

Funções Recursivas: Uma Abordagem Funcional ao Conceito  
de Computabilidade<sup>a</sup>

UDESC/CCT/DCC

Versão 1.6 - 19 de junho de 2006

Claudio Cesar de Sá (claudio@joinville.udesc.br)

<sup>a</sup>A versão original deve-se aos alunos: Marcos Bernardelli, Anderson Domingues Albino, Maiquel Bandeira Fiorentin, Matheus Lacorte Pietra, e Rafael Longo da turma de TEC de 2003-2.

Sítio pertinente ao assunto e a disciplina:

<http://plato.stanford.edu/entries/computability/> Com ajuda dos estudantes, e exercícios em sala de aula, todo semestre melhoramos um pouco este texto. Logo, este texto tem um “*up-grade*” constante.

## Sumário

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Noção de Funções Recursivas</b>	<b>2</b>
<b>3</b>	<b>Conceito de Algoritmo</b>	<b>3</b>
3.1	Operacional . . . . .	4
3.2	Axiomático . . . . .	4
3.3	Denotacional ou Funcional . . . . .	4
<b>4</b>	<b>Revisitando Alguns Conceitos</b>	<b>4</b>
4.1	Função Parcial . . . . .	4
4.2	Exemplos . . . . .	5
<b>5</b>	<b>Funções Recursivas Parciais e Total</b>	<b>6</b>
5.1	Funções Recursivas Primitivas . . . . .	7
5.2	Funções Recursivas Parciais . . . . .	8
5.3	Exemplos . . . . .	9
5.4	Comentários e Reflexões . . . . .	16
5.5	Função Recursiva Total . . . . .	16
5.6	Funções Recursivas x Funções Turing-Computáveis . . . . .	17
5.7	Importância das Funções Recursivas . . . . .	17
<b>6</b>	<b>Outras Computações</b>	<b>18</b>
6.1	Mais Exemplos . . . . .	18
6.2	Exercícios Propostos . . . . .	19
<b>7</b>	<b>Conclusões</b>	<b>19</b>

## 1 Introdução

As funções recursivas foram originalmente apresentadas por Church, Kleene e outros, com o propósito da busca de um “*procedimento mecânico*” que justificasse as transformações funcionais de números. Isto é, procedimentos para se realizar cálculos sobre números, símbolos etc. As Máquinas de Turing proporcionaram uma estreita aproximação com a palavra *computação*, na busca de tal “*procedimento mecânico*”. As operações elementares da Máquina de Turing definem um procedimento efetivo para computar os valores de uma função. Assim toda função  $f : N^k \rightarrow N$  é chamada *Turing-computável* se existe uma MT que reconhece  $f$ . Visto ao longo do curso vários exemplos sobre o cálculo de funções matemáticas, tais como:

1.  $\lambda x \lambda y (x + y)$
2.  $\lambda x \lambda y (x * y)$
3.  $\lambda x \lambda y (x/y)$
4. ...

Assim, neste texto busca-se estreitar esta equivalência computacional entre Máquinas de Turing, Post e Markov, com as funções recursivas.

As funções recursivas são objetos fundamentais para estudo, pois foram criadas a partir de elementos primitivos de recursão. Estas podem ser obtidas através da minimização de uma representação funcional de uma busca seqüencial. A ênfase da computação são os argumentos de uma função que se referem aos valores de entrada de uma função computacional.

Adicionalmente, as funções recursivas mostra-se como um método efetivo para encontrar valores de uma função. A análise de seu efeito computacional é demonstrada através de uma equivalência entre as noções computacionais de Turing e suas recursividades.

Ao final, se caracteriza o conceito de *Turing-computável* as funções recursivas parciais e totais por duas justificativas:

1. MTs reconhecíveis ou computáveis, já visto acima;
2. Um programa de computador  $P$  que possua primitivas tipo:  $+$ ,  $-$ , atribuição de variáveis, uma estrutura de comparação tipo  $==$ , uma estrutura de decisão tipo como *if – then – else*, uma estrutura de desvio de execução tipo *goto*; então  $P$  também é equivalente há uma função computável.

Esta segunda abordagem também é utilizada na área de computação, com as mesmas evidências fortes e problemas das MT sobre computabilidade.

Ver referências: [Davis and Weyuker, 1983, Engeler, 1973].

## 2 Noção de Funções Recursivas

Uma função recursiva é uma função que se refere a si própria. A idéia consiste em utilizar a própria função, aplicando a sua própria definição. Em todas as funções recursivas existe componentes:

1. Um passo básico (ou mais) cujo resultado é imediatamente conhecido;
2. Um passo recursivo (ou mais) em que se tenta resolver um sub-problema do problema inicial.

Se analisarmos a função fatorial, o caso básico é o teste de igualdade a zero (*zeropn*), o resultado imediato é 1, e o passo recursivo numa notação *Lisp-like* é dada por:  $(*n(\text{fact}(-n1)))$ . Geralmente, uma função recursiva só funciona se tiver uma expressão condicional, mas não é obrigatório que assim seja. A execução de uma função recursiva consiste em ir resolvendo sub-problemas sucessivamente mais simples até se atingir o caso mais simples de todos, cujo resultado é conhecido de imediato.

Desta forma, o padrão mais comum para escrever uma função recursiva é:

1. Começar por testar os casos mais simples;
2. Fazer chamada (ou chamadas) recursiva com sub-problemas cada vez mais próximos dos casos mais simples.

Dado este padrão, os erros e problemas mais comuns associados às funções recursivas são (naturalmente para alguns):

1. Não detectar os casos simples. Aqui, a recursão não diminui a complexidade do problema. No caso de erro em funções recursivas, o mais usual é a recursão nunca parar. O número de chamadas recursivas cresce indefinidamente até esgotar a memória (*stack*), e o programa gera um erro. Em certas linguagens como *Scheme* e implementações do *Common Lisp*, isto pode nunca ser identificado como um erro. A recursão infinita é o equivalente das funções recursivas aos ciclos infinitos dos métodos iterativos do tipo *while-do* e *repeat-until*. Repare-se que uma função recursiva que funciona perfeitamente para os casos para que foi prevista pode estar completamente errada para outros casos. A função *fact* é um exemplo. Quando o argumento é negativo, o problema torna-se cada vez mais complexo, cada vez mais longe do caso simples:  $\text{fact}(-1) \Rightarrow \text{fact}(-2) \Rightarrow \text{fact}(-3) \Rightarrow \dots$
2. O segundo ponto é a sua ineficiência a nível de implementação computacional. Custo de memória, contexto do programa, etc, cresce em complexidade exponencial;
3. Fazer a abstração necessária do problema à regra recursiva inferida por indução. Ou seja, a descoberta da regra geral recursiva nem sempre é imediata, mas os princípios são os mesmos;

Talvez se tenha muitas questões filosóficas e matemáticas sobre a *recursividade*. Contudo, no momento aqui, o que interessa é que ela será um dos paradigmas ao se definir *algoritmo*. Este fato é visto a seguir.

