow y] \qquad \alpha\text{-redução}$$

onde y não ocorre livre no sub-termo M.

A α-*Redução* formaliza a relação de *equivalência alfabética*,
o termo "redução" e o símbolo $\triangleright$ são usados para α-redução ou β-redução.

Definição 8.16
Redução Iterada

A *Redução Iterada* denotada por $\triangleright^*$, é a sucessiva aplicação da redução $\triangleright$zero ou mais vezes.

Exemplo 8.11
Redução

a) $(\lambda x.2 * x + 1)\ 3 \rhd$
$(2 * x + 1)\ [x \leftarrow 3] = 2 * 3 + 1 = 7$

b) $(\lambda x.\lambda y.x + y)\ 5 \rhd$
$(\lambda y.x + y)\ [x \leftarrow 5] = \lambda y.5 + y$

c) $((\lambda x.\lambda y.x + y)\ 5)\ 7 \rhd$
$((\lambda y.x + y)\ [x \leftarrow 5]\ )\ 7 = (\lambda y.5 + y)\ 7 \rhd$
$(5 + y)\ [y \leftarrow 7] = 5 + 7 = 12$

d) $(\lambda x.\lambda y.x - y)\ (\lambda z.z / 2) \rhd$
$(\lambda y.x - y)\ [x \leftarrow \lambda z.z / 2] = \lambda y.(\lambda z.z / 2) - y$

e) $(\lambda x.x\ x)\ (\lambda y.y) \rhd$
$x\ x\ [x \leftarrow \lambda y.y] = (\lambda y.y)\ (\lambda y.y) \rhd$
$y\ [y \leftarrow \lambda y.y] = \lambda y.y$

f) $(\ (\lambda x.\lambda y.xy)\ (\lambda y.z)\ )\ x \rhd$
$(\lambda y.xy\ [x \leftarrow \lambda y.z])\ x \rhd$
$(\ \lambda y.(\lambda y.z)y\ )\ x \rhd$
$(\lambda y.z)\ y\ [y \leftarrow x] \rhd$
$(\lambda y.z)\ [y \leftarrow x]\ y\ [y \leftarrow x] \rhd$
$(\lambda y.z)\ x \rhd$
$(z)\ [y \leftarrow x]\ \rhd$
z

g) $(\lambda x.x\ x)\ (\lambda x.x\ x) \rhd$
$x\ x\ [x \leftarrow \lambda x.x\ x] = (\lambda x.x\ x)(\lambda x.x\ x)$

h) $(\ (\lambda x.\lambda y.\lambda z.(x\ y)\ z)\ (\lambda x.\lambda y.(x + y)\ )\ 3\ )\ 7 \rhd$
$(\ (\lambda y.\lambda z.(x\ y\ )\ z)\ [x \leftarrow \lambda x.\lambda y.(x + y)]\ 3\ )\ 7$ $=$
$(\ (\lambda y.\lambda z.(\lambda x.\lambda y.(x + y)\ y)\ z)\ 3\ )\ 7 \rhd$
$(\ (\lambda z.(\lambda x.\lambda\ \mathbf{y}.(x + \mathbf{y})\ y)\ z)\ [y \leftarrow 3]\ )\ 7 = (\ (\lambda z.(\lambda x.\lambda\ \mathbf{y}.(x + \mathbf{y})\ 3)\ z)\ )\ 7 \rhd$
$(\lambda x.\lambda y.(x + \mathbf{y})\ 3)\ z)\ [z \leftarrow 7] = \lambda x.\lambda y.(x + \mathbf{y})\ 3)\ 7 \rhd$
$(\lambda y.(x + y)\ [x \leftarrow 3])\ 7 = (\lambda y.(3 + y)\ )\ 7 \rhd$
$(3 + y)\ [y \leftarrow 7] = 3 + 7 = 10$

Observe que, nos primeiros quatro itens bem como no último, foi feito o uso "informal" de operações Matemáticas e de constantes que não pertencem propriamente ao λ-Cálculo.

8.2.5 Tipos de Dados Básicos

Cálculo Lambda pode ser usado para modelar tipos de dados

Os valores possíveis para os elementos do tipo de dado *Booleano* ou *Lógico* são verdadeiro e falso.

Ambos valores são definidos como funções (λ-termos) e são ilustrados a seguir (modelagem conhecida como *Booleanos de Church*).

