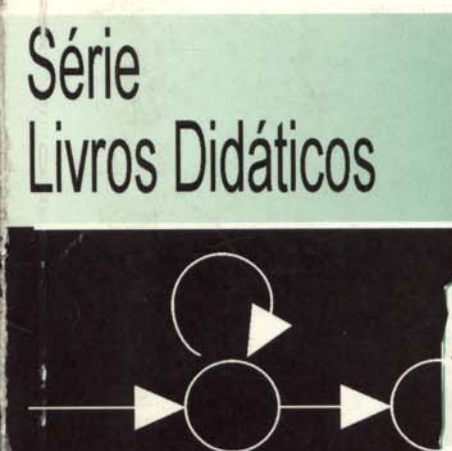
Teoria da Computação

Máquinas Universais e Computabilidade

Série Livros Didáticos

Número 5



Instituto de Informática da UFRGS

Editora Sagra Luzzatto

Tiarajú Asmuz Diverio
Paulo Blauth Menezes

Teoria da Computação

Máquinas Universais e Computabilidade

SÉRIE LIVROS DIDÁTICOS
Instituto de Informática
Universidade Federal do Rio Grande do Sul

Diretor
Prof. Philippe Olivier Alexandre Navaux

Vice-Diretor
Prof. Otacílio José Carollo de Souza

Comissão Editorial
Prof. Tiarajú Asmuz Diverio
Prof. Clesio Saraiva dos Santos
Prof. Ricardo Augusto da Luz Reis
Profª. Carla Maria Dal Sasso Freitas

Endereço
UFRGS – Instituto de Informática
Av. Bento Gonçalves, 9500 – Bloco IV – Bairro Agronomia
Caixa Postal 15064 – 91501-970 – Porto Alegre, RS
Telefone: 00 55 (051) 316 6165
Telefax: 00 55 (051) 319 1576
e-mail: informat@inf.ufrgs.br
http://www.inf.ufrgs.br

Visite a página da Editora Sagra Luzzatto na internet
http://www.sagra-luzzatto.com.br
atendimento@sagra-luzzatto.com.br

Tiarajú Asmuz Diverio

Doutorado em Ciência da Computação e Mestre em Ciência da Computação
junto ao Programa de Pós-Graduação em Computação da UFRGS.
Licenciado em Matemática pela UFRGS. Professor Adjunto e Pesquisador
do Departamento de Informática Teórica e Professor Orientador
Habilitado de Mestrado e Doutorado do PPGC da UFRGS

Paulo Blauth Menezes

Doutor em Matemática pelo IST/Universidade Técnica de Lisboa, Portugal.
Mestre em Ciência da Computação e Licenciado em Matemática pela UFRGS.
Professor Adjunto, Pesquisador e Chefe do Departamento
de Informática Teórica do Instituto de Informática da UFRGS.

Teoria da Computação

Máquinas Universais e Computabilidade

Série Livros Didáticos
Instituto de Informática da UFRGS
Número 5

Editora Sagra Luzzatto

1ª edição, 1999

Editora Sagra Luzzatto
Rua João Alfredo, 448 – Cidade Baixa
90050-230 – Porto Alegre, RS – Brasil
Fone (051) 227-5222 – Fax (051) 227-4438
www.sagra-luzzatto.com.br
atendimento@sagra-luzzatto.com.br

Editor: Darcy Caetano Luzzatto
Supervisão editorial: Elisa Schein Wenzel Luzzatto
Capa: Carlos Alberto Gravina
Ilustrações: Maria Lúcia Recena Menezes

Dados Internacionais de Catalogação na Publicação (CIP)
(Biblioteca do Instituto de Informática da UFRGS, Porto Alegre, RS)

Diverio, Tiarajú Asmuz
Teoria da computação: máquinas universais e computabilidade / Tiarajú Asmuz Diverio, Paulo Blauth Menezes. -- Porto Alegre : Instituto de Informática da UFRGS : Editora Sagra Luzzatto, 1999.

ISBN 85-241-0593-3
205 p.: il. (Série Livros Didáticos, n. 5)

1. Teoria da computação. 2. Informática teórica. 3. Matemática da computação. I. Menezes, Paulo Blauth. II. Título. III. Série.

Prefácio de Série

A série Livros Didáticos do Instituto de Informática da Universidade Federal do Rio Grande do Sul, é inspirada na idéia de desenvolver material didático para disciplinas ministradas no Bacharelado em Ciência da Computação e Bacharelado em Engenharia da Computação dessa universidade. Este material é resultante da experiência dos seus professores no ensino e na pesquisa.

Em seus primeiros volumes, a série era voltada para Matemática da Computação e Processamento Paralelo. Foram publicados três títulos: Fundamentos da Matemática Intervalar, Programando em Pascal XSC (estes dois primeiros títulos, foram resultado de pesquisas desenvolvidas dentro do Projeto ArInPar - Aritmética Intervalar Paralela, financiado pelo ProTeM-CC CNPq Fase II). O terceiro, Linguagens Formais e Autômatos foi o primeiro da série que voltado para o objetivo de suprir livros-texto para as disciplinas básicas dos cursos de Bacharelado em Ciência da Computação (ou Informática). O conteúdo desses livros é baseado no programa das disciplinas do Bacharelado em Ciência da Computação da UFRGS, sendo adotado, também, por diversas Universidades de todo o Brasil.

O sucesso da experiência com esses livros, bem como a responsabilidade que cabe ao Instituto de Informática na formação de professores e pesquisadores em Computação, conduziu à ampliação da abrangência e à institucionalização da série, que passa a ser de Livros Didáticos do Instituto de Informática.

Neste novo enfoque, foi publicada a segunda edição dos livros Linguagens Formais e Autômatos e Projeto de Banco de Dados de autoria, respectivamente, de Paulo Blauth Menezes e Carlos Alberto Heuser. Dando continuidade à série, estão sendo apresentados novos títulos: Teoria da Computação: Máquinas Universais e Computabilidade, de Tiarajú Asmuz Diverio e Paulo Blauth Menezes (quinto número) e Fundamentos de Arquiteturas de Computadores, de Raul Fernando Weber, todos professores do Instituto de Informática da UFRGS.

Outros títulos já se encontram em preparação, abordando assuntos tratados em outras disciplinas típicas de Ciência da Computação ou Informática. Todos os livros têm em comum a preocupação em manter nível compatível com a elevada qualidade do ensino e da pesquisa desenvolvidos no âmbito da UFRGS.

Comissão Editorial da Série Livros Didáticos

Instituto de Informática da UFRGS

Maio de 1999.

Este livro é dedicado:

à memória de Wanner Diverio, meu pai,

Tiaraju.

à Maria Fernanda, Maria Lúcia e Maria Luiza Menezes,

Paulo.

Prefácio dos Autores

"*Teoria da Computação: Máquinas Universais e Computabilidade*" pretende servir como um livro-texto para disciplinas dos cursos de Bacharelado em Ciência da Computação ou em Informática. O enfoque adotado neste livro não é histórico (cronológico) mas didático, visando a construção dos conceitos de Teoria da Computação.

São abordados os principais aspectos relativos à Teoria da Computação de forma sistematizada e acessível, fornecendo meios para um correto entendimento e aplicação dos conceitos de procedimento efetivo, computabilidade e solucionabilidade de problemas.

Trata-se de um trabalho baseado na experiência docente em diversos semestres no Curso de Bacharelado em Ciência da Computação da Universidade Federal do Rio Grande do Sul. É destinado, principalmente, para um primeiro curso de Teoria da Computação, sendo auto-contido e podendo ser adotado como bibliografia básica. Possui um texto simples, exemplos detalhados e exercícios em níveis crescentes de raciocínio. Recomenda-se, como pré-requisitos, conhecimentos básicos de lógica, teoria dos conjuntos e algoritmos.

A carga horária total recomendada é de 60 horas/aula. Este livro fornece, também, a base para uma disciplina de Linguagens Formais. A notação e o formalismo adotado são consistentes com o do livro "*Linguagens Formais e Autômatos*" de Paulo Blauth Menezes, também publicado na série de Livros Didáticos do Instituto de Informática da UFRGS.

O primeiro capítulo inicia com uma breve revisão histórica do surgimento e desenvolvimento dos conceitos básicos que serão utilizados nos demais capítulos.

Logo a seguir, no capítulo dois, é construído o conceito de equivalência de programas. Para tanto, é necessário que se formalize os conceitos de programas e de máquina, para que, então, sejam expostos os conceitos de computação e funções computáveis.

No terceiro capítulo é introduzido o conceito de Máquina Universal, para fundamentar o estudo de modelos como: Máquina de Turing, Máquina de Registradores Norma, Máquina de Post e Máquinas com Pilhas. Nesse contexto são introduzidos formalismos equivalentes e modificações sobre as máquinas universais, como o não-determinismo, mas que não alteram a classe de funções computadas. Aborda-se, ainda, processamento de funções e reconhecimento de linguagens. Por fim, apresenta-se a Hipótese de Church.

O quarto capítulo apresenta as funções recursivas. Aborda-se as funções recursivas de Kleene, onde a composição de três funções naturais simples juntamente com a recursão e minimização se constitui numa forma compacta e natural para definir muitas funções e suficientemente poderosa para descrever

qualquer função computável. Descreve-se, também, a classe das funções numéricas computáveis definidas recursivamente e a questão das formas de avaliação de funções por valor ou por variável.

O quinto capítulo trata da computabilidade e seus limites através do estudo da solucionabilidade de problemas, onde são definidas as classes de solucionabilidade e suas propriedades. São descritos alguns problemas típicos, como o Problema da Parada e o da Correspondência de Post.

Foram desenvolvidos simuladores, programas que emulam algumas das máquinas abstratas descritas neste livro. Eles são úteis para: ensino, demonstrações do funcionamento das máquinas, elaboração de exercícios e correção dos mesmos. Esses simuladores, juntamente com material de apoio ao professor e ao aluno, encontram-se disponíveis na página do Grupo de Matemática da Computação da UFRGS, no seguinte endereço da WEB: http://www.inf.ufrgs.br/~hgmc/ (ou por e-mail {diverio, blauth}@inf.ufrgs.br).

Os autores agradecem:

- aos colegas do Departamento de Informática Teórica do Instituto de Informática da Universidade Federal do Rio Grande do Sul, onde este livro foi elaborado, pelas sugestões dadas, especialmente aos professores Laira Vieira Toscani, Leila Ribeiro Korff e Vanderlei Moraes Rodrigues;
- aos bolsistas Caroline Carbonel Cintra, Thiago Moesch, Ingrid de Vargas Mito e Aline Vieira Malanovicz pelo trabalho de edição e correção do texto;
- à Maria Lúcia Recena Menezes pelas figuras e ilustrações;
- aos alunos das turmas de Teoria da Computação do Curso de Bacharelado em Ciência da Computação que colaboraram na revisão do livro e pelas sugestões dadas, especialmente os da turma do primeiro semestre de 1998, pelo trabalho de correção dos exercícios;
- à FAPERGS e ao CNPq, que têm financiado, através de bolsas de pesquisa (Iniciação Científica), os alunos que colaboram no Grupo de Matemática da Computação (GMC) da UFRGS;
- ao CNPq, a CAPES e a FAPERGS pelos auxílios financeiros aos projetos de pesquisa que viabilizaram a realização deste e outros trabalhos;
- à Viviane e à Maria Fernanda juntamente com nossos familiares. Aos nossos amigos e colegas.

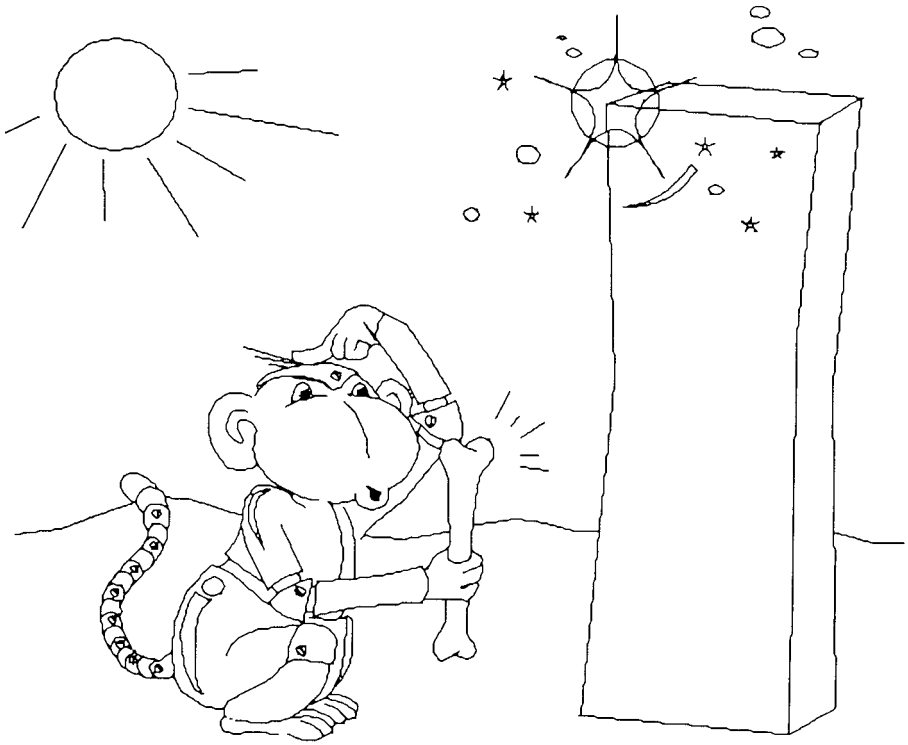
Um agradecimento especial ao Instituto de Informática da UFRGS que tem apoiado esta série de *Livros Didáticos*, e a Pró-Reitoria de Pesquisa da UFRGS, que sempre tem apoiado nossos projetos e, em especial, apoiou financeiramente a publicação deste livro.

Tiarajú Asmuz Diverio

Paulo Blauth Menezes

Maio de 1999

Sumário



"Há um teorema conhecido que diz que qualquer computador é capaz de emular qualquer outro computador"

Astronauta Frank Poole ao
explicar o princípio usado por Halman (computador HAL/astronauta Bowman)
para impedir o Monólito de executar qualquer ordem que ameaçasse a humanidade

Do livro 3001 - A Odisséia Final da Série iniciada por 2001 - Uma Odisséia no Espaço
Arthur C. Clarke

1 Introdução e Conceitos Básicos

Este capítulo inicia com uma breve história do surgimento e do desenvolvimento dos conceitos, resultados e formalismos nos quais a Teoria da Computação é baseada. A seguir, é apresentada a abordagem geral adotada nesta publicação. Por fim, são introduzidos alguns conceitos básicos que são usados ao longo de todo o texto.

1.1 Notas Históricas

Ciência da Computação é o conhecimento sistematizado relativo à computação. Sua origem é remota, tendo exemplos na antiga Grécia (século III A.C., no desenho de algoritmos por Euclides) e Babilônia (com estudos sobre complexidade e reducibilidade de problemas). O interesse atual possui duas ênfases: idéias fundamentais e modelos computacionais (ênfase teórica) e projeto de sistemas computacionais (ênfase prática), aplicando a teoria à prática.

A ênfase teórica da Ciência da Computação teve o seu início em uma grande diversidade de campos como, por exemplo, na biologia (modelos para redes de neurônios), na eletrônica (teoria do chaveamento), na matemática (lógica), na lingüística (gramáticas para linguagens naturais) e em outros. A partir destes estudos, surgiram os modelos que são hoje a base da Teoria da Computação.

No início do século XX, diversas pesquisas foram desenvolvidas com o objetivo de definir um modelo computacional suficientemente genérico, capaz de implementar qualquer *Função Computável*.

Um marco inicial da Teoria da Computação é o trabalho de David Hilbert, denominado de *Entscheidungsproblem* ([HIL1900]), o qual consistia em encontrar um procedimento para demonstrar se uma dada fórmula no cálculo de predicados de primeira ordem (onde a quantificação é restrita às variáveis que denotam elementos de conjuntos) era válida ou não. A procura de Hilbert por tal procedimento se justifica pelo fato de que se acreditava que todo problema bem definido poderia ser resolvido, possivelmente pela demonstração de falsidade. Admitindo-se que essa tese fosse verdadeira, o fracasso na resolução de um problema seria devido à escolha insuficiente de hipóteses ou a um raciocínio errôneo.

Entretanto, em 1931, Kurt Gödel publicou o trabalho denominado de *Incompleteness Theorem* (Teorema da Não-Completude), onde demonstrou que tal problema (mecanização do processo de prova de teoremas) não tinha solução ([GOD65]). Provou que um determinado sistema formal bem definido e consistente (isto é, onde não existe a possibilidade de "se A, então não A"), o qual define a multiplicação e adição no conjunto dos números naturais, não era suficiente para provar se toda a sentença nesse sistema é ou não um teorema.

Uma característica importante do trabalho de Gödel é o uso feito por ele de números naturais para codificar símbolos, fórmulas e seqüências de fórmulas. Na prova do *Incompleteness Theorem*, afirmações sobre fórmulas podiam ser representadas como determinadas classes de funções. Dessa forma, notou-se a correspondência entre a computabilidade efetiva de funções e a existência de procedimentos efetivos para a solução de problemas. A classe de funções usada por Gödel foi a das funções primitivas recursivas definidas anteriormente por Dedekind em 1888 ([DED1888]). Embora não tenha sido proposital, Gödel foi

(aparentemente) o primeiro a identificar um formalismo para definir a noção de *Procedimento Efetivo*.

Em 1936, Alonzo Church usou dois formalismos para mostrar que o problema de Hilbert não tem solução:

- Cálculo Lambda (Church, 1936) ([CHU36]);
- Funções Recursivas (Kleene, 1936) ([KLE56]).

De fato, a equivalência de ambos os formalismos foi verificada por Kleene (1936). Este fato levou Church a sugerir o que é conhecida como a *Hipótese de Church*:

> *"São caracterizações tão gerais da noção do efetivamente computável quanto consistentes com o entendimento intuitivo usual" (Church).*

A Hipótese de Church é assumida como verdadeira em todos os estudos relacionados com a Teoria da Computação. Note-se que não é demonstrável pois é fundamentada em uma noção intuitiva (não-formal) do que é efetivamente computável

Separadamente de Church, Alan Turing propôs, em 1936 ([TUR36]), um formalismo para a representação de procedimentos efetivos. O trabalho de Turing é particularmente significativo por ter sido o primeiro a identificar programas escritos para uma "máquina computacional", como noções intuitivas de efetivamente computável. A intenção do modelo de Turing, denominado *Máquina de Turing*, foi simular, tanto quanto possível, as atitudes humanas relacionadas à computação.

Desde então, muitos outros formalismos foram propostos, os quais, são provados possuírem (no máximo) o mesmo poder computacional das Funções Recursivas (ou o Cálculo Lambda) como, por exemplo:

- Máquina de Turing (1936);
- Máquina Norma (1976)
- Sistema Canônico de Post (1943);
- Algoritmo de Markov e a Linguagem Snobol (1954);
- Máquina de Registradores (1963);
- RASP (*Random Access Stored Programs* - 1964).

Assim, define-se programa como sendo um procedimento efetivo e, portanto, que pode ser descrito usando qualquer dos formalismos equivalentes como os citados acima. Ou seja, qualquer destes formalismos permite descrever todos os procedimentos possíveis que podem ser executados em um computador.

1.2 Abordagem

A abordagem desta publicação desenvolve os principais aspectos de Teoria da Computação combinando abordagens históricas com abordagens próximas dos sistemas computadores modernos. O objetivo é permitir um fácil entendimento e associação dos problemas abstratos com os problemas típicos da Ciência da Computação atual. Assim, por exemplo, questões como programas e máquinas ficam claramente caracterizadas e diferenciadas e são adequadamente tratadas no contexto de Máquinas Universais, juntamente com modelos tradicionais como a Máquina de Turing. Parte desta abordagem voltada para os sistemas computadores atuais é inspirada, entre outros, no trabalho de Richard Bird em *Programs and Machines - An Introduction to the Theory of Computation* ([BIR76]).

1.3 Conceitos Básicos

Linguagem é um conceito fundamental no estudo da Teoria da Computação, pois trata-se de uma forma precisa de expressar problemas, permitindo um desenvolvimento formal adequado ao estudo da computabilidade. Mais precisamente, em capítulos subseqüentes, será estudada a solucionabilidade de um problema, analisando-a como a investigação da existência de um algoritmo que determine se uma palavra pertence ou não à linguagem que traduz esse problema.

O dicionário Aurélio ([FER84]) define linguagem como:

> "*o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas*".

Entretanto, esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria baseada em linguagens. Assim, serão feitas, a seguir, algumas definições formais, necessárias aos estudos posteriores.

As definições que seguem são construídas usando como base a noção de *Símbolo* ou *Caractere*. Portanto, é uma entidade abstrata básica, não sendo definida formalmente. Letras e dígitos são exemplos de símbolos freqüentemente usados.

Definição 1.1 Alfabeto.

Um *Alfabeto* é um conjunto finito de *símbolos* ou *caracteres*. ❑

Portanto:

- um conjunto infinito *não* é um alfabeto;
- o conjunto vazio *é* um alfabeto.

EXEMPLO 1.1 Alfabeto.

a) Os seguintes conjuntos *são* exemplos de alfabetos:

{a, b, c}

∅ (conjunto vazio)

b) Os seguintes conjuntos *não* são exemplos de alfabetos:

$\mathbb{N}$ (conjunto dos números naturais)

{a, b, aa, ab, ba, bb, aaa,...} ❑

Definição 1.2 Cadeia de Símbolos, Palavra.

a) Uma *Cadeia de Símbolos* sobre um conjunto é uma seqüência de zero ou mais símbolos (do conjunto) justapostos;

b) Uma *Palavra* é uma cadeia de símbolos finita. ❑

Portanto, uma cadeia sem símbolos é uma palavra válida e o símbolo:

ε denota a *cadeia vazia* ou *palavra vazia*.

Se Σ representa um alfabeto, então:

Σ^* denota o conjunto de todas as palavras possíveis sobre Σ

Σ^+ denota $\Sigma^* - \{\varepsilon\}$

Definição 1.3 Comprimento ou Tamanho de uma Palavra.

O *Comprimento* ou *Tamanho* de uma palavra w, representado por $|w|$, é o número de símbolos que compõem a palavra. ❑

Definição 1.4 Prefixo, Sufixo, Subpalavra.

Um *Prefixo* (respectivamente, *Sufixo*) de uma palavra é qualquer seqüência inicial (respectivamente, final) de símbolos da palavra. Uma *Subpalavra* de uma palavra é qualquer seqüência de símbolos contígua da palavra. ❑

EXEMPLO 1.2 Palavra, Prefixo, Sufixo, Tamanho.

a) abcb é uma palavra sobre o alfabeto {a, b, c}

b) Se Σ = {a, b}, então:

Σ^+ = {a, b, aa, ab, ba, bb, aaa,...}

Σ^* = {ε, a, b, aa, ab, ba, bb, aaa,...}

c) $|abcb| = 4$ e $|\varepsilon| = 0$

d) Relativamente à palavra abcb, tem-se que:

ε, a, ab, abc, abcb são os prefixos;

ε, b, cb, bcb, abcb são os respectivos sufixos;

e) Qualquer prefixo ou sufixo de uma palavra é uma subpalavra. ❑

Definição 1.5 Linguagem Formal.

Uma *Linguagem Formal* ou simplesmente *Linguagem* é um conjunto de palavras sobre um alfabeto. ❑

EXEMPLO 1.3 Linguagem Formal.

Suponha o alfabeto $\Sigma = \{a, b\}$. Então:

a) O conjunto vazio e o conjunto formado pela palavra vazia são linguagens sobre Σ. Obviamente, $\emptyset \neq \{\varepsilon\}$;

b) O conjunto de palíndromos (palavras que têm a mesma leitura da esquerda para a direita e vice-versa) sobre Σ é um exemplo de linguagem infinita. Assim:

$$\varepsilon, a, b, aa, bb, aaa, aba, bab, bbb, aaaa,...$$

são palavras desta linguagem. ❑

Definição 1.6 Concatenação de Palavras.

A *Concatenação de Palavras* ou simplesmente *Concatenação* é uma operação binária, definida sobre uma linguagem, a qual associa a cada par de palavras uma palavra formada pela justaposição da primeira com a segunda.

Uma concatenação é denotada pela justaposição dos símbolos que representam as palavras componentes. A operação de concatenação satisfaz às seguintes propriedades (suponha v, w, t palavras):

a) *Associatividade.*

$v(wt) = (vw)t$

b) *Elemento Neutro à Esquerda e à Direita.*

$\varepsilon w = w = w\varepsilon$ ❑

Como a concatenação de palavras é uma operação associativa, é usual omitir os parênteses. Assim, v(wt) ou (vw)t pode ser denotado simplesmente por vwt. Uma operação de concatenação definida sobre uma linguagem L não é, necessariamente, fechada sobre L, ou seja, a concatenação de duas palavras de L não é, necessariamente, uma palavra de L.

EXEMPLO 1.4 Concatenação de Palavras.

a) Suponha o alfabeto $\Sigma = \{a, b\}$. Então, para as palavras v = baaaa e w = bb, tem-se que:

$vw = baaaabb$

$v\varepsilon = v = baaaa$

b) Suponha a linguagem L de palíndromos sobre $\Sigma = \{a, b\}$. A concatenação das palavras aba e bbb resulta na palavra ababbb a qual não é palíndromo. Portanto, a operação de concatenação não é fechada sobre L. ❑

Definição 1.7 Concatenação Sucessiva de uma Palavra.

A *Concatenação Sucessiva de uma Palavra* (com ela mesma) ou simplesmente *Concatenação Sucessiva*, representada na forma de um expoente (suponha w uma palavra):

w^n onde n é o número de concatenações sucessivas

é definida indutivamente a partir da operação de concatenação binária, como segue:

$w^0 = \varepsilon$

$w^n = w\,w^{n-1}$, para $n > 0$ ❑

EXEMPLO 1.5 Concatenação Sucessiva.

Sejam w uma palavra e a um símbolo. Então:

$w^3 = www$

$w^1 = w$

$a^5 = aaaaa$

$a^n = aaa...a$ (o símbolo a repetido n vezes) ❑

1.4 Exercícios

Exercício 1.1 Elabore uma linha de tempo sobre o desenvolvimento do conceito de função computável.

Exercício 1.2 Qual o marco inicial da Teoria da Computação?

Exercício 1.3 Em que se consistia o problema de Hilbert – *Entscheidungsproblem* e por que ele é sem solução?

Exercício 1.4 Qual a importância da Hipótese de Church e por que ela não é demonstrável?

Exercício 1.5 Marque os conjuntos que são alfabetos:

a) Conjunto dos números naturais [ ]

b) Conjunto dos números primos [ ]

c) Conjunto das letras do alfabeto brasileiro [ ]

d) Conjunto dos algarismos arábicos [ ]

e) Conjunto dos algarismos romanos [ ]

f) Conjunto {a, b, c, d} [ ]

g) Conjunto das partes de {a, b, c} [ ]

h) Conjunto das vogais [ ]

i) Conjunto das letras gregas [ ]

Exercício 1.6 Dê os possíveis prefixos e sufixos de cada uma das seguintes palavras:

a) teoria

b) universidade

c) aaa

d) abccba

e) abcabc

Exercício 1.7 Exemplifique, comprovando ou negando as seguintes propriedades algébricas da operação de concatenação de palavras:

a) Fechamento

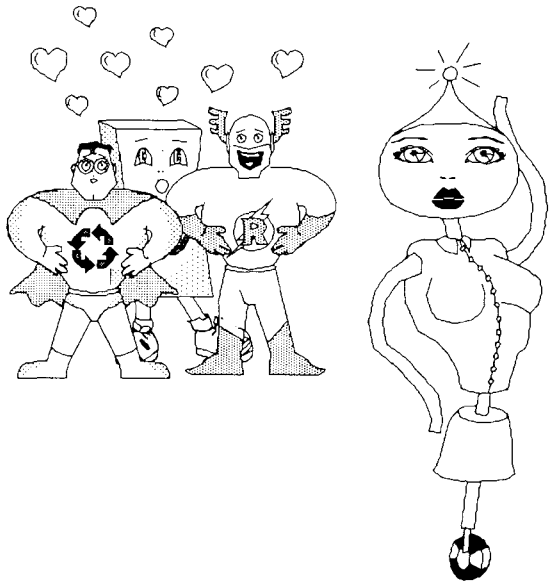
b) Comutatividade

c) Elemento neutro

d) Associatividade

e) Elemento inverso

Exercício 1.8 Quando se pode dizer que a estrutura algébrica da operação de concatenação sobre uma linguagem é análoga a estrutura da operação de adição nos naturais?



2 Programas, Máquinas e Computações

Neste capítulo é introduzida a formalização das noções de programa, de máquina, de computação e do que é computável em uma máquina e de relações entre formalismos como, equivalência e simulação.

Considerando que diferentes computadores podem ter diferentes arquiteturas e que os diversos tipos de linguagens de programação aparecem em abundância, a formalização dos conceitos de programa e de máquina não são baseadas em qualquer linguagem ou computador real. Assim, suas características essenciais são descritas em modelos matemáticos simples, permitindo um rápido entendimento de suas semânticas e facilitando a demonstração de resultados.

Inicialmente, é introduzido o conceito de *programa* o qual pode ser visto como um conjunto de operações e testes compostos de acordo com uma estrutura de controle. O tipo de estrutura de controle associada determina uma classificação de programas como segue:

- *monolítico*: baseada em desvios condicionais e incondicionais;
- *iterativo*: possui estruturas de iteração de trechos de programas;
- *recursivo*: baseado em sub-rotinas recursivas.

A seguir, é introduzido o conceito de *máquina* a qual interpreta os programas de acordo com os dados fornecidos. Uma máquina é capaz de interpretar um programa desde que possua uma interpretação para cada operação ou teste que constitui o programa.

Para um dado programa e uma dada máquina (capaz de interpretar este programa) é possível definir computação e função computada. Uma *computação* é, resumidamente, um histórico do funcionamento da máquina para o programa, considerando um valor inicial. Uma *função computada* é uma função (parcial) induzida a partir da máquina e do programa dados, a qual é definida sempre que, para um dado valor de entrada, existe uma computação finita (a máquina pára).

Funções computadas permitem introduzir algumas importantes noções de equivalências de programas e máquinas como segue:

- *programas equivalentes fortemente*: se as correspondentes funções computadas coincidem para qualquer máquina;
- *programas equivalentes*: se as correspondentes funções computadas coincidem para uma dada máquina;
- *máquinas equivalentes*: se as máquinas podem simular umas às outras.

Relativamente aos programas equivalentes fortemente, verifica-se que programas recursivos são mais gerais que os monolíticos os quais, por sua vez, são mais gerais que os iterativos, induzindo uma hierarquia de tipos de programas. Adicionalmente, tem-se que:

- existe um algoritmo para determinar se dois programas monolíticos (respectivamente, iterativos) são ou não equivalentes fortemente;
- até o momento, não é conhecido se existe ou não um algoritmo para mostrar equivalência forte de dois programas recursivos.

2.1 Programas

Um *programa* pode ser descrito como um conjunto estruturado de *instruções* que capacitam uma máquina a aplicar sucessivamente certas *operações* básicas e *testes* sobre os dados iniciais fornecidos, com o objetivo de transformar estes dados numa forma desejável.

Portanto, um programa deve explicitar como as operações ou testes devem ser *compostos*, ou seja, um programa deve possuir uma *estrutura de controle* de operações e testes. Nas linguagens de programação atuais, existem várias formas de estruturação do controle, com destaque para as seguintes, as quais são formalizadas adiante, na forma de tipos de programas:

a) *Estruturação Monolítica*. É baseada em desvios condicionais e incondicionais, não possuindo mecanismos explícitos de iteração, subdivisão ou recursão.

b) *Estruturação Iterativa*. Possui mecanismos de controle de iterações de trechos de programas. Não permite desvios incondicionais.

c) *Estruturação Recursiva*. Possui mecanismos de estruturação em sub-rotinas recursivas. Recursão é uma forma indutiva de definir programas. Também não permite desvios incondicionais.

Independentemente da estruturação do controle, duas ou mais operações ou testes podem ser compostos como segue:

a) *Composição Seqüencial*. A execução da operação ou teste subseqüente somente pode ser realizada após o encerramento da execução da operação ou teste anterior.

b) *Composição Não-Determinista*. Uma das operações ou testes compostos é escolhido para ser executada. A composição não-determinista também é denominada de *escolha*.

c) *Composição Concorrente*. As operações ou testes compostos podem ser executados em qualquer ordem, inclusive simultaneamente. Ou seja, a ordem de execução é irrelevante.

A interpretação considerada para a composição não-determinista é aquela que objetiva explorar os limites da capacidade de solucionar problemas como em [HOP79] e [MEN98]. Ou seja, se existe uma escolha que resolve o problema (mesmo que as demais não tenham esta capacidade), então se afirma que o programa resultante da composição é capaz de resolver o problema.