### 3 Conceito de Algoritmo

Na busca da formalização do conceito de algoritmo, enfoca-se o assunto sobre **três visões**: *Operacional*, *Axiomático* e *Denotacional*. Um fato validado é que uma função recursiva pode

ser construída para simular numericamente as computações sob uma Máquina de Turing. A construção pode ser facilmente generalizada através de funções de mais de uma variável. A configuração de uma máquina de Turing consiste no seu estado no posicionamento do marcador da fita e do segmento da fita para esquerda ou direita, aonde estiver o símbolo em branco mais próximo. Cada um desses componentes precisam ser representados por um número natural. Indicamos uma numeração para os estados e o alfabeto, os símbolos  $B$  e  $1$  são marcados com  $0$  e  $1$ , respectivamente. O local do marcador da fita pode ser encontrado utilizando a numeração da fita. Enfim, há outras implicações entre funções computáveis com uma MT e um formalismo único e universalmente aceito. Assim a visão do termo *algoritmo* cresce em “*várias dimensões*”. Estas abordagens de formalismos para especificar algoritmos são dadas por:

### 3.1 Operacional

Define-se uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado. Exemplos: formalismos tais como Máquina Norma e Máquina de Turing. Enfocando a questão do **paradigma** das Linguagens de Programação (LP's) dos dias de hoje, esta visão de algoritmo, está contemplada pelo paradigma procedural.

### 3.2 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: aqui é a visão das gramáticas, e como esquema de procedimento tem-se as Máquinas de Markov e Post sob esta montagem de regras.

### 3.3 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. Esta é a visão aqui tratada, bem como pela abstração de  $\lambda$ -Cálculo.

## 4 Revisitando Alguns Conceitos

### 4.1 Função Parcial

Dados dois conjuntos  $A$  e  $B$ , uma relação  $f : A \rightarrow B$  recebe o nome de *função parcial* definida em  $A$  com imagens em  $B$  se, para algum  $x \in A$ , tivermos  $(x, y) \in f$  e  $(x, z) \in f$  então  $y = z$ . Em outras palavras,  $f : A \rightarrow B$  é função parcial  $\forall_{x,y,z}(x, y) \in f \text{ e } (x, z) \in f \text{ então } y = z$ .

Seja a função parcial  $f : A * B$ , onde  $A$  é o conjunto de *partida* e  $B$  é o conjunto de *chegada*. Exemplo:

Sejam os conjuntos  $A = \{a, b, c\}$ ,  $B = \{3, 5, 7\}$  e as relações  $f, g$  de  $A$  em  $B$ .

- $f = \{(a, 5), (a, 7)\}$  não é uma função;
- $g = \{(b, 3), (c, 3)\}$  é uma **função parcial**, pois nem todos pares domínio  $\times$  imagem, foram definidos. No caso, há uma indefinição para o valor  $a$ , que não possui correspondente como imagem.

Observações:

Se uma função parcial ocorrer em que o domínio coincida com o conjunto de partida ( $A = B$ ) teremos uma *função total* (mapeamento ou transformação), como tradicionalmente é definida em matemática pura.

Desta forma podemos definir; dados dois conjuntos não vazios  $A$  e  $B$ , uma relação  $f : A \rightarrow B$  recebe o nome de *função total* definida em  $A$  com imagens em  $B$  se, e somente se, para todo  $x \in A$  existe um só  $y \in B$  tal  $(x, y) \rightarrow f$ . Em outras palavras,  $f : A \rightarrow B$  é uma função definida por  $\{x \in A, y \in B \mid y = f(x)\}$ .

## 4.2 Exemplos

Determine o tipo das seguintes funções:

1. Um  $f$  leva cada número real ao seu triplo. Neste caso podemos escrever  $f = \{(x, y) \in \mathfrak{R} \mid y = 3x\}$ , ou simplesmente  $f(x) = 3x$ . Como não há nenhuma limitação sobre  $f : A \rightarrow B$ , pois o par  $(x, y)$  conduzem há um valor em  $\mathfrak{R}$ , então esta  $f$  é total;
2. Um  $g$  leva cada número real ao seu quadrado mais 1. Aqui tem-se:  $g = \{(x, y) \in \mathfrak{R} \mid y = x^2 + 1\}$ , ou ainda  $g(x) = x^2 + 1$ . Pelas mesmas razões do item anterior, esta também é total;
3. Uma função  $g(x_1, x_2) = x_1 - x_2$ , definida  $g(x_1, x_2) : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$ . Contudo, esta é indefinida para  $x_1 < x_2$  em  $\mathcal{N}$ . Caso contrário,  $x_1 \geq x_2$  estaria no domínio  $\mathcal{N}$ , o que a tornaria total. Assim, esta é uma função parcial haja visto ao aplicarg aos argumentos  $(2, 3)$ , o mesmo é dado pelo valor  $-1$ , e  $-1 \notin \mathcal{N}$ ;
4. Idem para  $g(n) = \sqrt{n}$  tal que:  $g(n) :: \mathcal{N} \rightarrow \mathcal{N}$ . Aqui apenas um subconjunto de  $\mathcal{N}$  é mapeado sobre para o conjunto dos quadrados perfeitos.  $f : \mathcal{N} \xrightarrow{g(n)} \mathcal{S}$  onde  $\mathcal{S} = \{0, 4, 9, 16, 25, \dots\}$ . Logo,  $g(n)$  também é parcial, tal que  $\mathcal{S} \in \mathcal{N}$ . Por exemplo, aplicando  $g(n)$  ao valor 7, tem-se  $\sqrt{7} = 2.645751311064591$ , e  $2.645751311064591 \notin \mathcal{N}$ , e está em  $\mathcal{R}$ .

Ao leitor: a questão da função ser parcial ou total, se relaciona diretamente a definição de *funções computáveis*. Uma definição levantada no início do curso, e comentada, com evidência, em [Divério, 1999].

## 5 Funções Recursivas Parciais e Total

As Funções Recursivas Parciais foram introduzidas por Stephen Kleene (1936), as quais foram provadas que a Classe das *Funções Turing-Computáveis* era igual a esta Classe das Funções Recursivas Parciais. Verifica-se que a composição das funções naturais simples: *constante zero*, *sucessor* e *projeção*, juntamente com as estratégias de *composição*, *recursão* e *minimização*, constitui uma forma compacta e natural para definir novas funções e suficientemente poderosa para descrever “*toda*” e qualquer função intuitivamente computável.