Exemplo 8.12
Tipo Booleano

Os valores lógicos

verdadeiro := λx.λy.x

falso := λx.λy.y

verdadeiro seleciona o primeiro entre dois termos da entrada (projeção da primeira componente),
falso seleciona o segundo (projeção da segunda componente).

A operação de negação lógica ¬

¬ := ((λp.p falso) verdadeiro)

cálculo da negação lógica ¬ para o argumento falso é:

(λ**p**.p falso verdadeiro) **falso** ▷
falso falso verdadeiro ▷ falso:projeção da segunda componente
verdadeiro

cálculo da negação lógica ¬ para o argumento verdadeiro **é:**

(λ**p**.p falso verdadeiro) verdadeiro ▷
verdadeiro falso verdadeiro ▷ verdadeiro:projeção da primeira componente
falso

A operação de conjunção lógica ∧

$\wedge$(p,q) = (se p = falso, então falso; senão q)

pode ser representada pelo λ-termo:

∧ := λ p.λq. p q falso

Assim, se p for:

- verdadeiro, seleciona o primeiro argumento da lista seguinte, que será q, pois dele dependerá o valor da conjunção;
- falso, irá selecionar o segundo termo da lista, que é o λ-termo que, por definição, representa o valor falso.

cálculo da conjunção lógica ∧ p/ os argumentos verdadeiro e verdadeiro

(λ**p.**λq.p q falso) **verdadeiro** verdadeiro ▷
(λ**q.**verdadeiro q falso) **verdadeiro** ▷
verdadeiro verdadeiro falso ▷ projeção da primeira componente
verdadeiro

cálculo da conjunção lógica ∧ p/ os argumentos verdadeiro e falso

(λ**p.**λq.p q falso) **verdadeiro** falso ▷
(λ**q.**verdadeiro q falso) falso ▷
verdadeiro falso falso ▷ projeção da primeira componente
falso

cálculo da conjunção lógica ∧ p/ os argumentos falso e verdadeiro

(λ**p.**λq.p q falso) **falso** verdadeiro ▷
(λ**q.**falso q falso) **verdadeiro** ▷
falso verdadeiro falso ▷ projeção da segunda componente
falso

cálculo da conjunção lógica ∧ p/ os argumentos falso e falso

(λ**p.**λq.p q falso) falso falso ▷
(λ**q.**falso q falso) falso ▷
falso falso falso ▷ projeção da segunda componente
falso

A operação de disjunção lógica v

v (p,q) = (se p = verdadeiro, então verdadeiro; senão q)

pode ser representada pelo λ-termo:

v := λ p.λq. (p verdadeiro) q

cálculo da conjunção lógica v p/ os argumentos verdadeiro e verdadeiro

(λ**p**.λq.p verdadeiro q) **verdadeiro** verdadeiro ▷
(λ**q**. verdadeiro verdadeiro q) **verdadeiro** ▷
verdadeiro verdadeiro verdadeiro ▷ projeção da primeira componente
verdadeiro

cálculo da conjunção lógica v p/ os argumentos verdadeiro e falso

(λ**p**.λq.p verdadeiro q) **verdadeiro** falso ▷
(λ**q**. verdadeiro verdadeiro q) falso ▷
verdadeiro verdadeiro falso ▷ projeção da primeira componente
verdadeiro

cálculo da conjunção lógica v p/ os argumentos falso e verdadeiro

(λ**p**.λq.p verdadeiro q) falso verdadeiro ▷
(λ**q**. falso verdadeiro q) verdadeiro ▷
falso verdadeiro verdadeiro ▷ projeção da segunda componente
verdadeiro

cálculo da conjunção lógica v p/ os argumentos falso e falso

(λ**p**.λq.p verdadeiro q) falso falso ▷
(λ**q**. falso verdadeiro q) falso ▷
falso verdadeiro falso ▷ projeção da segunda componente
falso

A especificação: se p então x senão y

pode-se utilizar a construção apresentada para os Booleanos:

se-então-senão := λp. λx. λy.p x y

Modelagem conhecida como *Numerais de Church*.

Exemplo 8.13
Tipo Natural

Um número natural n

representa quantas vezes uma função f é aplicada a uma entrada x:

$$n := \lambda f.\lambda x.f^n x$$

$0 := \lambda f.\lambda x.x$

$1 := \lambda f.\lambda x.f(x)$

$2 := \lambda f.\lambda x.f(f(x))$

Observe que 0 e falso são representados pelo mesmo λ-termos - são equivalentes alfabeticamente.