Neste texto não são consideradas as composições concorrentes ou, mais precisamente, *composições concorrentes verdadeiras*. Entretanto, usando a composição não-determinista, é possível simular uma noção de concorrência verdadeira, denominada de *intercalação*. Na intercalação, uma concorrência é representada como composições não-deterministas de todas as combinações possíveis de composição seqüenciais, garantindo, portanto, que a ordem de execução é irrelevante. Uma boa referência sobre estes e outros aspectos referentes à concorrência é [WIN95].

Para o estudo de programas, não é necessário saber qual a natureza precisa das operações e dos testes os quais constituem as instruções. Eles são identificados pelos seus nomes. Portanto suponha que existam dois conjuntos de identificadores: de operações e de testes. Eles são descritos por:

a) *Identificadores de Operações*.

$$F, G, H, \ldots$$

b) *Identificadores de Testes*.

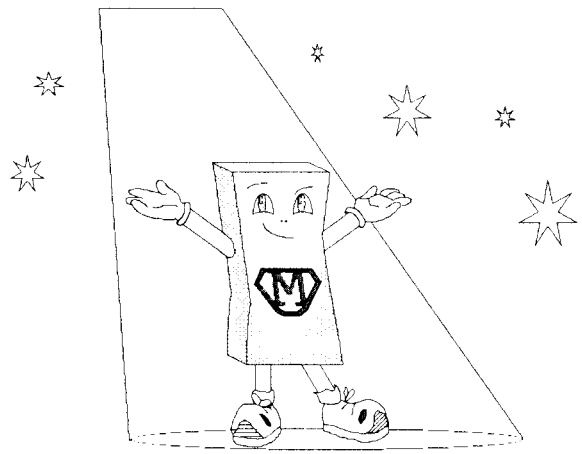
$$T_1, T_2, T_3, \ldots$$

Note-se que um teste é uma operação de um tipo especial a qual produz somente um dos dois possíveis valores verdade, ou seja *verdadeiro* ou *falso*,

usualmente denotados por v e f, respectivamente. Existe, ainda um tipo de operação que não faz coisa alguma, denominada:

operação vazia, denotada pelo símbolo ✔

No que segue, é suposto conhecimentos básicos de algoritmos.



2.1.1 Programa Monolítico

Um *programa monolítico* é estruturado usando desvios condicionais e incondicionais, não fazendo uso explícito de mecanismos auxiliares de programação que permitam uma melhor estruturação do controle como iteração, subdivisão ou recursão. Portanto, a lógica é distribuída por todo o bloco (monólito) que constitui o programa.

Uma das formas mais comuns e tradicionais de especificar programas monolíticos é através de *fluxogramas*. Informalmente, um fluxograma é um diagrama geométrico construído a partir de componentes (fluxogramas) elementares denominados partida, parada, operação e teste, introduzidos na Figura 2.1. No caso da operação vazia ✔, o retângulo correspondente à operação pode ser omitido, resultando simplesmente em uma seta.

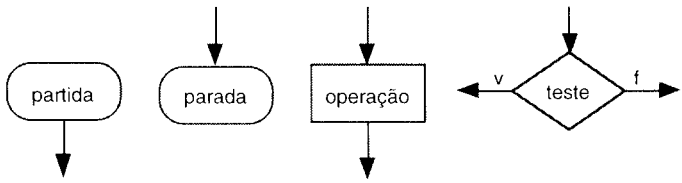


Figura 2.1 Componentes elementares de um fluxograma

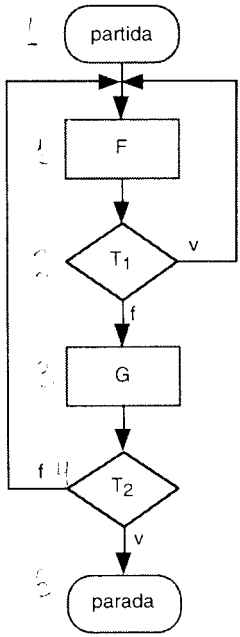


Figura 2.2 Fluxograma

EXEMPLO 2.1 Fluxograma.

Um exemplo de fluxograma é apresentado na Figura 2.2. Você é capaz de interpretar o fluxo de controle representado? ❑

Além da representação diagramática introduzida, um fluxograma pode ser denotado na forma de texto, usando *instruções rotuladas*. Como o próprio nome indica, cada instrução rotulada é identificada por um *rótulo*. Uma instrução rotulada pode ser como segue:

a) *Operação*. Indica a operação a ser executada seguida de um desvio incondicional para a instrução subseqüente.

b) *Teste*. Determina um desvio condicional, ou seja, que depende da avaliação de um teste.

No caso de um fluxograma denotado como um conjunto de instruções rotuladas, uma parada é especificada usando um desvio incondicional para um rótulo sem instrução correspondente.

EXEMPLO 2.2 Fluxograma como um Conjunto de Instruções Rotuladas.

O fluxograma da Figura 2.2 pode ser traduzido pelo conjunto de instruções rotuladas representado na Figura 2.3, supondo que a computação inicia pela instrução correspondente ao rótulo 1. Note-se que a parada é especificada usando um desvio para o rótulo 5. ❑

```
1:    faça F vá_para 2
2:    se T1 então vá_para 1 senão vá_para 3
3:    faça G vá_para 4
4:    se T2 então vá_para 5 senão vá_para 1
```

Figura 2.3 Instruções rotuladas

A definição formal de programa monolítico é melhor descrita usando a notação de instruções rotuladas do que diagramas geométricos. Inicialmente são introduzidas as definições de rótulo e instrução rotulada.

Definição 2.1 Rótulo, Instrução Rotulada.

a) Um *Rótulo* ou *Etiqueta* é uma cadeia de caracteres finita (palavra) constituída de letras ou dígitos.

b) Uma *Instrução Rotulada* `i` é uma cadeia de caracteres finita (palavra) de uma das duas formas a seguir (suponha que F e T são identificadores de operação e teste, respectivamente e que r_1, r_2 e r_3 são rótulos):

b.1) *Operação.*

`r1: faça F vá_para r2` ou `r1: faça ✔ vá_para r2`

b.2) *Teste.*

`r1: se T então vá_para r2 senão vá_para r3` ❑

EXEMPLO 2.3 Instrução Rotulada.

Na Figura 2.3, as cláusulas identificadas pelos rótulos `1`, `2`, `3` e `4` são instruções rotuladas. ❑

Definição 2.2 Programa Monolítico.

Um *Programa Monolítico* P é um par ordenado

$$P = (I, r)$$

onde:

`I` *Conjunto de Instruções Rotuladas* o qual é finito;

`r` *Rótulo Inicial* o qual distingue a instrução rotulada inicial em `I`.

Adicionalmente, relativamente ao conjunto `I` tem-se que:

- não existem duas instruções diferentes com um mesmo rótulo;
- um rótulo referenciado por alguma instrução o qual *não* é associado a qualquer instrução rotulada é dito um *Rótulo Final*. ❑

Portanto, a definição de programa monolítico requer a existência de pelo menos uma instrução, identificada pelo rótulo inicial.

EXEMPLO 2.4 Programa Monolítico.

São exemplos de programas monolíticos:

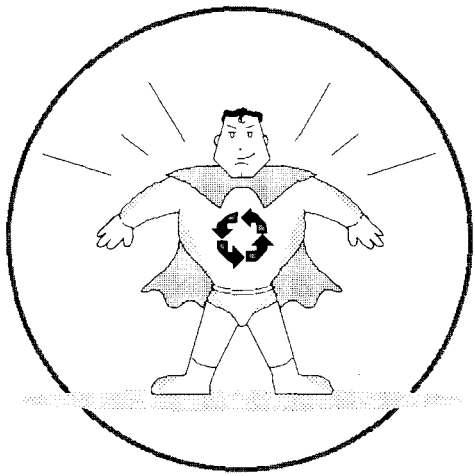
a) $P_1 = (I_1, 1)$ onde I_1 é o conjunto constituído pelas instruções rotuladas 1, 2, 3 e 4 na Figura 2.3. Neste caso, 5 é um rótulo final;

b) $P_2 = (\{r_1:$ faça ✔ vá_para $r_2\}, r_1)$ onde r_2 é um rótulo final (qual o correspondente fluxograma?). ❑

Observação 2.3 Programa Monolítico × Fluxograma.

Como um programa monolítico é definido usando a noção de fluxograma, ambos os termos são identificados e usados indistintamente. ❑

O mecanismo de composição seqüencial dado pelas instruções rotuladas é o tipo mais fundamental de controle de estrutura. É a estrutura básica utilizada pela maioria das linguagens de máquinas, bem como linguagens de baixo nível como *linguagem de montagem* (*assembly*) e é incorporada de certa maneira a outras linguagens de alto nível.



2.1.2 Programa Iterativo

A noção de programa com estruturas de controle iterativas tem sua origem na tentativa de solucionar os problemas decorrentes da dificuldade de entendimento e manutenção de programas monolíticos onde existe uma grande liberdade para definir desvios incondicionais ocasionando o que vulgarmente é conhecido como "quebras de lógica". A idéia básica é substituir desvios incondicionais por estruturas de controle de ciclos ou repetições resultando em uma melhor estruturação dos desvios. Estas noções deram origem ao que hoje é chamado de *Programação Estruturada* ([KNU69]) e inspiraram uma nova geração de linguagens de programação como Pascal ([JEN74]).

Programas iterativos são baseados em três mecanismos de composição (seqüenciais) de programas, os quais podem ser encontrados em um grande número de linguagens de alto nível, como, por exemplo, Algol 68 ([MAL69]), Pascal ([JEN74]), Ada ([UNI80]) e Fortran 90 ([ELL94]). Esses mecanismos de composição são:

- *seqüencial*: composição de dois programas, resultando em um terceiro, cujo efeito é a execução do primeiro e, após, a execução do segundo programa componente;
- *condicional*: composição de dois programas, resultando em um terceiro, cujo efeito é a execução de somente um dos dois programas componentes dependendo do resultado de um teste;
- *enquanto*: composição de um programa, resultando em um segundo, cujo efeito é a execução, repetidamente, do programa componente enquanto o resultado de um teste for *verdadeiro*.

Relativamente à composição enquanto, uma vez que o término da iteração é causado somente pelo retorno do valor falso para o teste, eventualmente pode ser conveniente uma outra composição da forma:

- *até*: análoga a composição enquanto, excetuando-se que a execução do programa componente ocorre enquanto o resultado de um teste for *falso*.

Definição 2.4 Programa Iterativo.

Um *Programa Iterativo* P é indutivamente definido como segue:

a) A operação vazia ✔ constitui um programa iterativo;

b) Cada identificador de operação constitui um programa iterativo;

c) *Composição Seqüencial.* Se V e W são programas iterativos, então a composição seqüencial denotada por:

$$V;W$$

resulta em um programa iterativo cujo efeito é a execução de V e, após, a execução de W;

d) *Composição Condicional.* Se V e W são programas iterativos e se T é um identificador de teste, então a composição condicional denotada por:

(se T então V senão W)

resulta em um programa iterativo cujo efeito é a execução de V se T é verdadeiro ou W se T é falso;

e) *Composição Enquanto.* Se V é um programa iterativo e se T é um identificador de teste, então a composição enquanto denotada por:

enquanto T faça (V)

resulta em um programa iterativo que testa T e executa V, repetidamente, enquanto o resultado do teste for o valor verdadeiro. Caso contrário, a iteração termina;

f) *Composição Até.* Se V é um programa iterativo e se T é um identificador de teste, então a composição até denotada por:

até T faça (V)

resulta em um programa iterativo que testa T e executa V, repetidamente, enquanto o resultado do teste for o valor falso. Caso contrário, a iteração termina. ❑

Assim, relativamente a composição seqüencial, tem-se que:

$$P_1; P_2; P_3; \ldots; P_n$$

denota o programa cujo efeito é a execução na ordem P_1, P_2, ..., P_n (da esquerda para a direita).

Os parênteses foram utilizados nas cláusulas das demais composições para possibilitar a interpretação, de forma unívoca, por partes consistentes. Por exemplo, a omissão de parênteses no seguinte programa:

enquanto T faça V;W

admite duas interpretações distintas, como segue:

(enquanto T faça V);W
enquanto T faça (V;W)

Assim, com o uso de parênteses, definem-se estruturas de blocos como no comando composto `begin-end` da linguagem Pascal ([JEN74]) ou nas chaves da linguagem C ([PLU83]).

A composição enquanto pode ser simulada pelo fluxograma na Figura 2.4 (qual seria o correspondente fluxograma para a composição até?). Note-se que, na execução de um `enquanto`, V pode não ser executado (em que condições isto ocorre?).

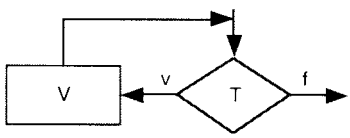


Figura 2.4 Fluxograma que simula a composição enquanto

EXEMPLO 2.5 Programa Iterativo.

a) A operação vazia ✔ constitui um programa iterativo (por quê?).

b) O programa na Figura 2.5 é do tipo iterativo. O uso, de certa forma, abusivo de linhas e identação objetiva facilitar a identificação visual da estrutura do programa. ❑

```
(se   T1
então enquanto T2
      faça (até  T3
            faça (V;W))
senão ✔)
```

Figura 2.5 Programa iterativo

Note-se que a tradução de um programa iterativo para fluxograma é trivial (por quê?). Entretanto, a inversa não é verdadeira, ou seja, fluxogramas nem sempre podem ser traduzidos para programas iterativos equivalentes, para qualquer máquina. A questão de equivalências de programas é detalhada adiante, neste capítulo.

2.1.3 Programa Recursivo

O terceiro tipo de estrutura de controle é encontrado, com diversas variações, na maioria das linguagem de alto nível que admite a definição de sub-rotinas recursivas. *Recursão* é uma forma indutiva de definir programas. *Sub-rotinas* permitem a estruturação hierárquica de programas, possibilitando níveis diferenciados de abstração.

Um dos grandes problemas da Ciência da Computação atual é o correto entendimento e implementação de mecanismos de abstração na presença de composições concorrentes. Entretanto, como afirmado anteriormente, esta publicação não trata de composições concorrentes. Uma boa referência sobre estes e outros aspectos relacionados a concorrência e abstração é [MEN98b].

Para o que segue, suponha um conjunto de *Identificadores de Sub-Rotinas* os quais são descritos por:

$$R_1, R_2, \ldots .$$

O conceito de expressão de sub-rotinas introduzido a seguir é usado na definição de programa recursivo.

Definição 2.5 Expressão de Sub-Rotinas.

Uma *Expressão de Sub-Rotinas* (ou simplesmente *Expressão*) E, é indutivamente definida como segue:

a) A operação vazia ✔ constitui uma expressão de sub-rotinas;

b) Cada identificador de operação constitui uma expressão de sub-rotinas;

c) Cada identificador de sub-rotina constitui uma expressão de sub-rotinas;

d) *Composição Seqüencial*. Se D_1 e D_2 são expressões de sub-rotinas, então a composição seqüencial denotada por:

$$D_1 ; D_2$$

resulta em uma expressão de sub-rotinas cujo efeito é a execução de D_1 e, após, a execução de D_2;

e) *Composição Condicional*. Se D_1 e D_2 são expressões de sub-rotinas e T é um identificador de teste, então a composição condicional denotada por:

$$(\texttt{se } T \texttt{ então } D_1 \texttt{ senão } D_2)$$

resulta em uma expressão de sub-rotinas cujo efeito é a execução de D_1 se T é verdadeiro ou D_2 se T é falso. ❑

Definição 2.6 Programa Recursivo.

Um *Programa Recursivo* P tem a seguinte forma:

$$P \text{ é } E_0 \text{ onde } R_1 \text{ def } E_1, R_2 \text{ def } E_2, \ldots, R_n \text{ def } E_n$$

onde (suponha $k \in \{1, 2, \ldots, n\}$):

E_0 *Expressão Inicial* a qual é uma expressão de sub-rotinas;

E_k *Expressão que Define* R_k, ou seja, a expressão que define a sub-rotina identificada por R_k.

Adicionalmente, para cada identificador de sub-rotina referenciado em alguma expressão, existe uma expressão que o define. ❑

Portanto, a operação vazia ✔ (que também é uma expressão de sub-rotina) constitui um programa recursivo. Neste caso, por simplicidade, como não existe qualquer sub-rotina a ser definida, é usual omitir a palavra onde, ou seja:

✔ denota o programa recursivo que não faz coisa alguma.

EXEMPLO 2.6 Programa Recursivo.

O programa na Figura 2.6 é do tipo recursivo. É importante observar que para cada identificador de sub-rotina referenciado existe uma expressão que o define. Note-se que a recursão é implícita, no sentido em que as sub-rotinas `R` e `S` se referenciam mutuamente e, portanto, `R` e `S` se auto-referenciam indiretamente.❑

```
P é R;S onde
      R def F;(se T então R senão G;S),
      S def (se T então ✔ senão F;R)
```

Figura 2.6 Programa recursivo

A computação de um programa recursivo será detalhada adiante, quando da introdução do conceito de *computação*. Intuitivamente, consiste na avaliação da expressão inicial onde cada identificador de sub-rotina referenciado é substituído pela correspondente expressão que o define, e assim sucessivamente (recursivamente), até que seja substituído pela expressão vazia ✔, determinando o fim da recursão.

Até agora, foram definidos três tipos de programas cujos modelos são características de linguagens de programação reais. Entretanto, esses programas são incapazes de descrever completamente uma computação, pois não se tem a natureza das operações ou dos testes, mas apenas um conjunto de identificadores. A natureza das operações e testes é especificada na definição de máquina.

2.2 Máquinas

O objetivo de uma máquina é suprir todas as informações necessárias para que a computação de um programa possa ser descrita. Portanto, cabe à máquina suprir o significado (dar semântica) aos identificadores das operações e testes.

Assim, cada identificador de operação e de teste interpretado pela máquina deve ser associado a uma transformação na estrutura de memória e a uma função verdade, respectivamente. Note-se que:

- nem todo o identificador de operação ou teste é definido em uma máquina;
- para cada identificador de operação ou teste definido em uma máquina, existe somente uma função associada.

Adicionalmente, a máquina deve descrever o armazenamento ou recuperação de informações na estrutura de memória.

Definição 2.7 Máquina.

Uma *Máquina* é uma 7-upla

$$M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$$

onde:

- V *Conjunto de Valores de Memória*;
- X *Conjunto de Valores de Entrada*;
- Y *Conjunto de Valores de Saída*;
- π_X *Função de Entrada* tal que:

$$\pi_X: X \rightarrow V$$

- π_Y *Função de Saída* tal que:

$$\pi_Y: V \rightarrow Y$$

- Π_F *Conjunto de Interpretações de Operações* tal que, para cada identificador de operação F interpretado por M, existe uma única função:

$$\pi_F: V \rightarrow V \text{ em } \Pi_F$$

- Π_T *Conjunto de Interpretações de Testes* tal que, para cada identificador de teste T interpretado por M, existe uma única função:

$$\pi_T: V \rightarrow \{\text{verdadeiro, falso}\} \text{ em } \Pi_T$$ ❑

As funções acima são totais, ou seja, definidas para todos os elementos do domínio. O conjunto de interpretações Π_F (respectivamente, Π_T) pode ser visto como um conjunto de funções indexado pelo subconjunto de identificadores de operações (respectivamente, testes) para as quais a máquina M é definida.

No texto que segue as seguintes notações são adotadas:

- $\mathbb{N}$ para o *conjunto dos números naturais*;
- S^2 para o produto cartesiano de um conjunto S, ou seja:

$$S^2 = S \times S$$

EXEMPLO 2.7 Máquina de Dois Registradores.

Suponha uma especificação de uma máquina com dois registradores a e b os quais assumem valores em $\mathbb{N}$, com duas operações e um teste como segue:

- subtração de 1 em a, se $a > 0$;
- adição de 1 em b;
- teste se a é zero.

Adicionalmente, valores de entrada são armazenados em a (zerando b) e a saída retorna o valor de b.

Dois registradores com valores em $\mathbb{N}$ podem ser definidos pelo produto cartesiano $\mathbb{N}^2$ onde os registradores a e b são representados pela primeira e segunda componente, respectivamente. Então a máquina na Figura 2.7 implementa a especificação acima. ❑

dois_reg = ($\mathbb{N}^2$, $\mathbb{N}$, $\mathbb{N}$, armazena_a, retorna_b, {subtrai_a, adiciona_b}, {a_zero})

onde:

$\mathbb{N}^2$ corresponde ao conjunto de valores de memória

$\mathbb{N}$ corresponde, simultaneamente, aos conjuntos de valores de entrada e saída

armazena_a: $\mathbb{N} \rightarrow \mathbb{N}^2$ é a função de entrada tal que, $\forall n \in \mathbb{N}$:

armazena_a(n) = (n, 0)

retorna_b: $\mathbb{N}^2 \rightarrow \mathbb{N}$ é a função de saída tal que, $\forall (n, m) \in \mathbb{N}^2$:

retorna_b(n, m) = m

subtrai_a: $\mathbb{N}^2 \rightarrow \mathbb{N}^2$ é interpretação tal que, $\forall (n, m) \in \mathbb{N}^2$:

subtrai_a(n, m) = (n-1, m), se $n \neq 0$; subtrai_a(n, m) = (0, m), se n = 0

adiciona_b: $\mathbb{N}^2 \rightarrow \mathbb{N}^2$ é interpretação tal que, $\forall (n, m) \in \mathbb{N}^2$:

adiciona_b(n, m) = (n, m+1)

a_zero: $\mathbb{N}^2 \rightarrow$ {verdadeiro, falso} é interpretação tal que, $\forall (n, m) \in \mathbb{N}^2$:

a_zero(n, m) = verdadeiro, se n = 0; a_zero(n, m) = falso, se $n \neq 0$

Figura 2.7 Máquina com dois registradores

Afirma-se que P é um *programa para a máquina* M se cada identificador de teste e operação em P tiver uma correspondente função de teste e operação em M, respectivamente. A definição formal é como segue (lembre-se que programas são constituídos por operações e testes e, portanto, não incluem funções de entrada e saída).

Definição 2.8 Programa para uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e P um programa onde P_F e P_T são os conjuntos de identificadores de operações e testes de P, respectivamente. P *é um Programa para a Máquina* M se, e somente, se:

- para qualquer $F \in P_F$, existe uma única função π_F: $V \rightarrow V$ em Π_F;
- para qualquer $T \in P_T$, existe uma única função π_T: $V \rightarrow$ {verdadeiro, falso} em Π_T.

Adicionalmente, a operação vazia ✔ sempre é interpretada em qualquer máquina. ❑

Portanto, para cada identificador de operação F de P_F (respectivamente, teste T de P_T) existe, na máquina M, uma única interpretação de função de operação indexada por F (respectivamente, função de teste indexada por T). Note-se que a máquina pode possuir operações ou testes sem correspondência no programa.

EXEMPLO 2.8 Programas para a Máquina de Dois Registradores.

Os programas iterativo itv_b←a na Figura 2.8 e recursivo rec_b←a na Figura 2.9, são programas para a máquina dois_reg (Figura 2.7) e atribuem o valor armazenado em a ao registrador b. ⌟

Programa Iterativo itv_b←a

```
até   a_zero
faça  (subtrai_a ; adiciona_b)
```

Figura 2.8 Programa iterativo para a máquina de dois registradores

Programa Recursivo rec_b←a

```
rec_b←a é R onde
      R def (se a_zero então ✔ senão S;R),
      S def subtrai_a ; adiciona_b
```

Figura 2.9 Programa recursivo para a máquina de dois registradores

2.3 Computações e Funções Computadas

Nesta seção é visto como as definições de programas e máquinas caminham juntas para a definição de computação. Uma *computação* é, resumidamente, um histórico do funcionamento da máquina para o programa, considerando um valor inicial.

Uma vez definida a noção de computação, pode-se inferir a natureza da função computada, por um dado programa, em uma dada máquina.

2.3.1 Computação

Inicialmente, é tratada a computação referente aos programas monolíticos e, a seguir, é apresentada a referente aos programas recursivos. A computação referente aos programas iterativos é sugerida como exercício.

Basicamente, uma computação de um programa monolítico em uma máquina é um histórico das instruções executadas e o correspondente valor de memória. O histórico é representado na forma de uma cadeia de pares onde:

- cada par reflete um estado da máquina para o programa, ou seja, a instrução a ser executada e o valor corrente da memória;
- a cadeia reflete uma seqüência de estados possíveis a partir do estado inicial (instrução inicial e valor de memória considerado).

Definição 2.9 Computação de Programa Monolítico em uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e $P = (I, r)$ um programa monolítico para M onde L é o seu correspondente conjunto de rótulos. Uma *Computação do Programa Monolítico* P *na Máquina* M é uma cadeia (finita ou infinita) de pares de $L \times V$:

$$(s_0, v_0)(s_1, v_1)(s_2, v_2)...$$

onde (s_0, v_0) é tal que $s_0 = r$ é o rótulo inicial do programa P e v_0 é o valor inicial de memória e, para cada par (s_k, v_k) da cadeia, onde $k \in \{0, 1, 2, ...\}$, tem-se que (suponha que F é um identificador de operação, T é um identificador de teste e r', r" são rótulos de L):

a) *Operação.*

a.1) Se s_k é o rótulo de uma operação da forma:

s_k: faça F vá_para r'

então $(s_{k+1}, v_{k+1}) = (r', \pi_F(v_k))$ é par subseqüente de (s_k, v_k) na cadeia

a.2) Se s_k é o rótulo de uma operação da forma:

s_k: faça ✔ vá_para r'

então $(s_{k+1}, v_{k+1}) = (r', v_k)$ é par subseqüente de (s_k, v_k) na cadeia

b) *Teste.* Se s_k é o rótulo de um teste da forma:

s_k: se T então vá_para r' senão vá_para r"

então (s_{k+1}, v_{k+1}) é par subseqüente de (s_k, v_k) na cadeia sendo que $v_{k+1} = v_k$ e:

$s_{k+1} = r'$ se $\pi_T(v_k)$ = verdadeiro

$s_{k+1} = r"$ se $\pi_T(v_k)$ = falso

Uma *Computação* é dita *Finita* ou *Infinita*, se a cadeia que a define é finita ou infinita, respectivamente. ❑

Note-se que:

- para um dado valor inicial de memória, a correspondente cadeia de computação é única, ou seja, a computação é determinística (por quê?);
- um teste e a operação vazia não alteram o valor corrente da memória;
- em uma computação infinita, rótulo algum da cadeia é final.

Nos exemplos que seguem, para facilitar o entendimento, uma cadeia de computação é representada na forma de coluna.

EXEMPLO 2.9 Computação Finita de Programa Monolítico na Máquina de Dois Registradores.

O programa monolítico mon_b←a na Figura 2.10 é um programa para a máquina dois_reg (Figura 2.7). Para o valor inicial de memória (3, 0), ou seja, onde os valores iniciais dos registradores a e b são 3 e 0, respectivamente, a correspondente computação finita é representada na Figura 2.11. Note-se que o registrador b recebeu o valor inicial de a (e a foi zerado). ❑

Programa Monolítico mon_b←a

```
1:    se a_zero então vá_para 9 senão vá_para 2
2:    faça subtrai_a vá_para 3
3:    faça adiciona_b vá_para 1
```

Figura 2.10 Programa monolítico cuja computação na máquina de 2 registradores é finita

(1, (3, 0))	instrução inicial e valor de entrada armazenado
(2, (3, 0))	em 1, como a ≠ 0, desviou para 2
(3, (2, 0))	em 2, subtraiu do registrador a e desviou para 3
(1, (2, 1))	em 3, adicionou no registrador b e desviou para 1
(2, (2, 1))	em 1, como a ≠ 0, desviou para 2
(3, (1, 1))	em 2, subtraiu do registrador a e desviou para 3
(1, (1, 2))	em 3, adicionou no registrador b e desviou para 1
(2, (1, 2))	em 1, como a ≠ 0, desviou para 2
(3, (0, 2))	em 2, subtraiu do registrador a e desviou para 3
(1, (0, 3))	em 3, adicionou no registrador b e desviou para 1
(9, (0, 3))	em 1, como a = 0, desviou para 9

Figura 2.11 Computação finita na máquina de 2 registradores

EXEMPLO 2.10 Computação Infinita de Programa Monolítico na Máquina de Dois Registradores.

O programa monolítico comp_infinita na Figura 2.12 é um programa para a máquina dois_reg (Figura 2.7). Para o valor inicial de memória (3, 0), a correspondente computação infinita é representada na Figura 2.13. ❑

Programa Monolítico comp_infinita

```
1:    faça adiciona_b vá_para 1
```

Figura 2.12 Programa monolítico cuja computação na máquina de 2 registradores é infinita

(1, (3, 0))	instrução inicial e valor de entrada armazenado
(1, (3, 1))	adicionou no registrador b e permanece em 1
(1, (3, 2))	adicionou no registrador b e permanece em 1
(1, (3, 3))	adicionou no registrador b e permanece em 1
...	repete 1, adicionando no registrador b, indefinidamente

Figura 2.13 Computação infinita na máquina de 2 registradores

A computação de um programa recursivo em uma máquina é análoga a de um monolítico. O histórico é representado na forma de uma cadeia de pares onde:

- cada par reflete um estado da máquina para o programa, ou seja, a expressão de sub-rotina a ser executada e o valor corrente da memória;
- a cadeia reflete uma seqüência de estados possíveis a partir do inicial.

Definição 2.10 Computação de Programa Recursivo em uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e P um programa recursivo para M tal que:

P é E_0 onde R_1 def E_1, R_2 def E_2, ..., R_n def E_n

Uma *Computação do Programa Recursivo* P *na Máquina* M é uma cadeia (finita ou infinita) de pares da forma:

$$(D_0, v_0)(D_1, v_1)(D_2, v_2)\ldots$$

onde (D_0, v_0) é tal que $D_0 = E_0 ; \checkmark$ e v_0 é o valor inicial de memória e, para cada par (D_k, v_k) da cadeia, onde $k \in \{0, 1, 2, \ldots\}$, tem-se que (suponha que F é um identificador de operação, T é um identificador de teste e C, C_1, C_2 são expressões de sub-rotina):

Caso 1. Se D_k é uma expressão de sub-rotina da forma:

$$D_k = \checkmark ; C$$

então $(D_{k+1}, v_{k+1}) = (C, v_k)$ é par subseqüente de (D_k, v_k) na cadeia;

Caso 2. Se D_k é uma expressão de sub-rotina da forma:

$$D_k = F ; C$$

então $(D_{k+1}, v_{k+1}) = (C, \pi_F(v_k))$ é par subseqüente de (D_k, v_k) na cadeia;

Caso 3. Se D_k é uma expressão de sub-rotina da forma:

$$D_k = R_i ; C$$

então $(D_{k+1}, v_{k+1}) = (E_i ; C, v_k)$ é par subseqüente de (D_k, v_k) na cadeia;

Caso 4. Se D_k é uma expressão de sub-rotina da forma:

$$D_k = (C_1 ; C_2) ; C$$

então $(D_{k+1}, v_{k+1}) = (C_1 ; (C_2 ; C), v_k)$ é par subseqüente de (D_k, v_k) na cadeia;

Caso 5. Se D_k é uma expressão de sub-rotina da forma:

$$D_k = E_k = (\texttt{se T então } C_1 \texttt{ senão } C_2) ; C$$

então (D_{k+1}, v_{k+1}) é par subseqüente de (D_k, v_k) na cadeia sendo que:

$v_{k+1} = v_k$
$D_{k+1} = C_1 ; C$ se $\pi_T(v_k)$ = verdadeiro
$D_{k+1} = C_2 ; C$ se $\pi_T(v_k)$ = falso

A *Computação* é dita *Finita* ou *Infinita*, se a cadeia que a define é finita ou infinita, respectivamente. ❑

Portanto:

- para um dado valor inicial de memória, a correspondente cadeia de computação é única, ou seja, a computação é determinística (por quê?);
- teste ou referência a uma sub-rotina não alteram o valor da memória;
- em uma computação finita, a expressão ✔ ocorre no último par da cadeia e não ocorre em qualquer outro par;
- em uma computação infinita, expressão alguma da cadeia é ✔.

EXEMPLO 2.11 Computação Infinita de Programa Recursivo.

O programa recursivo qq_máquina na Figura 2.14 é um programa para qualquer máquina. Adicionalmente, para qualquer valor inicial de memória, a correspondente computação é sempre infinita e é a cadeia:

$$(R, v_0)(R, v_0)(R, v_0)... \quad \square$$

Programa Recursivo qq_máquina

```
qq_máquina é R onde
      R def R
```

Figura 2.14 Programa recursivo cuja computação é infinita em qualquer máquina

No texto que segue, para um dado conjunto A, a *Função Identidade em* A é aquela que associa, a cada elemento do conjunto, o próprio elemento, ou seja:

$$id_A: A \rightarrow A$$

tal que, para qualquer $a \in A$, tem-se que $id_A(a) = a$.

EXEMPLO 2.12 Computação Finita de Programa Recursivo na Máquina de Um Registrador.