As funções recursivas parciais são definidas a partir das três funções primitivas, **constante zero**, **sucessor** e **projeção** mapeadas sobre o conjunto dos números naturais. A projeção não é uma função, mas uma família de funções, pois depende do número de componentes, bem como de qual a componente que se deseja projetar.

Esta definição matemática busca fundamentar o conceito de decidibilidade na Máquina de Turing, adicionalmente, demonstra-se que:

- Uma função é Turing-computável se, e somente se, a função é recursiva parcial;
- Uma função é Turing-computável por uma máquina que sempre pára se, e somente se, a função é recursiva total.

Assim, existe uma relação direta entre as seguintes Classes:

- Funções Recursivas Parciais e Funções Turing-Computáveis. As quais definem um conjunto de Linguagens Enumeráveis Recursivamente  $\Rightarrow$  Reconhecíveis;
- Funções Recursivas Totais e Funções Turing-Computáveis Totais. As quais definem um conjunto de Linguagens Recursivas  $\Rightarrow$  Decidíveis.

Estas provas de equivalências são matemáticas mesmo. Algumas delas podem ser encontradas em [Marta Sagastume, 2003]. Contudo, é fácil comprovar esta equivalência, toda vez que um modelo represente e efetue uma computação, e este estar simulando um outro modelo. Em termos práticos, podemos construir um simulador para Máquinas de Turing, Post, Markov, em uma linguagem tipo Haskell (“*100%  $\lambda$ -Cálculo*”), a qual está sob a visão denotacional/funcional.

### Definições

As funções recursivas parciais são construídas a partir funções básicas ou primárias, mais conhecidas como *primitivas*. A partir destas primitivas, aplica-se três tipos regras, ou estratégias de construções denominadas de **composição**, **recursão** e **minimização**. Esta última, empregada em funções e predicados lógicos, sem exemplos neste texto. Sobre estas aplicações é que funções mais elaboradas e/ou complexas são obtidas. Esta idéia é sumariada pela figura 1.

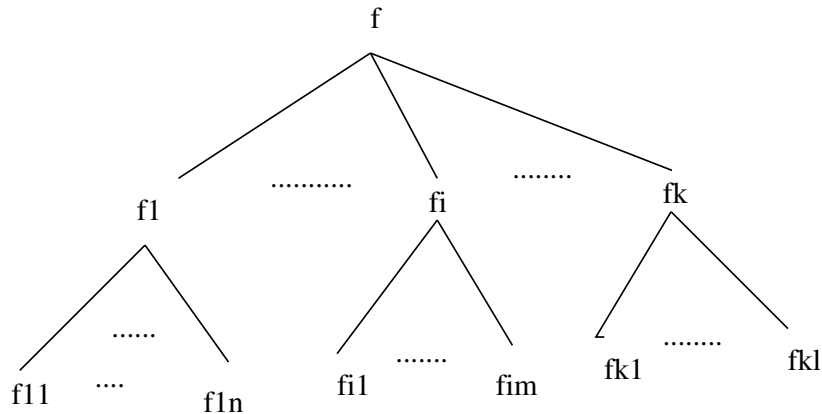


Figura 1: Funções e aplicações sobre elas e suas componentes

Assim, as recursivas parciais e totais são construídas. Salienta-se que a figura 1 não deve lembrar ou associar em nada aos conceitos de computação como: sub-rotinas, hierarquias de funções, etc. A proposta é a construção de novas funções [Marta Sagastume, 2003].

## 5.1 Funções Recursivas Primitivas

A classe das funções **recursivas primitivas** é a menor classe de funções contendo as funções iniciais e fechada pela composição e recursão.

Uma função  $f$  é recursiva primitiva se existe uma seqüência finita de funções tais que:  $f_0 \rightarrow f_1 \rightarrow f_2 \rightarrow \dots \rightarrow f_n$  onde  $f_n$  também é uma recursiva primitiva, isto é:

$$f_n = f$$

onde  $f_i$ , segue em  $i$ ,  $0 \leq i \leq n$ , e  $f_0$  é inicial ou obtida de funções anteriores na seqüência pela aplicação de composição e/ou recursão (a serem definidas na seção 5.2.).

As funções primitivas são definidas por:

1. Função zero:  $const_{zero} : n(x) = 0$  ou  $zero = \lambda x.0 :: N \Rightarrow N$   
Para qualquer  $x$  variando  $(0, 1, 2, 3, \dots, n)$ , tal que  $x$  esteja dentro do conjunto dos  $N$ , a função zero sempre dará como resposta  $0$ ;
2. Função sucessor:  $sucessor : s(x) = x + 1$  ou  $sucessor = \lambda x.x + 1 :: N \Rightarrow N$   
A função sucessor irá retornar o valor de entrada mais um, exemplo, se tiver como entrada  $1$  ela retorna o valor  $2$ , e assim sucessivamente;
3. Função projeção:  $projecao : u_i^n(x_1, \dots, x_n) = x_i$  ou  $proj_i^n = \lambda(x_1, x_2, \dots, x_n).x_i :: N^n \Rightarrow N$   
A função projeção seleciona a  $i$ -ésima componente de uma tupla, por exemplo, a  $3$  componente da tupla  $(1, 2, 7, 12, 4, 0, 99)$  é igual a  $7$ , e assim sucessivamente. No exemplo:  $proj_3^7$ .

Por considerar quase que trivial as funções acima, os exemplos encontram-se após a seção seguinte, quando esquemas mais elaborados são apresentados.

## 5.2 Funções Recursivas Parciais

Como visto nos exemplos iniciais, as funções recursivas parciais apresentam pontos os quais as  $f_{f_{rp}}$ <sup>1</sup> são não-definidas. Em outras palavras, estas  $f_{f_{rp}}$  são *computáveis parcialmente* ou apresentam pontos não-computáveis. Para isto, estas necessitam de termos aterrados que contornem as inflexões de das  $f_{f_{rp}}$ .

Adicionalmente, estas são construídas a partir de regras que possibilitam construir qualquer função matemática. Estas regras, **estratégias** ou **aplicações** sobre funções, são dadas por:

### 1. Composição:

As funções  $f$  (este representa uma *família* de funções  $f_i$  para  $\forall i \in \{1, 2, 3, \dots, k\}$ ) e  $g$  são recursivas parciais. Se  $g$  é uma função definida pela entrada  $\lambda(y_1, y_2, \dots, y_k)$ , onde há  $k$  termos ou argumentos, tal que:

$$g = \lambda(y_1, y_2, \dots, y_k).g(y_1, y_2, \dots, y_k) : N^k \Rightarrow N$$

e cada  $f_i = \lambda(x_1, x_2, \dots, x_n).f_i(x_1, x_2, \dots, x_n) : N^n \Rightarrow N$ , para todo  $i$  pertencente a  $\{1, 2, \dots, k\}$  então a seguinte função é recursiva parcial:

$$h = \lambda(x_1, x_2, \dots, x_n).h(x_1, x_2, \dots, x_n) : N^n \Rightarrow N$$

a qual é reescrita pela composição de funções a partir de  $g, f_1, f_2, \dots, f_k$  como:

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

Na qual outras composições podem ser feitas a partir destas. Os detalhes estão nos exemplos.

