Revista Eletrônica Paulista de Matemática

ISSN 2316-9664
Volume 14, fev. 2019
Edição Ermac
Iniciação Científica

Pseudolinguagem para caracterizar funções recursivas

Pseudo-language to characterize recursive functions

Pedro Henrique Paiola
UNESP - Universidade Estadual Paulista "Júlio de Mesquita Filho"
paiola@fc.unesp.br

Hércules de Araujo Feitosa
UNESP - Universidade Estadual Paulista "Júlio de Mesquita Filho"
haf@fc.unesp.br

Resumo

Iniciamos esse artigo com uma visão geral sobre a Teoria da Computabilidade e o conceito de algoritmo. Em seguida, definimos a classe de funções recursivas, uma tentativa de formalização da classe de funções algorítmicas, e mostramos como muitas funções usualmente disponíveis nas linguagens de programação são casos de funções recursivas. A partir daí, caminhamos para a construção de uma pseudolinguagem de programação que dê conta exatamente das funções recursivas, buscando criar uma ponte entre os conceitos da Teoria da Computabilidade e a programação prática. Desenvolvemos duas versões para essa pseudolinguagem, a *REC*, com um escopo menor de operações, e a *REC*++, que oferece alguns recursos que facilitam a codificação. Durante a apresentação dessas linguagens, procuramos demonstrar a equivalência entre a classe das funções *REC* e *REC*++ calculáveis com a classe das funções recursivas.

Palavras-chave: Matemática Discreta. Computabilidade. Funções recursivas. Pseudolinguagem.

Abstract

We begin this article with an overview on Computability Theory and the concept of algorithm. Then, we define the class of recursive functions, an attempt to formalize the class of algorithmic functions, and show how many functions usually available in programming languages are cases of recursive functions. From there, we move towards the construction of a programming pseudolanguage that accurately accounts for the recursive functions, seeking to create a bridge between the concepts of Computability Theory and the practical programming. We have developed two versions for this pseudo-language, the *REC*, with a minor scope of operations, and *REC*++, which offers some features that facilitate the coding. During the presentation of these languages, we demonstrate the equivalence between the class of functions *REC* and *REC*++ calculable with the class of recursive functions.

Keywords: Discrete mathematics. Computability. Recursive functions. Pseudo-language.

Artigo recebido em set. 2018 e aceito em jan. 2019.

1 Introdução

De acordo com Dias e Weber (2010), o ponto de partida da Computabilidade, ou ainda da Teoria da Recursão, é a análise conceitual, em termos matematicamente precisos, das noções intuitivas de algoritmo e função algorítmica. Essas são duas noções fundamentais, mas que não admitem uma definição cabal.

Ainda segundo Dias e Weber (2010), um algoritmo é, de forma breve, um conjunto finito de regras ou instruções, concebidas para serem aplicadas mecanicamente, ou seja, sem uso da criatividade. Assim, dada uma partida ou uma instrução inicial, após a execução de um número finito de operações elementares, programadas sobre os dados iniciais, o procedimento deve gerar uma saída ou resultado.

Devemos conceber o procedimento algorítmico como uma função algorítmica, em que o argumento da função são os dados de entrada, as instruções permitem efetivamente a determinação ou computação do valor da função, que é o valor de saída.

Segundo Cutland (1980), podemos dizer que a Teoria da Computabilidade, do ponto de vista da Ciência da Computação, parte da indagação sobre o que os computadores (sem considerar restrições de espaço, tempo ou dinheiro) podem fazer e, como consequência, quais as limitações inerentes.

Assim, fica claro que a Computabilidade se preocupa não apenas com o que um computador pode fazer, ou seja, o que pode ser computável, mas também sobre os limites da computação, com aquilo que não pode ser computável.

As noções apresentadas de algoritmos e funções algorítmicas são puramente intuitivas. Há uma série de versões formais para essas duas ideias. Nestas notas, teremos como base as funções recursivas, sistematizadas por Kleene a partir das ideias de Herbrand e Gödel (MENDELSON, 1964).

Inseridos nesse contexto, nos atuais cursos de Bacharelado em Ciência da Computação, temos disciplinas como Algoritmos e Modelos de Computação ou Teoria da Computação, que permitem não apenas o desenvolvimento lógico do aluno na resolução de problemas, através da construção de programas de computador, como também a abordagem sobre os limites teóricos da computação, anteriormente citados. Porém, observa-se um alto índice de reprovação nas duas disciplinas (FORTE; GUZDIAL, 2005; GUZDIAL, 2003; LIMA JUNIOR; VIEIRA, C.; VIEIRA, P., 2015).

Dessa forma, nosso objetivo com esse trabalho é a construção de uma pseudolinguagem de programação a partir da definição de funções recursivas, que possibilite estabelecer uma ponte entre os algoritmos e alguns conceitos matemáticos relativamente simples, com vistas a incentivar o aluno a criar a capacidade de traduzir processos matemáticos em procedimentos computacionais, ao mesmo tempo em que isso se associa ao estudo da Computabilidade.

Neste trabalho, iniciamos com uma visão bastante geral do conceito de algoritmo, com ênfase nos pseudocódigos. A seguir, apresentamos a definição e desenvolvimento teórico inicial sobre as funções recursivas. Existem definições distintas para este conceito, embora a classe de funções recursivas e recursivas parciais sejam unas.

Por fim, introduzimos o desenvolvimento da pseudolinguagem proposta, caminhando para demonstrar as equivalências entre o sistema formal por nós introduzido e as funções recursivas.

Este artigo é derivado do trabalho (PAIOLA; FEITOSA, 2018).

2 Sobre algoritmos

De um modo muito intuitivo, podemos entender o conceito de algoritmo como uma receita que indica passo a passo os procedimentos essenciais para a solução de um problema ou execução de uma tarefa. Este procedimento deve seguir uma ordenação que indica exatamente como proceder e se bem constituído deve ter pleno êxito na tarefa delineada.

Usamos algoritmos com enorme frequência em nossas vidas, como na receita para o preparo de um prato, ou o caminho para o envio de uma foto no WhatsApp. Algumas vezes ficam tão repetidos que acabam internalizados pelos humanos, como na ação de escovar os dentes ou trocar a marcha do carro.

Mas usamos algoritmos frequentes na Matemática. Para fazermos uma divisão de inteiros, seguimos uma sequência de ações muito bem treinadas nas aulas de Matemática, que quando bem executadas nos levam à resposta correta. Este é o conhecido Algoritmo da Divisão de Euclides. Temos inúmeros outros algoritmos no ambiente matemático.

Vemos, assim, que algoritmo não é instrumento só do contexto computacional, embora eles estejam na essência da computação.

2.1 O conceito de algoritmos

Como estamos iniciando um tópico, buscaremos o significado de 'algoritmo' num dicionário. Tomamos o verbete do (MICHAELIS, 2007):

algoritmo: al·go·rit·mo sm
1 MAT, OBSOL Sistema de notação aritmética com algarismos arábicos.
2 MAT Processo de cálculo que, por meio de uma sequência finita de regras, raciocínios e operações, aplicada a um número finito de dados, leva à resolução de grupos análogos de problemas.
3 MAT Operação ou processo de cálculo; sequência de etapas articuladas que produz a solução de um problema; procedimento sequenciado que leva ao cumprimento de uma tarefa.
4 LÓG Conjunto das regras de operação (conjunto de raciocínios) cuja aplicação permite resolver um problema enunciado por meio de um número finito de operações; pode ser traduzido em um programa executado por um computador, detectável nos mecanismos gramaticais de uma língua ou no sistema de procedimentos racionais finito, utilizado em outras ciências, para resolução de problemas semelhantes.
5 INFORM Conjunto de regras e operações e procedimentos, definidos e ordenados usados na solução de um problema, ou de classe de problemas, em um número finito de etapas.

2.2 Definição

Vamos assumir a seguinte definição de algoritmo que nos dá a noção essencial.

Definição 2.1 *Algoritmo é um encadeamento finito de regras ou instruções tal que a sua execução é feita de forma mecânica e determinista, em tempo finito, e resolve um problema ou cumpre uma tarefa.*

Podemos pensar num algoritmo como um procedimento realizado em etapas encadeadas, ou sequenciais, e precisas que atingem uma meta pretendida.

Exemplo simples e cotidiano de algoritmo é a receita para o preparo de um prato.