Considere o programa recursivo duplica na Figura 2.16 para a máquina um_reg na Figura 2.15. Para o valor inicial de memória 3, a correspondente computação finita é representada na Figura 2.17. Note-se que o valor final na memória é o valor inicial duplicado. Observe-se que, usando a noção de recursão, não foi necessário usar duas células de memória, uma para controlar o ciclo e outra para calcular o resultado. De fato, a facilidade de recursão é usada para determinar quantas vezes a operação ad necessita ser executada. Esta observação é importante adiante, quando é verificado que nem todos os tipos de programas são (fortemente) equivalentes. $\square$

$$um\_reg = (\mathbb{N}, \mathbb{N}, \mathbb{N}, id_{\mathbb{N}}, id_{\mathbb{N}}, \{ad, sub\}, \{zero\})$$

onde:

$\mathbb{N}$ corresponde, simultaneamente, aos conjuntos de valores de memória, entrada e saída

$id_{\mathbb{N}}: \mathbb{N} \rightarrow \mathbb{N}$ é a função de entrada e de saída

$ad: \mathbb{N} \rightarrow \mathbb{N}$ é interpretação tal que, $\forall n \in \mathbb{N}$, $ad(n) = n+1$

$sub: \mathbb{N} \rightarrow \mathbb{N}$ é interpretação tal que, $\forall n \in \mathbb{N}$:

$$sub(n) = n-1, \text{ se } n \neq 0; \quad sub(n) = 0, \text{ se } n = 0$$

$zero: \mathbb{N} \rightarrow \{verdadeiro, falso\}$ é interpretação tal que, $\forall n \in \mathbb{N}$:

$$zero(n) = verdadeiro, \text{ se } n = 0; \quad zero(n) = falso, \text{ caso contrário}$$

Figura 2.15 Máquina de um registrador

Programa Recursivo duplica

```
duplica é R onde
        R def (se zero então ✔ senão sub;R;ad;ad)
```

Figura 2.16 Programa recursivo cuja computação é finita na máquina de um registrador

(R;✔, 3)	valor de entrada armazenado
((se zero então ✔ senão (sub;R;ad;ad));✔, 3)	caso 3
((sub;R;ad;ad);✔, 3)	como n ≠ 0, executa senão
(sub;(R;ad;ad);✔, 3)	caso 4, composição seqüencial
((R;ad;ad);✔, 2)	subtraiu 1 da memória
(R;(ad;ad);✔, 2)	caso 4, composição seqüencial
((se zero então ✔ senão (sub;R;ad;ad));(ad;ad);✔, 2)	caso 3
((sub;R;ad;ad);(ad;ad);✔, 2)	como n ≠ 0, executa senão
(sub;(R;ad;ad);(ad;ad);✔, 2)	caso 4, composição seqüencial
((R;ad;ad);(ad;ad);✔, 1)	subtraiu 1 da memória
(R;(ad;ad);(ad;ad);✔, 1)	caso 4, composição seqüencial
((se zero então ✔ senão(sub;R;ad;ad));(ad;ad);(ad;ad);✔, 1)	caso 3
((sub;R;ad;ad));(ad;ad);(ad;ad);✔, 1)	como n ≠ 0, executa senão
(sub;(R;ad;ad);(ad;ad);(ad;ad);✔, 1)	caso 4, composição seqüencial
((R;ad;ad);(ad;ad);(ad;ad);✔, 0)	subtraiu 1 da memória
(R;(ad;ad);(ad;ad);(ad;ad);✔, 0)	caso 4, composição seqüencial
((se zero então ✔ senão (sub;R;ad;ad));(ad;ad);(ad;ad);(ad;ad);✔, 0)	caso 3
(✔;(ad;ad);(ad;ad);(ad;ad);✔, 0)	como n = 0, executa então
((ad;ad);(ad;ad);(ad;ad);✔, 0)	caso 1
(ad;(ad;(ad;ad);(ad;ad);✔), 0)	caso 4, composição seqüencial
((ad;(ad;ad);(ad;ad);✔), 1)	adicionou 1 na memória
(ad;((ad;ad);(ad;ad);✔), 1)	caso 4, composição seqüencial
(((ad;ad);(ad;ad);✔), 2)	adicionou 1 na memória
(ad;(ad;(ad;ad);✔), 2)	caso 4, composição seqüencial
((ad;(ad;ad);✔), 3)	adicionou 1 na memória
(ad;((ad;ad);✔), 3)	caso 4, composição seqüencial
(((ad;ad);✔), 4)	adicionou 1 na memória
(ad;((ad);✔), 4)	caso 4, composição seqüencial
(((ad);✔), 5)	adicionou 1 na memória
(ad;(✔), 5)	caso 4, composição seqüencial
((✔), 6)	adicionou 1 na memória
(✔, 6)	fim da recursão

Figura 2.17 Computação finita na máquina de um registrador

2.3.2 Função Computada

Em geral, a computação de um programa deve ser associada a uma entrada e uma saída. Adicionalmente, espera-se que a resposta (saída) seja gerada em um tempo finito. Estas noções induzem a definição de *função computada*.

Inicialmente, é tratada a função computada referente aos programas monolíticos e, a seguir, é apresentada a referente aos programas recursivos. A função computada referente aos programas iterativos é sugerida como exercício.

A função computada por um programa monolítico sobre uma máquina corresponde à noção intuitiva, ou seja:

- a computação inicia na instrução identificada pelo rótulo inicial com a memória contendo o valor inicial resultante da aplicação da função de entrada sobre o dado fornecido;
- executa, passo a passo, testes e operações, na ordem determinada pelo programa, até atingir um rótulo final, quando pára;
- o valor da função computada pelo programa é o valor resultante da aplicação da função de saída ao valor da memória quando da parada.

Definição 2.11 Função Computada por um Programa Monolítico em uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e P um programa monolítico para M. A *Função Computada pelo Programa Monolítico* P *na Máquina* M denotada por:

$$\langle P, M\rangle: X \to Y$$

é uma função parcial definida para $x \in X$ se a cadeia:

$$(s_0, v_0)(s_1, v_1)\ldots(s_n, v_n)$$

é uma computação finita de P em M, onde o valor inicial da memória é dado pela função de entrada, ou seja, $v_0 = \pi_X(x)$. Neste caso, a imagem de x é dada pela função de saída aplicada ao último valor da memória na computação, ou seja:

$$\langle P, M\rangle(x) = \pi_Y(v_n)$$ ❑

Note-se que, a função computada por um programa pode ser parcial, ou seja, não necessita ser definida para cada valor do domínio.

EXEMPLO 2.13 Função Computada por Programa Monolítico na Máquina de Dois Registradores.

Considere o programa monolítico mon_b←a (Figura 2.10) para a máquina dois_reg (Figura 2.7). A correspondente função computada é a função identidade, ou seja,

$$\langle \text{mon\_b}\leftarrow\text{a, dois\_reg}\rangle: \mathbb{N} \to \mathbb{N}$$

tal que, para qualquer $n \in \mathbb{N}$, tem-se que:

$$\langle \text{mon\_b}\leftarrow\text{a, dois\_reg}\rangle(n) = n$$

Por exemplo, para o valor entrada 3, tem-se que:

- $\pi_X(3) = (3, 0)$
- a correspondente computação é representada na Figura 2.11
- $\langle mon\_b \leftarrow a, dois\_reg \rangle(3) = \pi_Y(0, 3) = 3$

Portanto, ⟨mon_b←a, dois_reg⟩ é definida para 3 (a função é total?). ❑

EXEMPLO 2.14 Função Computada por Programa Monolítico na Máquina de Dois Registradores.

Considere o programa monolítico comp_infinita (Figura 2.12) para a máquina dois_reg (Figura 2.7). Relativamente à função computada ⟨comp_infinita, dois_reg⟩: $\mathbb{N} \rightarrow \mathbb{N}$, para a entrada de valor 3, tem-se que:

- $\pi_X(3) = (3, 0)$
- a correspondente computação infinita é representada na Figura 2.13

Como a cadeia é infinita, a função computada *não* é definida para o valor de entrada 3 (para quais valores do domínio é definida?). ❑

A função computada por um programa recursivo sobre uma máquina correspondente é análoga a de um monolítico:

- a computação inicia na expressão inicial com a memória contendo o valor inicial resultante da aplicação da função de entrada sobre o dado fornecido;
- executa, passo a passo, testes e operações, na ordem determinada pelo programa, até que a expressão de sub-rotina resultante seja a expressão vazia, quando pára;
- o valor da função computada pelo programa é o valor resultante da aplicação da função de saída ao valor da memória quando da parada.

Definição 2.12 Função Computada por um Programa Recursivo em uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e P um programa recursivo para M. A *Função Computada pelo Programa Recursivo* P *na Máquina* M denotada por:

$$\langle P, M \rangle: X \rightarrow Y$$

é uma função parcial definida para $x \in X$ se a seguinte cadeia é uma computação finita de P em M:

$$(D_0, v_0)(D_1, v_1)\ldots(D_n, v_n)$$

onde:

$D_0 = E_0$ é expressão inicial de P
$v_0 = \pi_X(x)$
$E_n = \checkmark$

Neste caso, tem-se que:

$$\langle P, M \rangle(x) = \pi_Y(v_n)$$ ❑

EXEMPLO 2.15 Função Computada por Programa Recursivo.

Considere o programa recursivo qq_máquina (Figura 2.14) e uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ qualquer. A correspondente função computada é:

$$\langle qq\_máquina, M\rangle: X \to Y$$

e é indefinida para qualquer entrada (por quê?), ou seja, é definida para o conjunto vazio. ❑

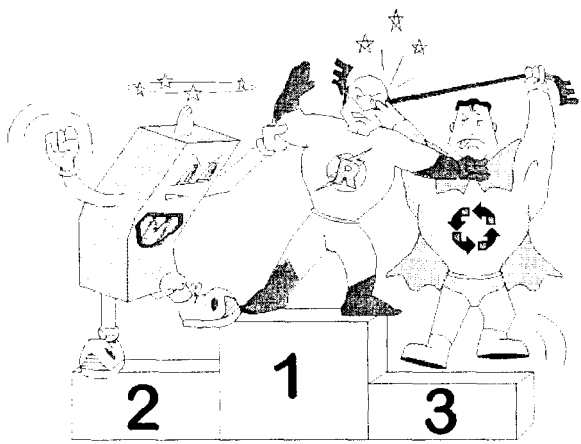
EXEMPLO 2.16 Função Computada por Programa Recursivo na Máquina de Um Registrador.

Considere o programa recursivo duplica (Figura 2.16) para a máquina um_reg (Figura 2.15). A correspondente função computada é:

$$\langle duplica, um\_reg\rangle: \mathbb{N} \to \mathbb{N}$$

e é tal que, para qualquer $n \in \mathbb{N}$:

$$\langle duplica, um\_reg\rangle(n) = 2n$$ ❑



2.4 Equivalências de Programas e Máquinas

Funções computadas permitem introduzir algumas importantes relações de equivalências de programas e máquinas como segue:

a) *Relação Equivalência Forte de Programas.* Um par de programas pertence à relação se as correspondentes funções computadas coincidem para *qualquer* máquina;

b) *Relação Equivalência de Programas em uma Máquina.* Um par de programas pertence à relação se as correspondentes funções computadas coincidem para uma *dada* máquina;

c) *Relação Equivalência de Máquinas*. Um par de máquina pertence à relação se as máquinas podem se simular mutuamente. A simulação de uma máquina por outra pode ser feita usando programas diferentes.

A Relação Equivalência Forte de Programas é especialmente importante pois, ao agrupar diferentes programas em classes de equivalências de programas cujas funções coincidem, fornece subsídios para analisar propriedades de programas como complexidade estrutural.

Um importante resultado é que, para a Relação Equivalência Forte de Programas, programas recursivos são mais gerais que os monolíticos os quais, por sua vez, são mais gerais que os iterativos.

Para o que segue, o seguinte deve ser considerado:

a) *Igualdade de Funções Parciais*. Duas funções parciais $f, g: X \rightarrow Y$ são ditas iguais, ou seja, $f = g$, se, e somente se, para cada $x \in X$:

- ou $f(x)$ e $g(x)$ são indefinidas;
- ou ambas são definidas e $f(x) = g(x)$;

b) *Composição Sucessiva de Funções*. Para uma dada função $f: S \rightarrow S$, a composição sucessiva de f com ela própria é denotada usando expoente, ou seja:

$$f^n = f\ f \ldots\ f \quad (n \text{ vezes})$$

2.4.1 Equivalência Forte de Programas

Definição 2.13 Relação Equivalência Forte de Programas, Programas Equivalentes Fortemente.

Sejam P e Q dois programas arbitrários, não necessariamente do mesmo tipo. Então o par (P, Q) está na *Relação Equivalência Forte de Programas*, denotado por:

$$P \equiv Q$$

se, e somente se, para qualquer máquina M, as correspondentes funções parciais computadas são iguais, ou seja:

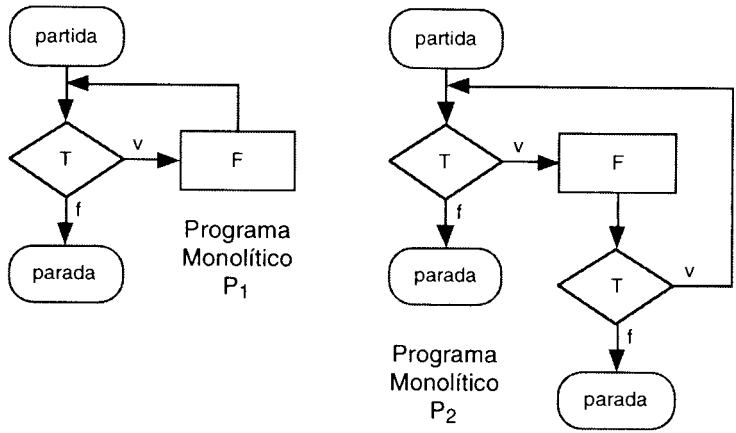
$$\langle P, M \rangle = \langle Q, M \rangle$$

Neste caso, P e Q são ditos *Programas Equivalentes Fortemente*. ❑

É fácil verificar que a Relação Equivalência Forte de Programas definida acima é uma relação de equivalência e que, portanto, induz uma partição do conjunto de todos os programas em classes de equivalências.

EXEMPLO 2.17 Programas Equivalentes Fortemente.

Considere os quatro programas na Figura 2.18. Os programas monolíticos P_1 e P_2 (acima), o iterativo P_3 (meio) e o recursivo e P_4 (abaixo) são todos equivalentes fortemente.



Programa Iterativo P_3

```
enquanto T
faça  (F)
```

Programa Recursivo P_4

```
P4 é R onde
      R def (se T então F;R senão ✔)
```

Figura 2.18 Programas equivalentes fortemente

Como ilustração, no que segue, é verificado que, de fato, os programas monolítico P_1 e o recursivo P_4 são equivalentes fortemente. O programa P_1 reescrito na forma de instruções rotuladas é **represe**ntado na Figura 2.19.

Programa Monolítico P_1

```
1:    se T então vá_para 2 senão vá_para 3
2:    faça F vá_para 1
```

Figura 2.19 Programa monolítico como um conjunto de instruções rotuladas

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina arbitrária e $x \in X$ tal que $\pi_X(x) = v$. Então, se $\langle P_1, M\rangle$ é definida para x, a correspondente computação é dada pela cadeia:

$$(1, v)(2, v)(1, \pi_F(v))(2, \pi_F(v))(1, \pi_F^2(v))(2, \pi_F^2(v))\ldots(1, \pi_F^n(v))(3, \pi_F^n(v))$$

supondo que n é o menor natural tal que $\pi_T(\pi_F^n(v)) =$ falso. Neste caso:

$$\langle P_1, M\rangle(x) = \pi_Y(\pi_F^n(v))$$

Analogamente, se $\langle P_4, M\rangle$ é definida para x, a correspondente computação é dada pela cadeia representada na Figura 2.20 supondo que n é o menor natural tal que $\pi_T(\pi_F{}^n(v)) = \text{falso}$. Neste caso:

$$\langle P_4, M\rangle(x) = \pi_Y(\pi_F{}^n(v))$$

Portanto, $P_1 \equiv P_4$ pois, para qualquer máquina M, tem-se que:

$$\langle P_1, M\rangle = \langle P_4, M\rangle$$ ❑

(R, v)
((se T então F;R senão ✔), v)
(F;R, v)
(R, $\pi_F(v)$)
((se T então F;R senão ✔), $\pi_F(v)$)
(F;R, $\pi_F(v)$)
(R, $\pi_F{}^2(v)$)
((se T então F;R senão ✔), $\pi_F{}^2(v)$)
(F;R, $\pi_F{}^2(v)$)
...
(R, $\pi_F{}^n(v)$)
((se T então F;R senão ✔), $\pi_F{}^n(v)$)
(✔, $\pi_F{}^n(v)$)

Figura 2.20 Computação

É importante que se considere a Relação Equivalência Forte de Programas por várias razões, como, por exemplo:

- permite identificar diferentes programas em uma mesma classe de equivalência, ou seja, identificar diferentes programas cujas funções computadas coincidem, para qualquer máquina;
- as funções computadas por programas equivalentes fortemente têm a propriedade de que os mesmos testes e as mesmas operações são efetuados na mesma ordem, independentemente do significado dos mesmos (por quê não diferentes testes ou operações ou, ainda, diferente ordem?);
- fornece subsídios para analisar a complexidade estrutural de programas. Por exemplo, analisando os programas monolíticos equivalentes fortemente P_1 e P_2 na Figura 2.18, pode-se concluir que P_1 é estruturalmente "mais otimizado" que P_2, pois contém um teste a menos.

No que segue, são introduzidos alguns resultados sobre tipos de programas introduzidos onde, verifica-se que:

- para todo iterativo, existe um monolítico equivalente fortemente;
- para todo monolítico, existe um recursivo equivalente fortemente.

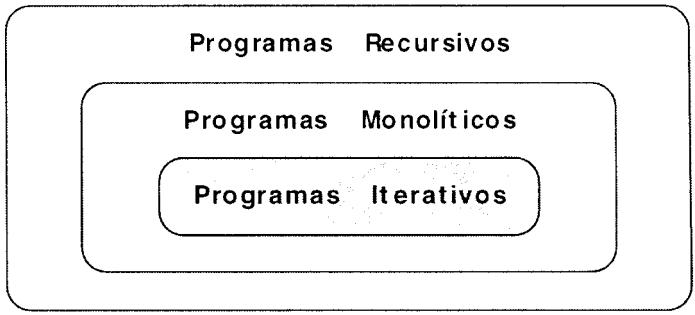


Figura 2.21 Hierarquia induzida pela Relação Equivalência Forte de Programas

Entretanto, a inversa não necessariamente é verdadeira, ou seja, relativamente à Relação Equivalência Forte de Programas, programas recursivos são mais gerais que os monolíticos os quais, por sua vez, são mais gerais que os iterativos, induzindo uma hierarquia de classes de programas como ilustrado na Figura 2.21, onde as inclusões são próprias.

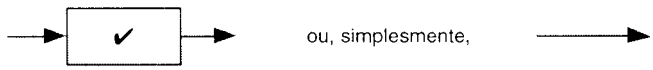
Teorema 2.14 Equivalência Forte de Programas: Iterativo → Monolítico.

Para qualquer programa iterativo P_i, existe um programa monolítico P_m, tal que $P_i \equiv P_m$.

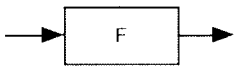
Prova:

Seja P_i um programa iterativo qualquer. Seja P_m um programa monolítico indutivamente construído como segue:

a) Para a operação vazia ✔ corresponde o seguinte fluxograma elementar:

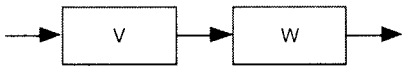


b) Para cada identificador de operação F de P_i corresponde o seguinte fluxograma elementar:

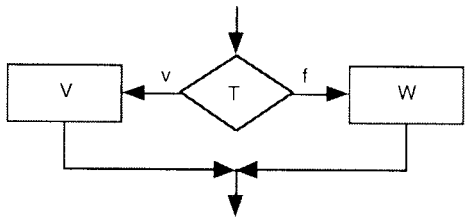


c) Suponha que T é um identificador de teste e que V, W são programas iterativos usados na construção de P_i. Então, para cada um dos seguintes tipos de composição é apresentado o correspondente fluxograma de P_m:

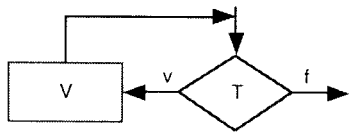
c.1) *Composição Seqüencial.* V ; W



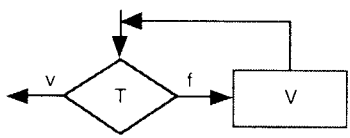
c.2) *Composição Condicional.* (se T então V senão W)



c.3) *Composição Enquanto.* enquanto T faça (V)



c.4) *Composição Até.* até T faça (V)



Adicionalmente, para cada uma das composições acima, dependendo se trata-se do início ou fim do programa iterativo, o correspondente fluxograma deve ser antecedido ou sucedido dos componentes elementares de partida ou parada, respectivamente.

A verificação que, de fato, $P_i \equiv P_m$ é sugerida como exercício. ❑

Teorema 2.15 Equivalência Forte de Programas: Monolítico → Recursivo.

Para qualquer programa monolítico P_m, existe um programa recursivo P_r, tal que $P_m \equiv P_r$.

Prova:

Seja P_m um programa monolítico qualquer onde $L = \{r_1, r_2, \ldots, r_n\}$ é o correspondente conjunto de rótulos. Suponha, sem perda de generalidade, que, em P_m, r_n é o *único* rótulo final (por quê esta suposição pode ser feita?). Então P_r é um programa recursivo construído a partir de P_m e é tal que:

P_r é R_1 onde R_1 def E_1, R_2 def E_2, ..., R_n def ✔

onde, para $k \in \{1, 2, \ldots, n-1\}$, E_k é como segue:

a) *Operação.* Se r_k é da forma:

r_k: faça F vá_para r_k'

então E_k é a seguinte expressão de sub-rotinas:

$F ; R_k'$

b) *Teste*. Se r_k é da forma:

r_k: se T então vá_para r_k' senão vá_para r_k"

então E_k é a seguinte expressão de sub-rotinas:

(se T então R_k' senão R_k")

A verificação que, de fato, $P_m \equiv P_r$ é sugerida como exercício. ❑

O seguinte corolário é conseqüência direta dos dois teoremas anteriores.

Corolário 2.16 Equivalência Forte de Programas: Iterativo → Recursivo.

Para qualquer programa iterativo P_i, existe um programa recursivo P_r, tal que $P_i \equiv P_r$. ❑

Teorema 2.17 Equivalência Forte de Programas: Recursivo ↛ Monolítico.

Dado um programa recursivo P_r qualquer, não necessariamente existe um programa monolítico P_m, tal que $P_r \equiv P_m$.

Prova: (Por Absurdo).

Para provar é suficiente apresentar um programa recursivo que, para uma determinada máquina, não existe programa monolítico equivalente fortemente.

Considere o programa recursivo duplica (Figura 2.16) e a máquina um_reg (Figura 2.15). A correspondente função computada ⟨duplica, um_reg⟩: $\mathbb{N} \to \mathbb{N}$ é tal que, para qualquer $n \in \mathbb{N}$:

$$\langle \text{duplica, um\_reg} \rangle(n) = 2n$$

Suponha que existe um programa monolítico P_m que computa a mesma função, ou seja, que $\langle P_m, \text{um\_reg} \rangle$: $\mathbb{N} \to \mathbb{N}$ e:

$$\langle \text{duplica, um\_reg} \rangle = \langle P_m, \text{um\_reg} \rangle$$

Suponha que P_m é constituído de k operações ad. Suponha $n \in \mathbb{N}$ tal que $n \geq k$. Então, para que $\langle P_m, \text{um\_reg} \rangle(n) = 2n$, é necessário que P_m execute n vezes a operação ad. Mas, como $n \geq k$, então pelo menos uma das ocorrências de ad será executada mais de uma vez, ou seja, existe um ciclo em P_m. Neste caso, o ciclo será executado indefinidamente. Lembre-se que:

- para uma função computada por dois programas equivalentes fortemente, os mesmos testes e as mesmas operações são efetuados na mesma ordem;
- portanto, o programa monolítico correspondente não pode intercalar testes de controle de fim de ciclo na seqüência de operações ad (no programa recursivo, as operações ad não são intercaladas por qualquer outra operação ou teste).

Portanto, a computação resultante é infinita e a correspondente função não é definida para n, o que é um absurdo, pois é suposto que os dois programas são equivalentes fortemente.

Logo, não existe um programa monolítico equivalente fortemente ao programa recursivo duplica. ❑

Para melhor entender o resultado acima, considere o seguinte:

- um programa de qualquer tipo não pode ser modificado dinamicamente, durante uma computação;
- um programa para ser equivalente fortemente a outro, não pode conter ou usar facilidades adicionais como memória auxiliar ou operações ou testes extras;
- para que um programa monolítico possa simular uma recursão sem um número finito e predefinido de quantas vezes a recursão pode ocorrer, seriam necessárias infinitas opções de ocorrências das diversas operações ou testes envolvidos na recursão em questão;
- infinitas opções implicam um programa infinito, o que contradiz a definição de programa monolítico, o qual é constituído por um conjunto *finito* de instruções rotuladas.

Entretanto, sugere-se como exercício a análise do teorema acima para um caso especial e ilustrativo: um programa recursivo onde a recursão (direta ou indireta) ocorre somente na última componente de uma composição seqüencial. Note-se que, no contra-exemplo usado na prova, a recursão de R não ocorre na última componente, como pode ser observado a seguir:

R def (se zero então ✔ senão sub;R;ad;ad)

A prova do teorema que segue é análoga à do teorema acima.

Teorema 2.18 Equivalência Forte de Programas: Monolítico ↛ Iterativo.

Dado um programa monolítico P_m qualquer, não necessariamente existe um programa iterativo P_i, tal que $P_m \equiv P_i$.

Prova: (Por Absurdo).

Para provar é suficiente apresentar um programa monolítico que, para uma determinada máquina, não existe programa iterativo equivalente fortemente.

Considere o programa monolítico par na Figura 2.22 e a máquina um_reg (Figura 2.15) onde a correspondente função computada $\langle par, um\_reg \rangle$: $\mathbb{N} \rightarrow \mathbb{N}$ é tal que, para qualquer $n \in \mathbb{N}$:

$\langle par, um\_reg \rangle(n) = 1$, se n é par;
$\langle par, um\_reg \rangle(n) = 0$, se n é ímpar.

Ou seja, retorna o valor 1 sempre que a entrada é par e zero, caso contrário.

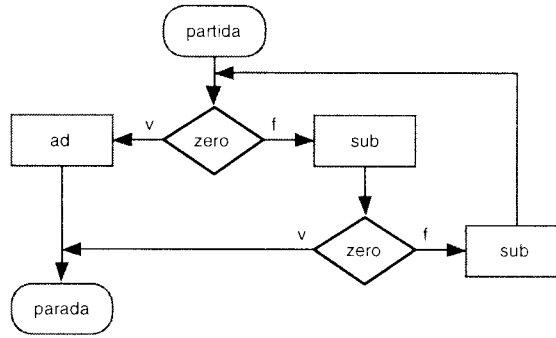


Figura 2.22 Programa monolítico

Suponha que existe um programa iterativo P_i que computa a mesma função, ou seja, que $\langle P_i, \text{um\_reg}\rangle$: $\mathbb{N} \to \mathbb{N}$ e:

$$\langle \text{par}, \text{um\_reg}\rangle = \langle P_i, \text{um\_reg}\rangle$$

Suponha que P_i é constituído de k operações sub. Suponha $n \in \mathbb{N}$ tal que $n \geq k$. Então, é necessário que P_i execute n vezes a operação sub. Mas, como $n \geq k$, então pelo menos uma das ocorrências de sub será executada mais de uma vez, ou seja, existe um ciclo iterativo (do tipo enquanto ou até) em P_i. Em qualquer caso, o ciclo terminará sempre na mesma condição, independentemente se o valor for par ou ímpar. Portanto, a computação resultante é incapaz de distinguir entre os dois casos, o que é um absurdo, pois é suposto que os dois programas são equivalentes fortemente.

Logo, não existe um programa iterativo equivalente fortemente ao programa monolítico par. ❑

Observação 2.19 Poder Computacional dos Diversos Tipos de Programas.

Os teoremas acima podem dar a falsa impressão de que o poder computacional da classe dos programas recursivos é maior que a dos monolíticos que, por sua vez, é maior que a dos iterativos. É importante destacar que consideram a Relação Equivalência Forte de Programas onde, para *qualquer* máquina, as correspondentes funções computadas devem coincidir. Entretanto, é importante constatar que as três classes de formalismos possuem o mesmo poder computacional, ou seja:

- para qualquer programa recursivo (respectivamente, monolítico) e para *qualquer* máquina, existe um programa monolítico (respectivamente, iterativo) e *existe* uma máquina tal que as correspondentes funções computadas coincidem.

Ou seja, para efeito de análise de poder computacional (tratada em capítulos subseqüentes), pode-se considerar máquinas distintas para programas distintos e não necessariamente existe uma relação entre as operações e testes (e a ordem de execução) dos programas. ❑

2.4.2 Equivalência de Programas

Eventualmente, pode ser desejado analisar a equivalência de programas em uma dada máquina, caracterizando uma noção de equivalência mais fraca que a apresentada até o momento.

Definição 2.20 Relação Equivalência de Programas em uma Máquina.

Sejam P e Q dois programas arbitrários, não necessariamente do mesmo tipo e uma máquina M qualquer. Então o par (P, Q) está na *Relação Equivalência de Programas na Máquina* M, denotado por:

$$P \equiv_M Q$$

se, e somente se, as correspondentes funções parciais computadas são iguais, ou seja:

$$\langle P, M \rangle = \langle Q, M \rangle$$

Neste caso, P e Q são ditos *Programas Equivalentes na Máquina* M, ou simplesmente *Programas* M-*Equivalentes*. ⌟

Existem máquinas nas quais não se pode provar a existência de um algoritmo para determinar se, dados dois programas, eles são ou não M-equivalentes. De fato, existem máquinas muito simples para as quais prova-se, este problema é não-solucionável. Este tópico será detalhado em outro capítulo.

2.4.3 Equivalência de Máquinas

Analogamente às equivalências de programas, pode-se estabelecer noções de equivalência de máquinas. Afirma-se que duas máquinas são equivalentes se uma pode simular a outra e vice-versa. Inicialmente é introduzido o conceito de simulação de máquinas.

Definição 2.21 Simulação Forte de Máquinas.

Sejam $M = (V_M, X, Y, \pi_{X_M}, \pi_{Y_M}, \Pi_{F_M}, \Pi_{T_M})$ e $N = (V_N, X, Y, \pi_{X_N}, \pi_{Y_N}, \Pi_{F_N}, \Pi_{T_N})$ duas máquinas arbitrárias. N *Simula Fortemente* M se, e somente se, para qualquer programa P para M, existe um programa Q para N tais que as correspondentes funções parciais computadas coincidem, ou seja:

$$\langle P, M \rangle = \langle Q, N \rangle$$ ⌟

Portanto, a simulação forte de uma máquina por outra pode ser feita usando programas diferentes. De fato, esta é a noção intuitiva de simulação de máquinas.

É importante observar que a igualdade de funções exige que os conjuntos de domínio e contradomínio sejam iguais. Portanto, é necessários que as duas máquinas possuam os mesmos conjuntos de valores de entrada e saída (observe

em detalhes na definição acima). Eventualmente, este fato pode ser um incômodo quando deseja-se comparar máquinas que determinam diferentes computações de entrada e saída, mas que estão relacionadas proximamente. Pode-se contornar esta dificuldade tornando menos restritiva a definição de simulação através da noção de codificações.

Definição 2.22 Simulação de Máquinas.

Sejam $M = (V_M, X_M, Y_M, \pi_{X_M}, \pi_{Y_M}, \Pi_{F_M}, \Pi_{T_M})$ e $N = (V_N, X_N, Y_N, \pi_{X_N}, \pi_{Y_N}, \Pi_{F_N}, \Pi_{T_N})$ duas máquinas arbitrárias. N *Simula* M se, e somente se, para qualquer programa P para M, existe um programa Q para N e existem:

Função de Codificação

$$c: X_M \rightarrow X_N$$

Função de Decodificação

$$d: Y_N \rightarrow Y_M$$

tais que:

$$\langle P, M\rangle = d \quad \langle Q, N\rangle \quad c$$ ❑

Definição 2.23 Relação Equivalência de Máquinas.

Sejam M e N duas máquinas arbitrárias. Então o par (M, N) está na *Relação Equivalência de Máquina*, se, e somente se:

$$M \text{ simula } N \quad \text{e} \quad N \text{ simula } M$$ ❑

2.5 Verificação da Equivalência Forte de Programas

Nesta seção, são descritas formas de se determinar se dois programas, sob determinadas condições, são ou não equivalentes fortemente. Uma vez que programas iterativos e monolíticos podem ser transformados em programas recursivos equivalentes fortemente, uma resposta satisfatória a essa questão poderia ser a obtenção de um método geral, aplicável a qualquer par de programas recursivos P e Q, o qual decidiria, em um número finito de passos, se $P \equiv Q$. Porém, como afirmado anteriormente, até o momento, não é conhecido se o problema da equivalência forte de programas recursivos é solucionável.