### 2. Recursão:

As funções  $f$  (este representa uma *família* de funções  $f_i$  para  $\forall i \in \{1, 2, 3, \dots, k\}$ ) e  $g$  são recursivas parciais. A função  $f$  é definida pela aplicação dela sobre ela mesma, ou seja, ela faz a recursão  $n$  vezes dela própria. Em detalhes:

$$f = \lambda(x_1, x_2, \dots, x_n).f(x_1, x_2, \dots, x_n) : N^n \Rightarrow N$$

Quanto a função  $g$  é definida por  $n+2$  termos, onde as entradas são  $\lambda(x_1, x_2, \dots, x_n, y, z)$ .

Assim  $g$  é definida por:

$$g = \lambda(x_1, x_2, \dots, x_n, y, z).g(x_1, x_2, \dots, x_n, y, z) : N^{n+2} \Rightarrow N$$

Então uma função  $h$  é definida como sendo uma recursiva parcial dada por:

$h = \lambda(x_1, x_2, \dots, x_n, y).h(x_1, x_2, \dots, x_n, y) : N^{n+1} \Rightarrow N$  a qual é definida por recursão (com regras recursivas) a partir de  $f$  e  $g$  como se segue:

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

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

Observe que há uma função  $g$  que tem como parâmetro um  $h$  (que se define recursivamente), e uma instância anterior a  $y + 1$ , que é  $y$ .

### 3. Minimização:

O conceito de minimização que segue não é tão intuitivo como os demais acima. Entretanto, é fundamental para garantir que a Classe de Funções Recursivas Parciais

<sup>1</sup>Uma notação própria a este texto,  $f_{f_{rp}}$  onde leia-se que  $f$  é uma *função recursiva parcial*.

definida adiante possa conter qualquer função intuitivamente computável, seja ela total ou parcial.

Se a função  $f$  é recursiva parcial dada por:

$$f = \lambda(x_1, x_2, \dots, x_n, y).f(x_1, x_2, \dots, x_n, y) : N^{n+1} \Rightarrow N$$

então uma função  $h$  é recursiva parcial, e é dada por:

$$h = \lambda(x_1, x_2, \dots, x_n).h(x_1, x_2, \dots, x_n) : N^n \Rightarrow N$$

a qual é definida pela regra de minimização sobre  $f$ , como segue:

$$h = \lambda(x_1, x_2, \dots, x_n).min_y \{y | f(x_1, x_2, \dots, x_n, y) = 0\} : N^n \Rightarrow N$$

A mesma definição acima, de acordo com D.A. Tiarajú [Divério, 1999]:

A função parcial  $h$ , dada por:

$h = \lambda(x_1, x_2, \dots, x_n).h(x_1, x_2, \dots, x_n) : N^n \rightarrow N$  é definida por minimização de  $f$ , tal que:

$h = \lambda(x_1, x_2, \dots, x_n).min\{y | f(x_1, x_2, \dots, x_n, y) = 0 \text{ e } \forall z \text{ tal que } z < y, f(x_1, x_2, \dots, x_n, z) \text{ é definida}\}$

Portanto, a função  $h$ , para o valor  $(x_1, \dots, x_n)$ , é definida como o menor natural  $y$  tal que  $f(x_1, \dots, x_n, y) = 0$ . Adicionalmente, a condição  $\forall z$  tal que  $z < y, f(x_1, x_2, \dots, x_n, z)$  definida garante que é possível determinar, em um tempo finito, se, para qualquer valor  $z$  menor do que  $y$ ,  $f(x_1, \dots, x_n, z)$  é diferente de zero. Note que a função  $h$  é parcial. Por simplicidade, a seguinte notação é adotada:

$$h = \lambda(x_1, x_2, \dots, x_n).min_y \{y | f(x_1, \dots, x_n, y) = 0\}$$

### 5.3 Exemplos

Esta seção reforça o aprendizado dos conceitos acima. Procure identificar (assinalando cada item) esses conceitos de funções básicas ou primitivas e recursivas parciais. Os exemplos pode estar duplicados, quando ocorrer, desconsidere.

1.  $zero(x) = \lambda x.0 : N \rightarrow N$  função constante zero. Esta também é uma *função primitiva recursiva*, pois pode ser obtida da função zero, visto anteriormente. Sem usar a notação  $\lambda$ , esta também pode ser definida como:  $zero(x) = const_{zero}$  ;
2.  $sucessor(x) = \lambda x.x + 1 : N \rightarrow N$  função sucessor. Idem, o sucessor é “*seguinte*” a  $x$  em  $N$ ;
3.  $um(x) = \lambda x.sucessor(zero(x)) : N \rightarrow N$  função constante um 1, foi obtido pela aplicação da composição, entre as funções  $sucessor(x)$  e  $zero(x)$ ;
4.  $dois(x) = \lambda x.sucessor(um(x)) : N \rightarrow N$ . Caso análogo ao anterior;
5.  $tres(x) = \lambda x.sucessor(dois(x)) : N \rightarrow N$ . Caso análogo ao anterior;
6.  $tres(x) = \lambda x.adicao(um(x), dois(x)) : N \rightarrow N$ . Interessante, mas, a função *adicao* ainda não foi provada. Até chegar lá, algumas outras ainda precisam ser provadas que são primitivas recursivas;

7. Função Identidade: A função identidade, definida como segue:

$$id = \lambda x.x : N \Rightarrow N$$

é recursiva primitiva, pois é se usou a estratégia da projeção, dado por:

$id(x) = proj_1^1(x)$ , ou ainda  $proj_1^1 = \lambda x.x = x$ . A notação aqui usada, foi abreviada, pois o correto é  $id(x)$ . Exemplo:  $id(77) = 77$ ;

8. Outra definição para identidade:

$id(x + 1) = suc(x)$ . Exemplo:  $id(7) = suc(6)$ . Logo,  $id(x)$  também é primitiva recursiva;

9. Agora sim, temos os elementos da prova de que a função da adição é uma função primitiva recursiva, observe porquê:

- (a)  $id(x) = \lambda x.x : N \rightarrow N$  função identidade;
- (b)  $suc(x) = \lambda x.x + 1 : N \rightarrow N$  função sucessor;
- (c)  $proj_3^3(x, y, z) = \lambda(x, y, z).z : N^3 \rightarrow N$  função projeção da 3<sup>a</sup> componente da tripla.