Receita do sanduíche Bauru: corta-se o pão francês ao meio e retira-se o miolo da parte superior, como se fosse uma pequena canoa; na metade inferior, colocam-se as fatias frias de rosbife e sal a gosto; por cima, distribuem-se algumas rodelas de tomate e pepino, polvilhando com orégano a gosto; à parte, coloca-se um pouco de água numa frigideira. Quando ferver, coloca-se a mussarela a ser derretida; retira-se a mussarela da água e coloca-se na metade da canoa da parte superior do pão, unindo-se as duas partes. O calor da mussarela vai aquecer os ingredientes da outra metade.

Agora um exemplo bastante conhecido da Matemática.

Algoritmo de Báskara: Usamos este algoritmo para o cômputo das raízes de uma equação do segundo grau. Consideremos o tratamento apenas sobre o conjunto dos números reais $\mathbb{R}$, em que uma equação do segundo grau tem a forma $ax^2+bx+c=0$, com a,b e c números reais, $a\neq 0$ e as raízes também devem ser, caso existam, números reais.

Iniciamos com o cálculo do discriminante da equação do segundo grau $\Delta = b^2-4ac$, que depende apenas dos coeficientes a,b e c.

Se $\Delta<0$, então a equação não tem raízes reais. Se $\Delta=0$, a equação tem exatamente uma raiz real. Se $\Delta>0$, a equação tem raízes reais distintas.

As raízes devem ser calculadas pela fórmula $x=\dfrac{-b\pm\sqrt{\Delta}}{2a}$. Quando $\Delta=0$, então a raiz única é $x=\dfrac{-b}{2a}$. Se $\Delta>0$, então as raízes são $x_1=\dfrac{-b-\sqrt{\Delta}}{2a}$ e $x_2=\dfrac{-b+\sqrt{\Delta}}{2a}$.

O algoritmo é completamente determinístico. Para toda equação $ax^2+bx+c=0$ podemos sempre determinar a solução do problema como acima.

2.3 Aplicação

Para uma quantidade enorme de tarefas executadas pelos humanos, parece haver um algoritmo subjacente. Mesmo que nem sempre sejam reconhecidos como norteadores das ações daquela tarefa.

Há mesmo quem defenda que toda tarefa exige um algoritmo subjacente. Não precisamos decidir sobre esta disputa.

De fato, precisamos reconhecer que executamos algoritmos para muitas tarefas cotidianas e que temos algoritmos para a solução de muitos problemas matemáticos, os quais aprendemos na nossa formação escolar.

2.4 Representação de algoritmos

Os algoritmos podem ser representados por formas diversas, alguns dos exemplos mais conhecidos são: as descrições narrativas, os fluxogramas e os pseudocódigos.

Não há uma melhor representação. A escolha depende de quem apresenta o tópico e a aplicação do algoritmo proposto. Algumas apresentações são mais detalhadas e outras menos, algumas têm maior apelo visual, outas estão mais próximas das linguagens de programação.

A seguir, apresentaremos brevemente uma forma geral de representar algoritmos com pseudocódigos, pois é a representação que mais nos interessa neste trabalho.

2.5 Pseudocódigo

A representação em pseudocódigo é rica em detalhes e tem formato próximo dos programas computacionais. Daí, o salto para a programação ser menor e sua relevância para a computação.

De modo geral, podemos desenvolver assim:

Algoritmo <nome_do_algoritmo>
    <declaração_de_variáveis>
    <subalgoritmos>
Início
    <corpo do algoritmo>
Fim_Algoritmo

A palavra **Algoritmo** determina o início do algoritmo na forma de pseudocódigo.

A expressão <nome_do_algoritmo> indica um espaço para nomear o algoritmo.

A expressão <declaração_de_variáveis> caracteriza parte opcional do algoritmo para a declaração de variáveis globais do algoritmo.

A expressão <subalgoritmos> separa uma parte opcional do pseudocódigo para construção de algoritmos parciais que podem compor o todo.

O pseudocódigo fica limitado pelo Início e Fim.

A seguir, mostramos uma representação em pseudocódigo do algoritmo do cômputo da média.

Algoritmo <Cômputo da Média>
Declare MP, NT, MF (real)
Início
    Leia MP, NT
    MF $\longleftarrow MP * 0.8 + NT * 0.2$
    Se MF ≥ 5.0 **então**
        Escreva 'Aprovado'
    Senão
        Escreva 'Reprovado'
Fim_Algoritmo.

3 Funções $\mu-$recursivas

Nesta seção, apresentamos as funções recursivas, as quais chamaremos a partir de agora por funções $\mu-$recursivas, para não haver problemas de interpretação mais a frente, quando trabalharmos com funções na nossa pseudolinguagem que fazem uso da recursão.

Na nossa pesquisa, mostramos que uma série de operações aritméticas, relacionais e lógicas como a soma, a relação de igualdade e a conjunção, podem ser dadas por funções recursivas, mas neste artigo omiteremos algumas, devido ao espaço.

As definições aqui apresentadas são baseadas em Mendelson (1964), Santiago e Bedregal (2004) e Dias e Weber (2010).

3.1 Definições

Inicialmente, apresentamos as funções recursivas iniciais, que são básicas para a definição da classe das funções $\mu-$recursivas.

Todas as funções $\mu-$recursivas são definidas sobre o conjunto dos números naturais $\mathbb{N}$, isto é, são funções $f : \mathbb{N}^m \to \mathbb{N}^k$.

Definição 3.1 *Para quaisquer $x, x_1, x_2, ..., x_n \in \mathbb{N}$, as funções recursivas iniciais ou básicas são:*
(i) Função nula: $N(x) = 0$;
(ii) Função sucessor: $S(x) = x + 1$;
(iii) Função projeção: $P_i^n(x_1, ..., x_n) = x_i$, para $1 \leq i \leq n$.

A seguir, definimos as operações que serão usadas para a obtenção de novas funções.

Definição 3.2 *Se g é uma função $\mu-$recursiva de aridade m e $h_1, ..., h_m$ são funções $\mu-$recursivas, todas de aridade n, então a função $\mu-$recursiva f de aridade n é obtida por composição a partir das funções g e $h_1, ..., h_m$ quando:*
$f(x_1, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_m(x_1, ..., x_n))$.

Definição 3.3 *Se g é uma função $\mu-$recursiva de aridade n e h é uma função $\mu-$recursiva de aridade $n+2$, então a função $\mu-$recursiva f de aridade $n+1$ é obtida por recursão a partir das funções g e h quando:*
$f(x_1, ..., x_n, 0) = g(x_1, ..., x_n)$
$f(x_1, ..., x_n, y+1) = h(x_1, ..., x_n, y, f(x_1, ..., x_n, y))$.

A próxima regra para obtenção de funções $\mu-$recursivas é o operador de busca do menor valor. Chamamos esta regra de operador de minimalização ou μ-operador.

Notação: Se g é uma função $\mu-$recursiva de aridade $n+1$, então o menor y para o qual $g(x_1, ..., x_n, y) = 0$ é indicado por $\mu_y(g(x_1, ..., x_n, y) = 0)$.

Em geral, para qualquer predicado ou relação $\varphi(x_1, ..., x_n, y)$, denotamos por $\mu_y \varphi(x_1, ..., x_n, y)$ ao menor y para o qual a relação $\varphi(x_1, ..., x_n, y)$ é válida.

Definição 3.4 *A função $\mu-$recursiva f de aridade n é obtida pelo μ-operador a partir da função $\mu-$recursiva g se:*
$f(x_1, ..., x_n) = \mu_y(g(x_1, ..., x_n, y) = 0)$.

Precisamos ainda, para podermos definir o conceito de funções $\mu-$recursivas, da definição de funções regulares.

Definição 3.5 *Uma função $g(x_1, ..., x_n, y)$, com $n \geq 1$, é regular se g é uma função total, isto é, ela é definida para todos $x_1, ..., x_n, y \in \mathbb{N}$, e é tal que para todos $x_1, ..., x_n \in \mathbb{N}$, existe $y \in \mathbb{N}$ de modo que $g(x_1, ..., x_n, y) = 0$.*

A partir das funções iniciais e das regras apresentadas, conseguimos finalmente apresentar uma definição para as funções recursivas.

Definição 3.6 *A classe das funções $\mu-$recursivas é a menor classe de funções $f : \mathbb{N}^m \to \mathbb{N}^k$ que contém as funções iniciais e é fechada sob as operações de composição, recursão e a aplicação do μ-operador a funções regulares.*

Podemos definir também a classe das funções $\mu-$recursivas parciais.