Na falta de um algoritmo genérico para programas recursivos, procura-se métodos para verificar a equivalência forte de classes mais simples de programas. De fato, mostra-se que existe um algoritmo para decidir a equivalência forte entre programas monolíticos. E, como todo o programa iterativo possui um programa monolítico equivalente fortemente, o mesmo pode ser afirmado para programas iterativos.

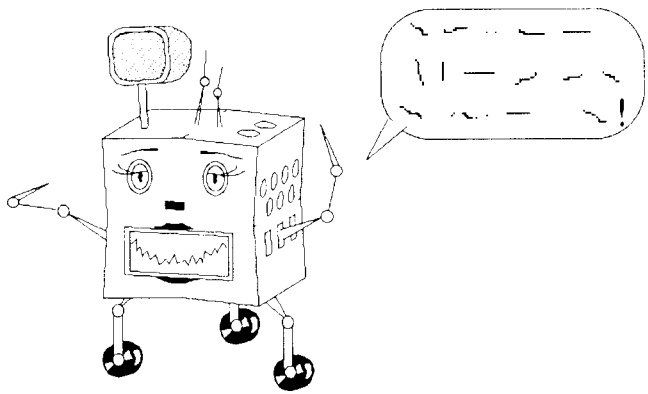
A verificação de que dois programas monolíticos são equivalentes fortemente usa os seguintes conceitos:

a) *Máquina de Traços.* Produz um rastro ou histórico (denominado *traço*) da ocorrência das operações do programa. Neste contexto, dois programas são equivalentes fortemente se são equivalentes em qualquer máquina de traços.

b) *Programa Monolítico com Instruções Rotuladas Compostas.* Instruções rotuladas compostas constituem uma forma alternativa de definir programas monolíticos. Basicamente, uma instrução rotulada composta é um teste da seguinte forma (compare com a instrução rotulada de teste):

```
r1: se T então faça F vá_para r2 senão faça G vá_para r3
```

De fato, usando máquinas de traços, é fácil verificar que, para um dado fluxograma é possível construir um programa equivalente fortemente usando instruções rotuladas compostas. Instruções rotuladas compostas induzem a noção de rótulos equivalentes fortemente a qual é usada para determinar se dois programas são ou não equivalentes fortemente.



2.5.1 Máquina de Traços

Uma máquina de traços não executa as operações propriamente ditas, mas apenas produz um histórico ou rastro da ocorrência destas, denominado de *traço*. Portanto, para computações finitas, um traço é uma palavra sobre um alfabeto de identificadores de operações. Essas máquinas são de grande importância para o estudo da equivalência de programas pois, como será verificado adiante, se dois programas são equivalentes em qualquer máquina de traços, então são equivalentes fortemente. A noção de traço também é importante no estudo dos modelos de concorrência e da semântica formal, os quais não são objetivos desta publicação. Uma boa referência sobre estes e outros aspectos referentes à noção de traço é [WIN95].

Definição 2.24 Máquina de Traços.

Uma *Máquina de Traços* é uma máquina:

$$M = (Op^*, Op^*, Op^*, id_{Op^*}, id_{Op^*}, \Pi_F, \Pi_T)$$

onde:

- Op^* *conjunto de palavras de operações* onde $Op = \{F, G, \ldots\}$ o qual corresponde, simultaneamente, aos conjuntos de valores de memória, entrada e saída;
- id_{Op^*} *função identidade* em Op^* a qual corresponde, simultaneamente, às funções de entrada e saída;
- Π_F *conjunto de interpretações de operações* onde, para cada identificador de operação F de Op, a interpretação π_F: $Op^* \rightarrow Op^*$ é tal que, para qualquer $w \in Op^*$, $\pi_F(w)$ resulta na concatenação do identificador F à direita de w, ou seja:

$$\pi_F(w) = wF$$

- Π_T *conjunto de interpretações de testes* tal que, para cada identificador de teste T:

$$\pi_T: Op^* \rightarrow \{\text{verdadeiro, falso}\} \text{ é função de } \Pi_T$$ ⌟

Portanto, o efeito de cada operação interpretada por uma máquina de traços é simplesmente o de acrescentar o identificador da operação à direita do valor atual da memória. O valor de saída da função computada consiste em um histórico das operações executadas durante a computação. Em suma, para definir uma máquina de traços, precisa-se apenas especificar as interpretações dos testes, pois as operações são predeterminadas.

Definição 2.25 Função Induzida por um Traço em uma Máquina.

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina, $Op = \{F, G, H, \ldots\}$ o conjunto de operações interpretadas em Π_F e $w = FG\ldots H$ um traço possível de M, ou seja, $w \in Op^*$. A *Função Induzida pelo Traço* w *na Máquina* M, denotada por:

$$[w, M]: X \rightarrow V$$

é a função (total):

$$[w, M] = \pi_H \ \ldots \ \pi_G \ \pi_F \ \pi_X$$

A função [w, M] aplicada a uma entrada $x \in X$ é denotada como segue:

$$[w_x, M] = \pi_H \ \ldots \ \pi_G \ \pi_F \ \pi_X(x)$$ ⌟

Portanto, a função induzida por um traço nada mais é do que a função resultante da composição das interpretações das diversas operações que constituem o traço. Note-se que a composição de funções é notada em ordem inversa da concatenação de símbolos no traço.

Teorema 2.26 Equivalência Forte de Programas $\longleftrightarrow$ Equivalência de Programas em Máquinas de Traços.

Sejam P e Q dois programas arbitrários, não necessariamente do mesmo tipo. Então:

$P \equiv Q$ se, e somente se, para qualquer máquina de traços M, $P \equiv_M Q$

Prova:

($\rightarrow$) É imediata, a partir da definição de equivalência (por quê?).

($\leftarrow$) A prova que segue é por absurdo e é para programas do tipo monolítico. Para os demais casos, a prova é similar e é sugerida como exercício.

Sejam P e Q programas monolíticos equivalentes em qualquer máquina de traços. Suponha que P e Q *não* são equivalentes fortemente. Então existe uma máquina $N = (V, X, Y, \pi_{X_N}, \pi_{Y_N}, \Pi_{F_N}, \Pi_{T_N})$ onde $Op = \{F, G, H, \ldots\}$ é o conjunto de operações interpretadas em Π_F, tal que $\langle P, N\rangle \neq \langle Q, N\rangle$. Então:

- seja $x \in X$ uma entrada tal que $\langle P, N\rangle(x) \neq \langle Q, N\rangle(x)$;
- seja $M = (Op^*, Op^*, Op^*, id_{Op^*}, id_{Op^*}, \Pi_{F_M}, \Pi_{T_M})$ uma Máquina e Traços tal que, para cada traço $w \in Op^*$ e para cada teste $T_N \in \Pi_{T_M}$:

$$T_M(w) = T_N([w_x, N])$$

onde $[w_x, N]$ é a função induzida pelo traço w na máquina N, aplicada à entrada x. Portanto, o teste de um histórico (traço) em M, é o correspondente teste em N, aplicado sobre o resultado da efetiva aplicação das funções que constituem o histórico. Lembre-se que, para definir uma máquina de traços, precisa-se apenas especificar as interpretações dos testes, pois as operações são predeterminadas. Portanto, M está completamente definida. Por simplicidade, no texto que segue a entrada x é omitida em $T_N([w_x, N])$ ou seja:

$$T_N([w_x, N]) \quad \text{é abreviado por} \quad T_N([w, N])$$

Para qualquer programa monolítico R, para qualquer entrada $x \in X$, tem-se que:

$\langle R, M\rangle(\varepsilon)$	resulta em um traço (M é uma máquina de traços)
$[\langle R, M\rangle(\varepsilon), N]$	é a função induzida por $\langle R, M\rangle(\varepsilon)$ em N para a entrada x

Neste contexto, prova-se que:

$$\langle R, N\rangle(x) = \pi_{Y_N}\ [\langle R, M\rangle(\varepsilon), N]$$

Assim, considerando-se que $\langle P, N\rangle(x) \neq \langle Q, N\rangle(x)$, tem-se que:

$\langle P, N\rangle(x) \neq \langle Q, N\rangle(x) \Rightarrow$

$\Rightarrow \pi_{Y_N}\ [\langle P, M\rangle(\varepsilon), N] \neq \pi_{Y_N}\ [\langle Q, M\rangle(\varepsilon), N] \Rightarrow$ $\quad \pi_{Y_N}$ é função

$\Rightarrow [\langle P, M\rangle(\varepsilon), N] \neq [\langle Q, M\rangle(\varepsilon), N] \Rightarrow$ $\quad$ (por quê?)

$\Rightarrow \langle P, M\rangle(\varepsilon) \neq \langle Q, M\rangle(\varepsilon)$

o que é um absurdo, pois foi suposto que os programas monolíticos P e Q são programas equivalentes em qualquer máquina de traços. Portanto, é absurdo supor que P e Q não são equivalentes fortemente.

Logo, P e Q são equivalentes fortemente.

A prova que, de fato, para qualquer programa monolítico $R = (I, r_0)$, para qualquer entrada $x \in X$, tem-se que $\langle R, N\rangle(x) = \pi_{Y_N}\ [\langle R, M\rangle(\varepsilon), N]$ é por indução número de pares que constituem as computações. No que segue, (r_k, v_k) e (m_k, w_k) representam pares das computações de $\langle R, N\rangle$ e $\pi_{Y_N}\ [\langle R, M\rangle(\varepsilon), N]$, respectivamente.

a) *Base de Indução.* Suponha $n = 0$. Então:

$r_0 = m_0$

$v_0 = \pi_{X_N}(x) = [\varepsilon, N] = [w_0, N]$

b) *Hipótese de Indução.* Suponha que, para algum $n \geq 0$ fixo, tem-se que:

$r_n = m_n$

$v_n = [w_n, N]$

c) *Passo de Indução.* Existem dois casos possíveis:

c.1) *Operação.* `r: faça F vá_para r'`

$r_{k+1} = m_{k+1} =$ `r'`

$v_{k+1} = \pi_{F_N}(v_k) = \pi_{F_N}([w_k, N]) = [w_kF, N] = [w_{k+1}, N]$

c.2) *Teste.* `r: se T então vá_para r' senão vá_para r"`

$r_{k+1} = m_{k+1}$ pois $T_M(w_k) = T_N([w_k, N]) = T_N(v_k)$

$v_{k+1} = v_k = [w_k, N] = [w_{k+1}, N]$

Portanto, $\langle R, N\rangle(x) = \pi_{Y_N}\ [\langle R, M\rangle(\varepsilon), N]$ ❑

A justificativa do seguinte corolário é sugerida como exercício:

Corolário 2.27 Equivalência Forte de Programas $\longleftrightarrow$ Equivalência de Programas em Máquinas de Traços.

Sejam P e Q dois programas arbitrários, não necessariamente do mesmo tipo. Então:

$P \equiv Q$ se, e somente se, para qualquer máquina de traços M, $\langle P, M\rangle(\varepsilon) = \langle Q, M\rangle(\varepsilon)$ ❑

2.5.2 Instruções Rotuladas Compostas

A verificação da equivalência forte de dois programas monolíticos pode ser mais facilmente realizada usando uma forma equivalente (fortemente) de representação baseada em conjuntos de instruções rotuladas compostas.

Instruções rotuladas compostas possuem somente uma única forma, ao contrário das instruções rotuladas as quais podem ser de duas formas: operação ou teste. De fato, uma instrução rotulada composta combina ambas em uma única forma.

Definição 2.28 Instrução Rotulada Composta.

Uma *Instrução Rotulada Composta* é uma seqüência de símbolos da seguinte forma (suponha que F e G são identificadores de operação e que T é um identificador de teste):

r_1: se T então faça F vá_para r_2 senão faça G vá_para r_3

Adicionalmente:

- r_2 e r_3 são ditos *rótulos sucessores* de r_1
- r_1 é dito *rótulo antecessor* de r_2 e r_3 ❑

Definição 2.29 Programa Monolítico com Instruções Rotuladas Compostas.

Um *Programa Monolítico com Instruções Rotuladas Compostas* P é um par ordenado

$$P = (I, r)$$

onde:

I *Conjunto de Instruções Rotuladas Compostas* o qual é finito;

r *Rótulo Inicial* o qual distingue a instrução rotulada inicial em I.

Adicionalmente, relativamente ao conjunto I tem-se que:

- não existem duas instruções diferentes com um mesmo rótulo;
- um rótulo referenciado por alguma instrução o qual *não* é associado a qualquer instrução rotulada é dito um *Rótulo Final*. ❑

Para simplificar o entendimento da determinação da equivalência forte de programas monolíticos, no que segue, é considerado somente o caso particular em que os programas possuem um *único* identificador de teste, denotado por T, como o representado na Figura 2.23. O caso geral (mais de um identificador de teste) pode ser facilmente estendido.

Considerando um único identificador de teste, uma instrução rotulada composta da forma:

r_1: se T então faça F vá_para r_2 senão faça G vá_para r_3

pode ser abreviada simplesmente por:

$$r_1: (F, r_2), (G, r_3)$$

A seguir, é apresentado um algoritmo para traduzir fluxogramas em instruções rotulas compostas. Para um melhor entendimento, sugere-se intercalar a leitura do algoritmo com o exemplo que o segue.

Definição 2.30 Algoritmo: Fluxograma → Rotuladas Compostas.

Os componentes elementares de partida, parada e operação de um fluxograma são genericamente denominados de *Nó*. O *Algoritmo para Traduzir um Fluxograma* P *para um Programa Monolítico* P' *Constituído por Instruções Rotuladas Compostas* é como segue:

a) *Rotulação de Nós*. Rotula-se cada nó do fluxograma. Suponha, sem perda de generalidade, que existe um único componente elementar de parada, ao qual é associado o identificador ε (palavra vazia). O rótulo correspondente ao nó partida é o Rótulo Inicial do programa P';

b) *Instruções Rotuladas Compostas*. A construção de uma instrução rotulada composta parte do nó partida e segue o caminho do fluxograma. Dependendo do próximo componente elementar, tem-se que:

   b.1) *Teste*. Para um teste como na Figura 2.23, a correspondente instrução rotulada composta é como segue:

$$r_1 : (F, r_2), (G, r_3)$$

   b.2) *Operação*. Para uma operação como na Figura 2.23, a correspondente instrução rotulada composta é como segue:

$$r_1 : (F, r_2), (F, r_2)$$

   b.3) *Parada*. Para uma parada da forma como na Figura 2.23, a correspondente instrução rotulada composta é como segue:

$$r : (\text{parada}, \varepsilon), (\text{parada}, \varepsilon)$$

   b.4) *Testes Encadeados*. No caso de testes encadeados como na Figura 2.23, segue-se o fluxo até que seja encontrado um nó, resultando na seguinte instrução rotulada composta:

$$r_1 : (F, r_2), (G, r_3)$$

   Analogamente para encadeamentos sucessivos de testes. Note-se que a ocorrência da operação H é impossível, pois, como é suposto que existe somente um identificador de teste, equivale a afirmar que T é uma contradição, ou seja, é simultaneamente verdadeira e falsa;

   b.5) *Testes Encadeados em Ciclo Infinito*. Para um ciclo infinito determinado por testes encadeados como na Figura 2.23, a correspondente instrução rotulada composta é como segue:

$$r_1 : (F, r_2), (\text{ciclo}, \omega)$$

   Neste caso, deve ser incluída, adicionalmente, uma instrução rotulada composta correspondente ao ciclo infinito, ou seja:

$$\omega : (\text{ciclo}, \omega), (\text{ciclo}, \omega)$$ ❑

No texto que segue, a rotulação dos nós de um fluxograma usa números naturais, fixando o número 1 para o nó de partida.

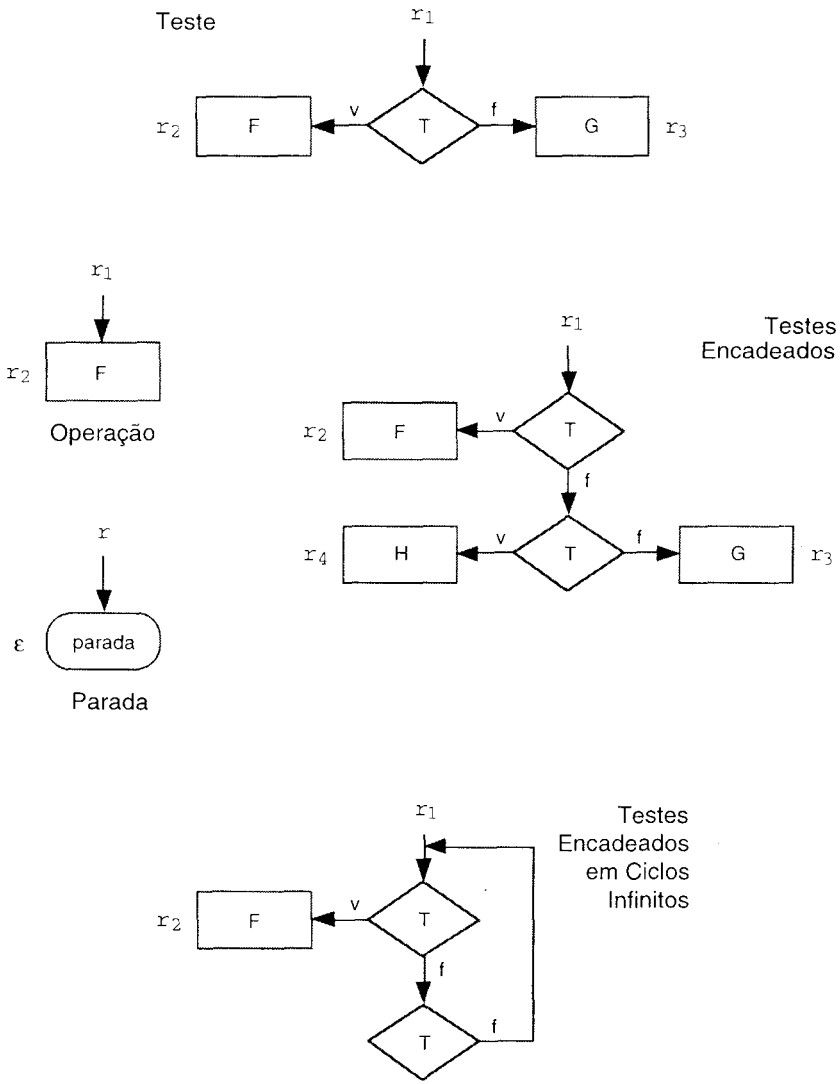


Figura 2.23 Instruções rotuladas compostas

EXEMPLO 2.18 Algoritmo: Fluxograma Rotuladas Compostas.

Considere o programa monolítico especificado na forma de fluxograma na Figura 2.24 cujos nós já estão rotulados. O correspondente programa com instruções rotuladas compostas é representado na Figura 2.25, supondo que 1 é o rótulo inicial. Note-se que:

- o rótulo 2 é sucessor dele mesmo. O mesmo ocorre com os rótulos 4, 7 e ω;
- existem dois caminhos no fluxograma que atingem nó parada. Entretanto, somente um é representado no conjunto de instruções rotuladas compostas. O segundo caminho é impossível, pois trata-se de uma contradição de T;
- na instrução rotulada por 7, ocorre um ciclo infinito (compare com o caso de ciclo infinito determinado por testes encadeados como na instrução rotulada por ω). ❑

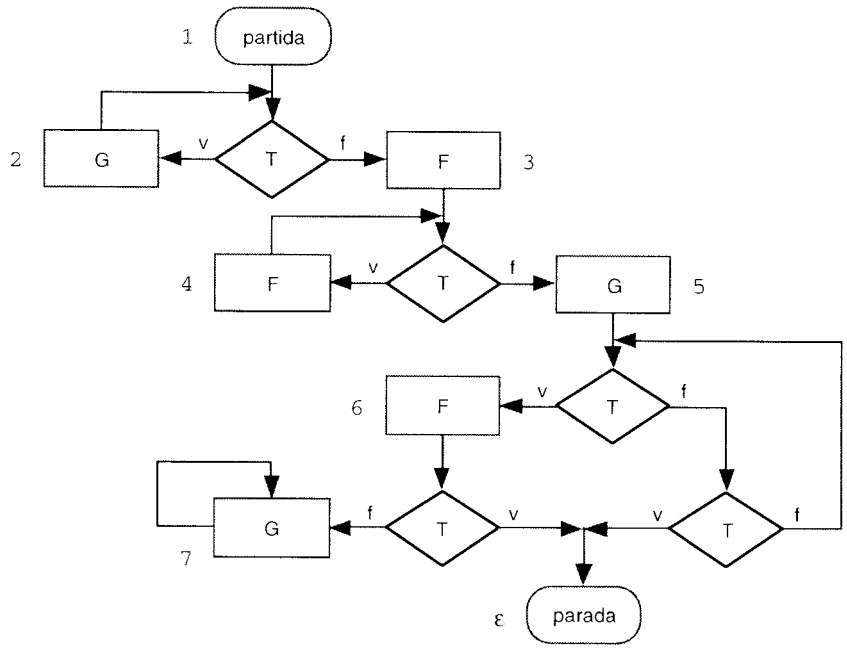


Figura 2.24 Fluxograma com os nós rotulados

```
1:    (G, 2), (F, 3)
2:    (G, 2), (F, 3)
3:    (F, 4), (G, 5)
4:    (F, 4), (G, 5)
5:    (F, 6), (ciclo, ω)
6:    (parada, ε), (G, 7)
7:    (G, 7), (G, 7)
ω:    (ciclo, ω), (ciclo, ω)
```

Figura 2.25 Conjunto de instruções rotuladas compostas

Relativamente aos programas especificados usando instruções rotuladas compostas, as noções de computação e função computada são análogas às dos programas monolíticos.

A seguir, verifica-se que, de fato, um fluxograma e o seu correspondente programa com instruções rotuladas compostas são equivalentes fortemente, excetuando-se para computações finitas, devido ao nó de parada.

Lema 2.31 Equivalência Forte: Fluxograma → Rotuladas Compostas.

Sejam P é um programa monolítico especificado na forma de fluxograma e P' = (I, 1) o seu correspondente programa construído usando o algoritmo de tradução de fluxograma para instruções rotuladas compostas, supondo o rótulo 1 associado ao nó de partida. Então, para qualquer máquina de traços M, tem-se que:

$$\langle P, M\rangle(\varepsilon) = \langle P', M\rangle(\varepsilon)$$

Prova: A prova é trivial (por quê?). Lembre-se que:

- uma máquina de traços simplesmente concatena identificadores de operações;
- a palavra vazia ε, correspondente ao rótulo de parada, é o elemento neutro da operação de concatenação;
- o identificador ciclo ocorre somente em computações infinitas, quando as correspondentes funções computadas são indefinidas. ❑

2.5.3 Equivalência Forte de Programas Monolíticos

Para introduzir a equivalência forte de programas monolíticos é necessário, antes, apresentar o conceito de união disjunta. A *união disjunta* de conjuntos garante que todos os elementos dos conjuntos componentes constituem o conjunto resultante, mesmo que possuam a mesma identificação. Neste caso, considera-se que os elementos são distintos, mesmo que possuam a mesma identificação. Por exemplo, para os conjuntos A = {a, x} e B = {b, x}, o conjunto resultante da união disjunta é:

$$\{a_A, x_A, b_B, x_B\}$$

No exemplo, o índice é usado para assegurar que as identificações dos elementos são distintas. Por simplicidade, quando a identificação for única, o índice pode ser omitido. Assim, conjunto resultante do exemplo acima fica como segue:

$$\{a, x_A, b, x_B\}$$

A verificação da equivalência forte de programas monolíticos é baseada no seguinte corolário sobre união disjunta o qual é decorrência direta do Lema 2.31 - Equivalência Forte: Fluxograma → Rotuladas Compostas.

Corolário 2.32 Equivalência Forte: União Disjunta.

Sejam $Q = (I_Q, q)$ e $R = (I_R, r)$ dois programa monolíticos especificados usando instruções rotuladas compostas e sejam $P_q = (I, q)$ e $P_r = (I, r)$ programas monolíticos onde I é o conjunto resultante da união disjunta de I_Q e I_R. Então:

$$P_q \equiv P_r \text{ se, e somente se, } Q \equiv R$$ ❑

Assim, o algoritmo para verificação da equivalência forte de Q e R resume-se a verificação se P_q e P_r são equivalentes fortemente. Entretanto, para desenvolver o algoritmo, é necessário considerar:

- *cadeia de conjunto*: seqüência de conjuntos ordenada pela relação de inclusão;
- *programa monolítico simplificado*: instruções rotuladas compostas que determinam ciclos infinitos são excluídas (excetuando-se a instrução rotulada por ω, se existir);
- *rótulos equivalentes fortemente*: o algoritmo de verificação se P_q e P_r são equivalentes fortemente baseia-se em rótulos consistentes.

Definição 2.33 Cadeia de Conjuntos, Cadeia Finita de Conjuntos, Limite de uma Cadeia Finita de Conjuntos.

Uma seqüência de conjuntos $A_0A_1\ldots$ é dita:

a) Uma *Cadeia de Conjuntos* se, para qualquer $k \geq 0$,

$$A_k \subseteq A_{k+1}$$

b) Uma *Cadeia Finita de Conjuntos* é uma cadeia de conjuntos onde existe n, para todo $k \geq 0$, tal que:

$$A_n = A_{n+k}$$

Neste caso, define-se o *Limite da Cadeia Finita de Conjuntos* como segue:

$$\lim A_k = A_n$$ ❑

A lema a seguir fornece um algoritmo para determinar se existem ciclos infinitos em um conjunto de instruções rotuladas compostas. A idéia básica é partir da instrução parada, rotulada por ε, determinando os seus antecessores. Por exclusão, uma instrução que não é antecessora da parada determina um ciclo infinito. As provas dos lemas e teorema que seguem são omitidas e podem ser encontradas em [BIR76].

Lema 2.34 Identificação de Ciclos Infinitos em Programa Monolítico.

Seja I é um conjunto de n instruções rotuladas compostas. Seja $A_0A_1\ldots$ uma seqüência de conjuntos de rótulos indutivamente definida como segue:

$A_0 = \{\varepsilon\}$

$A_{k+1} = A_k \cup \{r \mid r$ é rótulo de instrução antecessora de alguma instrução rotulada por $A_k\}$

Então $A_0A_1\ldots$ é uma cadeia finita de conjuntos e, para qualquer rótulo r de instrução de I, tem-se que

$$(I, r) \equiv (I, \omega) \quad \text{se, e somente se,} \quad r \notin \lim A_k$$ ❑

O lema acima proporciona uma maneira fácil de determinar se algum rótulo caracteriza ciclos infinitos.

EXEMPLO 2.19 Identificação de Ciclos Infinitos em Programa Monolítico.

Considere o conjunto I de instruções rotuladas compostas representado na Figura 2.25. A correspondente cadeia finita de conjuntos é como segue:

$A_0 = \{\varepsilon\}$
$A_1 = \{6, \varepsilon\}$
$A_2 = \{5, 6, \varepsilon\}$
$A_3 = \{3, 4, 5, 6, \varepsilon\}$
$A_4 = \{1, 2, 3, 4, 5, 6, \varepsilon\}$
$A_5 = \{1, 2, 3, 4, 5, 6, \varepsilon\}$

Logo:

$\lim A_k = \{1, 2, 3, 4, 5, 6, \varepsilon\}$
$(I, 7) \equiv (I, \omega)$, pois $7 \notin \lim A_k$ ❑

Portanto, pode-se simplificar um conjunto de instruções rotuladas compostas eliminando qualquer instrução de rótulo $r \neq \omega$ que determine um ciclo infinito.

Definição 2.35 Algoritmo de Simplificação de Ciclos Infinitos.

Seja I um conjunto finito de instruções rotulas compostas. O *Algoritmo de Simplificação de Ciclos Infinitos* é como segue:

a) Determina-se a correspondente cadeia finita de conjuntos $A_0A_1\ldots$ como no lema acima;

b) Para qualquer rótulo r de instrução de I tal que $r \notin \lim A_k$, tem-se que:
- a instrução rotulada por r é excluída
- toda referência a pares da forma (F, r) em I é substituída por (ciclo, ω)
- $I = I \cup \{\omega: (\text{ciclo}, \omega), (\text{ciclo}, \omega)\}$ ❑

EXEMPLO 2.20 Simplificação de Ciclos Infinitos em Programa Monolítico.

Considere conjunto de instruções rotuladas compostas representado na Figura 2.25 o qual corresponde ao fluxograma representado na Figura 2.24. O correspondente conjunto de instruções rotuladas compostas simplificadas é representado na Figura 2.26. ❑

```
1:      (G, 2), (F, 3)
2:      (G, 2), (F, 3)
3:      (F, 4), (G, 5)
4:      (F, 4), (G, 5)
5:      (F, 6), (ciclo, ω)
6:      (parada, ε), (ciclo, ω)
ω:      (ciclo, ω), (ciclo, ω)
```

Figura 2.26 Conjunto de instruções rotuladas compostas e simplificadas

Lema 2.36 Rótulos Consistentes.

Seja I um conjunto finito de instruções rotuladas compostas e simplificadas. Sejam r e s dois rótulos de instruções de I, ambos diferentes de ε. Suponha que as instruções rotuladas por r e s são da seguinte forma, respectivamente:

r: $(F_1, r_1), (F_2, r_2)$
s: $(G_1, s_1), (G_2, s_2)$

Então, r e s são *rótulos consistentes* se, e somente se:

$$F_1 = G_1 \quad \text{e} \quad F_2 = G_2$$ ❑

O lema acima induz a seguinte definição.

Definição 2.37 Rótulos Equivalentes Fortemente.

Seja I um conjunto finito de instruções rotuladas compostas e simplificadas. Sejam r e s dois rótulos de instruções de I. Então, r e s são *Rótulos Equivalentes Fortemente* se, e somente, se:

- ou r = s = ε;
- ou r e s são ambos diferentes de ε e consistentes. ❑

Teorema 2.38 Determinação de Rótulos Equivalentes Fortemente.

Seja I um conjunto de n instruções compostas e simplificadas. Sejam r e s dois rótulos de instruções de I. Define-se, indutivamente, a seqüência de conjuntos $B_0B_1\ldots$ como segue:

$B_0 = \{(r, s)\}$
$B_{k+1} = \{(r'', s'') \mid r''$ e s'' são rótulos sucessores de r' e s', respectivamente, $(r', s') \in B_k$ e $(r'', s'') \notin B_i, i \in \{0, 1, \ldots, k\}\}$

Então $B_0B_1\ldots$ é uma seqüência de conjuntos que converge para o conjunto vazio e r, s são rótulos equivalentes fortemente se, e somente se, qualquer par de B_k é constituído por rótulos consistentes. ❑

O seguinte algoritmo é induzido pelo teorema acima bem como pelo Corolário 2.32 - Equivalência Forte: União Disjunta.

Definição 2.39 Algoritmo de Equivalência Forte de Programas Monolíticos.

Sejam Q = (I_Q, q) e R = (I_R, r) dois programas monolíticos especificados usando instruções rotuladas compostas e simplificados. O *Algoritmo de Equivalência Forte de Programas Monolíticos* Q *e* R é como determinado pelos seguintes passos:

Passo 1. Sejam $P_q = (I, q)$ e $P_r = (I, r)$ programas monolíticos onde I é o conjunto resultante da união disjunta de I_Q e I_R, excetuando-se a instrução rotulada ω, se existir, a qual ocorre, no máximo, uma vez em I. Lembre-se que $P_q \equiv P_r$ se, e somente se, Q ≡ R;

Passo 2. Se q e r são rótulos equivalentes fortemente, então $B_0 = \{(q, r)\}$. Caso contrário, Q e R *não são equivalentes fortemente* e o algoritmo termina;

Passo 3. Para $k \geq 0$, define-se o conjunto B_{k+1}, contendo somente os pares (q", r") de rótulos sucessores de cada (q', r') $\in B_k$, tais que:

- q' ≠ r'
- q' e r' são ambos diferentes de ε
- os pares sucessores (q", r") não são elementos de $B_0, B_1, \ldots, B_k$

Passo 4. Dependendo de B_{k+1}, tem-se que:

a) $B_{k+1} = \emptyset$: Q *e* R *são equivalentes fortemente* e o algoritmo termina;

b) $B_{k+1} \neq \emptyset$: se todos os pares de rótulos de B_{k+1} são equivalentes fortemente, então vá para o *Passo 3*; caso contrário, Q e R *não são equivalentes fortemente* e o algoritmo termina. ❑

Note-se que, no algoritmo acima, a construção dos conjuntos referida no Teorema 2.38 - Determinação de Rótulos Equivalentes Fortemente é implícita.

EXEMPLO 2.21 Algoritmo de Equivalência Forte de Programas Monolíticos.