A função adição nos naturais na abstração  $\lambda$  é dada por:  $adicao(x, y) = \lambda x \lambda y.x + y : N^2 \rightarrow N$

Essa é reescrita, usando a definição da *recursão*, como se segue:

- (a)  $adicao(x, 0) = id(x)$
- (b)  $adicao(x, y + 1) = proj_3^3(x, y, suc(adicao(x, y)))$

Certifique-se que tudo está claro até aqui. Por exemplo,  $adicao(3, 3)$  é como abaixo. Note-se que, nas instâncias de

$proj_3^3(x, y, suc(adicao(x, y)))$ , somente a componente  $suc(adicao(x, y))$  é importante na lógica apresentada. A função  $proj_3^3$ , bem como as componentes  $x$  e  $y$ , estão presentes somente para satisfazer a definição de recursão. De fato, a função projeção é fundamental na regra da recursão.

$$\begin{aligned}
 & adicao(3, 3) \\
 &= proj_3^3(3, 2, suc(adicao(3, 2))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(adicao(3, 1)))))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(proj_3^3(3, 0, suc(adicao(3, 0))))))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(proj_3^3(3, 0, suc(id(3))))))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(proj_3^3(3, 0, suc(3)))))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(suc(3)))))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, suc(4)))) \\
 &= proj_3^3(3, 2, suc(proj_3^3(3, 1, 5))) \\
 &= proj_3^3(3, 2, suc(5)) \\
 &= proj_3^3(3, 2, 6) \\
 &= 6
 \end{aligned}$$

Ao estudante, verifique e avalie a necessidade deste formalismo.

10. Simplificando<sup>2</sup> a notação da prova acima de **adição** = **adição**( $x, y$ ).

<sup>2</sup>Esta notação é encontrada em muitos livros da área, é mais legível, mas de igual valor semântico.

- (a)  $adição(x, 0) = x$   
 (b)  $adição(x, y + 1) = adição(x, y) + 1$

Ex:  $2 + 2$

$$\begin{aligned}
 adicao(2, 2) &= adicao(2, 1 + 1) = adicao(2, 1) + 1 \\
 &= (adicao(2, 0) + 1) + 1 \\
 &= (2 + 1) + 1 \\
 &= 3 + 1 \\
 &= 4
 \end{aligned}$$

11. Suponha a constante zero  $const_{zero}$ , bem como a seguinte função de projeção:  
 $proj_1^2 = \lambda(x, y).x : N^2 \rightarrow N$  função projeção da 1<sup>a</sup> componente do par (tupla-2). A  
 função antecessor(a) nos naturais é definida por:  
 $antecessor = \lambda x.antecessor(x) : N \rightarrow N$   
 pode ser definida, usando recursão (supondo que antecessor de 0 é 0), como segue:

- (a)  $antecessor(0) = const_{zero}$   
 (b)  $antecessor(y + 1) = proj_1^2(y, antecessor(y))$

Por exemplo, calculando o  $antecessor(2)$ , este é dado por:

$$\begin{aligned}
 antecessor(2) &= proj_1^2(1, antecessor(1)) \\
 &= proj_1^2(1, proj_1^2(0, antecessor(0))) \\
 &= proj_1^2(1, proj_1^2(0, const_{zero})) \\
 &= proj_1^2(1, proj_1^2(0, 0)) \\
 &= proj_1^2(1, 0) \\
 &= 1
 \end{aligned}$$

Novamente, avalie bem este exemplo, pois a projeção só pode ser aplicada mediante a  
 redução aos números naturais!

12. Suponha a constante zero ( $const_{zero}$ ), bem como as seguintes funções:  $id = \lambda x.x : N \rightarrow N$  função identidade,  $proj_3^3 = \lambda(x, y, z).z : N^3 \rightarrow N$  função projeção da 3<sup>a</sup> componente da tripla. Assim, uma **função subtração** nos naturais é definida como:  
 $sub = \lambda(x, y).sub(x, y) : N^2 \rightarrow N$

pode ser redefinida usando recursão dado por:

- (a)  $sub(x, 0) = id(x)$   
 (b)  $sub(x, y + 1) = proj_3^3(x, y, antecessor(sub(x, y)))$

Por exemplo, a avaliação  $sub(3, 2)$  é dada por:

$$\begin{aligned}
 & sub(3, 2) \\
 &= proj_3^3(3, 1, antecessor(sub(3, 1))) \\
 &= proj_3^3(3, 1, antecessor(proj_3^3(3, 0, antecessor(sub(3, 0)))) \\
 &= proj_3^3(3, 1, antecessor(proj_3^3(3, 0, antecessor(id(3)))) \\
 &= proj_3^3(3, 1, antecessor(proj_3^3(3, 0, antecessor(3)))) \\
 &= proj_3^3(3, 1, antecessor(proj_3^3(3, 0, 2))) \\
 &= proj_3^3(3, 1, antecessor(2)) \\
 &= proj_3^3(3, 1, 1) \\
 &= 1
 \end{aligned}$$

13. Ampliando estas idéias, podemos simplificar a multiplicação em **multiplicação** =  $mult(x, y)$ :

1.  $mult(x, 0) = 0$
2.  $mult(x, y + 1) = mult(x, y) + x$

Ex:  $2 * 2$

$$\begin{aligned}
 mult(2, 2) &\Rightarrow mult(2, 1 + 1) = mult(2, 1) + 2 \\
 &= (mult(2, 0) + 2) + 2 \\
 &= (0 + 2) + 2 \\
 &= 4
 \end{aligned}$$

14. Uma versão mais rigorosa para a multiplicação é dada por (onde a multiplicação é reescrita por  $mult(x, y)$ ):

1.  $mult(x, 0) = const_{zero}$
2.  $mult(x, y + 1) = proj_3^3(x, y, adicao(x, mult(x, y)))$

Ao estudante: “ $mult(7, 1000)$  é igual a  $mult(1000, 7)$ ”? Isto é:  $7 \times 1000 = 1000 \times 7$ . A resposta é igual sim, mas . . . , avalie a ordem de complexidade em ambos os cálculos, dada as duas definições acima.