Definição 3.7 *A classe das funções $\mu-$recursivas parciais é a menor classe de funções $f : \mathbb{N}^m \to \mathbb{N}^k$ que contém as funções iniciais e é fechada sob as operações de composição, recursão e a aplicação irrestrita do μ-operador.*

Em 1936, Alonzo Church apresentou num artigo o que chamamos de a "Tese de Church". A Tese de Church diz que toda função algorítmica é $\mu-$recursiva, e em sua versão mais ampla, que toda função parcial algorítmica é parcial $\mu-$recursiva.

Como a definição de algoritmo é um conceito intuitivo, não há como provar essa tese, porém existe na literatura uma vasta discussão sobre ela, como exemplo, em Carnielli e Epstein (2006), com bom desenvolvimento teórico sobre Computabilidade em geral e a Tese de Church em particular.

3.2 Demonstrações

De posse da definição de funções $\mu-$recursivas, e também de funções $\mu-$recursivas parciais, então podemos colocar em prática as demonstrações propostas no começo dessa seção, que serão muito importantes para a construção da nossa pseudolinguagem, na qual devem constar os operadores aritméticos, relacionais e lógicos, usuais das pseudolinguagens.

3.2.1 Adição

Intuitivamente, conseguimos definir recursivamente a soma da seguinte forma:

$$x+0 = x$$
$$x+(y+1) = (x+y)+1.$$

A adição é definida a partir da função sucessor que, como função básica, é $\mu-$recursiva.

Para verificarmos que a função soma, dada por $+(x,y) = x+y$, é $\mu-$recursiva faremos a seguinte derivação recursiva:

Demonstração:

(1) $S(x) = x+1$	função inicial
(2) $P^3_3(x,y,z) = z$	função inicial
(3) $P^1_1(x) = x$	função inicial
(4) $+(x,0) = P^1_1(x)$	recursão base: 3
(5) $+(x,S(y)) = S(P^3_3(x,y,+(x,y)))$	recursão passo indutivo: 1, 2.

■

Com uma notação menos carregada, omitindo as funções de projeção, temos que a função soma é dada pela regra de recursão por:

$+(x,0) = x$

$+(x,S(y)) = S(+(x,y))$.

3.2.2 Somatório

Uma vez que a adição é $\mu-$recursiva, podemos concluir também que o operador de somatório é recursivo. Intuitivamente, podemos definir o somatório da seguinte forma:

$$\sum_{y<z} f(x_1,...,x_n,y) = \begin{cases} 0 & \text{se } z = 0 \\ f(x_1,...,x_n,0) + ... + f(x_1,...,x_n,z-1) & \text{se } z > 0 \end{cases}$$

Segue a derivação recursiva para verificarmos que a função somatório, dada por $\sigma(x_1,...,x_n,z) = \sum_{y<z} f(x_1,...,x_n,y)$, é $\mu-$recursiva, dado que f é uma função $\mu-$recursiva qualquer.

Demonstração:

(1) $N(x) = 0$	função inicial
(2) $+(x,y) = x+y$	adição
(3) $g(x_1,...,x_n,y,z) = +(f(x_1,...,x_n,y),z)$	composição de (2), projeção e f
(4) $\sigma(x_1,...,x_n,0) = N(x)$	recursão base: 1
(5) $\sigma(x_1,...,x_n,S(z)) = g(x_1,...,x_n,z,\sigma(x_1,...,x_n,z))$	recursão passo indutivo: 3.

■

De forma simplificada, temos que a função somatório é dada por recursão a partir da adição do seguinte modo:

$\sigma(x_1,...,x_n,0) = 0$

$\sigma(x_1,...,x_n,S(z)) = +(f(x_1,...,x_n,z),\sigma(x_1,...,x_n,z))$.

3.2.3 Subtração

A subtração não é tão intuitiva quanto os casos anteriores, pois como as funções $\mu-$recursivas são definidas sobre os números naturais, não temos números negativos. Logo, temos que decidir como tratar subtrações como $x-y$ em que $x < y$. Para isso, definiremos a subtração truncada que será usada em nossa linguagem como uma subtração própria da funções $\mu-$recursivas, que é definida intuitivamente da seguinte forma:

$$x \dot{-} y = \begin{cases} x-y, & \text{se } x \geqslant y \\ 0, & \text{se } x < y. \end{cases}$$

Segue a derivação recursiva para demonstrar que a função de subtração truncada, dada por $-(x,y) = x \dot{-} y$, é $\mu-$recursiva, considerando o fato que para $x \geqslant y$, $x-(y+1) = (x-y)-1$.

Demonstração:

(1) $N(x) = 0$	função inicial
(2) $P_1^1(x) = x$	função inicial
(3) $P_1^2(x,y) = x$	função inicial
(4) $f(x_1,x_2,0) = 0$	recursão base: 1
(5) $f(x_1,x_2,S(y)) = P_1^2(y,f(x_1,x_2,y))$	recursão passo indutivo: 3, 4
(6) $\dot{-}(x,0) = x$	recursão base: 2
(7) $\dot{-}(x,S(y)) = f(x,y,\dot{-}(x,y))$	recursão passo indutivo: 5

■

Caso seja necessário, a diferença absoluta é facilmente obtida a partir da subtração truncada:

$$|x-y| = (x \dot{-} y) + (y \dot{-} x)$$

3.2.4 Constante

Para qualquer número natural n, a função constante $C_n(x) = n$ é $\mu-$recursiva. Esse resultado é importante para as próximas demonstrações, assim como será importante na construção da nossa pseudolinguagem. A prova é feita por indução sobre n.

Naturalmente, $C_0(x) = N(x)$ que é $\mu-$recursiva. Agora, se $C_n(x)$ é $\mu-$recursiva, então $C_{n+1}(x) = S(C_n(x))$ que também é $\mu-$recursiva, por composição. ■

3.2.5 Sinal e contrassinal

As funções sinal e contrassinal não serão implementadas diretamente na nossa pseudolinguagem, porém são importantes para as próximas demonstrações.

A função sinal é definida intuitivamente como:

$$sg(x) = \begin{cases} 1, & \text{se } x \neq 0 \\ 0, & \text{se } x = 0 \end{cases}$$

Segue a derivação recursiva para a verificação de que a função sinal é $\mu-$recursiva:

Demonstração:

(1) $N(x) = 0$	função inicial
(2) $P_1^2(x,y) = x$	função inicial
(3) $f(x,y) = C_1(P_1^2(x,y))$	composição de função constante e projeção
(4) $sg(0) = 0$	recursão base: 1
(5) $sg(S(y)) = f(y, sg(y))$	recursão passo indutivo: 3

■

De forma simplificada, temos que a função sinal é dada por
$sg(0) = 0$
$sg(S(y)) = 1$.

A função contrassinal, por sua vez, é definida intuitivamente como:

$$\overline{sg}(x) = \begin{cases} 0, & \text{se } x \neq 0 \\ 1, & \text{se } x = 0 \end{cases}$$

Segue a derivação recursiva para a verificação de que a função sinal é $\mu-$recursiva:

Demonstração:

(1) $N(x) = 0$	função inicial
(2) $P_1^2(x,y) = x$	função inicial
(3) $f(x,y) = N(P_1^2(x,y))$	composição: 1, 2
(4) $\overline{sg}(0) = C_1(0)$	recursão base: constante
(5) $\overline{sg}(S(y)) = f(y, \overline{sg}(y))$	recursão passo indutivo: 3

■

De forma simplificada, temos que a função contrassinal é dada por
$\overline{sg}(0) = 1$
$\overline{sg}(S(y)) = 0$.

3.2.6 Operadores relacionais

Iniciando as demonstrações dos predicados que representarão os operadores relacionais da pseudolinguagem, temos a relação "$x = y$".

A função característica desse predicado é dada por $\overline{sg}(|x-y|)$. Como as funções contrassinal e de diferença absoluta são $\mu-$recursivas, então o predicado "$x = y$" também é $\mu-$recursivo.

Analogamente, temos que "$x \neq y$" é um predicado $\mu-$recursivo, e sua função característica é dada por $sg(|x-y|)$.

O predicado "$x < y$" por sua vez tem sua função característica dada por $sg(y \dot{-} x)$. Como as funções sinal e subtração truncada são $\mu-$recursivas, segue que o predicado "$x < y$" também é $\mu-$recursivo.

Analogamente, temos que "$x \geqslant y$" é um predicado $\mu-$recursivo, e sua função característica é dada por $\overline{sg}(y \dot{-} x)$.

Da mesma forma, a relação "$x > y$" tem sua função característica dada por $sg(x \dot{-} y)$, e como as funções sinal e subtração truncada são $\mu-$recursivas, então o predicado "$x > y$" também é $\mu-$recursivo.