Considere os programas monolíticos Q e R especificados na forma de fluxograma representados nas Figura 2.24 e Figura 2.27, respectivamente. Relativamente aos pré-requisitos do algoritmo, tem-se que:

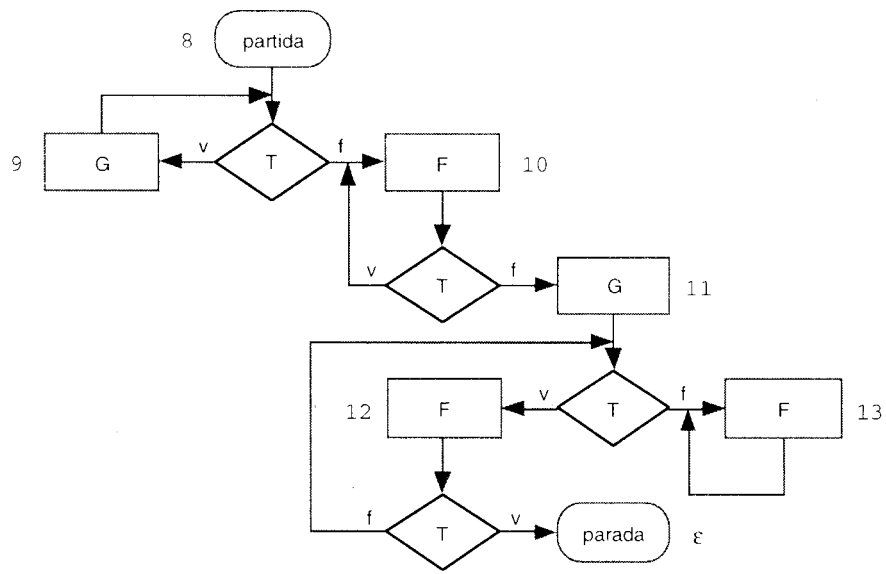


Figura 2.27 Fluxograma

a) A especificação do programa Q usando instruções rotuladas compostas, já simplificado, é representado na Figura 2.26.

b) Em relação ao programa R, tem-se que:

b.1) *Conjunto de instruções rotuladas compostas.*

```
8:    (G, 9), (F, 10)
9:    (G, 9), (F, 10)
10:   (F, 10), (G, 11)
11:   (F, 12), (F, 13)
12:   (parada, ε), (F, 13)
13:   (F, 13), (F, 13)
```

b.2) *Identificação de ciclos infinitos.*

$A_0 = \{\varepsilon\}$
$A_1 = \{12, \varepsilon\}$
$A_2 = \{11, 12, \varepsilon\}$
$A_3 = \{10, 11, 12, \varepsilon\}$
$A_4 = \{8, 9, 10, 11, 12, \varepsilon\}$
$A_5 = \{8, 9, 10, 11, 12, \varepsilon\}$

Portanto:

$\lim A_k = \{8, 9, 10, 11, 12, \varepsilon\}$
$(I_R, 13) \equiv (I, \omega)$, pois $13 \notin \lim A_k$

b.3) *Simplificação de ciclos infinitos.*

```
8:    (G, 9), (F, 10)
9:    (G, 9), (F, 10)
10:   (F, 10), (G, 11)
11:   (F, 12), (ciclo, ω)
12:   (parada, ε), (ciclo, ω)
ω:    (ciclo, ω), (ciclo, ω)
```

Relativamente à aplicação do algoritmo, tem-se que:

Passo 1. Seja I a união disjunta dos conjuntos I_Q e I_R, excetuando-se a instrução rotulada ω, como segue:

```
1:    (G, 2), (F, 3)
2:    (G, 2), (F, 3)
3:    (F, 4), (G, 5)
4:    (F, 4), (G, 5)
5:    (F, 6), (ciclo, ω)
6:    (parada, ε), (ciclo, ω)
8:    (G, 9), (F, 10)
9:    (G, 9), (F, 10)
10:   (F, 10), (G, 11)
11:   (F, 12), (ciclo, ω)
12:   (parada, ε), (ciclo, ω)
ω:    (ciclo, ω), (ciclo, ω)
```

Para verificar se Q ≡ R é suficiente verificar se (I, 1) ≡ (I, 8).

Passo 2. Como 1 e 8 são rótulos equivalentes fortemente, então:

$B_0 = \{(1, 8)\}$

Passos 3 e 4. Para $k \geq 0$, construção de B_{k+1} é como segue:

$B_1 = \{(2, 9), (3, 10)\}$ pares de rótulos equivalentes fortemente
$B_2 = \{(4, 10), (5, 11)\}$ pares de rótulos equivalentes fortemente
$B_3 = \{(6, 12), (\omega, \omega)\}$ pares de rótulos equivalentes fortemente
$B_4 = \{(\varepsilon, \varepsilon)\}$ pares de rótulos equivalentes fortemente
$B_5 = \varnothing$

Logo (I, 1) ≡ (I, 8) e, portanto, Q ≡ R ❑

EXEMPLO 2.22 Algoritmo de Equivalência Forte de Programas Monolíticos.

Foi afirmado anteriormente que os programas monolíticos (fluxogramas) representados na Figura 2.18, reproduzidos novamente na Figura 2.28 com os nós rotulados, são equivalentes fortemente. De fato, relativamente aos pré-requisitos do algoritmo, tem-se que:

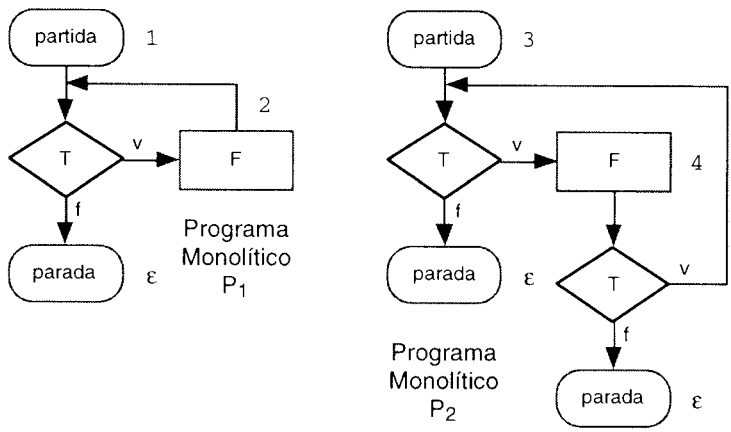


Figura 2.28 Fluxogramas equivalentes fortemente

a) A especificação de P_1 usando instruções rotuladas compostas (e já simplificado) é como segue:

```
1:    (F, 2), (parada, ε)
2:    (F, 2), (parada, ε)
```

b) A especificação de P_2 usando instruções rotuladas compostas (e já simplificado) é como segue:

```
3:    (F, 4), (parada, ε)
2:    (F, 4), (parada, ε)
```

Portanto, os correspondentes conjuntos de instruções rotuladas compostas são iguais, a menos dos rótulos.

Relativamente à aplicação do algoritmo, é fácil de verificar que (I, 1) ≡ (I, 3). Logo, também como afirmado anteriormente, a relação equivalente fortemente fornece subsídios para analisar a complexidade estrutural de programas. No caso, P_1 é estruturalmente "mais otimizado" que P_2. ❑

2.6 Conclusão

Neste capítulo, são introduzidos os conceitos de programa e máquina os quais são usados para construir as definições de computação e função computada. Em particular, estuda-se três tipos de programas: monolítico, iterativo e recursivo.

Baseadas em função computada as seguintes noções de equivalência de programas e máquinas são apresentadas: programas equivalentes fortemente, programas equivalentes (em uma máquina) e máquinas equivalentes.

A relação programas equivalentes fortemente induz uma hierarquia de tipos de programa: recursivos são mais gerais que os monolíticos os quais, por sua vez, são mais gerais que os iterativos. Adicionalmente, mostra-se a existência de um algoritmo para verificar se programas monolíticos (ou iterativos) são equivalentes fortemente. Entretanto, até o momento, não é conhecido se existe um algoritmo análogo para programas recursivos.

No próximo capítulo, é visto o conceito de máquina universal, a caracterização do que é computável e a correspondência entre os diferentes formalismos de máquinas e computações.

A relação entre os conceitos deste e demais capítulos é como na Figura 2.29.

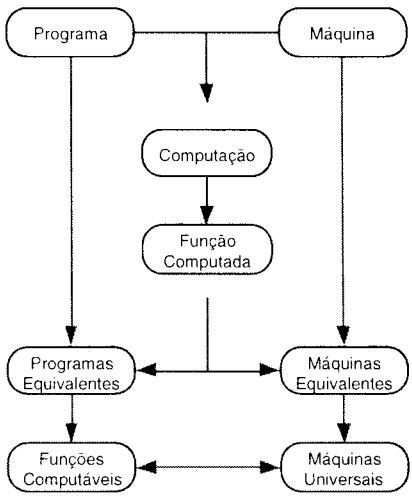


Figura 2.29 Relação entre os conceitos deste e demais capítulos

2.7 Exercícios

Exercício 2.1 Identifique e compare construções análogas às usadas nas definições de programas monolítico, iterativo e recursivo em linguagens de programação como: Pascal, C ou outra de seu conhecimento.

Exercício 2.2 Desenhe um fluxograma que corresponde a cada um dos seguintes programas

a) $P_2 = (\{r_1:$ faça ✔ vá_para $r_2\}, r_1)$

b) Composição até (programa iterativo);

c) Programa sem instrução alguma;

d) Programa sem instrução de parada.

Exercício 2.3 Relativamente a programas iterativos:

a) Em que situação a execução de:

enquanto T faça V

V pode não ser executado?

b) Por que a operação vazia ✔ constitui um programa iterativo?

c) Por que pode-se afirmar que:

a tradução de um programa iterativo para um monolítico é trivial?

Exercício 2.4 Relativamente a computação:

a) Por que é possível afirmar que a computação de um programa monolítico em uma máquina, para um dado valor inicial de memória, é determinística?

b) Analogamente para um programa iterativo?

c) Analogamente para um programa recursivo?

Exercício 2.5 Caracterize e diferencie computação e função computada.

Exercício 2.6 Defina computação de programas iterativos em uma máquina.

Exercício 2.7 Dê a definição formal da função computada ⟨W, M⟩ de um programa iterativo W em uma máquina M.

Exercício 2.8 Defina computação finita para programas iterativos.

Exercício 2.9 Escreva um programa iterativo onde a computação seja infinita.

Exercício 2.10 Relativamente à função computada por um programa em uma máquina:

a) Considere o programa monolítico mon_b←a (Figura 2.10) para a máquina dois_reg (Figura 2.7). A correspondente função computada ⟨mon_b←a, dois_reg⟩ é total?

b) Considere o programa monolítico comp_infinita (Figura 2.12) para a máquina dois_reg (Figura 2.7). Para quais valores do domínio a função computada ⟨comp_infinita, dois_reg⟩ é definida?

c) Considere o programa recursivo qq_máquina (Figura 2.14) e uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ qualquer. Por que a correspondente função computada ⟨qq_máquina, M⟩ é indefinida para qualquer entrada?

Exercício 2.11 Relativamente aos seguintes corolários e teoremas:

a) Corolário 2.27 - Equivalência Forte de Programas ↔ Equivalência de Programas em Máquinas de Traços.

   a.1) Justifique a afirmação de que a prova (→) é imediata;

   a.2) Esboce a prova (←) para programas iterativo e recursivos;

b) Justifique a afirmação de que a prova do Lema 2.31 - Equivalência Forte: Fluxograma → Rotuladas Compostas é imediata;

c) Por quê o Lema 2.31 - Equivalência Forte: Fluxograma → Rotuladas Compostas garante que $P_q \equiv P_r$ se, e somente se, $Q \equiv R$?

d) Justifique o Corolário 2.27 - Equivalência Forte de Programas ↔ Equivalência de Programas em Máquinas de Traços.

Exercício 2.12 Traduza os programas monolíticos representados por fluxogramas em programas recursivos e simplifique se possível.

a) Fluxograma 1 representado na Figura 2.30;

b) Fluxograma 2 representado na Figura 2.31;

c) Fluxograma 3 representado na Figura 2.32;

d) Fluxograma 4 representado na Figura 2.33;

e) Fluxograma 5 representado na Figura 2.34.

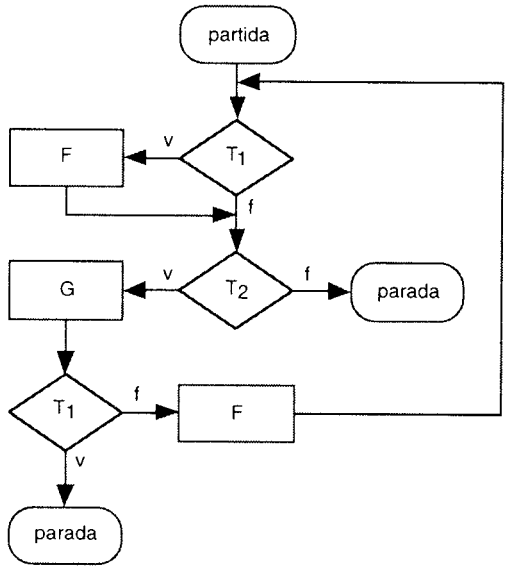


Figura 2.30 Fluxograma 1

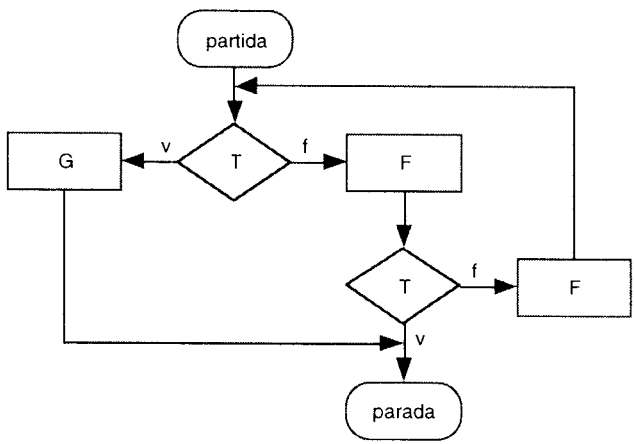


Figura 2.31 Fluxograma 2

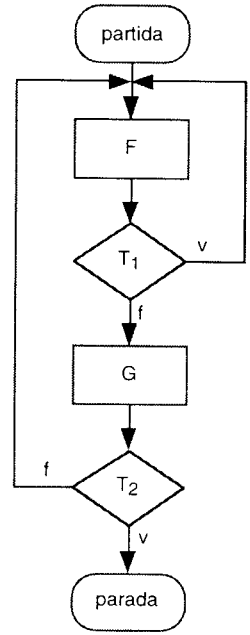


Figura 2.32 Fluxograma 3

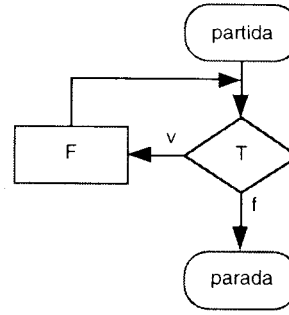


Figura 2.33 Fluxograma 4

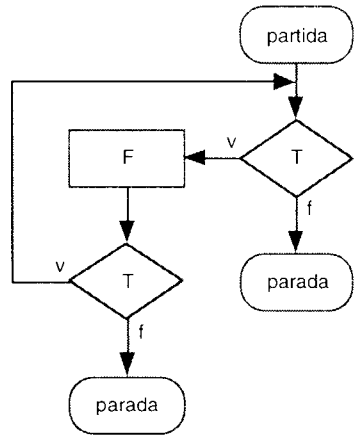


Figura 2.34 Fluxograma 5

Exercício 2.13 Traduza o programa iterativo representado na Figura 2.35 em programa monolítico, nas formas de :

a) Fluxograma;

b) Instruções rotuladas.

```
(se    T1
então enquanto T2
      faça  (até  T3
            faça  (V;W))
senão (✔))
```

Figura 2.35 Programa iterativo

Exercício 2.14 Traduza o programa recursivo representado na Figura 2.36 em programa iterativo.

```
P é R1 onde
      R1 def (se T então F;R2 senão R1),
      R2 def G; (se T então F;R1 senão ✔)
```

Figura 2.36 Programa recursivo

Exercício 2.15 Suponha que se escreva $M_1 \leq M_2$ se a máquina M_2 simula M_1, mas M_1 pode ter na sua definição outros testes e operações a mais. Qual é a relação entre $\langle P, M_1 \rangle$ e $\langle P', M_2 \rangle$ se $M_1 \leq M_2$? Qual a relação do poder computacional da máquina M_1 em relação a máquina M_2?

Exercício 2.16 Suponha que na definição de uma máquina, seja admitido funções parciais na especificação dos identificadores de operações e testes. Dê a definição apropriada da função computada por um programa em uma máquina em tal caso.

Exercício 2.17 Mostre que os seguintes programas P e Q representados na Figura 2.37 são equivalentes.

Programa Iterativo P

```
até   T
faça  (✔);
enquanto T
faça  (F;G;( se     T
              então F;
                    até     T
                    faça (✔)
              senão ✔))
```

Programa Monolítico Q

```
1:   se T então vá_para 2 senão vá_para 1
2:   faça F vá_para 3
3:   faça G vá_para 4
4:   se T então vá_para 5 senão vá_para 6
5:   faça F vá_para 1
```

Figura 2.37 Programas iterativo (acima) e monolítico (abaixo)

Exercício 2.18 Verifique se os programas monolíticos M_1 e M_2 representados na Figura 2.38 são equivalentes fortemente.

Programa Monolítico M_1

```
1:   faça F vá_para 2
2:   se T então vá_para 3 senão vá_para 5
3:   faça G vá_para 4
4:   se T então vá_para 1 senão vá_para 0
5:   faça F vá_para 6
6:   se T então vá_para 7 senão vá_para 2
7:   faça G vá_para 8
8:   se T então vá_para 6 senão vá_para 0
```

Programa Monolítico M_2

```
1:   faça F vá_para 2
2:   se T então vá_para 3 senão vá_para 1
3:   faça G vá_para 4
4:   se T então vá_para 1 senão vá_para 0
```

Figura 2.38 Programas monolíticos

Exercício 2.19 Qual a importância da relação Equivalência Forte de Programas?

Exercício 2.20 Verifique se os programas iterativos W_1 e W_2 definidos na Figura 2.39 e Figura 2.40, respectivamente, são equivalentes fortemente.

Programa Iterativo W_1

```
enquanto T
faça  (F;(se T então faça ✔ senão faça G))
```

Figura 2.39 Programa iterativo

Programa Iterativo W_2

```
enquanto T
faça  (F;enquanto T faça (F);G)
```

Figura 2.40 Programa iterativo

Exercício 2.21 Traduza os programas iterativos W_1 e W_2 definidos na Figura 2.39 e na Figura 2.40 para programas recursivos.

Exercício 2.22 Traduza o programa monolítico da Figura 2.41 na forma de instruções rotuladas compostas. Como existem dois testes, cada instrução rotulada composta terá quatro possíveis sucessores, um para cada possível combinação de valores-verdade dos testes T_1 e T_2.

Programa Monolítico Dois_Testes

```
1:    faça F vá_para 2
2:    se T1 então vá_para 1 senão vá_para 3
3:    faça G vá_para 4
4:    se T2 então vá_para 0 senão vá_para 1
```

Figura 2.41 Programa Monolítico

Exercício 2.23 Adapte para o caso do programa monolítico da Figura 2.41, os seguintes itens:

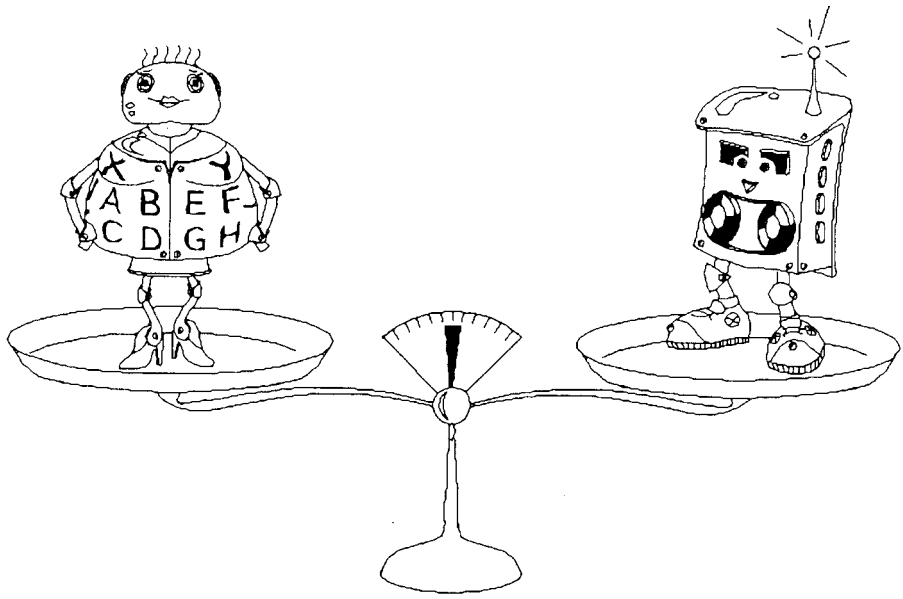
a) Lema 2.34 - Identificação de Ciclos Infinitos em Programa Monolítico;

b) Teorema 2.38 - Determinação de Rótulos Equivalentes Fortemente;

c) Definição 2.39 - Algoritmo de Equivalência Forte de Programas Monolíticos;

Exercício 2.24 Generalize a definição de instruções rotuladas compostas para o caso de três testes distintos.

Exercício 2.25 Traduza os fluxogramas da Figura 2.30 e da Figura 2.32 em instruções rotuladas compostas.



3 Máquinas Universais

Até o momento, o termo *algoritmo* foi intuitivamente usado como solução de um problema, ou seja, como uma forma de descrever se determinada propriedade é verificada ou não para uma dada classe de entrada. Portanto, a investigação da solucionabilidade de um problema é a investigação da existência de um algoritmo capaz de resolvê-lo. Entretanto, se a noção algoritmo é intuitiva (não-formal), qualquer afirmação sobre a não-solucionabilidade de determinado problema é questionável.

Relativamente à noção intuitiva de algoritmo, o seguinte pode ser afirmado:

- sua descrição deve ser finita e não-ambígua;
- deve consistir de passos discretos, executáveis mecanicamente em um tempo finito.

Limitações de tempo ou de espaço podem, eventualmente, determinar se um algoritmo pode ou não ser utilizado na prática. Entretanto, estas não são restrições teóricas pois a inexistência de limitações *não* implica recursos ou descrições infinitas. Assim, recursos de tempo e espaço são "tanto quanto necessários". O correto entendimento dessa observação é de fundamental importância para todo o estudo que segue.

Considerando que um algoritmo deve possuir uma descrição finita, alguns tipos de dados podem não satisfazer tal condição como, por exemplo, os número irracionais, onde qualquer descrição finita de seus valores constitui apenas uma aproximação. O número π é um exemplo bem conhecido que ilustra esta afirmação. Assim, no que segue, o estudo é restrito aos algoritmos naturais, ou seja, definidos sobre o conjunto dos números naturais.

Na realidade, qualquer *conjunto contável* (existe uma bijeção com $\mathbb{N}$) poderia ser equivalentemente considerado. Mesmo para os tipos de dados cujos valores possuem descrição finita, uma grande variedade de tipos deve ser considerada como, por exemplo, inteiros, cadeia de caracteres, valores-verdade, etc., bem como tipos baseados em conjuntos estruturados como vetores. Adiante, é exemplificado como alguns destes tipos de dados podem ser codificados como naturais.

O conceito de programa, como introduzido anteriormente, satisfaz à noção intuitiva de algoritmo. Entretanto, neste caso, era necessário definir a máquina a ser considerada. Tal máquina deveria ser suficientemente:

- *simples*, para permitir estudos de propriedades sem a necessidade de considerar características não-relevantes, bem como permitir estabelecer conclusões gerais sobre a classe de funções computáveis;
- *poderosa*, capaz de simular qualquer característica de máquinas reais ou teóricas, de tal forma que os resultados provados sejam válidos para modelos aparentemente com mais recursos e para que qualquer função computável possa ser nela representada.

Se for possível representar qualquer algoritmo como um programa em tal máquina, então esta é denominada de *Máquina Universal*. As evidências de que uma máquina é, de fato, universal, podem ser classificadas como:

a) *Evidência Interna*. Consiste na demonstração de que qualquer extensão das capacidades da máquina proposta computa, no máximo, a mesma classe de funções, ou seja, não aumenta o seu poder computacional;

b) *Evidência Externa*. Consiste no exame de outros modelos que definem a noção de algoritmo, juntamente com a prova de que são, no máximo, computacionalmente equivalentes.

Neste contexto, é introduzida uma Máquina Universal, denominada *Máquina Norma*, proposta por Richard Bird, a qual, resumidamente, possui um conjunto de registradores naturais e somente três instruções sobre os registradores: adição e subtração do valor um e o teste se o valor armazenado é zero. Trata-se de um exemplo de *Máquina de Registradores*. Como ilustração, diversas características de máquinas reais são simuladas usando a Máquina Norma, reforçando as evidências internas e externas de que, de fato, trata-se de uma Máquina Universal.

Provavelmente, o modelo mais utilizado como formalização de algoritmo é a *Máquina de Turing*, proposta em 1936 por Alan Turing. Basicamente, uma Máquina de Turing é um mecanismo simples que formaliza a idéia de uma pessoa que realiza cálculos, usando um instrumento de escrita e um apagador. O modelo formal é baseado em uma *fita* (usada para entrada, saída e rascunho), uma unidade de controle e um programa, de forma muito similar aos atuais computadores, embora tenha sido proposto aproximadamente 20 anos antes do primeiro computador digital. Na realidade, não se trata de uma máquina no sentido dado a esta palavra, mas sim de um programa para uma Máquina Universal.

No que se refere aos recursos "tanto quanto necessários", tem-se que:

- os registradores da Máquina Norma podem assumir "qualquer valor natural tão grande quanto necessário". Adicionalmente, existem "tantos registradores quanto necessários";
- a Máquina de Turing possui "tantas células de armazenamento de dados quanto necessário".

Em 1936, Alonzo Church apresentou a *Hipótese de Church*, a qual afirma que qualquer função computável pode ser processada por uma Máquina de Turing, ou seja, que existe um algoritmo expresso na forma de Máquina de Turing capaz de processar a função. Contudo, como a noção intuitiva de algoritmo não é matematicamente precisa, é impossível formalizar uma demonstração de que a Máquina de Turing é, efetivamente, o mais genérico dispositivo de computação. Entretanto, todas as evidências internas e externas imaginadas foram sempre verificadas, reforçando a Hipótese de Church, ou seja, que os demais modelos de máquinas propostos, bem como qualquer extensão de suas capacidades, possuem, no máximo, a mesma capacidade computacional da Máquina Turing. Em particular, é verificado, neste capítulo, que os seguintes modelos são equivalentes à Máquina de Turing (note o tipo de estrutura de dados de cada modelo):

a) *Máquina Norma*. Uma Máquina de Registradores, sendo que o conjunto de registradores é infinito;

b) *Máquina de Post*. Baseada na estrutura de dados do tipo *fila* (o primeiro dado armazenado é o primeiro a ser recuperado);

c) *Máquinas com Pilhas*. Baseada na estrutura de dados do tipo *pilha* (o último dado armazenado é o primeiro a ser recuperado), onde são necessárias pelo menos duas pilhas para simular o mesmo poder computacional de uma fita ou fila.

Também é verificado que algumas extensões da Máquina de Turing não aumentam o seu poder computacional como, por exemplo:

a) *Não-Determinismo*. Permite que a máquina possa tentar diversos caminhos alternativos para uma mesma situação;

b) *Múltiplas Fitas*. Mais de uma fita;

c) *Múltiplas Unidades de Controle*. Mais de uma unidade de controle;

d) *Fitas Infinitas nas duas Extremidades*. As fitas não possuem fim em qualquer das duas extremidades.

Existem três maneiras de abordar o estudo das Máquinas de Turing e de seus modelos equivalentes, como segue:

a) *Processamento de Funções*. Funções computáveis e suas propriedades;

b) *Reconhecimento de Linguagens*. Linguagens que podem ser reconhecidas e suas propriedades;

c) *Solucionabilidade de Problemas*. Problemas solucionáveis e não-solucionáveis, problemas parcialmente solucionáveis (computáveis) e completamente insolúveis (não-computáveis), bem como suas propriedades.

As três abordagens são usadas ao longo deste livro. Entretanto, a terceira (solucionabilidade de problemas) constitui um dos problemas fundamentais da Ciência da Computação e é tratada em capítulo específico. De fato, existe um grande número de funções para as quais não é possível desenvolver algoritmos capazes de computá-las. Ou seja, existem funções que são não-computáveis, sendo algumas relativamente simples de serem enunciadas como, por exemplo, uma função que nomeia todas as funções.

3.1 Codificação de Conjuntos Estruturados

Nesta seção, é considerado, de forma breve e através de exemplos, o problema da *codificação de conjuntos estruturados*, onde elementos de tipos de dados estruturados são representados como números naturais. Para um dado conjunto estruturado X, a idéia básica é definir uma função injetora:

$$c: X \rightarrow \mathbb{N}$$

ou seja, uma função tal que, para todo $x, y \in X$, tem-se que:

$$\text{se } c(x) = c(y), \text{ então } x = y$$

Neste caso, o número natural $c(x)$ é a codificação do elemento estruturado x.

EXEMPLO 3.1 Codificação de n-Uplas Naturais.

Suponha que é desejado codificar, de forma unívoca, elementos de $\mathbb{N}^n$ como números naturais, ou seja, deseja-se uma função injetora:

$$c: \mathbb{N}^n \rightarrow \mathbb{N}$$

Uma codificação simples é a seguinte:

- lembre-se que, pelo *Teorema Fundamental da Aritmética*, cada número natural é univocamente decomposto em seus fatores primos;
- suponha os n primeiros números primos denotados por $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ e assim sucessivamente. Então, a codificação c: $\mathbb{N}^n \rightarrow \mathbb{N}$ definida como segue é unívoca (suponha $(x_1, x_2, ..., x_n) \in \mathbb{N}^n$ e que o símbolo • denota a operação de multiplicação nos naturais):

$$c(x_1, x_2, ..., x_n) = p_1^{x1} \bullet p_2^{x2} \bullet ... \bullet p_n^{xn}$$

Deve-se reparar que esta codificação não constitui uma função bijetora, ou seja, nem todo número natural corresponde a uma n-upla (qual seria um exemplo?). Entretanto, todo número natural decomponível nos n primeiros números primos corresponde a uma n-upla. ❑

EXEMPLO 3.2 Codificação de Programas Monolíticos.

De forma análoga, um programa monolítico (fluxograma) pode ser codificado como um número natural. Suponha um programa monolítico P = (I, r) com m instruções rotuladas onde { $F_1, F_2, ..., F_f$ } e { $T_1, T_2, ..., T_t$ } são os correspondentes conjuntos de identificadores de operações e de testes, respectivamente. Seja, P' = (I, 1) como P, exceto pelos rótulos, os quais são renomeados como números naturais, onde 1 é o rótulo inicial e 0 o único rótulo final (se existir). Assim, uma instrução rotulada pode ser de uma das duas seguintes formas:

a) *Operação.*

r_1: faça F_k vá_para r_2

b) *Teste.*

r_1: se T_k então vá_para r_2 senão vá_para r_3

Cada instrução rotulada pode ser denotada por uma quádrupla ordenada como segue, onde a primeira componente identifica o tipo da instrução:

a) *Operação.* Instrução tipo zero:

$$(0, k, r_2, r_2)$$

b) *Teste.* Instrução tipo um:

$$(1, k, r_2, r_3)$$

Usando a codificação do Exemplo 3.1, o programa monolítico P' visto como quádruplas ordenadas pode ser codificado como segue:

- cada quádrupla (instrução rotulada) é codificada como um número natural. Assim, o programa monolítico P' com m instruções rotuladas pode ser visto como uma m-upla;
- por sua vez, a m-upla correspondente ao programa monolítico P' é codificada como um número natural.

Analogamente ao Exemplo 3.1, a codificação de programas monolíticos apresentada não é uma função bijetora (por quê?).

A seguir, é ilustrado como decodificar um número natural que denota um programa monolítico em seu correspondente programa. Suponha o número:

$$p = (2^{150}) \bullet (3^{105})$$

Portanto, o programa possui duas instruções rotuladas correspondentes aos números 150 e 105. Relativamente às decomposições em seus fatores primos, tem-se que:

$$150 = 2^1 \bullet 3^1 \bullet 5^2 \bullet 7^0 \quad \text{e} \quad 105 = 2^0 \bullet 3^1 \bullet 5^1 \bullet 7^1$$

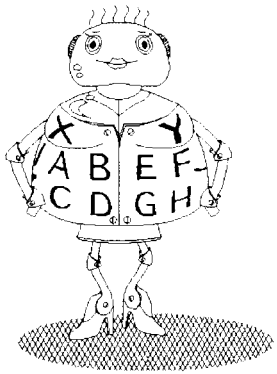
o que corresponde às quádruplas:

$$(1, 1, 2, 0) \quad \text{e} \quad (0, 1, 1, 1)$$

Logo, as instruções rotuladas decodificadas são como segue:

```
1:    se T1 então vá_para 2 senão vá_para 0
2:    faça F1 vá_para 1
```

Sugere-se como exercício a generalização do caso ilustrado para qualquer número natural correspondente à codificação de um programa monolítico. ❑



3.2 Máquina de Registradores - Norma

A *Máquina Norma* (*Number TheOretic Register MAchine*), proposta por Richard Bird ([BIR76]), é uma Máquina de Registradores. Máquinas de registradores são modelos propostos mais recentemente (comparativamente com a Máquina de Turing e com outros formalismos universalmente conhecidos), sendo definidas de forma a lembrar a arquitetura básica dos computadores atuais. Em particular, a Máquina Norma é especialmente interessante pois distingue as noções de programa e máquina (tal fato ficará evidente quando do estudo dos demais modelos). Assim, por razões didáticas, a Máquina Norma é o primeiro formalismo de Máquina Universal introduzido nesta publicação.