15. Versões simplificadas, também válidas, para função **antecessor(a)** =  $antecessor(y)$
1.  $antecessor(0) = 0$
  2.  $antecessor(y + 1) = id(y)$

Esta definição está em conformidade com a definição de recursão, logo, ela é uma primitiva recursiva. Ex:

$$antecessor(5) = antecessor(4 + 1) = id(4) = 4$$

16. A exemplo da adição, uma versão simplificada para **sub**( $x, y$ ) = **subtração**( $x, y$ ):

1.  $subtracao(x, 0) = id(x)$
2.  $subtracao(x, y + 1) = antecessor(subtracao(x, y))$

Ex:  $2 - 4$

$$\begin{aligned}
 &= subtracao(2, 4) = subtracao(2, 3 + 1) = antecessor(subtracao(2, 3)) \\
 &= antecessor(antecessor(subtracao(2, id(2)))) \dots \dots \\
 &= antecessor(antecessor \dots antecessor(subtracao(2, 0))) \\
 &= \dots \dots \\
 &= antecessor(antecessor(2))
 \end{aligned}$$

$$= \text{antecessor}(1)$$

$$= 0$$

Este exercício tem uma notação mais flexível que o apresentado em 12, mas, a sua operacionalidade é a mesma. Contudo, estes dois exemplos evidenciam/ilustram o conceito de que a subtração é uma *função parcial*. Isto é, pois  $2 - 4 = 0$ , pois o domínio é  $\mathcal{N}$ . Assim, essa subtração restrita é uma função conhecida como função *monus*( $x, y$ ), ou subtração positiva a qual tem uma definição computacional equivalente há:

$$\text{monus}(x, y) = \text{if } x \geq y \text{ then } x - y \text{ else } 0$$

Confirme se voce entendeu essa definição informal acima. Sim? Assim, a função *monus*( $x, y$ ) é dada por:

$$\text{monus}(x, y) = \begin{cases} x - y & \text{se } x \geq y \\ 0 & \text{caso contrário ou } x < y \end{cases}$$

Agora ficou claro? Em caso de negativo, tenha um pouco de paciência, chegaremos lá!

17. Funções mais complexas: **exponenciação** = *expoente*( $x, y$ )

1. *expoente*( $x, 0$ ) = 1
2. *expoente*( $x, y + 1$ ) = *expoente*( $x, y$ ) \*  $x$

Ex:  $2^2$

$$\begin{aligned} \text{expoente}(2, 2) &= \text{expoente}(2, 1 + 1) = \text{expoente}(2, 1) * 2 \\ &= (\text{expoente}(2, 0) * 2) * 2 \\ &= (1 * 2) * 2 \\ &= 4 \end{aligned}$$

Ao estudante reescreva esta definição com o formalismo exigido e necessário.

18. Idem: **fatorial** = *fatorial*( $x$ )

1. *fatorial*(0) = 1
2. *fatorial*( $x + 1$ ) = *mult*(*fatorial*( $x$ ), ( $x + 1$ ))

Ex: ao estudante, resolva agora o *fatorial*(4)

Ao estudante reescreva esta definição com o formalismo exigido e necessário.

19. Suponha a função constante *zero* =  $\lambda x. 0 : N \rightarrow N$ . Considere a seguinte função que identifica o número zero nos naturais como:

$$\text{const}_{\text{zero}} : \rightarrow N$$

Note que é uma função sem variáveis, ou seja, é uma constante. A constante *const<sub>zero</sub>* é definida usando minimização como segue:

$$\text{const}_{\text{zero}} = \min_y \{y | \text{zero}(y) = 0\}$$

resultando em: *const<sub>zero</sub>* = 0

20. Seja  $f(x) = \min_y \{y + x = 4\}$ , quanto vale  $f(0)$ ,  $f(2)$  e  $f(4)$ ?

Respostas: 4, 2 e 0 respectivamente.

21. Seja  $h(x, y) = \min_z \{x = y + z\}$ , quanto vale  $h(5, 2)$  e  $h(2, 5)$ ?

Respostas:

- (a)  $h(5, 2) = 3$  dado que  $5 = 2 + z$  então tem-se que  $z = 3$ ;
- (b)  $h(2, 5)$  é sem solução mesmo, pois  $2 = 5 + z$ , e tal  $z$  não existe, haja visto que a imagem é em  $\mathcal{N} = \{0, 1, 2, 3, \dots\}$ .
22. Definir<sup>3</sup> função  $maior(x, y)$  que tem a mesma resposta que a relação  $x > y$  (“ $x$  maior que  $y$ ”). Mas,  $>(x, y)$  é uma *relação binária*, e  $maior(x, y)$  é uma *função*. Onde sem muito rigor esta é dada por:

$$maior(x, y) = \text{if } x > y \text{ then } 1 \text{ else } 0$$

Na definição acima, necessita-se da relação “ $>$ ” para que mesma seja definida. Então contornar desviar desta “*necessidade recursiva*”<sup>4</sup>, o passo seguinte é reescrever  $maior(x, y)$  em termos de primitivas recursivas. Uma solução proposta é dada por:

- (a)  $maior(0, 0) = 0$  ou ainda: *const\_zero*
- (b)  $maior(0, y) = 0$  ou ainda: *const\_zero*
- (c)  $maior(x, 0) = 1$  ou ainda: *id(1)*
- (d)  $maior(x + 1, y + 1) = proj_3^3(x, y, maior(x, y))$

Obviamente que os valores de 0 e 1 poderiam ser substituídos por,  $zero(x) = conts_{zero}$  e  $um(x) = suc(zero(x))$ , respectivamente. Uma outra definição alternativa, é dada pelo uso da função recursiva “*monus(x, y)*”, que é subtração positiva. Esta é deixada como exercício.

23. Definir<sup>5</sup> função  $menor\_igual(x, y)$ , cuja funcionalidade é a mesma da relação  $x \leq y$  (“ $x$  menor ou igual a  $y$ ”). Todas as considerações anteriores são válidas aqui, logo:
- (a)  $menor\_igual(x, 0) = 0$
- (b)  $menor\_igual(0, 0) = 1$
- (c)  $menor\_igual(0, y) = 1$
- (d)  $menor\_igual(x + 1, y + 1) = proj_3^3(x, y, menor\_igual(x, y))$

24. Uma função condicional do tipo:

$$cond(x_1, x_2, x_3) = \begin{cases} x_2 & \text{se } x_1 = 0 \\ x_3 & \text{se } x_1 \geq 1 \end{cases}$$

Cuja interpretação para uma máquina sequencial, ou paradigma procedural, teria uma equivalência semântica dada por:

$$cond(x_1, x_2, x_3) = \text{if } x_1 = 0 \text{ then } x_2 \text{ else } x_3$$

Essa é reescrita em termos de primitiva recursivas, dado por:

---

<sup>3</sup>Quem tiver uma solução melhor, me avise. Esta é uma solução; algo parecido se encontra em [Hein, 1999].