Analogamente, temos que "$x \leqslant y$" é um predicado $\mu-$recursivo, e sua função característica é dada por $\overline{sg}(x \dot{-} y)$.

3.2.7 Operadores lógicos

Sejam $\varphi(x_1,...,x_n)$ e $\psi(x_1,...,x_n)$ dois predicados $\mu-$recursivos de aridade n e sejam $\chi_\varphi(x_1,...,x_n)$ e $\chi_\psi(x_1,...,x_n)$ suas funções características, respectivamente. Então:

(1) $\varphi(x_1,...,x_n) \vee \psi(x_1,...,x_n)$ é um predicado $\mu-$recursivo, e sua função característica é dada por $sg(\chi_\varphi(x_1,...,x_n) + \chi_\psi(x_1,...,x_n))$;

(2) $\varphi(x_1,...,x_n) \wedge \psi(x_1,...,x_n)$ é um predicado $\mu-$recursivo, e sua função característica é dada por $\chi_\varphi(x_1,...,x_n).\chi_\psi(x_1,...,x_n)$;

(3) $\neg\varphi(x_1,...,x_n)$ é um predicado $\mu-$recursivo, e sua função característica é dada por $\overline{sg}(\chi_\varphi(x_1,...,x_n))$.

4 Linguagem REC

Esta seção é destinada à pseudolinguagem proposta.

De fato, desenvolveremos duas versões, a primeira, denotada por *REC*, conterá entre suas operações apenas aquelas definidas pelas funções iniciais das funções $\mu-$recursivas. Assim, não teremos ainda operações como soma, multiplicação, subtração, divisão, módulo, relações de maior, menor, entre outras, com exceção apenas da estrutura de seleção, apresentada mais a frente, pois caso contrário os códigos em *REC* poderiam ficar desnecessariamente complexos e a linguagem perderia seu objetivo.

A segunda linguagem, denotada por *REC++*, por sua vez, fornecerá todos esses operadores diretamente, simplificando a programação e a legibilidade do código.

Nesta seção apresentaremos a linguagem *REC*, e tudo o que é válido para a linguagem *REC* também é válido para a linguagem *REC*++.

Daqui por diante usaremos a expressão *REC*-calculável ou *REC*-computável para designar toda função para qual existe um algoritmo escrito na linguagem *REC* que a calcule. Um dos nossos principais objetivos é provar os seguintes teoremas:

Teorema 4.1 *Uma função é REC-calculável se, e somente se, é $\mu-$recursiva.*

Teorema 4.2 *Uma função é parcial REC-calculável se, e somente se, é parcial $\mu-$recursiva.*

A prova desses teoremas se dará naturalmente durante a apresentação da linguagem neste capítulo, pois iremos incluir todas as funções básicas das funções $\mu-$recursivas e as operações de composição, recursão primitiva e minimalização na linguagem *REC*, o que garantirá que toda função $\mu-$recursiva seja *REC*-calculável, e para tudo aquilo que será acrescentado de diferente, será demonstrado que pode ser definido por uma função $\mu-$recursiva, tornando impossível fazer algo na linguagem *REC* que já não possa ser feito por alguma função $\mu-$recursiva. Isso será enfatizado novamente no Capítulo 5.

4.1 Estrutura de um código em REC

Um código ou um programa escrito na linguagem *REC* será estruturado em funções. Uma função principal, que é a função pela qual o programa começa a execução, e uma série de funções auxiliares. Uma função auxiliar, dados alguns parâmetros, realiza operações sobre eles e retorna um valor.

A função principal será identificada por um prefixo 'main' antes do identificador da função. De modo geral, desenvolvemos assim:

```
<declaração_de_constantes>

<funções_auxiliares>

main <nome_da_função>()
{
    <declaração_de_variáveis>
    <comandos>
}
```

Podemos definir uma estrutura geral para uma função auxiliar da seguinte forma:

```
<nome_da_função>([parâmetro, ...])
{
    <declaração_de_variáveis>
    <comandos>
}
```

A seguir, de forma intuitiva, apresentaremos os componentes específicos da linguagem *REC*. Na seção seguinte, introduzimos o que é acrescentado na linguagem *REC*++ e, ao fim de cada

uma das seções, introduzimos gramáticas livres de contextos em notação BNF para a representação formal da sintaxe dessas linguagens.

4.2 Tipos de dados

Nossa linguagem contempla, basicamente, dois tipos de dados: os dados numéricos (inteiros e não negativos) e os dados alfanuméricos (*strings*).

Estes últimos, porém, só estarão disponíveis para uso em comandos de saída, para interação com o usuário, pensando na execução de um programa. Dados alfanuméricos serão delimitados por aspas.

Exemplo 4.3 *Dados:*
13 - dado numérico
-15 - não aceito pela linguagem
12.5 - não aceito pela linguagem
"texto" - dado alfanumérico
"13" - dado alfanumérico.

4.3 Identificadores

Identificadores são nomes usados para representar uma variável ou uma função.

Para determinar os identificadores aceitos por nossas linguagens, usaremos as seguintes regras:

1. O primeiro caractere de um identificador deve ser uma letra ou o sublinhado _.
2. Os demais caracteres do identificador podem ser letras, números ou o sublinhado, mas nenhum outro tipo de caractere é permitido.

Porém, há algumas palavras que não podem ser usadas como identificadores, pois são próprias da linguagem e serão usadas para outros fins, sendo elas: 'main', 's', 'if', 'else', 'var' e 'rec'. A essas palavras damos o nome de palavras reservadas.

4.4 Declaração de variáveis

Nossas linguagens trazem apenas variáveis para armazenamento de dados numéricos.

Dessa forma, para declararmos uma variável ou um conjunto de variáveis, usaremos o prefixo "var", seguido dos identificadores de cada variável, separados por vírgula, e ao fim, um ponto e vírgula para encerrar o comando.

Exemplo 4.4 *Exemplos de declaração de variáveis:*
var x, y, z;
var contador;

Além disso, podemos declarar um vetor para armazenar um conjunto de dados numéricos.

A declaração de um vetor será feita de forma semelhante às variáveis comuns, mas com o acréscimo de um "[ ]" na frente do identificador do vetor.

Exemplo 4.5 *Exemplos de declaração de vetores:*
var x[ ], vetor[ ];

As variáveis em *REC* são todas variáveis locais, ou seja, elas são restritas à função em que foram declaradas, não podendo ser utilizadas em outra função.

Duas funções distintas podem, inclusive, terem a declaração de variáveis com o mesmo nome, sem nenhum problema. Afinal estão em contextos diferentes. Em uma dada função, porém, duas variáveis não podem ter o mesmo nome.

4.5 Iniciação de variáveis

Na linguagem *REC*, possibilitaremos que uma variável receba um valor inicial, dado por uma expressão qualquer, logo na sua declaração. Tal procedimento é conhecido como iniciação de variável. Toda variável não inicializada consideraremos que se inicia com o valor zero.

Exemplo 4.6 *Iniciação de variáveis em REC:*
$var\ x, y = 0, z = s(s(s(0)));$ $\quad //x = 0, y = 0, z = 4$
$var\ contador = s(0);$ $\quad //contador = 1$

A notação $//$ indica um comentário no trecho do algoritmo, que podemos usar para explicar determinadas partes do código e torná-lo mais claro para quem o está lendo.

4.6 Constantes

O uso de constantes em um algoritmo é bastante recorrente e o nosso é como usual.

Na linguagem *REC*, a única constante pré-definida será o número 0, para representar a função nula.

4.7 Funções iniciais

Visto que a nossa linguagem é motivada pelas funções recursivas, temos que implementar, de alguma forma, estruturas equivalentes às funções iniciais das funções recursivas, que são as funções nula, sucessor e as projeções.

A função nula será representada pela constante 0, como já dito anteriormente.

A função sucessor, por sua vez, será representada pela função $s(x)$, que retornará o sucessor do argumento x.

Já para as funções de projeção, não precisamos de uma estrutura específica para ela, pois o uso das funções de projeção será feito de forma natural durante o programa, de forma parecida com que o que acontecia com as demonstrações feitas no capítulo de funções recursivas quando as omitíamos.