A Máquina Norma possui como memória um conjunto infinito de registradores naturais e três instruções sobre cada registrador:

- adição do valor um;
- subtração do valor um;
- teste se o valor armazenado é zero.

No texto que segue, $\mathbb{N}^\infty$ denota o conjunto de todas as uplas com infinitos (mas contáveis) componentes sobre o conjunto dos números naturais. Por exemplo, as seguintes uplas são elementos de $\mathbb{N}^\infty$:

$$(0, 1, 2, 3, ...) \quad \text{e} \quad (5, 5, 5, 5, ...)$$

Para evitar subscritos, as componentes das uplas são denotadas por letras maiúsculas como A, B, X, Y, ... as quais denotam os registradores na Máquina Norma.

A definição formal de $\mathbb{N}^\infty$ é omitida, mas pode ser resumidamente entendida como segue:

- suponha a upla $(a_0, a_1, a_2, ...)$ em $\mathbb{N}^\infty$;
- assim, pode-se afirmar que, para cada $n \in \mathbb{N}$, a_n é a n-ésima componente da upla $(a_0, a_1, a_2, ...)$;
- tal fato pode ser formalmente denotado como uma função $f: \mathbb{N} \to \mathbb{N}$ tal que, para qualquer $n \in \mathbb{N}$, $f(n) = a_n$;
- portanto, um elemento de $\mathbb{N}^\infty$ pode ser visto como uma função nos naturais, ou seja:

$$\mathbb{N}^\infty = \{f: \mathbb{N} \to \mathbb{N} \mid f \text{ é função}\}$$

No caso específico das uplas (0, 1, 2, 3, ...) e (5, 5, 5, 5, ...) as correspondentes funções são as seguintes, respectivamente:

- função identidade de $\mathbb{N}$, ou seja, $id_{\mathbb{N}}: \mathbb{N} \to \mathbb{N}$ tal que, para qualquer $n \in \mathbb{N}$, tem-se que $id_{\mathbb{N}}(n) = n$;
- função constante 5, ou seja, $const_5: \mathbb{N} \to \mathbb{N}$ tal que, para qualquer $n \in \mathbb{N}$, tem-se que $const_5(n) = 5$.

Definição 3.1 Máquina Norma.

A *Máquina Norma* é uma 7-upla (suponha que $K \in \{A, B, X, Y, ...\}$):

$$\text{Norma} = (\mathbb{N}^\infty, \mathbb{N}, \mathbb{N}, \text{ent}, \text{sai}, \{ad_K, sub_K\}, \{zero_K\})$$

onde:

a) Cada elemento do conjunto de valores de memória $\mathbb{N}^\infty$ denota uma configuração de seus infinitos *registradores*, os quais são denotados por:

$$A, B, X, Y, ...$$

b) A *função de entrada*:

$$\text{ent}: \mathbb{N} \to \mathbb{N}^\infty$$

é tal que carrega no registrador denotado por X o valor de entrada, inicializando todos os demais registradores com zero;

c) A *função de saída*:

$$sai: \mathbb{N}^{\infty} \rightarrow \mathbb{N}$$

é tal que retorna o valor corrente do registrador denotado por Y;

d) O *conjunto de interpretações de operações* é uma família de operações indexada pelos registradores onde, para cada registrador K, tem-se que:

$$ad_K: \mathbb{N}^{\infty} \rightarrow \mathbb{N}^{\infty}$$

adiciona um à componente correspondente ao registrador K, deixando as demais com seus valores inalterados, e:

$$sub_K: \mathbb{N}^{\infty} \rightarrow \mathbb{N}^{\infty}$$

subtrai um da componente correspondente ao registrador K, se o seu valor for maior que zero (caso contrário, mantém o valor zero), deixando as demais com seus valores inalterados;

e) O *conjunto de interpretações de testes* é uma família de testes indexada pelos registradores onde, para cada registrador K, tem-se que:

$$zero_K: \mathbb{N}^{\infty} \rightarrow \{verdadeiro, falso\}$$

resulta em verdadeiro, se a componente correspondente ao registrador K for zero e falso, caso contrário. ❑

Por simplicidade, para um registrador K, as correspondentes operações ou teste ad_K, sub_K, $zero_K$ são denotadas, respectivamente, como segue:

K := K + 1
K : = K - 1
K = 0

3.3 Máquina Norma como Máquina Universal

A Máquina Norma é uma máquina extremamente simples, de tal forma que parece difícil acreditar que o seu poder computacional é, no mínimo, o de qualquer computador moderno. Inclusive, é suficiente considerar somente os programas monolíticos. Ou seja, mecanismos como os de recursão podem ser simulados por fluxogramas em Norma. A seguir, como ilustração, diversas características de máquinas reais são simuladas usando a Máquina Norma, reforçando as evidências de que, de fato, trata-se de uma Máquina Universal. As seguintes simulações por Norma são exemplificadas:

a) *Operações e Testes*. Definição de operações e testes mais complexos como adição, subtração, multiplicação e divisão de dois valores e tratamento de valores diversos como testes sobre números primos;

b) *Valores Numéricos*. Armazenamento e tratamento de valores numéricos de diversos tipos como inteiros (negativos e não-negativos) e racionais;

c) *Dados Estruturados.* Acesso, armazenamento e tratamento de dados estruturados como arranjos (vetores uni e multidimensionais), pilhas, etc.;

d) *Endereçamento Indireto e Recursão.* Desvio para uma instrução determinada pelo conteúdo de um registrador e simulação de mecanismos de recursão;

e) *Cadeia de Caracteres.* Definição e manipulação de cadeias de caracteres.

3.3.1 Operações e Testes

As seguintes operações e teste não-definidos na Máquina Norma são exemplificadas:

- atribuição de valor a um registrador;
- adição de dois registradores;
- multiplicação de dois registradores;
- teste se o valor de um registrador é um número primo;
- atribuição do n-ésimo número primo.

EXEMPLO 3.3 Atribuição do Valor Zero a um Registrador.

A atribuição do valor zero para um registrador A, denotada por:

A := 0

pode ser obtida com o seguinte programa iterativo:

```
até   A = 0
faça  (A := A – 1)
```

❑

Dessa forma, pode-se tratar a operação A := 0 como uma *macro*, ou seja, um trecho de programa que é substituído pela sua definição sempre que referenciado.

Usando a macro A := 0 é fácil construir macros para definir operações de atribuição de um valor qualquer.

EXEMPLO 3.4 Atribuição de um Valor Natural a um Registrador.

A seguinte macro de atribuição de um valor natural n a um registrador A, denotada por:

A := n

pode ser definida pelo seguinte programa iterativo, exemplificado para n = 3:

```
A := 0;
A := A+1;
A := A+1;
A := A+1
```

❑

EXEMPLO 3.5 Adição de Dois Registradores.

Uma macro correspondente à operação de adição do valor do registrador B em A, denotada por:

A := A + B

pode ser obtida com o seguinte programa iterativo:

```
até   B = 0
faça  (A := A + 1;  B := B - 1)
```

❑

Note-se que, no exemplo acima, ao somar o valor de B em A, o registrador B é zerado. Para preservar o valor original de B, é necessário utilizar um registrador de trabalho.

EXEMPLO 3.6 Adição de Dois Registradores, Preservando o Conteúdo.

Uma macro correspondente à operação de adição do valor do registrador B em A, preservando o valor em B, necessita usar um registrador auxiliar C, como no seguinte programa iterativo:

```
C := 0;
até   B = 0
faça  (A := A + 1;  C := C + 1;  B := B - 1);
até   C = 0
faça  (B := B + 1;  C := C - 1)
```

Entretanto, como este programa não preserva o conteúdo original do registrador de trabalho C, faz-se necessário explicitar o uso deste registrador. Portanto, a seguinte notação é adotada para a macro de adição de dois registradores, preservando o conteúdo:

A := A + B usando C ❑

É importante destacar que, ao simular, na Máquina Norma, um programa P de outra máquina que contenha uma operação da forma A := A + B, é necessário escolher um registrador de trabalho C que não seja referenciado em P. Desta forma, quando da execução da macro A := A + B usando C a alteração do valor armazenado em C não terá efeito em qualquer outra parte do programa P.

EXEMPLO 3.7 Atribuição do Conteúdo de um Registrador.

Usando a macro A := A + B usando C, pode-se facilmente construir a seguinte macro de atribuição entre registradores:

A := B usando C

a qual pode ser definida pelo seguinte programa iterativo:

```
A := 0;
A := A + B usando C
```

❑

A definição de uma macro de multiplicação requer dois registradores de trabalho.

EXEMPLO 3.8 Multiplicação de Dois Registradores.

Uma macro correspondente à operação de multiplicação do valor do registrador B em A, usando dois registradores de trabalho C e D, denotada por:

A := A × B usando C, D

pode ser definida pelo seguinte programa iterativo:

```
C := 0;
até   A = 0
faça  (C := C + 1;  A := A - 1);
até   C = 0
faça  (A := A + B usando D;  C := C - 1)
```

❑

EXEMPLO 3.9 Teste se o Valor de um Registrador é um Número Primo.

Uma macro de teste que verifica se o valor de um registrador A é um número primo, usando um registrador de trabalho C, denotada por:

teste_primo(A) usando C

pode ser obtido pelo seguinte programa iterativo, o qual retorna o valor verdadeiro, se primo, e falso, caso contrário:

```
(se     A = 0
então   falso
senão   C := A;
        C := C - 1;
        (se     C = 0
        então   verdadeiro
        senão   até     teste_mod(A, C)
                faça    (C := C – 1)
                C := C - 1;
                (se     C = 0
                então   verdadeiro
                senão   falso) ) )
```

onde teste_mod(A, C) é um teste (sugerido como exercício) que retorna o valor verdadeiro, se o resto da divisão inteira do conteúdo de A por C é zero, e o valor falso, caso contrário. ❑

EXEMPLO 3.10 Atribuição do n-ésimo Número Primo a um Registrador.

A atribuição do n-ésimo número primo a um registrador A, usando um registrador de trabalho D, denotada por (suponha que o conteúdo de B é n):

A := primo(B) usando D

pode ser obtida pelo seguinte programa iterativo, o qual usa a macro teste_primo construída acima (suponha que 1 é o 0-ésimo número primo):

```
A := 1;
D := B;
até   D = 0
faça  (D := D - 1;
       A := A + 1;
       até   teste_primo(A)
       faça  (A := A + 1) )
```

❑

É sugerido como exercício desenvolver os programas iterativos para as seguintes operações e testes:

- fatorial;
- potenciação;
- teste "menor";
- teste "menor ou igual".

3.3.2 Valores Numéricos

Os seguintes tipos de dados numéricos não-definidos na Máquina Norma são exemplificados:

- inteiros (negativos e não-negativos);
- racionais.

EXEMPLO 3.11 Inteiros.

Um valor inteiro m pode ser representado como um par ordenado:

$$(s, |m|)$$

onde:

- $|m|$ denota a *magnitude* dada pelo valor absoluto de m;
- s denota o sinal de m, como segue:

$$\text{se } m < 0, \text{ então } s = 1; \text{ senão } s = 0$$

Duas formas simples de representar tal par ordenado na Máquina Norma podem ser:

- usando a codificação de n-uplas naturais como introduzido em 3.1 - Codificação de Conjuntos Estruturados;
- usando dois registradores: o primeiro referente ao sinal, e o segundo, à magnitude (valor absoluto).

No segundo caso (dois registradores), a simulação da operação inteira:

$$A := A + 1$$

supondo que A denota o par de registradores A_1 (sinal) e A_2 (magnitude), pode ser realizada pelo seguinte programa iterativo:

```
(se     A1 = 0
então   A2 := A2 + 1
senão   A2 := A2 - 1;
        (se     A2 = 0
        então   A1 := A1 – 1
        senão   ✔) )
```

❑

Note-se que outras operações e testes sobre inteiros podem ser tratadas de forma similar. Sugere-se como exercício a operação de subtração (A := A - 1) e o teste se é zero (A = 0).

EXEMPLO 3.12 Racionais.

Um valor racional r pode ser denotado como um par ordenado:

$$(a, b)$$

tal que $b > 0$ e $r = a/b$. A representação não é única pois, por exemplo, o valor racional 0.75 pode ser representado pelos pares (3, 4) e (6, 8), entre outros (na realidade, os pares (3, 4) e (6, 8) pertencem à classe de equivalência que denota o número racional 0.75 - por simplicidade, tal questão não será discutida). Neste contexto, as operações de adição, subtração, multiplicação e divisão, bem como o teste de igualdade, podem ser definidos como segue:

$$(a, b) + (c, d) = (a \bullet d + b \bullet c, b \bullet d)$$
$$(a, b) - (c, d) = (a \bullet d - b \bullet c, b \bullet d)$$
$$(a, b) \times (c, d) = (a \bullet c, b \bullet d)$$
$$(a, b) \div (c, d) = (a \bullet d, b \bullet c) \quad \text{com } c \neq 0$$
$$(a, b) = (c, d) \quad \text{se, e somente se,} \quad a \bullet d = b \bullet c$$

A construção dos correspondentes programas na Máquina Norma é sugerida como exercício. ❑

3.3.3 Dados Estruturados

A seguir, é exemplificado como uma estrutura do tipo arranjo unidimensional pode ser definida na Máquina Norma. Sugere-se como exercício a generalização para arranjos multidimensionais. Adicionalmente, é esboçada uma solução para pilhas, usando arranjos.

EXEMPLO 3.13 Arranjo Unidimensional.

Uma estrutura do tipo arranjo unidimensional da forma A(1), A(2), ..., pode ser definida por um único registrador A, usando a codificação de n-uplas naturais como introduzido em 3.1 - Codificação de Conjuntos Estruturados. Note-se que o arranjo não necessita ter tamanho máximo (número de posições indexáveis) predefinido. Lembre-se que a função de entrada é tal que carrega o valor da entrada no registrador X, zerando todos os demais, incluindo, portanto, o arranjo.

Em uma estrutura do tipo arranjo, é desejável indexar as suas posições de forma direta (número natural) ou indireta (conteúdo de um registrador). Para ambos os casos e para manter a coerência com a definição da Máquina Norma, é importante definir as operações de adição e subtração do valor 1, bem como do teste se o valor é zero, como segue, supondo que:

- o arranjo é implementado usando o registrador A;
- p_n denota o n-ésimo número primo;
- o seguinte teste (sugerido como exercício) é tal que retorna o valor verdadeiro, se o conteúdo do registrador C é um divisor do conteúdo do registrador A e falso, caso contrário:

  teste_div(A, C)

- a seguinte macro (a definição é sugerida como exercício) denota a divisão de dois registradores:

  teste_div(A, C)

- por simplicidade, na referência a uma macro definida anteriormente, é omitida a referência aos registradores usados. Por exemplo:

  A := A × B usando C, D é abreviada simplesmente por A := A × B

a) *Indexação Direta*. As macros:

   $ad_{A(n)}$ usando C

   $sub_{A(n)}$ usando C

   $zero_{A(n)}$ usando C

   onde A(n) denota a n-ésima posição do arranjo A, podem ser definidas pelos seguintes programas iterativos, respectivamente:

Programa Iterativo $ad_{A(n)}$ usando C

C := p_n;
A := A × C

Programa Iterativo $sub_{A(n)}$ usando C

C := p_n;
(se teste_div(A, C)
então A := A / C
senão ✔)

Programa Iterativo $zero_{A(n)}$ usando C

C := p_n;
(se teste_div(A, C)
então falso
senão verdadeiro)

b) *Indexação Indireta.* As macros:

$ad_{A(B)}$ usando C
$sub_{A(B)}$ usando C
$zero_{A(B)}$ usando C

onde A(B) denota a b-ésima posição do arranjo A, onde b é o conteúdo do registrador B, podem ser definidas pelos seguintes programas iterativos, respectivamente:

Programa Iterativo $ad_{A(B)}$ usando C

C := primo(B)
A := A × C

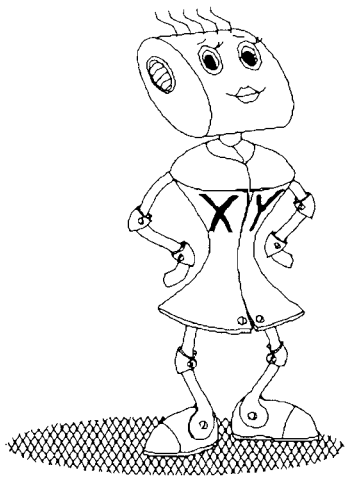
Programa Iterativo $sub_{A(B)}$ usando C

C := primo(B)
(se teste_div(A, C)
então A := A / C
senão ✔)

Programa Iterativo $zero_{A(B)}$ usando C

C := primo(B)
(se teste_div(A, B)
então falso
senão verdadeiro)

❑



Observação 3.2 Arranjo Unidimensional × Norma com 2 Registradores.

Um resultado interessante (e que será usado adiante) é que, usando a estrutura de arranjo unidimensional com indexação direta, como ilustrado no Exemplo 3.13, pode-se mostrar que os registradores X e Y são suficientes para realizar qualquer processamento (ou seja, os registradores A, B,... não são necessários). De fato, é suficiente usar X para armazenar um arranjo unidimensional onde cada posição corresponde a um registrador (por exemplo: X, Y, A, B,... correspondem às posições do arranjo indexadas por 0, 1, 2, 3,...). Assim, para um determinado registrador K, as operações e testes de Norma:

ad_K
sub_K
$zero_K$

podem ser simulados pelas operações e testes indexados introduzidos no Exemplo 3.13:

$ad_{X(k)}$ usando Y
$sub_{X(k)}$ usando Y
$zero_{X(k)}$ usando Y

onde X(k) denota a k-ésima posição do arranjo em X. ❑

EXEMPLO 3.14 Pilha.

Estruturalmente, a principal característica de uma *pilha* é que o último valor gravado é o primeiro a ser lido, como ilustrado na Figura 3.1. A *base* de uma pilha é fixa e define o seu início. O *topo* é variável e define a posição do último símbolo gravado. Uma pilha pode facilmente ser simulada usando um arranjo (como definido acima) e um registrador de índice (indexação indireta do arranjo) que

aponta para o topo da pilha. Sugere-se como exercício o detalhamento desta solução, bem como, a definição das operações *empilha* (adiciona o conteúdo de um registrador ao topo da pilha) e *desempilha* (retira o valor do topo e armazena-o em um registrador). ❑

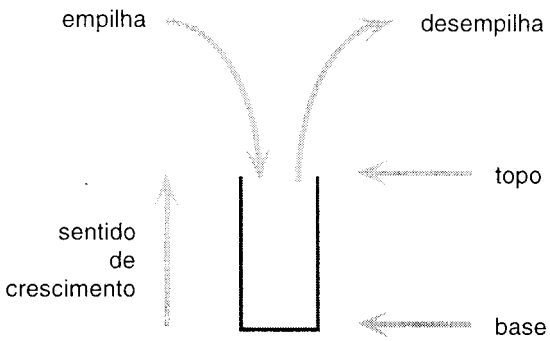


Figura 3.1 Estrutura do tipo pilha

3.3.4 Endereçamento Indireto e Recursão

A seguir, é exemplificado, em programas do tipo monolítico, como definir desvios, usando endereçamento indireto (determinado pelo conteúdo de um registrador).

EXEMPLO 3.15 Endereçamento Indireto em um Programa Monolítico.

Uma operação com endereçamento indireto da seguinte forma, onde A é um registrador:

```
r:    faça F vá_para A
```

pode ser definida pelo seguinte programa monolítico:

```
r:    faça F vá_para End_A
```

onde a macro End_A, representada na forma de fluxograma na Figura 3.2, trata o endereçamento indireto de A (e onde cada circunferência rotulada representa um desvio incondicional para a correspondente instrução). De forma análoga (sugere-se como exercício), é possível definir um teste com endereçamento indireto como segue, onde A e B são registradores:

```
r: se T então vá_para A senão vá_para B
```

❑

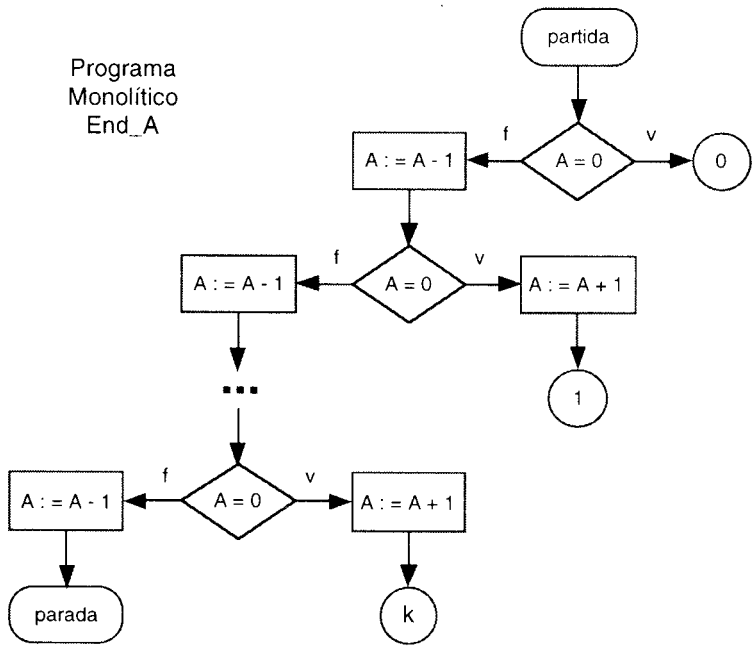


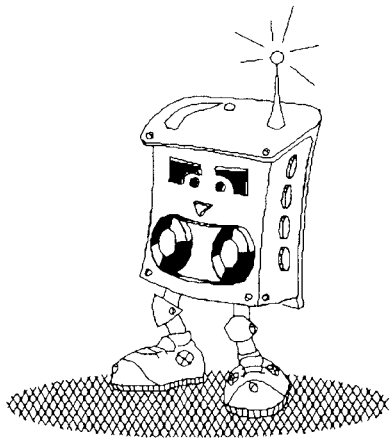
Figura 3.2 Fluxograma para tratar endereçamento indireto

Observação 3.3 Programa Monolítico × Programa Recursivo.

Em Norma, sub-rotinas e mecanismos de recursão como os definidos nos programas recursivos podem ser simulados por programas monolíticos, usando endereçamento indireto. Esta demonstração pode ser encontrada em ([BIR76]).❑

3.3.5 Cadeias de Caracteres

Cadeia de caracteres é um tipo de dado não-predefinido na Máquina Norma. O tratamento da definição e da manipulação de cadeias de caracteres será realizado através de uma outra Máquina Universal, denominada Máquina de Turing, a qual, prova-se, é equivalente à Norma.



3.4 Máquina de Turing

A Máquina de Turing, proposta por Alan Turing em 1936 ([TUR36]) é universalmente conhecida e aceita como formalização de algoritmo. Trata-se de um mecanismo simples que formaliza a idéia de uma pessoa que realiza cálculos. Lembra, em muito, os computadores atuais, embora tenha sido proposta anos antes do primeiro computador digital. Apesar de sua simplicidade, o modelo Máquina de Turing possui, no mínimo, o mesmo poder computacional de qualquer computador de propósito geral. No que segue, é importante observar que uma Máquina de Turing não constitui uma máquina, como definida anteriormente, mas sim um programa para uma Máquina Universal.

3.4.1 Noção Intuitiva

O ponto de partida de Turing foi analisar a situação na qual uma pessoa, equipada com um instrumento de escrita e um apagador, realiza cálculos numa folha de papel, organizada em quadrados.

Inicialmente, suponha que a folha de papel contém somente os dados iniciais do problema. O trabalho da pessoa pode ser resumido em seqüências de operações simples como segue:

- ler um símbolo de um quadrado;
- alterar um símbolo em um quadrado;
- mover os olhos para outro quadrado.

Quando é encontrada alguma representação satisfatória para a resposta desejada, a pessoa termina seus cálculos. Para viabilizar esse procedimento, as seguintes hipóteses são aceitáveis:

- a natureza bidimensional do papel não é um requerimento essencial para os cálculos. Pode ser assumido que o papel consiste de uma fita infinita organizada em quadrados;

- o conjunto de símbolos pode ser finito, pois se pode utilizar seqüências de símbolos;
- o conjunto de estados da mente da pessoa durante o processo de cálculo é finito. Mais ainda, entre esses estados, existem dois em particular: "estado inicial" e "estado final", correspondendo ao início e ao fim dos cálculos, respectivamente;
- o comportamento da pessoa a cada momento é determinado somente pelo seu estado presente e pelo símbolo para o qual sua atenção está voltada;
- a pessoa é capaz de observar e alterar o símbolo de apenas um quadrado de cada vez, bem como de transferir sua atenção somente para um dos quadrados adjacentes.

3.4.2 Noção como Máquina

Esta noção de uma pessoa calculando pode ser vista como uma máquina, constituída de três partes, como segue:

a) *Fita*. Usada simultaneamente como dispositivo de entrada, de saída e de memória de trabalho;

b) *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma unidade de leitura e gravação (cabeça da fita), a qual acessa uma célula da fita de cada vez e movimenta-se para a esquerda ou para a direita;

c) *Programa* ou *Função de Transição*. Função que define o estado da máquina e comanda as leituras, as gravações e o sentido de movimento da cabeça.

A fita é finita à esquerda e infinita (tão grande quanto necessário) à direita, sendo dividida em células, onde cada uma armazena um símbolo. Os símbolos podem pertencer ao alfabeto de entrada, ao alfabeto auxiliar ou ainda ser "branco" ou "marcador de início de fita". Inicialmente, a palavra a ser processada (ou seja, a informação de entrada para a máquina) ocupa as células mais à esquerda após o marcador de início de fita, ficando as demais com "branco", como ilustrado na Figura 3.3, onde ß e ⌾ representam "branco" e "marcador de início de fita", respectivamente.

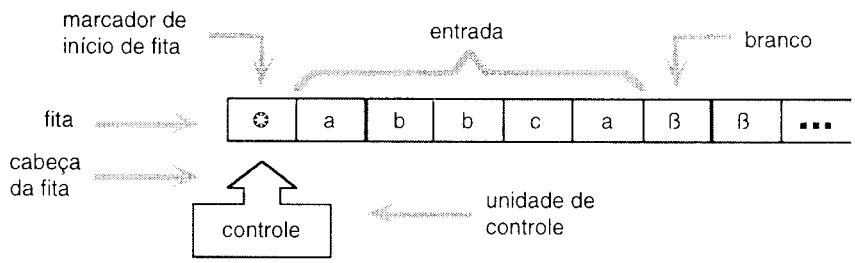


Figura 3.3 Fita e unidade de controle de uma Máquina de Turing

A unidade de controle possui um número finito e predefinido de estados. A *cabeça da fita* lê o símbolo de uma célula de cada vez e grava um novo símbolo. Após a leitura/gravação (a gravação é realizada na mesma célula de leitura), a cabeça move uma célula para a direita ou para a esquerda. O símbolo gravado e o sentido do movimento são definidos pelo programa.

O programa é uma função que, dependendo do estado corrente da máquina e do símbolo lido, determina o símbolo a ser gravado, o sentido do movimento da cabeça e o novo estado.

3.4.3 Modelo Formal

Definição 3.4 Máquina de Turing.

Uma *Máquina de Turing* é uma 8-upla:

$$M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis da máquina, o qual é finito;
- Π *programa* ou *função de transição*:

  $$\Pi: Q \times (\Sigma \cup V \cup \{ß, ⊙\}) \rightarrow Q \times (\Sigma \cup V \cup \{ß, ⊙\}) \times \{E, D\}$$

  a qual é uma função parcial;
- q_0 *estado inicial* da máquina tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q;
- V *alfabeto auxiliar*;
- ß símbolo especial *branco*;
- ⊙ símbolo especial *marcador de início* ou *símbolo de início* da fita. ❑

O símbolo de início de fita ocorre exatamente uma vez e sempre na célula mais à esquerda da fita, auxiliando na identificação de que a cabeça da fita se encontra na célula mais à esquerda da fita. A função programa:

considera	*para determinar*
• estado corrente • símbolo lido da fita	• novo estado • símbolo a ser gravado • sentido de movimento da cabeça, onde esquerda e direita são representados por E e D, respectivamente

Assim, tem-se que:

Π(estado corrente, símbolo lido) =
(novo estado, símbolo gravado, sentido do movimento)

ou seja (suponha que $p, q \in Q$, $a_u, a_v \in (\Sigma \cup V \cup \{ß, ⊙\})$ e $m \in \{E, D\}$):

$$\Pi(p, a_u) = (q, a_v, m)$$

A função programa pode ser interpretada como um grafo finito direto, como ilustrado na Figura 3.4. Neste caso, os estados iniciais e finais são representados como ilustrado na Figura 3.5.

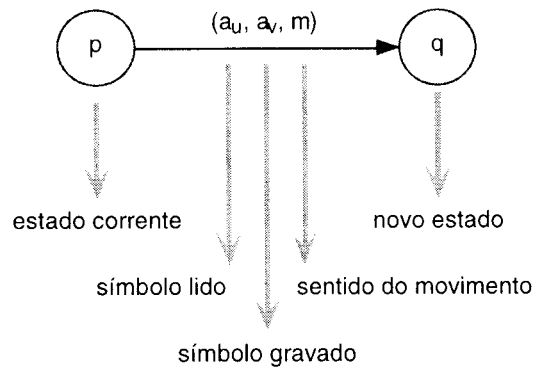


Figura 3.4 Representação da função programa como um grafo

Figura 3.5 Representação de um estado inicial (esq.) e final (dir.) como nodos de grafos

Adicionalmente, a função programa pode ser representada na forma de tabela como na Figura 3.6 (ilustrada para o caso $\Pi(p, a_u) = (q, a_v, m)$).

Π	⊙	...	a_u	...	a_v	...	ß
p			(q, a_v, m)				
q							
...							

Figura 3.6 Representação da função programa como uma tabela

O processamento de uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$ para uma palavra de entrada w consiste na sucessiva aplicação da função programa a partir do estado inicial q_0 e da cabeça posicionada na célula mais à esquerda da fita, até ocorrer uma condição de parada. O processamento de M para a entrada w pode parar ou ficar em *loop* infinito. A parada pode ser de duas maneiras: aceitando ou rejeitando a entrada w. As condições de parada são as seguintes:

a) *Estado Final.* A máquina assume um estado final: a máquina pára e a palavra de entrada é aceita;

b) *Função Indefinida.* A função programa é indefinida para o argumento (símbolo lido e estado corrente): a máquina pára e a palavra de entrada é rejeitada;

c) *Movimento Inválido.* O argumento corrente da função programa define um movimento à esquerda e a cabeça da fita já se encontra na célula mais à esquerda: a máquina pára e a palavra de entrada é rejeitada.

Observação 3.5 Máquina de Turing × (Programa e Máquina).

É importante reparar que a definição da Máquina de Turing não distingue os conceitos de programa e máquina. Na realidade, trata-se de um programa para uma Máquina Universal. ❑

Observação 3.6 Variações sobre a Definição de Máquina de Turing.

Diversas variações sobre a definição de Máquina de Turing são adotadas. Note-se que essas variações não alteram o poder computacional do formalismo. As variações mais significativas estão nas características da fita e no movimento da cabeça como, por exemplo:

a) *Inexistência do Marcador de Início de Fita.* É freqüente não se incluir um marcador de início de fita. Assim, a célula mais à esquerda da fita contém o primeiro símbolo da entrada (ou branco, se a entrada for vazia). Neste caso, ao definir uma função programa, deve-se tomar cuidado especial para controlar quando a cabeça da fita atinge o fim da mesma;

b) *Cabeça de Fita não se Move em uma Leitura/Gravação.* Na função programa, é possível especificar, adicionalmente ao movimento para esquerda ou direita, que a cabeça permaneça parada (na célula de leitura/gravação). O principal objetivo desta variação é facilitar a especificação da função programa, bem como reduzir o número de transições necessárias;

c) *Estado Final de Rejeição.* Pode ser definido um estado final de rejeição, para explicitar a condição de parada com rejeição da entrada. Tal modificação é especialmente interessante pois facilita a compreensão da lógica da função programa. ❑

3.4.4 Máquinas de Turing como Reconhecedores de Linguagens

Uma das abordagens do estudo das Máquinas de Turing ou máquinas universais em geral é como reconhecedores de linguagens, ou seja, dispositivos capazes de determinar se uma dada palavra sobre o alfabeto de entrada pertence ou não a uma certa linguagem.

Definição 3.7 Linguagem Aceita por uma Máquina de Turing.