<sup>4</sup>O contexto é pejorativo mesmo, pois se precisa de algo para definir ele próprio!

<sup>5</sup>Idem a nota anterior.

- (a)  $cond(0, x_2, x_3) = id(x_2)$
- (b)  $cond(1, x_2, x_3) = id(x_3)$
- (c)  $cond((x_1 + 1, x_2, x_3) = proj_3^4(x_1, x_2, x_3, cond((x_1, x_2, x_3)))$

Ainda falta definir a função *igual* (=) para que a demonstração fique completa.

Com este exemplo, percebe-se a analogia direta com linguagens de programação, haja visto que esta **frp** foi escrita por uma linha de código equivalente a: *if*  $x_1 == 0$  *then*  $x_2$  *else*  $x_3$ . Para este caso, basta definir a relação “ $x == y$ ” ou “ $x = y$ ”, tarefa esta deixada a cargo do estudante.

25. Finalmente, reusando todo esse conhecimento, mostre que a função<sup>6</sup> definida por:

$$f(x, y) = \begin{cases} 0 & : x = 7 \\ 7 * x + y & : x > 7 \end{cases}$$

é também uma função primitiva recursiva. A função *maior*( $x, y$ ) já foi definida, de modo análogo a função *igual*( $x, y$ ) é construída. Falta construir  $f(x, y)$ . Como essa função tem condições sobre o valor de  $x$  (mas, que poderia ser sobre  $y$  também), essa é dividida em duas partes:

- (a)  $f(7, y) = const\_zero$  ora, quando  $x$  casar com 7, esta é uma função. No caso a função constante zero;
- (b)  $f(x, y) = id(g(maior(x, 7), x, y))$  onde uma função auxiliar  $g$  é definida por:
- (c)  $g(1, x, y) = id(adicao(mult(7, x), y))$

Convém observar que  $f(x, y)$  não é uma função definida recursivamente, mas a sua demonstração é uma primitiva recursiva, pois necessitou de outras que são **fpr**, no caso: *adicao*, *mult*, *maior*, *igual* (que para esta solução não foi necessária), etc.

Reescrevendo esta expressão em termos de uma notação clássica de linguagem sequencial, tem-se:

$$f(x, y) = \text{if } igual(7, y) \text{ then } const\_zero \text{ else } id(g(maior(x, 7), x, y))$$

Na condição *igual*(7,  $y$ ), é a idéia de casamento de  $x$  com 7, cuja resposta sendo verdade *const\\_zero* é a resposta. Assim sendo, a expressão acima pode ser reescrita como:

$$f(x, y) = \text{if } x == 7 \text{ then } const\_zero \text{ else } id(g(maior(x, 7), x, y))$$

Neste caso, “==” é um operador relacional muito simples, e que tem uma equivalência semântica com a função *igual*(7,  $y$ ). Cabe observar que este exemplo é uma função primitiva recursiva parcial, pois há uma indefinição de  $x$  em  $\mathcal{N}$ , quando  $0 \leq x < 7$ . O conceito de parcial é enfatizado no sentido de *computabilidade*, aqui é o caso em que para esse intervalo ( $0 \leq x < 7$ ), a função é *não-computável*. Para torná-la computável,  $x$  e  $y$  devem estar definidos em  $\mathcal{N}$ , e o mapeamento de  $f(x, y)$  igualmente levar a  $\mathcal{N}$ .

<sup>6</sup>Exercício resolvido por André Körbes, turma de TEC 2005-2. O exemplo poderia merecer uma outra abordagem de solução, mas esta, está muito boa.

A definição dessas últimas relações em funções primitivas recursivas, vai auxiliar a resolução de várias outras funções mais complexas tais como: a divisão, o resto, operadores lógicos, etc. Todas pertinentes a área de computação.

## 5.4 Comentários e Reflexões

**Sugere-se como exercício pensar sobre a avaliação de  $sub(2, 3)$  ? O que esperar dessa avaliação? Quais as limitações? Em seguida pense sobre o uso do domínio dos reais  $\mathbb{R}$  como alternativa a enumeração infinita (ou infinito contável) ao domínio dos naturais  $\mathcal{N}$ . Estes pontos questionados em sala de aula, mas é pré-requisito de todo este texto. Porquê o domínio dos naturais  $\mathcal{N}$  é suficientemente poderoso (forte) para usar nas definições das funções recursivas? Porquê estas conseguem responder ao conceito de Turing-computável ao domínio dos reais  $\mathbb{R}$ ? E conseqüentemente, quais os limites da teoria das funções recursivas?**

Algumas das respostas acima existem provas, usando principalmente, a Diagonalização de Cantor. Como o domínio dos reais  $\mathbb{R}$  é infinito, as limitações das MTs, são aqui também evidenciadas para as funções recursivas.

## 5.5 Função Recursiva Total

Uma Função Recursiva Total é uma Função Recursiva Parcial definida para todos os elementos do domínio. Se esta foi obtida a partir de estratégias e regras sobre funções primitivas, então toda função recursiva total é também uma primitiva recursiva. Assim, um gráfico que resume esta “*hierarquia*”<sup>7</sup> é dada pela figura 2.

---

<sup>7</sup>Não é bem uma hierarquia, mas sim um conceito de classes, e redução entre as mesmas.