Exemplo 4.7 *Trecho de código e, como comentário, as funções $\mu-$recursivas equivalentes:*
$x_1 = 0;$ $\quad //x_1 = N(0)$
$x_2 = s(x_3);$ $\quad //x_2 = S(x_3)$ *ou* $x_2 = S(P_3^3(x_1, x_2, x_3))$
$x_3 = x_1;$ $\quad //x_3 = P_1^3(x_1, x_2, x_3)$

Em um caso particular do uso das funções de projeção, podemos considerar o acesso a uma posição específica de um vetor. Para fazer tal acesso, usaremos o identificador do vetor seguido de "$[i]$", quando i representa a posição desejada, podendo ser uma expressão qualquer que contenha constantes, variáveis e/ou funções.

Exemplo 4.8 *Trecho de código em REC:*

var $pos, x[\,]$;
$x[0] = s(s(s(0)))$; $\quad //x[0] = 3$
$pos = s(0)$; $\quad //pos = 1$
$x[pos] = s(x[0])$; $\quad //x[1] = s(3) = 4$
$x[s(pos)] = s(s(pos))$; $\quad //x[2] = s(s(1)) = 3$

As funções de projeção, porém, são definidas para posições constantes, e aqui estamos permitindo o uso de variáveis e funções. Porém a demonstração que tal operação é dada por uma função recursiva é muito simples.

O nosso problema é determinar que a função $P^n(x_1, x_2, ..., x_n, i) = x_i \mid i \leq n$ é uma função $\mu-$recursiva. Tal função pode ser dada da seguinte forma:

$$P^n(x_1, x_2, ..., x_n, i) = \sum_{j=1}^{n} \overline{sg}(i \dot{-} j).x_j.$$

Uma vez que já verificamos que são $\mu-$recursivas as operações: somatório, contrassinal, subtração própria e multiplicação, então a função $P^n(x_1, x_2, ..., x_n, i)$ também é $\mu-$recursiva. ■

4.8 Comandos de entrada e saída

Para a entrada de dados em nossos códigos em *REC*, usaremos o comando *'read(<variáveis>)'*, em que o trecho *<variáveis>* será substituído pelas variáveis que serão lidas, separadas por vírgulas.

Já para a saída de dados, o comando usado será *'print(<string>)'*, em que *<string>* representa uma *string*, um dado alfanumérico qualquer.

O comando *print* pode ser usado para exibir o conteúdo de uma ou mais variáveis. Para isso, usaremos o identificador da variável em questão e consideraremos que a conversão do dado numérico para alfanumérico está implícita.

Também é possível concatenar várias *strings*. Para tanto, usaremos o símbolo +.

Exemplo 4.9 *Uso dos comandos read e print em REC:*

```
main teste()
{
    var x, y, resultado;
    print("Digite dois números naturais:")
    read(x, y);
    resultado = s(x);
    print("O sucessor de " + x + " é " + resultado);
    resultado = s(y);
    print("O sucessor de " + y + " é " + resultado);
}
```

Para facilitar as demonstrações que serão feitas nos próximos itens desta seção, não iremos considerar o uso do comando *read* dentro de funções auxiliares. Consideraremos que todos os dados de entrada necessários para aquela função estão sendo recebidos por algum parâmetro.

Não há nenhum problema nessa simplificação, já que uma função auxiliar que apresente o comando *read* pode ser facilmente alterada para receber as variáveis em questão por parâmetro ao invés de serem lidas dentro da própria função.

4.9 Operadores

A seguir, apresentamos os operadores que farão parte da linguagem *REC*, que são basicamente o operador de atribuição e o operador de minimalização.

Em alguns momentos falaremos de expressões aritméticas, relações e expressões lógicas. Todas essas expressões podem ser expressadas por funções, no caso das relações e expressões lógicas por funções que retornam sempre o valor 0, representando que o valor da expressão lógica ou relação é falso, ou 1, representando que a expressão lógica ou relação é verdadeira.

4.9.1 Operador de atribuição

Através de um comando de atribuição podemos armazenar dados, valores, em uma determinada variável. O símbolo de atribuição empregado em nossas linguagens será "=", e o comando em si será composto pelo símbolo "=", um operando à esquerda (a variável que receberá o valor dado) e um operando à direita (uma expressão qualquer que terá seu valor armazenado na variável).

Exemplo 4.10 *Atribuição em REC:*
$x = s(0)$;
$y = s(x)$;
$z = s(y)$;

Quando atribuímos uma expressão a uma variável é como se estivéssemos atribuindo um nome a uma determinada função ou uma composição de funções. Podemos obter facilmente a função $\mu-$recursiva dessas expressões substituindo o identificador da variável por seu valor.

Exemplo 4.11 *Considerando o exemplo anterior temos que* $x = S(0)$, $y = S(x)$ *e* $z = S(y)$, *logo* $z = S(S(S(0)))$.

4.9.2 Operador de minimalização

O operador de minimalização é parte fundamental da linguagem *REC*, cujo conceito vem diretamente da definição do μ-operador das funções $\mu-$recursivas.

O operador de minimalização funcionará como um operador de atribuição, ou seja, atribuirá um valor a uma determinada variável, e sua sintaxe será dada por:
$y << R(x_1, x_2, ..., x_n, y)$;

Nesse caso, $R(x_1, x_2, ..., x_n, y)$ representa uma relação ou uma função que retorna o valor de uma expressão lógica que envolve as variáveis $x_1, x_2, ..., x_n$ e y, de forma que y receba o menor número natural tal que $R(x_1, x_2, ..., x_n, y)$ retorne 1, ou seja, que a expressão lógica correspondente

seja verdadeira.

Podemos também considerar o μ-operador limitado, com a seguinte sintaxe:
$y << R(x_1, x_2, ..., x_n, y), z;$

A ideia é a mesma do caso anterior, porém agora temos um teto para o valor de y. Dessa forma, y irá receber o menor número natural, menor que z, tal que $R(x_1, x_2, ..., x_n, y)$ retorne 1. Caso não exista tal valor, y receberá z.

Exemplo 4.12 *Uso do operador de minimalização em REC, supondo que foram implementadas a função auxiliar igual*(x,y) *que retorna* 1 *se* $x = y$ *e* 0 *caso contrário, e a função auxiliar mult*(x,y) *que retorna o produto entre x e y:*

$raiz << igual(mult(raiz, raiz), x);$ $\qquad //raiz = raiz\ de\ x$

A diferença entre as funções *REC*-calculáveis das funções parciais *REC*-calculáveis é a mesma das funções $\mu-$recursivas e as parciais $\mu-$recursivas, que é em relação a aplicação do operador de minimalização.

As funções *REC*-calculáveis são aquelas em que o uso do operador de minimalização se restringe às funções regulares.

As funções parciais *REC*-calculáveis são aquelas que o uso do operador de minimalização é irrestrito. Nesse caso pode ocorrer a seguinte situação: dada uma função não regular, se ela for chamada para receber um determinado argumento como parâmetro, o programa em *REC* pode entrar em 'loop' e nunca terminar sua execução.

4.10 Estrutura de seleção

A linguagem *REC* apresenta uma estrutura de seleção que possibilita a escolha de um conjunto de comandos a partir de condições, tal qual a maioria das linguagens de programação.

Há dois tipos de estrutura de seleção principais, a simples, usando o comando *'if'*, e a composta, adicionando ao comando *'if'* a estrutura *'else'*.

Na estrutura de seleção simples, usamos o comando *'if'*, da seguinte forma:

```
if (condição) {
    <comandos>
} else {
    <comandos>
}
```

Em que 'condição' representa uma expressão lógica qualquer. As chaves podem ser omitidas se o bloco de comandos tiver apenas um comando. O *'else'* é opcional. O primeiro bloco de comandos só será executado caso a 'condição' seja verdadeira, enquanto o segundo apenas se ela for falsa.

Essa estrutura de seleção permite uma série de desvios condicionais em nosso programa, o que basicamente nos permite criar funções definidas por partes.

Para provarmos que tal operação é uma função $\mu-$recursiva, vamos considerar uma função f qualquer escrita em *REC* com diversos desvios condicionais, ou seja, caso certas condições sejam satisfeitas, ou não, uma série de comandos diferentes podem ser executados.