Seja $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ❂)$ uma Máquina de Turing. Então:

a) A *Linguagem Aceita* por M, denotada por ACEITA(M) ou L(M), é o conjunto de todas as palavras pertencentes a Σ^* aceitas por M, ou seja:

$$ACEITA(M) = \{w \mid M \text{ ao processar } w \in \Sigma^* \text{ pára em um estado } q_f \in F\}$$

b) A *Linguagem Rejeitada* por M, denotada por REJEITA(M) é o conjunto de todas as palavras de Σ^* rejeitadas por M, ou seja:

$$REJEITA(M) = \{w \mid M \text{ ao processar } w \in \Sigma^* \text{ pára em um estado } q \notin F\}$$

c) A linguagem para qual M fica em *loop* infinito, denotada por LOOP(M), é o conjunto de todas as palavras de Σ^* para as quais M fica processando indefinidamente. ❑

As seguintes afirmações são verdadeiras:

$ACEITA(M) \cup REJEITA(M) \cup LOOP(M) = \Sigma^*$
$ACEITA(M) \cap REJEITA(M) = \varnothing$
$ACEITA(M) \cap LOOP(M) = \varnothing$
$REJEITA(M) \cap LOOP(M) = \varnothing$

e, portanto:

$$ACEITA(M) \cap REJEITA(M) \cap LOOP(M) = \varnothing$$

Conseqüentemente, o complemento de:

ACEITA(M) é $REJEITA(M) \cup LOOP(M)$
REJEITA(M) é $ACEITA(M) \cup LOOP(M)$
LOOP(M) é $ACEITA(M) \cup REJEITA(M)$

Ou seja, uma Máquina de Turing como reconhecedor particiona o conjunto de palavras sobre o alfabeto Σ em classes de equivalência, como ilustrado na Figura 3.7.

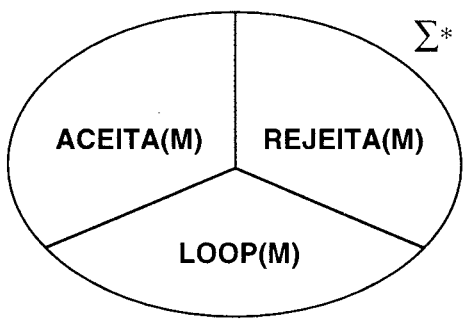


Figura 3.7 Particionamento do conjunto de palavras em classes de equivalência induzido por uma Máquina de Turing

EXEMPLO 3.16 Máquina de Turing – Duplo Balanceamento.

Considere a linguagem:

$$Duplo\_Bal = \{a^n b^n \mid n \geq 0\}$$

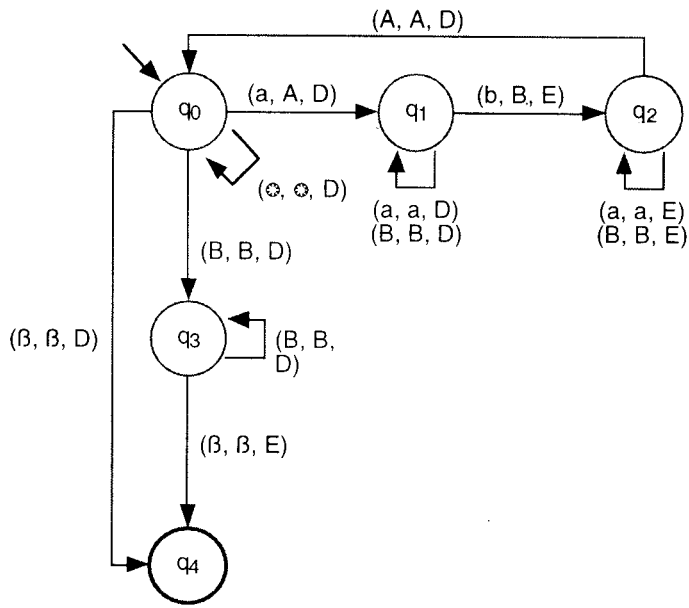
A Máquina de Turing:

$$MT\_Duplo\_Bal = (\{a, b\}, \{q_0, q_1, q_2, q_3, q_4\}, \Pi, q_0, \{q_4\}, \{A, B\}, ß, ⊙)$$

ilustrada na Figura 3.8, é tal que:

$$ACEITA(MT\_Duplo\_Bal) = Duplo\_Bal$$
$$REJEITA(MT\_Duplo\_Bal) = \Sigma^* - Duplo\_Bal$$

e, portanto, $LOOP(MT\_Duplo\_Bal) = \varnothing$



Π	⊙	a	b	A	B	ß
q_0	$(q_0, ⊙, D)$	(q_1, A, D)			(q_3, B, D)	$(q_4, ß, D)$
q_1		(q_1, a, D)	(q_2, B, E)		(q_1, B, D)	
q_2		(q_2, a, E)		(q_0, A, D)	(q_2, B, E)	
q_3					(q_3, B, D)	$(q_4, ß, E)$
q_4						

Figura 3.8 Grafo e tabela de transições da Máquina de Turing – Duplo Balanceamento

O algoritmo apresentado reconhece o primeiro símbolo a, o qual é marcado como A, e movimenta a cabeça da fita à direita, procurando o b correspondente, o qual é marcado como B. Este ciclo é repetido sucessivamente até identificar para cada a o seu correspondente b. Adicionalmente, o algoritmo garante que qualquer outra palavra que não esteja na forma $a^n b^n$ é rejeitada. Note-se que o símbolo de início de fita não tem influência na solução proposta.

A Figura 3.9 ilustra a seqüência do processamento da Máquina de Turing Duplo_Bal para a entrada w = aabb.

Esta linguagem é um exemplo clássico e de fundamental importância no estudo das linguagens, pois permite estabelecer analogia com linguagens que possuem duplo balanceamento em sua estrutura como, por exemplo:

a) Linguagens bloco-estruturadas do tipo $BEGIN^nEND^n$, como a linguagem de programação Pascal;

b) Linguagens com parênteses balanceados na forma $(^n)^n$, como as expressões aritméticas, presentes na maioria das linguagens de programação. ❑

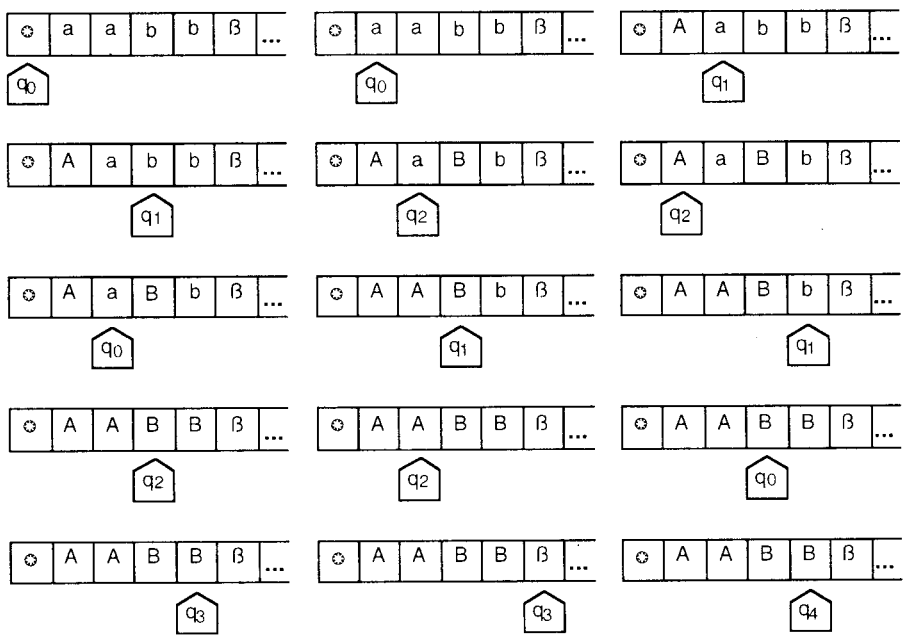


Figura 3.9 Computação de uma Máquina de Turing

Uma linguagem aceita por uma Máquina de Turing é dita Enumerável Recursivamente. Historicamente, o termo "enumerável" deriva do fato de que as palavras de qualquer Linguagem Enumerável Recursivamente podem ser enumeradas ("listadas") por uma Máquina de Turing; e "recursivamente" é um

termo matemático anterior ao computador, com significado similar ao de "recursão", usado em informática. Tal terminologia ficará mais clara quando do estudo do Capítulo 4 - Funções Recursivas.

Definição 3.8 Linguagem Enumerável Recursivamente.

Uma linguagem aceita por uma Máquina de Turing é dita *Linguagem Enumerável Recursivamente.* ❑

EXEMPLO 3.17 Linguagem Enumerável Recursivamente.

As seguintes linguagens são exemplos de linguagens enumeráveis recursivamente sendo que, para a primeira, a correspondente Máquina de Turing foi construída no Exemplo 3.16 (as demais são sugeridas como exercício):

a) Duplo_Bal = $\{a^n b^n \mid n \geq 0\}$

b) Triplo_Bal = $\{a^n b^n c^n \mid n \geq 0\}$

c) Palavra_Palavra = {ww | w é palavra sobre os símbolos a e b}

A linguagem Palavra_Palavra é outro exemplo clássico e de fundamental importância no estudo das linguagens, pois permite estabelecer analogia com linguagens que possuem duas (ou mais) ocorrências de um mesmo trecho de código em um programa. Por exemplo, em linguagens como Algol ou Pascal, é necessário declarar variáveis antes de referenciá-las ao longo dos comandos. ❑

Como, segundo a Hipótese de Church (ver 3.8 - Hipótese de Church), a Máquina de Turing é o dispositivo mais genérico de computação, a Classe das Linguagens Enumeráveis Recursivamente define a classe de todas as linguagens que podem ser reconhecidas mecanicamente.

Entretanto, é importante destacar que Classe das Linguagens Enumeráveis Recursivamente, inclui algumas para as quais é impossível determinar mecanicamente se uma palavra *não* pertence à linguagem. Se L é uma dessas linguagens, então para qualquer máquina M que aceita L, existe pelo menos uma palavra w que não pertencente a L que, ao ser processada por M, a máquina entra em *loop* infinito. Assim, pode-se afirmar que:

- se w pertence a L, M pára e aceita a entrada;
- se w não pertence a L, M pode parar, rejeitando a palavra, ou permanecer em *loop* infinito, processando indefinidamente.

Portanto, é conveniente definir uma subclasse da Classe das Linguagens Enumeráveis Recursivamente, denominada Classe das Linguagens Recursivas, composta pelas linguagens para as quais existe pelo menos uma Máquina de Turing que pára para qualquer entrada, aceitando ou rejeitando.

Definição 3.9 Linguagem Recursiva.

Uma linguagem L é dita *Linguagem Recursiva* se existe uma Máquina de Turing M tal que:

ACEITA(M) = L
REJEITA(M) = Σ^* - L
LOOP(M) = $\varnothing$ ❑

Portanto, a Classe das Linguagens Recursivas define a classe de todas as linguagens que podem ser reconhecidas mecanicamente e para as quais *sempre existe* um reconhecedor ("compilador") que *sempre pára*, para qualquer entrada, reconhecendo ou rejeitando. É importante destacar que a grande maioria das linguagens aplicadas são recursivas e para elas, conseqüentemente, sempre existe um reconhecedor que sempre pára. Entretanto, tais reconhecedores podem ser muito ineficientes (ver [MEN98]).

EXEMPLO 3.18 Linguagem Recursiva.

É fácil verificar que as seguintes linguagens são recursivas:

a) Duplo_Bal = $\{a^n b^n \mid n \geq 0\}$

b) Triplo_Bal = $\{a^n b^n c^n \mid n \geq 0\}$

c) $\{w \mid w \in \{a, b\}^*$ e tem o dobro de símbolos a que b$\}$ ❑

A seguir, são enunciadas algumas das principais propriedades das linguagens enumeráveis recursivamente e das linguagens recursivas, tais propriedades podem ser resumidas como segue e são detalhas em [MEN98]:

a) O complemento de uma linguagem recursiva é uma linguagem recursiva. Conseqüentemente, existe um algoritmo que sempre pára e que reconhece o complemento da linguagem;

b) Uma linguagem L é recursiva se, e somente se, L e seu complemento são enumeráveis recursivamente;

c) A Classe das Linguagens Recursivas está contida propriamente na Classe das Linguagens Enumeráveis Recursivamente.

3.4.5 Máquinas de Turing como Processadores de Funções

Uma das principais abordagens das Máquinas de Turing é como processador de funções. O estudo é restrito às funções de mapeamento de palavras de um alfabeto Σ em uma palavra do mesmo alfabeto. Em termos práticos, a maioria das funções pode ser facilmente transformadas em funções desse tipo.

A seguir, é apresentada a definição de função computada para uma Máquina de Turing. Sugere-se como exercício comparar e diferenciar com a definição de função computada introduzida no Capítulo 2 – Programas, Máquinas e Computações.

Definição 3.10 Função Turing-Computável.

Uma função parcial:

$$f: (\Sigma^*)^n \rightarrow \Sigma^*$$

é dita *Função Turing-Computável* ou simplesmente *Função Computável* se existe uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$ que computa f, ou seja:

a) Para $(w_1, w_2, \ldots, w_n) \in (\Sigma^*)^n$, tem-se que a palavra de entrada para M é:

$$⊙\ w_1\ w_2\ \ldots\ w_n$$

b) Se f é definida para $(w_1, w_2, \ldots, w_n)$, então o processamento de M para a entrada $⊙\ w_1\ w_2\ \ldots\ w_n$ é tal que:

- pára (aceitando ou rejeitando);
- o conteúdo da fita é (excetuando-se os símbolos brancos):

$$⊙\ w$$

c) Se f é indefinida para $(w_1, w_2, \ldots, w_n)$, então M, ao processar a entrada $⊙\ w_1\ w_2\ \ldots\ w_n$, fica em *loop* infinito. ❑

Observação 3.11 Função Turing-Computável × Condição de Parada.

Na definição acima, é considerado como resultado do processamento de M somente o conteúdo gravado na fita. Portanto, relativamente à função computável, um processamento que pára em um estado não-final é perfeitamente válido. ❑

Observação 3.12 Símbolo Auxiliar de Entrada .

Ao especificar uma n-upla na fita de entrada, pode haver a necessidade de delimitar suas componentes. Neste caso, é usual utilizar # como símbolo especial de delimitação, como exemplificado a seguir para a n-upla $(w_1, w_2, \ldots, w_n)$:

$$⊙\ w_1\ \#\ w_2\ \#\ \ldots\ \#\ w_n$$ ❑

EXEMPLO 3.19 Máquina de Turing – Concatenação.

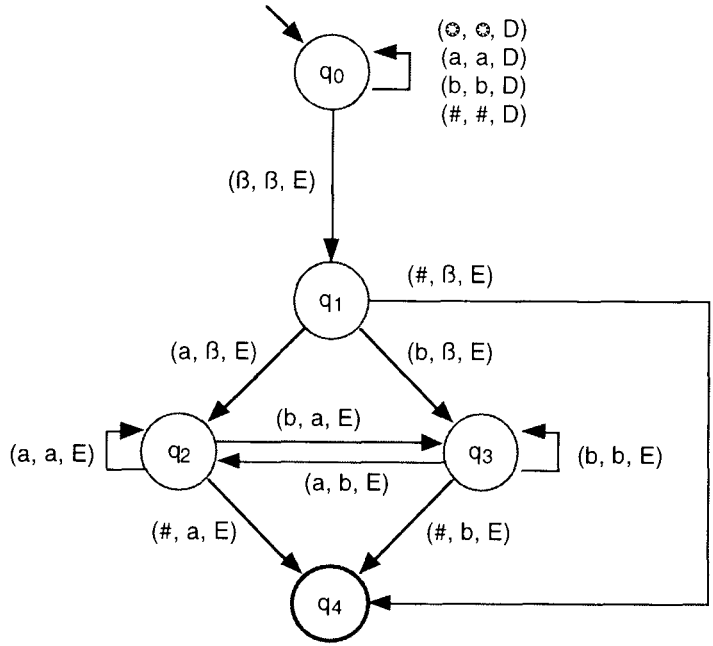
Considere a função (total):

$$\text{concatenação}: (\{a, b\}^*)^n \rightarrow \{a, b\}^*$$

tal que associa ao par (w_1, w_2) a palavra $w_1 w_2$. A Máquina de Turing:

$$\text{Conc} = (\{a, b, \#\}, \{q_0, q_1, q_2, q_3, q_4\}, \Pi, q_0, \{q_4\}, \varnothing, ß, ⊙)$$

é como ilustrada na Figura 3.10.



Π	⊙	a	b	#	ß
q_0	$(q_0, \odot, D)$	(q_0, a, D)	(q_0, b, D)	$(q_0, \#, D)$	$(q_1, ß, E)$
q_1		$(q_2, ß, E)$	$(q_3, ß, E)$	$(q_4, ß, E)$	
q_2		(q_2, a, E)	(q_3, a, E)	(q_4, a, E)	
q_3		(q_2, b, E)	(q_3, b, E)	(q_4, b, E)	
q_4					

Figura 3.10 Grafo e tabela de transições da Máquina de Turing - Concatenação

O algoritmo apresentado recebe como entrada a palavra:

$$\odot\, w_1 \,\#\, w_2$$

Inicialmente, posiciona a cabeça no último símbolo da palavra de entrada. Após, move a cabeça para a esquerda até encontrar o símbolo #, quando pára. Enquanto move a cabeça, ao ler um símbolo, grava sobre este o símbolo lido anteriormente. A memorização do símbolo anterior é realizada pelos estados como segue:

q_2 memoriza que o símbolo anterior é a
q_3 memoriza que o símbolo anterior é b ❑

EXEMPLO 3.20 Máquina de Turing – Função Quadrado.

Considere a função (total):

$$\text{quadrado: } \{1\}^* \rightarrow \{1\}^*$$

tal que associa o valor natural n, representado em unário, ao valor n^2 (também em unário). A Máquina de Turing:

$$\text{Quadr} = (\{1\}, \{q_0, q_1, q_2, \ldots, q_{13}\}, \Pi, q_0, \{q_{13}\}, \{A, B, C, D\}, ß, ⊙)$$

ilustrada na Figura 3.12 e onde Π é como na Figura 3.11.

Π	⊙	1	A	B	C	ß
q_0	$(q_0, ⊙, D)$	(q_1, A, D)		(q_0, B, D)		$(q_3, ß, E)$
q_1		$(q_1, 1, D)$		(q_1, B, D)		(q_2, B, E)
q_2		$(q_2, 1, E)$	(q_0, A, D)	(q_2, B, E)		
q_3	$(q_{13}, ⊙, D)$			$(q_4, ß, E)$		
q_4			(q_5, A, D)	(q_4, B, E)		
q_5				(q_6, C, E)		$(q_{12}, ß, E)$
q_6	$(q_7, ⊙, D)$		$(q_6, 1, E)$		(q_6, C, E)	
q_7		(q_8, A, D)				
q_8		(q_9, A, D)			(q_{11}, C, D)	
q_9		$(q_9, 1, D)$		(q_9, B, D)	(q_9, C, D)	$(q_{10}, 1, E)$
q_{10}		$(q_{10}, 1, E)$	(q_8, A, D)	(q_{10}, B, E)	(q_{10}, C, E)	
q_{11}		$(q_{12}, 1, E)$		(q_6, C, E)	(q_{11}, C, D)	
q_{12}	$(q_{13}, ⊙, D)$		$(q_{12}, 1, E)$		$(q_{12}, 1, E)$	
q_{13}						

Figura 3.11 Tabela de transições da Máquina de Turing – Função Quadrado

O algoritmo apresentado recebe como entrada a palavra:

$$⊙\ n_1$$

onde n_1 denota o valor n representado em unário sobre {1}. Assim, $(n_1)^2$ é simplesmente n_1 concatenado consigo mesmo n vezes, ou seja:

$$(n_1)^2 = n_1\, n_1 \ldots n_1 \quad (\text{n vezes})$$

Tal concatenação é obtida como segue:

- no ciclo em q_0, q_1 e q_2, é gerado $⊙ n_A\, n_B$ (n_A é em unário sobre {A} e n_B, sobre {B});
- em q_0, q_3 e q_4, é retirado um símbolo B de n_B, resultando em $n_A\, (n\text{-}1)_B$

- em q_5 até q_{11}, a subpalavra $(n\text{-}1)_B$ é usada para controlar concatenações sucessivas, resultando em:

  $\odot\, n_A (n\text{-}1)_C (n\text{-}1)_1 (n\text{-}1)_1 \ldots (n\text{-}1)_1$ onde $(n\text{-}1)_1$ é repetida n-1 vezes
- por fim, em q_{12}, as subpalavras $n_A (n\text{-}1)_C$ são substituídas por $n_1 (n\text{-}1)_1$, resultando em:

  $\odot\, n_1 (n\text{-}1)_1 (n\text{-}1)_1 (n\text{-}1)_1 \ldots (n\text{-}1)_1$ onde $(n\text{-}1)_1$ é repetida n vezes
- ou seja, o comprimento da palavra resultante é:

$$n + (n\text{-}1) * n = n + (n^2 - n) = n^2$$ ❑

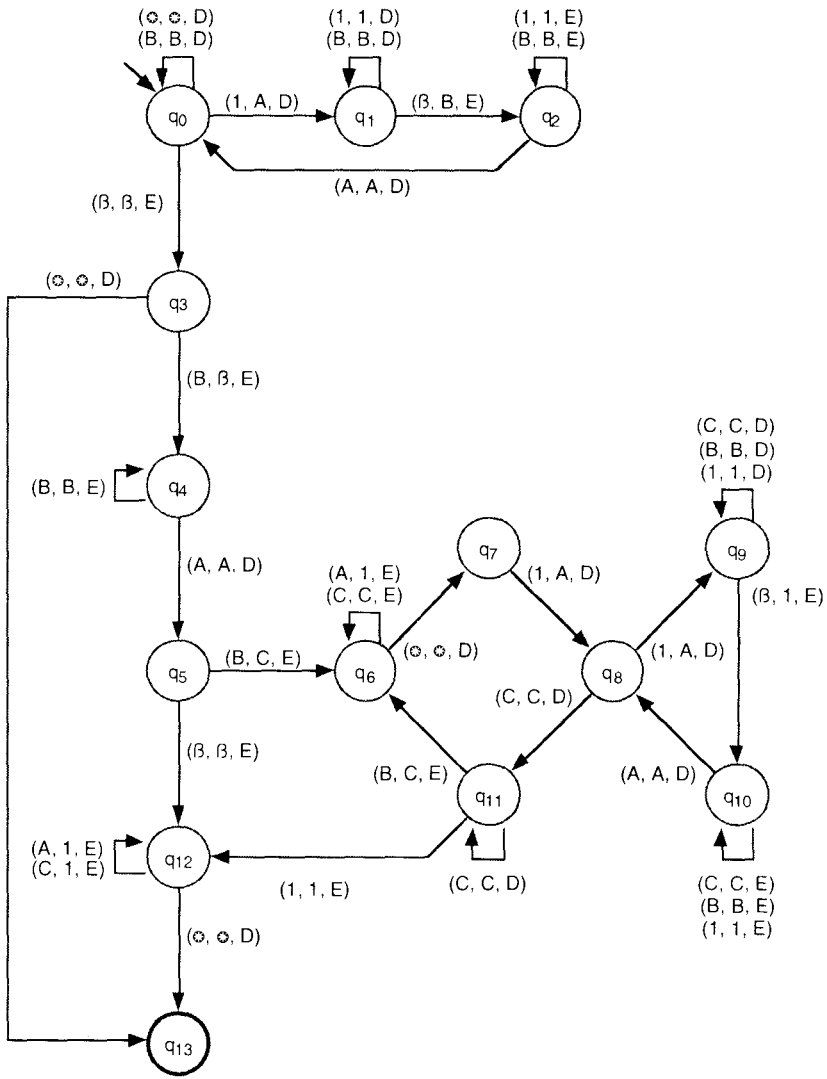


Figura 3.12 Grafo da Máquina de Turing – Função Quadrado

Definição 3.13 Função Turing-Computável Total.

Uma função total:

$$f: (\Sigma^*)^n \rightarrow \Sigma^*$$

é dita *Função Turing-Computável Total* ou simplesmente *Função Computável Total* se existe uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$ que computa f e sempre pára para qualquer entrada. ❑

A definição de uma função Turing-computável total é intuitiva, ou seja, existe uma Máquina de Turing que a computa e que sempre pára para qualquer entrada. Sugere-se como exercício verificar se os exemplos acima de funções Turing-computáveis são totais.

Observação 3.14 Máquina de Turing como Reconhecedor × Função Turing-Computável.

Toda Máquina de Turing vista como um reconhecedor de linguagem também pode ser vista como um processador de função. De fato, é suficiente modificar a Máquina de Turing reconhecedora para, ao identificar uma condição de aceitação ou rejeição, gravar na fita uma informação sobre tal condição e, somente após, parar. Esta informação pode ser do tipo binária como, por exemplo (suponha uma Máquina de Turing sobre o alfabeto {a, b}):

⊙ a	saída que identifica aceitação
⊙ b	saída que identifica rejeição

Tal modificação não é tão trivial como aparenta inicialmente, pois o reconhecimento de uma entrada pode gerar qualquer informação na fita, inclusive brancos intercalados com outros símbolos. Assim, o algoritmo modificado, na tentativa de substituir todos os demais símbolos (excetuando-se os que codificam a condição aceita ou rejeita), pode entrar em *loop* infinito. Um detalhamento correto de tal modificação é sugerido como exercício.

A abordagem inversa também pode ser feita. Ou seja, toda função Turing-computável pode ser vista como uma Máquina de Turing como reconhecedor. Neste caso, é suficiente modificar a função como um problema do tipo sim (aceita) ou não (rejeita). Tal questão é detalhada quando do estudo da solucionabilidade de problemas.

Conseqüentemente, é usual unificar as nomenclaturas como, por exemplo:

- Função Turing-Computável e *Função Enumerável Recursivamente*;
- Função Turing-Computável Total e *Função Recursiva*. ❑

3.4.6 Equivalência entre as Máquinas de Turing e Norma

A seguir prova-se que a Máquina de Turing é equivalente à Máquina Norma. Além de reforçar as evidências de que ambas são máquinas universais, também fica caracterizado que, de fato, Norma pode definir e manipular cadeias de caracteres (conforme introduzido em 3.3.5 - Cadeias de Caracteres).

Resumidamente, a prova é como segue:

a) *Turing* ≤ *Norma*. A estrutura de fita da Máquina de Turing é simulada em Norma usando uma estrutura de arranjo unidimensional;

b) *Norma* ≤ *Turing*. Conforme introduzido na Observação 3.2, na Máquina Norma, os registradores X e Y são suficientes para realizar qualquer processamento. Assim, uma Máquina de Turing pode simular os dois registradores como segue:

   b.1) O conteúdo de cada registrador (valor natural) é implementado de forma unária em Turing (ou seja, a repetição de um mesmo símbolo ao longo da fita, tantas vezes quanto for o valor do número natural);

   b .2) O registrador X ocupa as células ímpares da fita, e Y, as pares.

Lembre-se que, no conceito de simulação, é necessário considerar funções de codificação e decodificação para permitir comparar máquinas com diferentes conjuntos de entrada e saída.

Teorema 3.15 Máquina de Turing ≤ Máquina Norma.

O formalismo Máquina de Turing pode ser simulado pelo formalismo Máquina Norma.

Prova:

Como sugerido em exercício, sem perda de generalidade, pode-se supor que a função programa de uma Máquina de Turing é total. Suponha uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$. Então, a simulação de M por um programa P em Norma pode ser definida como segue:

Fita. A fita é simulada como um arranjo unidimensional em X, sendo que cada célula da fita corresponde a uma posição do arranjo. Adicionalmente, o símbolo de cada célula é codificado como um número natural como segue: para um alfabeto $\Sigma = \{a_1, a_2, \ldots, a_n\}$, o símbolo a_i é codificado como o natural i, e os símbolos especiais ß e ⊙ como zero e n+1, respectivamente;

Estados. Para os estados de $Q = \{q_0, q_1, \ldots, q_n\}$ em M, o programa P em Norma possui correspondentes instruções rotuladas por 0, 1, ..., n. Portanto, o rótulo inicial de P é 0 (pois q_0 é o estado inicial de M) e, para qualquer $q_f \in F$, f é rótulo final de P;

Estado Corrente. O estado corrente de M é simulado em Norma, usando o registrador Q, o qual assume valores em {0, 1, ..., n} (correspondendo aos estados $q_0, q_1, \ldots, q_n$);

Cabeça da Fita. A posição corrente da cabeça da fita de M é simulada usando o registrador C de Norma, o qual contém a posição corrente do arranjo em X e é inicializado com o valor 1;

Função Programa. A simulação de uma transição de M da forma:

$$\Pi(q_u, a_r) = (q_v, a_s, m)$$

onde m assume valores em {E, D} (esquerda e direita), é dada pelo trecho de programa de P em Norma abaixo, onde cada instrução rotulada u é, na realidade, uma chamada de macro, a qual implementa a transição de M, contendo instruções rotulas para o novo estado, gravação na fita e movimento da cabeça (lembre-se que é suposto que a função programa é total). Note-se que:

a) No trecho de programa é suposto que o movimento da cabeça da fita é para a direita e, portanto é adicionado 1 ao registrador C; caso o movimento seja para a esquerda, é necessário subtrair 1;

b) A transição depende do estado corrente q_u e do símbolo lido a_r. Assim, na instrução rotulada por u, é especificado um desvio incondicional para uma instrução rotulada pelo par (u, r), usando a codificação de n-uplas do Exemplo 3.1 (lembre-se que o conteúdo de Q e X(C) são u e r, respectivamente). Por simplicidade, os detalhes de como implementar o cálculo da expressão $2^Q \times 3^{X(C)}$ são omitidos, mas podem ser facilmente deduzidos a partir dos exemplos discutidos;

c) As macros End_A e End_Q referem-se ao endereçamento indireto definido no Exemplo 3.15. O conteúdo do registrador A é denotado por a.

```
u:    faça A := 2^Q × 3^X(C) vá_para End_A
...
a:    faça X(C) := s vá_para a1          grava na fita
a1:   faça adC vá_para a2          move a cabeça para a direita
a2:   faça Q := v vá_para End_Q          novo estado
```

Adicionalmente, a cada rótulo final f corresponde o seguinte trecho de programa em P, o qual especifica que o conteúdo de X ("fita") é atribuído a Y (pois, em Norma, a função de saída retorna o valor do registrador Y):

```
f:    faça Y := X vá_para fim
...
fim:
```

Codificação. O conteúdo inicial da fita é codificado em X, visto como um arranjo unidimensional (ver o item *Fita* acima) ;

Decodificação. Decodifica de Y, o conteúdo final da fita. A decodificação é o inverso do especificado no item *Fita* acima.

Sugere-se, como exercício, a simulação, por Norma, das condições de ACEITA e REJEITA de parada da Máquina de Turing. ❑

Teorema 3.16 Máquina Norma ≤ Máquina de Turing.

O formalismo Máquina Norma pode ser simulado pelo formalismo Máquina de Turing.

Prova:

Lembre-se que, de acordo com a Observação 3.3 - Programa Monolítico × Programa Recursivo, é suficiente considerar o caso de programas monolíticos. Então, suponha um programa monolítico P para Norma, mas com somente dois registradores X e Y (como na Observação 3.2). Então, a simulação de P para Norma por uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⊙)$, onde o alfabeto Σ é o conjunto unário { 1 }, pode ser definida como segue:

Registrador X. O conteúdo do registrador X é armazenado em unário nas células pares da fita de M. Assim, se o natural em X é x, então x células pares da fita possuem o símbolo 1;

Registrador Y. Analogamente ao registrador X, o registrador Y é armazenado na fita em unário, mas nas células ímpares (excetuando-se a primeira, reservada para o marcador de início de fita ⊙);

Rótulos de Instruções. A cada rótulo r de instrução de P corresponde um estado q_r de M. Aos rótulos inicial e finais correspondem os estados inicial e finais, respectivamente;

Programa. O programa P de Norma pode ser simulado por M como segue:

a) *Adição.* Uma instrução rotulada de P da seguinte forma:

```
r:    faça ad_K vá_para s
```

pode ser simulada por um trecho da função programa de M resumido como segue:

a.1) No estado q_r, move a cabeça, pesquisando as células pares (caso K = X) ou ímpares (caso K = Y), até encontrar o primeiro branco, o qual é substituído pelo símbolo 1;

a.2) Reposiciona a cabeça no início da fita e assume o estado q_s;

b) *Subtração.* Uma instrução rotulada de P da seguinte forma:

```
r:    faça sub_K vá_para s
```

pode ser simulada por um trecho da função programa de M resumido como segue:

b.1) No estado q_r, move a cabeça, pesquisando as células pares (caso K = X) ou ímpares (caso K = Y), até encontrar o último símbolo 1, o qual é substituído por branco. Caso a primeira célula pesquisada já contenha o símbolo branco, nada é substituído;

b.2) Reposiciona a cabeça no início da fita e assume o estado q_s

c) *Teste.* Uma instrução rotulada de P da seguinte forma:

```
r:    se zero_K vá_para s senão vá_para t
```

pode ser simulada por um trecho da função programa de M resumido por:

c.1) No estado q_r, move a cabeça, pesquisando a primeira célula par (caso K = X) ou segunda célula ímpar (caso K = Y - lembre-se que a primeira célula ímpar é reservada para o marcador de início de fita ⊙);

c.2) Caso a célula pesquisa contenha o símbolo branco, reposiciona a cabeça no início da fita e assume o estado q_s; caso contrário, reposiciona a cabeça no início da fita e assume o estado q_t.