A operação sucessor: $\text{sucessor} := \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$

Essa definição de sucessor prevê a utilização de equivalência alfabéticas para o termo que define o natural n (entrada).

se n é definido por $\lambda g.\lambda y.g^n(y)$, no cálculo do sucessor acarretará no termo final $\lambda f.\lambda x.f(f^n(x))$, que é equivalente à representação $\lambda f.\lambda x.f^{n+1}(x)$, definida para $n + 1$.

Por exemplo, o cálculo do sucessor para o argumento $1 := \lambda g.\lambda x.g(x)$ é:

$(\lambda n.\lambda f.\lambda x.f\ (n\ f\ x))\ (\lambda g.\lambda y.g(y)) \triangleright$

$\lambda f.\lambda x.f\ ((\lambda g.\lambda y.g(y))\ f\ x) \triangleright$ (observação: $[g \leftarrow f]$)

$\lambda f.\lambda x.f\ ((\lambda y.f(y))\ x) \triangleright$ (observação: $[y \leftarrow x]$)

$\lambda f.\lambda x.f\ (f(x)) =$

$\lambda f.\lambda x.f^{2}\ (x) = 2$

A operação de adição

pode ser definida através da aplicação repetida da função sucessor.

adição := λn.λm.n sucessor m

na adição de x com y o argumento x é associado com n que receberá como primeiro argumento a função (sucessor) que será aplicada repetida vezes (x vezes) ao segundo argumento (y que é associado com m).

Por exemplo, o cálculo da adição para os argumentos
2 := λf.λx.f^2(x) e
3 := λf.λx.f^3(x),

(λ**n** λm.n sucessor m) **2** 3 ▷
(λ**m**.2 sucessor m) **3** ▷
2 sucessor 3 = λ**f**.λx.f^2(x) **sucessor** 3▷
λ**x**.sucessor^2(x) **3** ▷
sucessor^2(3) = sucessor(sucessor(3)) = sucessor(4) = 5

8.2.6 Recursão e Ponto Fixo

- uma função não pode referenciar a si mesma diretamente,
- mas pode usar a si mesmo como parâmetro, permitindo então, uma recursão.
- o operador de ponto fixo implementa esta idéia, permitindo definir recursão no Cálculo Lambda.

Definição 8.17
Ponto Fixo

Seja $f: A \to A$ uma função.
Então $x \in A$ é um *Ponto Fixo* de f se: **f(x) = x**

- um ponto fixo de uma função é um elemento do domínio tal que, quando aplicado à função, resulta no próprio elemento.

Exemplo 8.14
Ponto Fixo

Para as seguintes funções, os correspondentes pontos fixos são apresentados na coluna da direita:

$id: N \to N$	qualquer $n \in N$
$sucessor: N \to N$	não tem
$n^2: N \to N$	0 e 1
$n^2 - 2n: N \to N$	0 e 3

Definição 8.18
Operador de Ponto Fixo

Um *Operador de Ponto Fixo* M é um λ–termo tal que, para qualquer λ–termo F, é fato que: **M F = F (M F)**

- um operador de ponto fixo é um λ–termo que gera um ponto fixo para qualquer função.

Exemplo 8.15
Operador de Ponto Fixo

O seguinte λ–termo é um operador de ponto fixo:

M := λf.(λx.f (x x)) (λx.f (x x))

De fato, M f é um ponto fixo de f, pois:

M f =
(λf.(λx.f (x x)) (λx.f (x x)) f ▷
(λx.f (x x)) (λx.f (x x)) [f ← f] = (λx.f (x x)) (λx.f (x x)) ▷

(f (x x)) [x ← λx.f (x x)] = f (λx.f (x x) λx.f (x x)) =
f (M f)

Exemplo 8.21
Fatorial

Considere o seguinte λ–termo

F := λr.λn.(se n = 0 então 1 senão n • (r (n - 1)))

o fatorial

Fat := M F

Por exemplo, o cálculo de Fat para o argumento 3, tem-se que:

M F 3 =
F (M F) 3 =
λr.λn.(se n = 0 então 1 senão n • (r (n - 1))) (M F) 3 =
(λn.(se n = 0 então 1 senão n • ((M F) (n - 1)))) 3 =
se 3 = 0 então 1 senão 3 • ((M F) (3 - 1)) =
3 • ((F (M F)) 2) =
3 • (λr.λn.(se n = 0 então 1 senão n • (r (n - 1))) (M F) 2) =
3 • ((λn.(se n = 0 então 1 senão n • ((M F) (n - 1)))) 2) =
3 • (se 2 = 0 então 1 senão 2 • ((M F) (2 - 1))) =
3 • (2 • (F (M F) 1)) =
3 • (2 • (λr.λn.(se n = 0 então 1 senão n • (r (n - 1))) (M F) 1))) =
3 • (2 • ((λn.(se n = 0 então 1 senão n • ((M F) (n - 1)))) 1)) =
3 • (2 • (se 1 = 0 então 1 senão 1 • ((M F) (1 - 1)))) =
3 • (2 • (1 • ((F (M F)) 0)) =
3 • (2 • (1 • (λr.λn.(se n = 0 então 1 senão n • (r (n - 1))) (M F) 0)) =
3 • (2 • (1 • ((λn.(se n = 0 então 1 senão n • ((M F) (n - 1)))) 0)) =
3 • (2 • (1 • (se 0 = 0 então 1 senão 0 • ((M F) (0 - 1))))) =
3 • (2 • (1 • 1)) =
3 • (2 • 1) =
3 • 2 =
6

8.2.7 Cálculo Lambda e Computabilidade

O λ-Cálculo é equivalente a qualquer linguagem de programação

Se um λ-termo for visto como um programa, e o processo de redução como sua execução, pode-se associar a cada termo da forma λx.M um programa que computa uma função com entrada x e corpo de comandos M (normalmente dependente de x).

O resultado de tal programa seria o termo obtido após um número finito de reduções e tal que não possa ser mais reduzido

a) O processo de reduções, em alguns casos, não pára;
b) O processo de reduções não é deterministico, pois as reduções não têm ordem preferencial de aplicação.

Exemplo 8.21
Redução não determinística

A seqüência escolhida de redução, produz "resultados" diferentes.

Seja: (λx.z) ((λy.y y) (λy.y y))

- aplicando-se sempre a redução ao redex mais à direita, resulta em uma seqüência infinita de aplicações.

  (λx.z) ((λy.y y) (λy.y y)) ▷
  (λx.z) ((λy.y y) (λy.y y)) ▷
  (λx.z) ((λy.y y) (λy.y y)) …

- aplicando-se a redução mais à esquerda, tem-se

  (λx.z) ((λy.y y) (λy.y y)) ▷ z

se um λ-termo possui resultado definido a partir de alguma seqüência de reduções, então esse resultado é único, isto é, qualquer outra seqüência que produza resultado definido produz o mesmo resultado.

Definição 8.19
Igualdade Induzida por Redução Beta

Sejam M e N dois λ-termos.
A *Igualdade Induzida por* β-*Redução*,
denotada por:

$$M =_\beta N$$

ocorre se e somente se existe um λ-termo K tal que (considerando a α-redução como incluída na relação $\triangleright$):

$$M \triangleright K \quad \text{e} \quad N \triangleright K$$

□

Teorema 8.20
Redução Cálculo Lambda e Função Computável

Uma função $f: N \to N$ é computável se e somente se existe um termo lambda F tal que, para qualquer $x, y \in N$:

$$F(x) = y \quad \text{se e somente se} \quad x =_\beta y$$

O teorema estabelece que o formalismo Cálculo Lambda é equivalente ao formalismo Máquina de Turing bem como aos demais formalismos equivalentes estudados.

8.3 Funções Recursivas e Ciência da Computação

8.3.1 Importância das Funções Recursivas

- O estudo das funções recursivas e da recursão em geral é de fundamental importância na Ciência da Computação.
- Não só são formalismos tão poderosos como as máquinas universais como fornecem uma abordagem (denotacional) diferente da operacional.
- A quase totalidade das linguagens de programação modernas como Pascal ou C possui recursão como um construtor básico de programas.
- As arquiteturas da maioria dos atuais computadores possuem facilidades para implementar recursão.

8.3.2 Linguagem de Programação Funcional

- aplicações das funções recursivas e do Cálculo Lambda, no contexto da programação funcional, usando como exemplo a linguagem *Haskell*.