Podemos mapear todas as possíveis execuções dessa função, sendo que para cada caminho i possível, definiremos uma função g_i, e este só será executado caso a condição (ou predicado) R_i for verdadeiro. Todas essas condições são, evidentemente, mutuamente exclusivas. Ou seja, matematicamente temos a seguinte situação:

$$f(x_1,...,x_n) = \begin{cases} g_1(x_1,...,x_n) & \text{se } R_1(x_1,...,x_n) \\ g_2(x_1,...,x_n) & \text{se } R_2(x_1,...,x_n) \\ ... & \\ g_m(x_1,...,x_n) & \text{se } R_m(x_1,...,x_n) \end{cases}$$

A partir disso podemos definir a função f da seguinte forma, considerando que cada predicado R_i tem sua função característica χ_{R_i}:

$$f(x_1,...,x_n) = g_1(x_1,...,x_n).\chi_{R_1}(x_1,...,x_n) + ... + g_m(x_1,...,x_n).\chi_{R_m}(x_1,...,x_n)$$

Sabendo que a adição é dada por uma função $\mu-$recursiva e que $g_1, g_2, ..., g_m$ são funções $\mu-$recursivas e $R_1, R_2, ..., R_m$ são predicados $\mu-$recursivos, então a função f também é $\mu-$ recursiva. ■

4.11 Funções auxiliares ou sub-algoritmos

No início deste capítulo já apresentamos a estrutura geral para definir uma função auxiliar.

Toda função auxiliar em *REC* deve retornar um valor inteiro não negativo, e para isso será usado o comando *'return'* ao fim de cada possível caminho de execução, que será responsável por retornar o valor de uma dada expressão.

Deve-se ressaltar que quaisquer comandos da função auxiliar após a execução de um *'return'* serão ignorados.

Uma função pode ser chamada dentro de uma expressão, da seguinte forma: *'nome_da_função*([*parâmetros*])*'*. No lugar de *'*[*parâmetros*]*'* serão passados quantos forem os parâmetros exigidos pela função, sendo que esses parâmetros podem ser expressões quaisquer. Porém, só pode ser feita uma chamada de função se a função em questão foi definida anteriormente, ou seja, se está acima da função em que a chamada está ocorrendo.

Podem haver funções auxiliares com o mesmo nome em *REC*, desde que a quantidade de parâmetros seja diferente de uma para outra. Na chamada da função podemos distinguir qual função será chamada justamente por esse fator.

Uma particularidade é quando precisamos receber um vetor como parâmetro. Na definição da função será indicado que um dos parâmetros deve ser um vetor através do uso de colchetes ("[ ]") à frente do identificador. Na chamada da função, porém, deve-se colocar apenas o identificador do vetor, sem os colchetes.

Exemplo 4.13 *Função auxiliar com vetor como parâmetro em REC, supondo que foi implementada a função auxiliar igual*(x,y) *que retorna* 1 *se* $x = y$ *e* 0 *caso contrário:*

```
primeiroElemento(x[],n){
    if (igual(n,0))
        return 0;
    else
        return x[s(0)];
}

main teste(){
    var x[];
    x[s(0)] = 0;
    x[s(s(0))] = s(0);
    print(primeiroElemento(x,s(s(0))));
}
```

Dentre as operações básicas das funções recursivas, falta apresentar as operações de composição e a recursão primitiva. Agora que já apresentamos como funcionarão as funções auxiliares em *REC*, podemos mostrar como essas operações aparecerão nas nossas linguagens.

4.11.1 Composição

A operação de composição se dá de forma bastante simples, uma vez que uma expressão qualquer pode ser enviada como parâmetro na chamada de uma função, pois que qualquer expressão em *REC* pode ser descrita por funções $\mu-$recursivas. Em vários dos exemplos anteriores usamos a composição de funções.

4.11.2 Recursão primitiva

No caso da recursão, temos uma situação um pouco mais delicada. No geral, em linguagens usuais de programação, uma função pode ser chamada dentro dela mesmo, com quaisquer que sejam os parâmetros e a qualquer momento. O único cuidado que o programador deve ter ao fazê-lo é garantir que o programa não entre em 'loop' eterno ou que haja algum outro problema de compatibilidade lógica.

Porém, essa liberdade faz com que a recursão primitiva por si só nem sempre dê conta desses casos, e demonstrar que essas funções são $\mu-$recursivas pode ser um tanto complicado. Um exemplo é a função de Ackermann, que possui a seguinte definição:

$$\psi(x,y) = \begin{cases} y+1 & se\ x = 0 \\ \psi(x-1,1) & se\ x > 0\ e\ y = 0 \\ \psi(x-1,\psi(x,y-1)) & se\ x > 0\ e\ y > 0 \end{cases}$$

Segundo Santiago e Bedregal (2004), essa definição apresenta uma espécie de dupla recursão, que é mais forte que a função recursiva simples, e demonstrar que esta é $\mu-$recursiva é possível, mas bastante complexo.

Sendo assim, e lembrando que o que estamos propondo é uma ferramenta que estabeleça uma ponte fácil com as funções recursivas, iremos fazer um pouco diferente das linguagens de programação usuais e vamos engessar um pouco a recursão, limitando-a apenas à recursão primitiva.

Dessa maneira, uma função que faça uso da recursão primitiva terá a seguinte estrutura:

```
nome_da_função(x_1, x_2, ..., x_n, rec y)
{
    <declaração_de_variáveis>

    //base da recursão
    if (y == 0) {
        <comandos>
        return <expressão>;
    }

    //passo indutivo
    y = y − 1;
    nome_da_função = nome_da_função(x_1, x_2, ..., x_n, y);
    <comandos>
    return <expressão>;
}
```

Vale lembrar que na linguagem *REC*, os operadores aritméticos e relacionais como *'-'* e *'=='* não estão implementados (eles serão definidos e explicados na próxima seção sobre a linguagem *REC++*), porém permitiremos seu uso nesse caso em específico, visando a simplicidade e, como é um caso bem específico e até mesmo engessado, não interfere no objetivo na linguagem *REC*, que é apresentar um número bem reduzido de operações básicas.

O prefixo *'rec'* deve ser usado em apenas um dos parâmetros, indicando qual será o argumento em que se aplicará a recursão, não sendo necessariamente o último parâmetro, embora seja aconselhável para manter o código padronizado.

Após isso, a estrutura da função é auto-explicativa, mas vale ressaltar as duas primeiras linhas do passo indutivo. Nesse trecho temos a chamada da própria função (e essa é a única situação em que uma função permite autorreferência) com apenas o parâmetro demarcado com o prefixo *'rec'* modificado, enviando o seu antecessor. O resultado dessa chamada é atribuído para o identificador *'nome_da_função'* que consideraremos como uma variável que poderá ser usada livremente nas próximas linhas de código.

Mostrar que tal estrutura é equivalente à operação de recursão primitiva é bastante simples. Vamos dizer que uma função auxiliar qualquer com a estrutura apresentada acima seja equivalente a uma função $\mu-$recursiva f.

Nessa função temos o uso de uma estrutura de seleção, que define dois caminhos de execução possíveis, um que é executado quando a variável y é igual a 0, que podemos definir como uma função g; e outro caminho em que y seja diferente de 0 (embora não tenhamos usado a estrutura *'if-else'*, como o último comando dentro da estrutura de seleção é um *'return'* o que estiver após a estrutura de seleção só será executado caso a condição do *'if'* seja falsa) e, então, como estamos trabalhando com os números naturais, é equivalente a dizer que é o caminho executado caso y seja maior que 0, ao qual definiremos como uma função h.

Como dados de entrada para as funções g e h temos, evidentemente, todos os parâmetros recebidos pela função f, porém, no caso da função g, esta só será executada caso y seja 0. Portanto, não é necessário que ela receba o valor da variável y; e a função h, por sua vez, recebe o ante-

cessor da variável y e também o valor da chamada recursiva da função f, de forma que podemos definir a função f da seguinte forma:

$$f(x_1,x_2,...,x_n,y) = \begin{cases} g(x_1,x_2,...,x_n) & se\ y = 0 \\ h(x_1,x_2,...,x_n,y-1,f(x_1,x_2,...,y-1)) & se\ y > 0 \end{cases}$$

Ou ainda:

$$f(x_1,x_2,...,x_n,0) = g(x_1,x_2,...,x_n)$$
$$f(x_1,x_2,...,x_n,y+1) = h(x_1,x_2,...,x_n,y,f(x_1,x_2,...,x_n,y))$$

Esta apresentação nada mais é do que a definição que apresentamos anteriormente para a operação de recursão primitiva. ■

Exemplo 4.14 *Uso da recursão primitiva em REC:*

```
soma(x,rec y)
{
    var res;

    if (y == 0)
        return x;

    y = y - 1;
    soma = soma(x,y);

    res = s(soma);
    return res;
}

mult(x, rec y)
{
    if (y == 0)
        return 0;

    y = y - 1;
    mult = mult(x,y);

    return soma(mult,x);
}
```

5 Linguagem REC++

A linguagem *REC++*, além de todas as partes presentes na linguagem *REC*, contará com a adição dos operadores aritméticos, relacionais e lógicos que serão apresentados a seguir, além da possibilidade de um uso mais irrestrito de constantes.