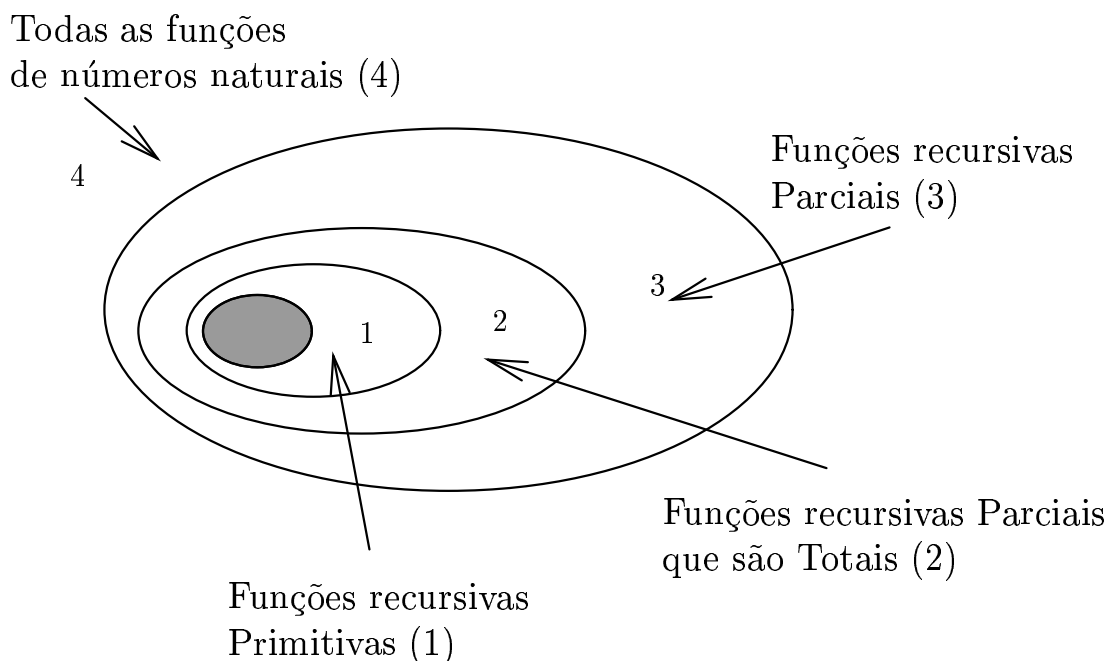


Figura 2: Hierarquia das Funções Recursivas

Verifique bem se entendeu esta figura, caso contrário vá a referência [Hein, 1999]. Adicionalmente pesquise e pense sobre a questão de que a função de Ackermann, de que esta função é recursiva parcial e total, mas não é primitiva!

## 5.6 Funções Recursivas x Funções Turing-Computáveis

As seguintes classes de funções são equivalentes:

1. Funções Recursivas Parciais e Funções Turing-Computáveis (Reconhecível);
2. Funções Recursivas Totais e Funções Turing-Computáveis Totais (Decidível).

A nível de funções uma relação entre classes também pode ser estabelecida (em outras palavras, idem a classificação anterior):

1. Funções Recursivas Parciais  $\Leftrightarrow$  Linguagens Enumeráveis Recursivamente;
2. Funções Recursivas Totais  $\Leftrightarrow$  Funções Turing-Computáveis Totais  $\Leftrightarrow$  Linguagens Recursivas.

Anote os comentários do professor em sala de aula sobre as conclusões acima.

## 5.7 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, definições em hardware.

## 6 Outras Computações

### 6.1 Mais Exemplos

Desenvolva funções recursivas totais sobre  $N$  para as seguintes funções (desconsidere as duplicações anteriores, e valem as definições mais formais/rigorosas):

- Definindo a função recursiva da **adição** em termos da função **sucessor**:

$$\begin{aligned} \text{(a)} \quad & a + 0 = a \\ \text{(b)} \quad & a + s(b) = s(a + b) \end{aligned}$$

Essa definição tem uma base e uma parte recursiva. A base define a adição com 0, e a parte recursiva define a adição do sucessor de  $b$  em termos da adição de  $b$ .

Calculando um exemplo numérico:

$$4 + 3 = s(4 + 2) = s(s(4 + 1)) = s(s(s(4 + 0))) = s(s(s(4))) = s(s(5)) = s(6) = 7$$

- Multiplicação:**  $3 * 4$

Como já foi definido a multiplicação anteriormente, logo:

$$mult(3, 4) = mult(3, 3 + 1) = mult(3, 3) + 3$$

$$(mult(3, 2) + 3) + 3$$

$$((mult(3, 1) + 3) + 3) + 3$$

$$(((mult(3, 0) + 3) + 3) + 3) + 3$$

$$(((0 + 3) + 3) + 3) + 3$$

$$12$$

- Quadrado ( $n^2$ ):** Como definido anteriormente o exponencial, podemos então computar o quadrado:

$$4^2$$

$$expoente(4, 2) = expoente(4, 1 + 1) = expoente(4, 1) * 4$$

$$(expoente(4, 0) * 4) * 4$$

$$(1 * 4) * 4$$

$$16$$

- Fatorial ( $n!$ ):** Sabendo-se que  $fatorial(x) = 0$ , logo:

$$4!$$

$$fatorial(4) = fatorial(3 + 1) = fatorial(3) * (3 + 1)$$

$$(fatorial(2) * (2 + 1)) * (3 + 1)$$

$$((fatorial(1) * (1 + 1)) * 2 + 1) * (3 + 1)$$

$$(((1 * 2) * 3) * 4)$$

$$24$$

## 6.2 Exercícios Propostos

1. Usando as funções zero, sucessor e adição, verifique se existem outras formas de definir equivalentemente as funções um, dois e três apresentadas anteriormente;
2. Compare e diferencie a constante  $const_{zero}$  e a função constante zero;
3. Determine o valor de  $subtração(2, 3)$ ;
4. Construa a definição das seguintes funções primitivas recursivas:
  - (a) Da divisão inteira:  $x \text{ div } y$ ;
  - (b) Da função resto da divisão inteira:  $x \text{ mod } y$ ;
  - (c) Do máximo divisor comum entre dois números;
  - (d) Reescreva, com outras definições de funções primitivas, para:  $menor(x, y)$ ,  $maior(x, y)$ ,  $menor\_igual(x, y)$ ,  $maior\_igual(x, y)$ , etc.
  - (e) Reescreva as definições em **frp** para as funções lógicas:  $ou(x, y)$ ,  $e(x, y)$ , e  $ou - exclusivo(x, y)$ ;
  - (f)

## 7 Conclusões

Nesse texto ilustra-se que qualquer função recursiva pode ser computada em uma máquina de Turing. Utiliza-se várias funções simples para se construir funções mais complexas, e com isto também computáveis por uma máquina de Turing. A construção dessas funções requer uma mudança no **domínio da máquina** para o **domínio dos números naturais**. Provamos assim a importância do estudo da teoria das funções recursivas aplicadas a computação, e com isso reescreve-se expressões fortes do cotidiano, tais como: computabilidade, computável, algoritmo, e computação!

## Referências

- [Davis and Weyuker, 1983] Davis, M. D. and Weyuker, E. J. (1983). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, San Diego, CA.
- [Divério, 1999] Divério, T. A.; Menezes, P. F. B. (1999). *Teoria da Computação: Máquinas Universais e Computabilidade*. Sagra-Luzzatto, . Porto Alegre - RS. Série Livros Didáticos do Instituto de Informática da UFRGS, n.5.
- [Engeler, 1973] Engeler, E. (1973). *Introduction to the Theory of Computation*. Academic Press, New York.
- [Hein, 1999] Hein, J. L. (1999). *Theory of Computation: An Introduction*. Jones and Barlett Publishers, United States – USA.

Atualizado em: 19 de junho de 2006

---

[Marta Sagastume, 2003] Marta Sagastume, Gabriel Baum, G. M. (2003). *Problemas, Lenguages y Algoritmos*, volume 37. Publicações CLE - UNICAMP - BR. Uma 1<sup>a</sup> versão da I EBAI - 1988.