- *Programação funcional* é um estilo de programação baseada em funções e composição de funções em vez de comandos.
- um **programa** é uma expressão funcional e não uma seqüência de comandos a serem executados.
- Uma ***linguagem de programação*** é ***funcional*** se suporta esse estilo de programação.
  - operações tipadas (ou seja, com tipos associados)
  - construtores que permitem definir novos tipos e operações mais complexos.
  - Uma linguagem de programação funcional considerada *pura* não possui variável e nem atribuições:

- tipos primitivos de dados da linguagem;
- constantes de cada tipo primitivo de dado;

- operações as quais são funções sobre os tipos primitivos de dados da linguagem;
- construtores que permitem definir novos tipos e operações derivados dos tipos e operações da linguagem.

Haskell é um exemplo de linguagem de programação funcional pura na qual todas as funções devem seguir o conceito matemático de função.

Exemplos:

- **função constante** x = 1

  Trata-se de uma função constante a qual sempre retorna o valor 1.
- **Função sucessor** : f x = x + 1

  Trata-se de uma função f que, para o parâmetro x retorna o valor x + 1.

são exemplos de funções em Haskell que satisfazem o conceito de função matemática.

- função para calcular as raízes de uma equação polinomial de segundo grau $a\,x^2 + b\,x + c = 0$, pela fórmula de Baskara,

```
baskara a b c =
let delta = b*b - 4*a*c
in      ( (-b + sqrt(delta))/(2*a),
        (-b - sqrt(delta))/(2*a) )
```

- Trata-se de uma função que, para a, b e c, retorna o par ordenado de valores correspondendo às raízes da equação.
- A palavra chave let é usada para declarar delta a qual fica acessível apenas dentro do escopo da função baskara.

Aplicando os valores: `1`, `−5` e `6`, a expressão seria reduzida pelo computador de forma similar à seguinte seqüência:

```
baskara 1 -5 6
```

```
let delta = (-5)*(-5) - 4*1*6
in      ( (5 + sqrt(delta))/(2*1),
        (5 - sqrt(delta))/(2*1) )
```

```
let delta = 1
in      ( (5 + sqrt(delta))/2,
        (5 - sqrt(delta))/2 )
```

```
( (5 + 1)/2,
(5 - 1)/2 )
```

```
( 3, 2 )
```

8.4 Conclusões

Neste capítulo foram estudados dois formalismos do tipo denotacional, baseados em funções recursivas:

- Funções Recursivas Parciais de Kleene as quais são funções parciais definidas recursivamente sobre três funções naturais básicas: zero, sucessor e projeção, juntamente com as operações substituição composicional, recursão primitiva e minimização;
- Cálculo Lambda, formalismo para definição de função, aplicação de função e recursão.
  - O cálculo Lambda é uma linguagem e um cálculo.
  - O cálculo objetiva verificar a igualdade de termos da linguagem e pode ser interpretado como um "passo computacional" na busca de um "valor" representativo para um λ-termo qualquer.
  - Portanto, a redução pode ser vista como uma operacionalização do λ-Cálculo.

A importância do Cálculo Lambda transcende o estudo da computabilidade, tendo importantes aplicações no estudo da lógica e das linguagens de programação.

No contexto das linguagens de programação funcionais, uma breve introdução da linguagem Haskell é apresentada.

Ambos formalismos são equivalentes ao formalismo Máquina de Turing bem como aos demais formalismos equivalentes estudados o que significa dizer que formalizaram o que é possível computar em um computador.

8.5 Exercícios

Exercício 8.1

Usando as funções recursivas parciais f_{zero}, sucessor e adição, verifique se existem outras formas de definir equivalentemente as funções f_{um}, fdois e ftrês apresentadas no **exemplo 8.1** Substituição de Funções: Funções Constantes

Exercício 8.2

Compare e diferencie claramente as funções f_{zero} (**exemplo 8.1** Substituição de Funções: Funções Constantes) e a função constante $const_{zero}$ (**exemplo 8.4** Minimização: Função Número Zero).

Exercício 8.3

Determine o valor de sub(2, 3) para a função recursiva do **exemplo 8.6** Minimização, Recursão Primitiva: Subtração.

Exercício 8.4

Desenvolva funções recursivas totais sobre **N** para as seguintes operações:

a) Multiplicação;
b) Quadrado (n^2);
c) Fatorial ($n!$).

Sugestão: use a função recursiva de adição definida nos exemplos.

Exercício 8.5

As funções recursivas de Kleene são definidas sobre **N**. Como poderia ser uma função recursiva para tratar palavras?