Será mostrado que todos esses operadores também podem ser descritos por funções da linguagem *REC*, de forma que os conjuntos de funções computáveis pela linguagem *REC* e pela linguagem *REC++* são equivalentes, mesmo que na realidade isso nem fosse necessário, uma vez que todas as operações acrescidas podem ser descritas por funções $\mu-$recursivas.

Como a linguagem *REC++* apresenta todas as estruturas da linguagem *REC*, é imediato concluir que o seguinte teorema é válido.

Teorema 5.1 *Toda função REC-calculável é também REC++ calculável.*

Uma vez que todos os operadores acrescentados podem ser descritos por funções em *REC*, ou seja, por funções *REC*-calculáveis, então também podemos concluir que a recíproca é verdadeira, e então podemos estender o teorema.

Teorema 5.2 *Uma função é REC++ calculável se, e somente se, é REC-calculável.*

Por fim, pelo Teorema 4.1, como o conjunto de funções *REC*-calculáveis é equivalente ao conjunto de funções $\mu-$recursivas, junto ao Teorema 5.2, podemos dizer o mesmo em relação as funções *REC++* calculáveis.

Teorema 5.3 *Uma função é REC++ calculável se, e somente se, é $\mu-$recursiva.*

Teorema 5.4 *Uma função é parcial REC++ calculável se, e somente se, é parcial $\mu-$recursiva.*

Garantimos que o Teorema 4.1 é verdade durante a apresentação da linguagem *REC*, quando inserimos nessa linguagem todas as funções iniciais e operações básicas das funções $\mu-$recursivas, e tudo o que foi inserido além disso pode ser derivado das funções $\mu-$recursivas (com as devidas demonstrações apresentadas) ou são recursos próprios da programação que não intereferem diretamente na classe das funções computáveis e que normalmente apenas traduzem ações simples que fazemos quando estamos trabalhando com essas funções, como a entrada e saída de dados.

A vantagem da linguagem *REC++* em relação a *REC* é que ela é mais versátil, simples e intuitiva para a construção de alguns algoritmos.

5.1 Constantes

Na linguagem *REC++* poderemos usar qualquer número natural como uma constante, além de podermos atribuir identificadores para algumas constantes, com o intuito de facilitar a legibilidade do código, assim como possíveis alterações futuras; de forma que se o valor da constante for alterado, só teremos que adaptar a declaração da constante, e não substituí-la pelo novo valor em todas as partes em que ela era usada.

É importante ressaltarmos que, diferentemente de uma variável, uma constante não pode ter seu valor alterado durante a execução do algoritmo.

Para declararmos uma constante, usaremos a instrução *"#define"* seguida do identificador da constante e do seu valor.

Embora não seja necessário, é uma boa prática usarmos sempre caracteres maiúsculos no identificador da constante, para diferenciá-la das variáveis.

Exemplo 5.5 *Definição de constantes:*
#define MAX 20
#define DEZ 10
#define QUANTIDADE_ALUNOS 35

5.2 Operadores

A seguir, apresentamos os operadores que serão inclusos na linguagem *REC++*. Durante nossa pesquisa, demonstramos que cada um deles pode ser descrito por uma função em *REC*.

5.2.1 Operadores aritméticos

A linguagem *REC++* conterá os operadores aritméticos dados na tabela a seguir, sendo que todos podem ser definidos por funções recursivas, como já demonstrado.

Operação	Símbolo
Adição	+
Multiplicação	*
Subtração própria	-
Módulo (resto da divisão inteira)	%
Divisão	/

Uma expressão aritmética será formada por um operando à esquerda, um operador binário e um operando à direita, em que o operador é um dos símbolos da tabela acima, e o operando uma constante, uma variável, uma função ou mesmo uma outra expressão aritmética.

Para as expressões aritméticas com mais de um operador, a ordem em que as operações serão executadas seguirá as regras de precedência de operadores, que são as mesmas da matemática elementar.

Os operadores de multiplicação, divisão e módulo têm precedência sobre os operadores de adição e subtração.

A ordem de precedência dos operadores pode ser quebrada pelo uso de parênteses, quando podemos considerar operadores com mais alta precedência.

Exemplo 5.6 *Expressões aritméticas em REC++:*

$x = 10 * z + 2;$

$y = (x\%7 + z)/3;$

5.2.2 Operadores relacionais

A linguagem *REC++* conterá os operadores relacionais dados na tabela seguinte, que permitirão a construção de expressões relacionais (lógicas .0., .1.), sempre predicados recursivos, como já verificado.

Relação	Símbolo
Igualdade	$==$
Menor	$<$
Maior	$>$
Menor ou igual	$<=$
Maior ou igual	$>=$
Diferente	$!=$

Uma expressão lógica será formada por um operando à esquerda, um operador e um operando à direita, de modo que o operador seja um dos símbolos da tabela acima e cada operando seja uma constante, uma variável, uma função e/ou uma expressão aritmética.

Assim como na linguagem *C*, não definiremos um tipo lógico. Portanto, o resultado de uma expressão lógica qualquer será um valor numérico: 1 para uma expressão avaliada como verdadeira e 0 para uma expressão avaliada como falsa.

Exemplo 5.7 *Expressões relacionais em REC++:*
$x = y < 10$;
$y = z\ != x$; // *z é diferente de x*

5.2.3 Operadores lógicos

A linguagem *REC++* conterá os operadores lógicos da tabela seguinte, sendo que todos podem ser definidos por funções recursivas, como já verificado.

Operação	Símbolo
Negação	!
Disjunção (Ou)	||
Conjunção (E)	&&

Uma expressão lógica formada por operadores lógicos pode ser formada por um operador e um operando à direita, no caso da negação (unária), ou por um operando à esquerda, um operador e um operando à direita para as outras operações (binárias).

Em todo caso, o operador é um dos símbolos apresentados na tabela acima e os operandos são expressões lógicas.

Assim como para as expressões aritméticas, para expressões lógicas com mais de um operador, seguiremos regras de precedência para a ordem em que as operações devem ser feitas.

O operador de negação é o operador de maior prioridade, seguido pelo operador de conjunção e por último pelo operador de disjunção.

Exemplo 5.8 *Expressões lógicas em REC++:*
$x = y < 10$ && $y > 5$;
$y = (z\ != x)\ ||\ (x == w)$;

5.3 Exemplos

Agora que temos todos os elementos das linguagens *REC* e *REC++*, podemos dar alguns exemplos de programas que podem ser feitos nessas linguagens.

Na realidade, nos nossos exemplos nos limitaremos a usar a linguagem *REC++* devido a sua versatilidade, porém todos os exemplos poderiam ser reescritos em *REC*.

Estes exemplos também serão úteis para visualizar como o uso de certas estruturas que apresentamos na linguagem *REC*, como o uso de vetores e de estruturas de seleção, ficam mais fáceis de uso e compreensão na linguagem *REC++*.

Alguns exemplos estarão com o algoritmo completo, incluindo a função principal, em outros colocaremos apenas as funções auxiliares responsáveis pelos cálculos, sem nos preocupar com a função principal, que ficaria responsável apenas pela interface com o usuário e a chamada da função.

5.3.1 X é divisível por Y

```
ehDiv(x, y)
{
    if (x%y == 0)
        return 1;
    return 0;
}
```

5.3.2 Quantidades de divisores de X

Neste exemplo, usaremos a função *'ehDiv(x,y)'* definida anteriormente.

```
qtdeDivAux(x, rec div)
{
    if (div == 0)
        return 0;

    div = div - 1;
    qtdeDivAux = qtdeDivAux(x, div);

    if (ehDiv(x, div+1))
        return 1 + qtdeDivAux;
    return qtdeDivAux;
}

qtdeDivisores(x)
{
    return qtdeDivAux(x, x);
}
```

5.3.3 MMC

Neste exemplo, usaremos a função *'ehDiv(x, y)'* definida anteriormente para criar uma função que determine o mínimo múltiplo comum entre 2 ou n números.

```
somaDivs(x[], num, rec n)
{
    if (n == 0)
        return 0;

    n = n - 1;
    somaDivs = somaDivs(x, num, n);

    return ehDiv(num, x[n+1]) + somaDivs;
}
```

```
mmc(x[], n)
{
    var res;
    res << somaDivs(x, res,n) == n;
    return res;
}

mmc(x, y)
{
    var res;
    res << ehDiv(res, x) && ehDiv(res, y);
    return res;
}
```

5.3.4 MDC

Neste exemplo, usaremos a função *'ehDiv(x, y)'* definida anteriormente para criar uma função que retorne o máximo divisor comum de dois números.

```
mdc_aux(x, b, rec div)
{
    if (div == 0)
        return 0;

    div = div - 1;
    mdc_aux = mdc_aux(x, b, div);

    if (ehDiv(a,div+1) && ehDiv(b, div+1))
        return div + 1;
    return mdc_aux;
}

mdc(x, y)
{
    if (x > y)
        return mdc_aux(x, y, y);
    return mdc_aux(x, y, x);
}
```

5.3.5 Teste de primalidade

Neste exemplo, usaremos a função *'qtdeDivisores(x)'* definida anteriormente para criar uma função que retorne 1 se o número recebido como parâmetro for primo e 0 caso contrário.

```
primo(x)
{
```

```
    return (qtdeDivisores(x) == 2);
}
```

5.3.6 I-ésimo primo

Neste exemplo, usaremos as funções *'primo(x)'* e *'fatorial(x)'* que devem estar previamente definidas para criar uma função que retorne o i-ésimo número primo.

```
p(rec i)
{
    var y;
    if (i == 0)
        return 2;

    i = i - 1;
    p = p(i);

    y << p < y && primo(y) , fatorial(p) + 1;
    return y;
}
```

5.3.7 I-ésimo expoente de uma fatoração

Neste exemplo, usaremos as funções *'pow(x,y)'*, *'p(i)'* e *'ehDiv(x,y)'* que devem estar previamente definidas para criar uma função que, dada a decomposição de um número x em fatores primos, retorne o expoente do i-ésimo primo dessa fatoração.

```
i_exp(x,i)
{
    var y;
    y << ( ehDiv( x, pow(p(i), y) ) && !ehDiv( x , pow(p(i), y+1) ) ),x;
    return y;
}
```

5.3.8 Fibonacci

Neste exemplo, usaremos as funções *'pow(x,y)'*,*'p(i)'* e *'i_exp(x,i)'* definidas anteriormente para criar uma função que retorne o n-ésimo termo de Fibonacci, que podemos definir intuitivamente pela seguinte função:

$$f(n) = \begin{cases} 1 & se\ n = 0 \\ 2 & se\ n = 1 \\ f(n-1) + f(n-2) & se\ n >= 2 \end{cases}$$

Em casos como este, não podemos usar a operação de recursão primitiva diretamente pois o valor de $f(n+1)$ não depende apenas de $f(n)$, mas também de $f(n-1)$ ou, no caso de outras funções, até mesmo de outros valores $f(s)$ em que $s < n$.

Para resolver isso, usaremos a técnica de recursão por curso de valores, em que definimos uma função auxiliar que codifica os valores $f(0)$, ..., $f(s)$, para $s <= n$. Claro que isso também é válido para qualquer função $f(x_1,...,x_n,y)$, $n \geq 0$ em que $f(x_1,...,x_n,y+1)$ dependa de alguns ou todos os valores $f(x_1,...,x_n,s)$, para $s <= y$.

Mais detalhes sobre esta técnica e como ela funciona podem ser encontradas em Dias e Weber (2010) e Carnielli e Epstein (2006).

```
h(x,y)
{
    if (x == 0)
        return 1;
    else if (x == 1)
        return 2;
    return i_exp(y,x-1) + i_exp(y,x-2);
}

fibC(rec n)
{
    var a;
    if (n == 0)
        return 1;

    n = n - 1;
    fibC = fibC(n);

    a = h(n,fibC);
    a = pow(p(n),a);
    return a*fibC;
}

fib(n)
{
    return h(n,fibC(n));
}
```

6 Considerações finais

O objetivo inicial deste projeto era construir uma pseudolinguagem de programação que possibilitasse criar uma ponte entre conceitos matemáticos importantes associados ao estudo da Computabilidade, usando o modelo das funções $\mu-$recursivas; e a programação em si, a qual os estudantes dos cursos de Bacharelado em Ciência da Computação estão mais acostumados.

Dessa forma, teríamos uma ferramenta mais simples e intuitiva para esses estudantes, ao invés de trabalhar pura e simplesmente com as funções $\mu-$recursivas, com notações matemáticas que são muitas vezes variáveis de autor para autor e que podem acabar gerando algumas dúvidas. Mas, ao mesmo tempo, como a linguagem proposta implementaria apenas as operações e funções iniciais das funções recursivas, para qualquer função que o estudante fosse desenvolver nessa

linguagem, ele teria que se debruçar sobre os mesmos conceitos matemáticas necessários caso estivesse trabalhando com as funções $\mu-$recursivas, apenas com uma linguagem facilitada.

Tal proposta foi cumprida e, ao fim, construímos não apenas uma, mas duas linguagens. A primeira, a linguagem *REC*, implementa apenas as funções iniciais e as operações básicas das funções $\mu-$recursivas, adicionando apenas a estrutura de seleção *'if-else'*, mas do contrário essa linguagem acabaria perdendo o intuito de simplificar a notação e acabaríamos com expressões bastante complexas e de difícil leitura.

Com foco na facilidade de construção de algoritmos, criamos também a linguagem *REC*++, que implementa também outras operações, como a soma e a multiplicação. Todavia, foi demonstrado antes que cada uma delas é caso de função $\mu-$recursiva, e só então foram implementadas. Ainda assim, justificamos que são também funções *REC*-calculáveis.

Foi verificado que essas linguagens são equivalentes às funções $\mu-$recursivas durante todo o desenvolvimento do trabalho, pois cada estrutura dessas linguagens veio diretamente dos conceitos relacionados a funções $\mu-$recursivas, e sempre com as devidas demonstrações.

E quanto a ideia de ainda estimular o estudante a se debruçar sobre certos conceitos matemáticos para poder desenvolver algoritmos nessas linguagens, podemos perceber que foi satisfeita, quando vemos exemplos como o da série de Fibonacci e na conversão de um número decimal para binário. Tais funções poderiam ser feitas em poucas linhas em qualquer linguagem de programação convencional, porém em *REC* ou *REC*++ ambas só poderiam ser feitas se o estudante dominasse conceitos como o de codificação e de recursão por curso de valores.

7 Agradecimentos

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), processo nº 2017/05030-6, pela bolsa fornecida a esse trabalho.

8 Referências bibliográficas

CARNIELLI, W.; EPSTEIN, R. L. **Computabilidade, funções computáveis, lógica e os fundamentos da matemática**. São Paulo: Editora Unesp, 2006.

CUTLAND, N. J. **Computability:** an introduction to recursive function theory. Cambridge: Cambridge University Press, 1980.

DIAS, M. F.; WEBER, L. **Teoria da recursão**. São Paulo: Editora Unesp, 2010.

FORTE, A.; GUZDIAL, M. Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses. **IEEE Transaction on Education**, v. 48, n. 2, p. 248-253, 2005.

GUZDIAL, M. A media computation course for non-majors. In: ANUAL CONFERENCE ON INNOVATION AND TECHNOLOGY IN COMPUTER SCIENCE EDUCATION, 8., 2003, Thessaloniki, **Proceedings...** Thessaloniki: [s.n.], 2003. v. 35, n. 3, p. 104-108.

LIMA JUNIOR, J. A. T. de; VIEIRA, C. E. C.; VIEIRA, P. de P. Dificuldades no processo de aprendizagem de algoritmos: uma análise dos resultados na disciplina de AL1 do Curso de Sistemas de Informação da FAETERJ - Campus Paracambi. **Cadernos UniFOA**, v. 10, n. 27, p. 5-15, 2015.

MENDELSON, E. **Introduction to mathematical logic**. New York: Van Nostrand Reinhold Company, 1964.

MICHAELIS. **Dicionário Brasileiro da Língua Portuguesa**. São Paulo: Editora Melhoramentos, 2007.

PAIOLA, P. H.; FEITOSA, H. A. Pseudolinguagem para caracterizar funções recursivas. In: ENCONTRO REGIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, 5., 2018, Bauru. **Caderno de trabalhos completos e resumos**... Bauru: Unesp, Faculdade de Ciências, 2018. p. 243-249. Disponível em: <http://www.fc.unesp.br/index.php#!/departamentos/matematica/eventos2341/ermac-2018/caderno-de-trabalhos-e-resumos/>. Acesso em: 6 ago. 2018.

SANTIAGO, R. H. N.; BEDREGAL, B. R. C. **Computabilidade:** os limites da computação. São Paulo: SBMAC, 2004.