Exercício 8.6

Usando a solução do exercício acima, demonstre que as funções que seguem são recursivas de Kleene. Em cada caso, verifique se é total ou parcial (suponha $\Sigma = \{ a, b \}$):

a) esquerda: $\Sigma^* \rightarrow \Sigma^*$ tal que retorna o primeiro símbolo de x;

b) direita: $\Sigma^* \rightarrow \Sigma^*$ tal que retorna o último símbolo de x;

c) comprimento: $\Sigma^* \rightarrow \{ a \}^*$ tal que retorna o número de símbolos que compõem x, em unário;

d) ordem_lexicográfica: $\{ a \}^* \to \Sigma^*$ tal que retorna a x-ésima palavra (valor em unário) na ordem lexicográfica;

e) compara: $(\Sigma^*)^2 \to \Sigma^*$ tal que retorna:

ε, se $x = y$
a, se $x > y$
b, se $x < y$

Exercício 8.7

Função de Ackernmann. A função de Ackernmann é um importante exemplo no estudo das funções recursivas e das funções *Recursivas Primitivas* (funções recursivas, mas totais e sem minimização). Historicamente, foi considerada a possibilidade de que as funções recursivas primitivas definissem a Classe das Funções Totais Computáveis. Entretanto, a função de Ackernmann (prova-se) é um exemplo de função recursiva (total) a qual não é primitiva.

A função de Ackernmann ack: $\mathbf{N}^2 \to \mathbf{N}$ é tal que:

$ack(0, y) = y + 1$
$ack(1, 0) = 2$
$ack(x, 0) = x + 2$, para $x \geq 2$
$ack(x + 1, y + 1) = ack(ack(x, y + 1), y)$

a) A definição acima satisfaz a definição de função recursiva?

b) Calcule, passo a passo:

$ack(0, 0) = 1$

$ack(2, 1) = 5$

$ack(2, 2) = 7$

c) Defina, formalmente, a função recursiva de um argumento $ack(x, 2)$

Exercício 8.8

Relativamente ao **exemplo 8.11**, detalhe as substituições, conforme a **Definição 8.12** Substituição.

Exercício 8.9

Usando os exemplos referentes aos tipos Booleano e Natural, desenvolva λ-termo para:

a) Operação ou lógico;
b) Teste se é zero;
c) Teste se é menor ou igual;
d) Operação predecessor, na qual:
   pred(0) = 0
   pred(n) = n - 1, para n > 0
e) Teste se é número par;
f) Operação Multiplicação;
g) Operação Exponencial.

Exercício 8.10

Relativamente a composição de funções:

a) Desenvolva um λ-termo correspondente à composição de uma função f consigo mesma, ou seja, f o f
b) Aplique a solução do item acima para f(x) = x + 1

Exercício 8.11

Desenvolva um λ-termo correspondente à seguinte função:

$$f(x) = \prod_{y=0}^{x} g(y$$

Exercício 8.12

Considere os seguintes λ-termo:

   $teste_{zero}$(n) := λn.n(λx.falso) verdadeiro
   predecessor(n) := λn.λf.λx(n(λz.λw.w(z f))(λw.x) (λu.u))

a) Compare os termos $teste_{zero}$ e predecessor definidos acima com os seus termos desenvolvidos nos correspondentes exercícios anteriores;
b) Usando os exemplos referentes aos tipos Booleano, Natural e operador de ponto fixo bem como os termos definidos no enunciado deste exercício, desenvolva um λ-termo correspondente à seguinte função:

$$f(x) = \begin{cases} 0, \text{ se } x = 0 \\ 1, \text{ se } x = 1 \\ f(x-1) + f(x-2), \text{caso c} \end{cases}$$

Exercício 8.13

Considere o seguinte λ-termo:

$teste_{par}(n) := \lambda n.n(\lambda x.x \text{ falso verdadeiro}) \text{ verdadeiro}$

a) Compare o termo $teste_{par}$ definido acima com o seu termo desenvolvido no correspondente exercício anterior;

b) Usando os exemplos referentes aos tipos Booleano, Natural e operador de ponto fixo bem como os termos definidos no enunciado deste exercício e dos anteriores, desenvolva um λ-termo correspondente à seguinte função:

$$f(x)=\begin{cases}0, & \text{se } x=0\\ f(x-1)+1, & \text{se } x \text{ for í}\\ f(x-2)+2, & \text{caso co}\end{cases}$$