Codificação. O conteúdo inicial do registrador X é codificado em unário nas células pares da fita de M (ver item *Registrador X*);

Decodificação. Decodifica da fita, o valor final do registrador Y. A decodificação é o inverso do especificado no item *Registrador X* acima. ⌟

3.5 Outros Modelos de Máquinas Universais

Uma das razões para considerar-se a Máquina de Turing (ou Norma) como o mais geral dispositivo de computação é o fato de que todos os demais modelos de máquinas propostos possuem, no máximo, a sua capacidade computacional.

A seguir, são introduzidos os seguintes modelos de máquinas universais:

a) *Máquina de Post.* A principal característica da Máquina de Post é que usa uma estrutura de dados do tipo *fila* para entrada, saída e memória de trabalho. Estruturalmente, a principal característica de uma fila é que o primeiro valor gravado é também o primeiro a ser lido (uma leitura exclui o dado lido), como ilustrado na Figura 3.13. Sugere-se como exercício uma comparação das estruturas de fita e fila.

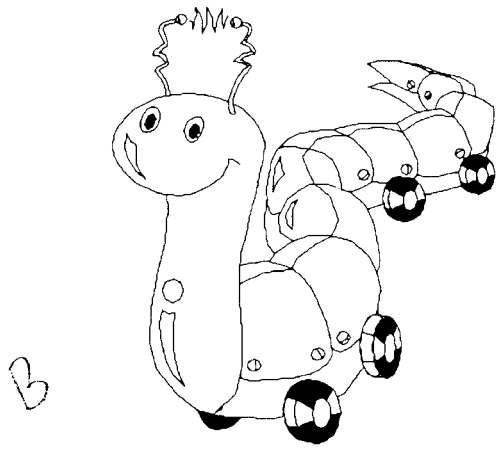
b) *Máquina com Pilhas.* Como o próprio nome indica, a principal característica da Máquina com Pilhas é que usa estrutura do tipo pilha (introduzida no Exemplo 3.14) como memória de trabalho. Entretanto, são necessárias pelo menos duas pilhas para que a Máquina seja, de fato, Universal.

Figura 3.13 Estrutura do tipo fila

Os modelos Máquina de Post e Máquina Finita ou Autômato Finito com duas Pilhas constituem classes de máquinas equivalentes à da Máquina de Turing. Assim, para cada Máquina de Turing existe uma Máquina de Post e um Autômato Finito com duas Pilhas capaz de realizar o mesmo processamento e vice-versa.

Um resultado importante que pode ser estabelecido das equivalências destas máquinas universais é que, referente às estruturas de dados:

- infinitos registradores (Norma), fita infinita (Máquina de Turing) e fila (Máquina de Post) são estruturas que podem simular umas às outras;
- são necessárias pelo menos duas pilhas (Máquina com Duas Pilhas) para simular as demais estruturas de dados. Tal fato é detalhado adiante em 3.7 - Hierarquia de Classes de Máquinas.



3.5.1 Máquina de Post

Assim como Turing, Emil Leon Post propôs, também em 1936, um modelo de Máquina Universal denominado *Máquina de Post* ([POS36]). Uma Máquina de Post consiste, basicamente, de duas partes:

a) *Variável X*. Trata-se de uma variável do tipo fila e é utilizada como entrada, saída e memória de trabalho.

b) *Programa*. É uma seqüência finita de instruções, representado como um diagrama de fluxos (espécie de fluxograma), onde cada vértice é uma instrução. As instruções podem ser de quatro tipos:

partida
parada
desvio
atribuição

A variável X não possui tamanho nem limite fixos. Seu comprimento é igual ao comprimento da palavra corrente armazenada. Os símbolos podem pertencer ao alfabeto de entrada ou a {#}, único símbolo auxiliar. Inicialmente, o valor de X é a palavra de entrada. Caso X não contenha símbolos, a entrada é vazia, representada por ε.

Definição 3.17 Máquina de Post.

Uma *Máquina de Post* é uma tripla:

$$M = (\Sigma, D, \#)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- D *programa* ou *diagrama de fluxos* construído a partir de componentes elementares denominados partida, parada, desvio e atribuição;
- # *símbolo auxiliar*.

As componentes elementares de um diagrama de fluxos são como segue:

a) *Partida*. Existe somente uma instrução de início (partida) em um programa a qual é representada como ilustrado na Figura 3.14 (esquerda);

b) *Parada*. Existem duas alternativas de instruções de parada em um programa, uma de aceitação (aceita) e outra de rejeição (rejeita), representadas como ilustrado na Figura 3.14 (direita);

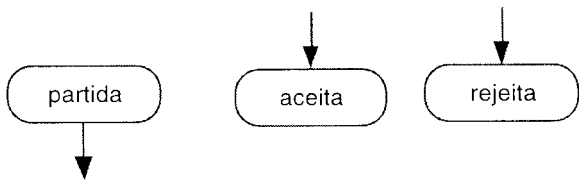


Figura 3.14 Partida (esquerda) e parada (direita) em um diagrama de fluxos

c) *Desvio* ou *Teste*. Determina o fluxo do programa de acordo com o símbolo mais à esquerda da palavra armazenada na variável X (início da fila). Também deve ser prevista a possibilidade de X conter a palavra vazia. Portanto, é um desvio condicional, e trata-se de uma função total, ou seja, definida para todos os valores do domínio. Assim, se o cardinal de Σ é n, então existem n+2 arestas de desvios condicionais, pois se deve incluir as possibilidades # e ε, como ilustrado na Figura 3.15, onde X←ler(X) denota uma *leitura destrutiva* ou seja, que lê o símbolo mais à esquerda da palavra, retirando da mesma o símbolo lido.

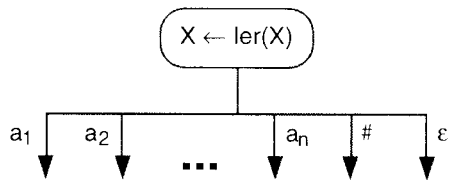


Figura 3.15 Desvio em um diagrama de fluxos

d) *Atribuição*. Concatena o símbolo indicado (pertencente a $\Sigma \cup \{\#\}$) à direita da palavra armazenada na variável X (fim da fila). A operação de atribuição é representada como ilustrado na Figura 3.16, supondo que $s \in \Sigma \cup \{\#\}$. ❑

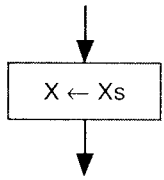


Figura 3.16 Atribuição em um diagrama de fluxos

Em um diagrama de fluxos, existe somente uma instrução de partida, mas podem existir diversas (zero ou mais) instruções de parada, tanto de aceitação como de rejeição. Uma palavra de entrada é aceitada ou rejeitada, se a computação, iniciada com a variável X, contendo a entrada, atingir uma instrução aceita ou rejeita, respectivamente. Note-se que é perfeitamente possível uma Máquina de Post ficar em *loop* infinito (sugere-se, como exercício, exemplificar um *loop* infinito).

Em um desvio, se X contém a palavra vazia ε, então segue o fluxo correspondente. Caso contrário, lê o símbolo mais à esquerda da palavra em X e o remove após a decisão de qual aresta do fluxo indica a próxima instrução.

EXEMPLO 3.21 Máquina de Post – Duplo Balanceamento.

Considere a seguinte linguagem introduzida no Exemplo 3.16:

$$Duplo\_Bal = \{a^n b^n \mid n \geq 0\}$$

A Máquina de Post:

$$Post\_Duplo\_Bal = (\{a, b\}, D, \#)$$

onde D é como ilustrado na Figura 3.17, é tal que:

$ACEITA(Post\_Duplo\_Bal) = Duplo\_Bal$

$REJEITA(Post\_Duplo\_Bal) = \Sigma^* - Duplo\_Bal$

e, portanto, $LOOP(Post\_Duplo\_Bal) = \varnothing$

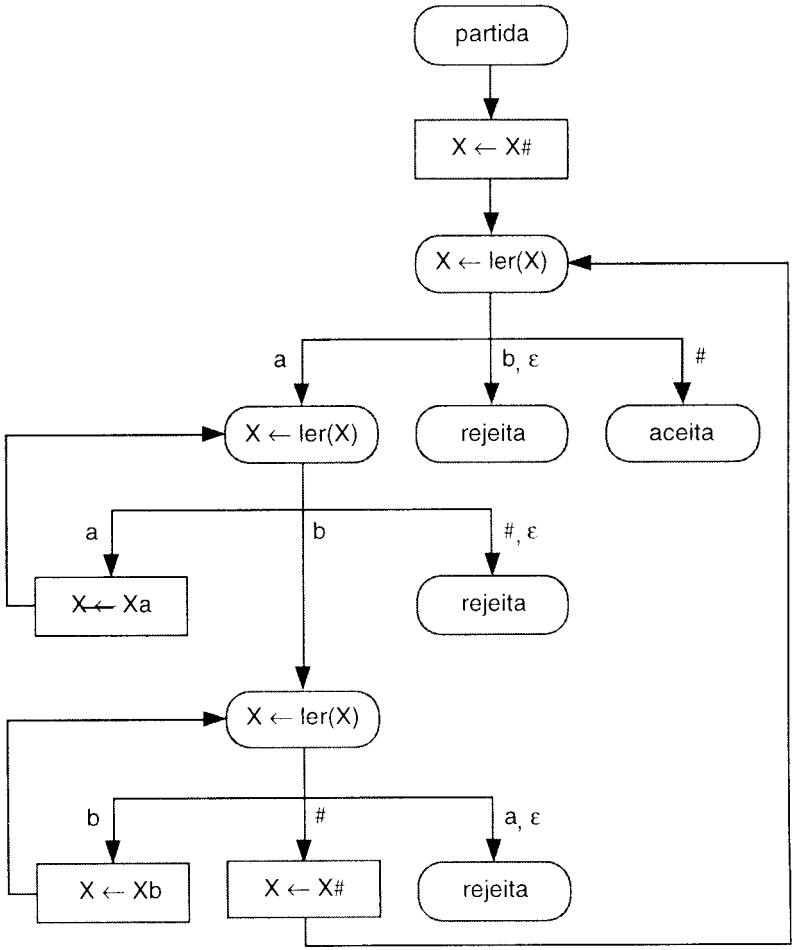


Figura 3.17 Diagrama de Fluxo da Máquina de Post – Duplo Balanceamento

O algoritmo lê e remove o primeiro símbolo a. Após, realiza uma varredura circular em busca do correspondente b. Esta varredura é realizada através de sucessivas leituras (e remoções), armazenando o símbolo lido à direita de X. Ao encontrar o b, este é removido, e uma nova varredura circular é realizada até o fim da palavra de entrada (identificado pelo símbolo auxiliar #, atribuído a X no início do processamento). Este ciclo é repetido até restar a palavra vazia ou ocorrer alguma condição de rejeição. Compare este algoritmo com o apresentado no Exemplo 3.16 para a Máquina de Turing. ⌟

A seguir prova-se que a Máquina de Post é equivalente à Máquina de Turing, reforçando a evidência de que são máquinas universais. Resumidamente, a prova é como segue:

a) *Turing* ≤ *Post*. A estrutura de fita da Máquina de Turing é simulada em Post, usando a variável X, onde a posição corrente da cabeça corresponde à primeira posição da fila;

b) *Post* ≤ *Turing*. A variável X da Máquina de Post é simulada em Turing, usando a estrutura de fita, onde a primeira posição da fila corresponde à posição corrente da cabeça da fita.

Nos teoremas a seguir, são apresentados os resumos dos algoritmos que realizam a construção da máquina equivalente. A demonstração completa necessita detalhar tais algoritmos e provar que funcionam sempre, usando a técnica de indução.

Teorema 3.18 Máquina de Turing ≤ Máquina de Post.

O formalismo Máquina de Turing pode ser simulado pelo formalismo Máquina de Post.

Prova:

Suponha uma Máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, ß, ⌾)$. Então, a simulação de M por uma Máquina de Post $M' = (\Sigma \cup V \cup \{⌾\}, D, \#)$ pode ser definida como segue:

a) *Fita*. A fita é simulada pela variável X, e a posição corrente da cabeça da fita é representada pela primeira posição da fila. A fita ilustrada na Figura 3.18 é simulada pela variável X com o seguinte conteúdo:

$$X = a_3\, a_4 \ldots a_n \,\#\, a_1\, a_2$$

Observa-se que o símbolo especial # foi introduzido, para indicar na variável X o que estava à esquerda da cabeça da fita, a partir do inicio da fita;

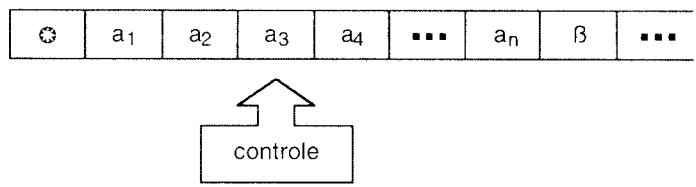


Figura 3.18 Fita e posição da cabeça da fita

b) *Movimento para a Esquerda*. Considere o movimento para a esquerda da cabeça da fita ilustrado na Figura 3.19. Assim, o conteúdo da variável X antes do movimento é o seguinte:

$$X = a_3\, a_4 \ldots a_n \,\#\, a_1\, a_2$$

Para simular o movimento para a esquerda, bem como a substituição do símbolo a_3 por A_3, é necessário alterar o conteúdo de X como segue:

$$X = a_2\, A_3\, a_4 \ldots a_n \,\#\, a_1$$

Para tal, é necessário executar tantos testes e atribuições quantos forem os símbolos da palavra corrente. O detalhamento de um diagrama de fluxos de uma Máquina de Post que simula este movimento é sugerido como exercício:

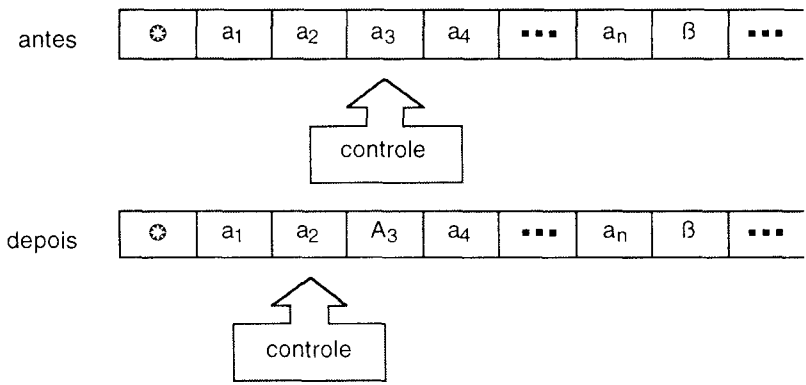


Figura 3.19 Movimento para a esquerda: antes (acima) e depois (abaixo)

c) *Movimento para a Direita*. Considere o movimento para a direita da cabeça da fita ilustrado na Figura 3.20. Assim, o conteúdo da variável X antes do movimento é o seguinte:

$$X = a_2\, a_3\, a_4 \ldots a_n \# a_1$$

Para simular o movimento para a direita, é necessário alterar o conteúdo de X como segue, o que é trivial:

$$X = a_3\, a_4 \ldots a_n \# a_1\, A_2$$

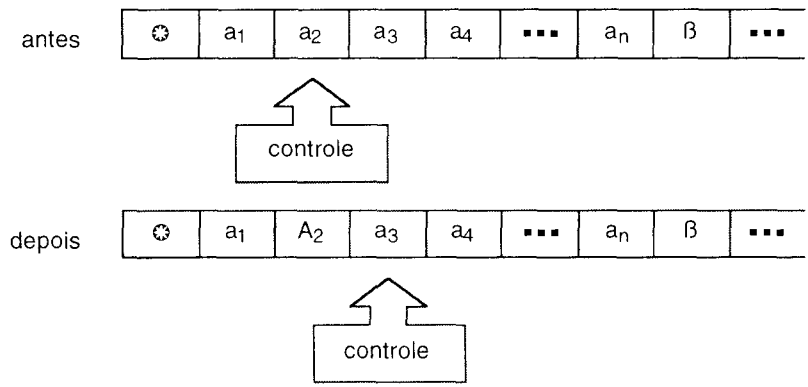


Figura 3.20 Movimento para a direita: antes (acima) e depois (abaixo)

d) *Estados*. A simulação dos estados é como segue:

d.1) *Estado Inicial*. Simulado pela instrução partida;

d.2) *Estados Finais*. Simulados pela instrução aceita;

d.3) *Demais Estados*. Cada estado corresponde a uma instrução teste;

e) *Condições de Rejeição*. As condições de rejeição da Máquina de Turing (função programa indefinida ou movimento inválido) são simuladas em Post pela instrução rejeita. ⌟

Teorema 3.19 Máquina de Post ≤ Máquina Turing

O formalismo Máquina de Post pode ser simulado pelo formalismo Máquina de Turing.

Prova:

Suponha uma Máquina de Post $M = (\Sigma, D, \#)$. Então, a simulação de M por uma Máquina de Turing $M' = (\Sigma, Q, \Pi, q_0, F, \{\#\}, ß, ⌾)$ pode ser definida como segue:

a) *Variável X*. A variável X é simulada pela fita, e a posição da cabeça da fita representa a posição mais à esquerda da fila. Assim, para

$$X = a_1\, a_2\, a_3 \ldots a_m \,\#\, a_{m+1} \ldots a_n$$

a fita e a cabeça da fita são como ilustrado na Figura 3.21.

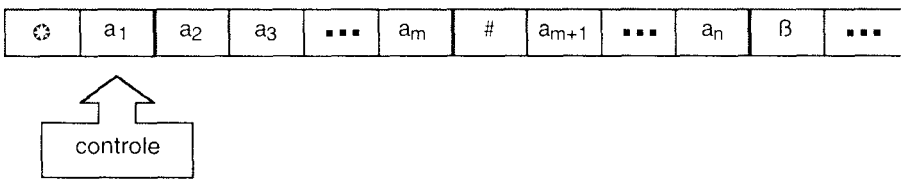


Figura 3.21 Simulação da fila pela fita

b) *Desvio*. Suponha que o conteúdo da variável X é:

$$X = a_1\, a_2\, a_3 \ldots a_m \,\#\, a_{m+1} \ldots a_n$$

Portanto, a leitura e remoção do símbolo mais à esquerda resulta no seguinte conteúdo de X:

$$X = a_2\, a_3 \ldots a_m \,\#\, a_{m+1} \ldots a_n$$

Isto pode ser simulado pela alteração da fita e da cabeça da fita ilustrada na Figura 3.22.

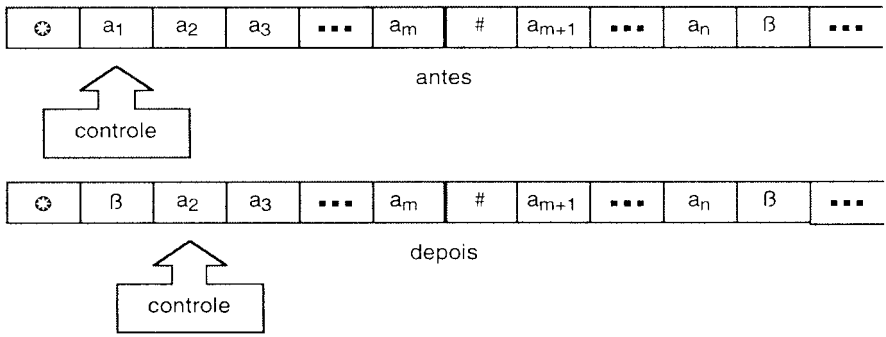


Figura 3.22 Simulação da de uma leitura e remoção: antes (acima) e depois (abaixo)

c) *Atribuição*. A concatenação de um símbolo s deve sempre ser à direita do conteúdo da variável X (ou seja, no fim da fila). Assim, para o conteúdo de X como segue:

$$X = a_1 \ldots a_m \# a_{m+1} \ldots a_n$$

resulta no seguinte conteúdo de X:

$$X = a_1 \ldots a_m \# a_{m+1} \ldots a_n s$$

o que pode ser simulado pela Máquina de Turing como ilustrado na Figura 3.23. Para tal, é necessário mover a cabeça para o fim da fita, gravar o símbolo s e retornar para a posição correspondente ao primeiro símbolo da fila.

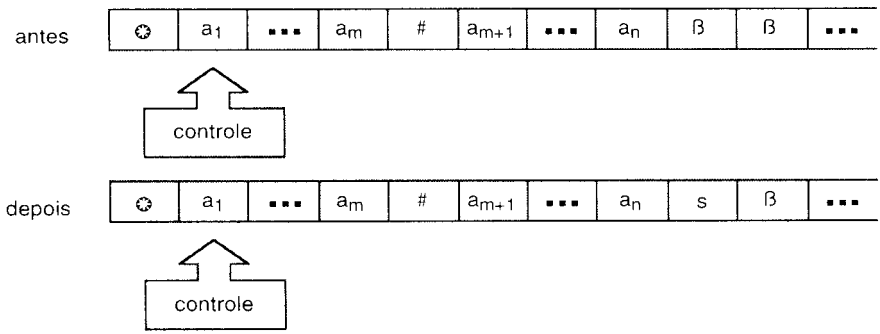
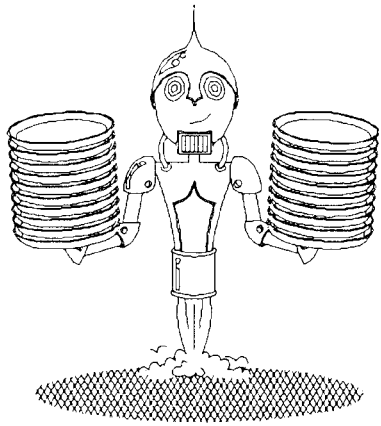


Figura 3.23 Simulação de uma concatenação: antes (acima) e depois (abaixo)

d) *Partida*. A instrução partida pode ser simulada em uma Máquina de Turing usando o estado inicial;

e) *Aceita*. Uma instrução aceita pode ser simulada em uma Máquina de Turing usando um estado final;

f) *Rejeita*. Uma instrução rejeita pode ser simulada em uma Máquina de Turing usando uma condição excepcional de parada (como um movimento inválido).❑



3.5.2 Máquina com Pilhas

Máquina com Pilhas diferencia-se das Máquinas de Turing e de Post principalmente pelo fato de possuir a memória de entrada separada das memórias de trabalho e de saída. Essas estruturas auxiliares são do tipo pilha, e cada máquina possui zero ou mais pilhas.

Um resultado importante que será detalhado adiante é o fato de que a Classe das Máquinas com Duas Pilhas possui o mesmo poder computacional que a Classe das Máquinas de Turing. Assim, o uso de mais de duas pilhas não aumenta o poder das máquinas com pilhas. Entretanto, com somente uma pilha, o poder computacional é reduzido significativamente. Veja a Figura 3.36, onde é feita uma comparação do poder computacional dos diversos formalismos.

As seguintes conclusões podem ser estabelecidas sobre o número de pilhas e o poder computacional das máquinas com pilhas:

a) *Máquina Finita*. Uma Máquina Finita, que corresponde a uma *Máquina Sem Pilha*, possui um poder computacional relativamente restrito, pois não tem memória auxiliar para armazenar informações de trabalho. Por exemplo, não existe Máquina Sem Pilha capaz de reconhecer um duplo balanceamento como em $\{a^n b^n \mid n \geq 0\}$. Entretanto, é uma classe de máquinas de fundamental importância no estudo das *Linguagens Formais* ([MEN98]) como em editores de textos e analisadores léxicos;

b) *Máquina com Uma Pilha*. A Classe das Máquinas com Uma Pilha, embora mais poderosa que a Classe dos Máquinas Finitas, ainda possui uma capacidade computacional restrita. Por exemplo, não existe Máquinas com Uma Pilha capaz de reconhecer um triplo balanceamento como em $\{a^n b^n c^n \mid n \geq 0\}$ (procure intuir por quê não existe tal máquina). Entretanto, também é uma classe de máquinas de fundamental importância no estudo das *Linguagens Formais* ([MEN98]) como em compiladores de linguagens tipo Pascal;

c) *Máquina com Duas Pilhas*. Conforme será verificado adiante, a Classe das Máquinas com Duas Pilhas possui o mesmo poder computacional que a Classe das Máquinas de Turing ou de Post;

d) *Máquina com Mais de Duas Pilhas*. A Classe dos Máquinas com Mais de Duas Pilhas possui o mesmo poder computacional que a Classe dos Máquinas com Duas Pilhas. Ou seja, para qualquer máquina com mais de duas pilhas, é possível construir uma equivalente com, no máximo, duas pilhas que realiza o mesmo processamento.

Pilhas constituem um tipo de estrutura de dados usado em Ciência da Computação há algum tempo. Entretanto, somente na década de 1960, o modelo Máquina com Pilhas foi formalizado e devidamente estudado ([OET61], [SCH67] e [EVE63]).

Uma Máquina com Pilhas consiste, basicamente, de três partes:

a) *Variável X*. Trata-se de uma variável de entrada, similar à da Máquina de Post, mas usada somente para entrada;

b) *Variáveis* Y_i. Trata-se de variáveis do tipo pilha, e são utilizadas como memória de trabalho;

c) *Programa*. É uma seqüência finita de instruções, e é representado como um diagrama de fluxos (espécie de fluxograma), onde cada vértice é uma instrução. As instruções podem ser de cinco tipos:

   partida
   parada
   desvio
   desempilha
   empilha

A variável X não possui tamanho nem limite fixos. Os símbolos pertencem ao alfabeto de entrada. Inicialmente, o valor de X é a palavra de entrada e seu comprimento é igual ao comprimento da palavra corrente armazenada. Caso X não contenha símbolos, a entrada é vazia, representada por ε.

As variáveis Y_i, em número variável mas finito, também não possuem tamanho nem limite fixos, e os símbolos pertencem ao alfabeto de entrada (não existem símbolos auxiliares). Inicialmente, o valor de cada pilha Y_i é a palavra vazia, e seu comprimento, é igual ao comprimento da palavra corrente armazenada.

É importante reparar que, embora os modelos Máquina de Post e Máquina com Pilhas possuam alguma semelhança, existe uma grande diferença na forma de manipular seus dados. Na Máquina de Post, em uma fila, pode-se ler (e remover) símbolos em uma extremidade e armazenar na outra. Assim, os dados "circulam". Em uma Máquina com Pilhas, o armazenamento e leitura (e remoção) é sempre na mesma extremidade.

Definição 3.20 Máquina com Pilhas.

Uma *Máquina com Pilhas* é uma dupla:

$$M = (\Sigma, D)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- D *programa* ou *diagrama de fluxos* construído a partir de componentes elementares denominados partida, parada, desvio, empilha e desempilha.

As componentes elementares de um diagrama de fluxos são como segue:

a) *Partida*. Existe somente uma instrução de início (partida) em um programa a qual é representada como ilustrado na Figura 3.24 (esquerda);

b) *Parada*. Existem duas alternativas de instruções de parada em um programa, uma de aceitação (aceita) e outra de rejeição (rejeita), representadas como ilustrado na Figura 3.24 (direita);

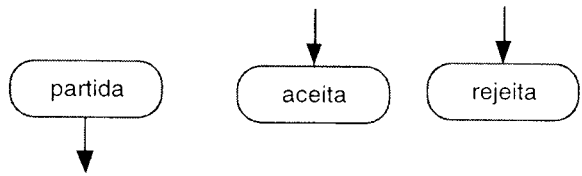


Figura 3.24 Partida (esquerda) e parada (direita) em um diagrama de fluxos

c) *Desvio* (ou *Teste*) e *Desempilha*. Determina o fluxo do programa de acordo com o símbolo mais à esquerda da palavra armazenada na variável X (desvio) ou no topo da pilha Y_i (desempilha). Também deve ser prevista a possibilidade de a variável conter a palavra vazia. Portanto, é um desvio condicional, e trata-se de uma função total, ou seja, definida para todos os valores do domínio. Assim, se o cardinal de Σ é n, então existem n+1 arestas de desvios condicionais, pois se deve incluir a possibilidade ε, como ilustrado na Figura 3.25, onde $X \leftarrow$ ler(X) denota uma *leitura destrutiva*, ou seja, que lê o símbolo mais à esquerda de X ou do topo de Y_i, retirando da estrutura o símbolo lido.

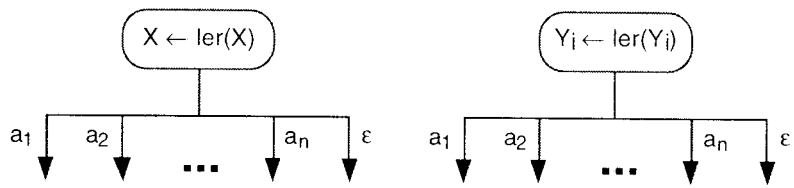


Figura 3.25 Desvio (esquerda) e desempilha (direita) em um diagrama de fluxos

d) *Empilha*. Empilha um símbolo (pertencente a Σ) no topo da pilha indicada, ou seja, concatena o símbolo na extremidade da palavra armazenada na variável Y_i. A operação empilha é representada como ilustrado na Figura 3.26, supondo que $s \in \Sigma$. ❑

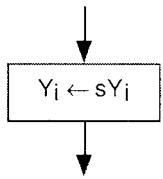


Figura 3.26 Empilha em um diagrama de fluxos

Analogamente à Máquina de Post, em uma Máquina com Pilhas, tem-se que:

- em um diagrama de fluxos, existe somente uma instrução de partida, mas podem existir diversas (zero ou mais) instruções de parada, tanto de aceitação como de rejeição;
- uma palavra de entrada é aceitada ou rejeitada, se a computação, iniciada com a variável X, contendo a entrada, atingir uma instrução aceita ou rejeita, respectivamente;
- é perfeitamente possível uma Máquina com Pilhas ficar em *loop* infinito (sugere-se, como exercício, exemplificar um *loop* infinito);
- em um desvio (respectivamente, desempilha), se X (respectivamente, Y_i) contém a palavra vazia ε, então segue o fluxo correspondente; caso contrário, lê o símbolo mais à esquerda de X (respectivamente, no topo de Y_i) e o remove após a decisão de qual aresta do fluxo indica a próxima instrução.

EXEMPLO 3.22 Máquina com Pilhas - Duplo Balanceamento.

Considere a seguinte linguagem introduzida no Exemplo 3.16:

$$\text{Duplo\_Bal} = \{a^n b^n \mid n \geq 0\}$$

A Máquina com Pilhas:

$$\text{Pilhas\_Duplo\_Bal} = (\{a, b\}, D)$$

onde D é como ilustrado na Figura 3.27, é tal que:

$$\text{ACEITA(Pilhas\_Duplo\_Bal)} = \text{Duplo\_Bal}$$
$$\text{REJEITA(Pilhas\_Duplo\_Bal)} = \Sigma^* - \text{Duplo\_Bal}$$

e, portanto, LOOP(Pilhas_Duplo_Bal) = $\varnothing$

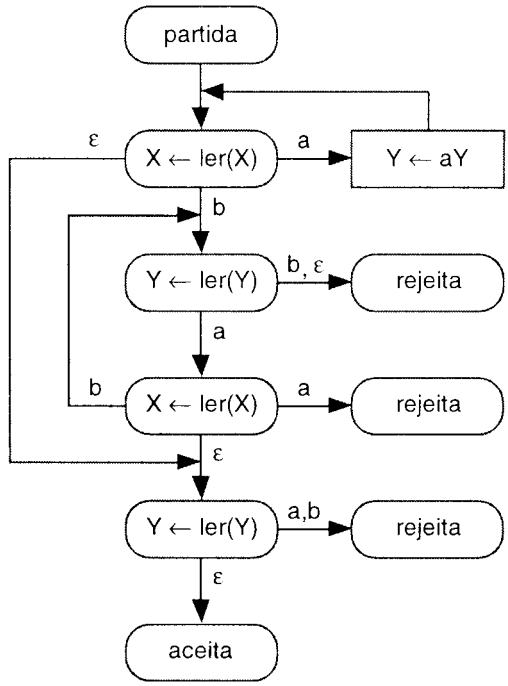


Figura 3.27 Diagrama de Fluxo da Máquina com Pilhas – Duplo Balanceamento

O algoritmo lê o prefixo de símbolos a e empilha na única pilha utilizada Y. Após, para cada símbolo b em X deve existir um correspondente a em Y. Compare este algoritmo com o apresentado no Exemplo 3.21 para a Máquina de Post. ❑

EXEMPLO 3.23 Máquina com Pilhas – Triplo Balanceamento.

Considere a seguinte linguagem introduzida no Exemplo 3.18:

$$Triplo\_Bal = \{ a^n b^n c^n \mid n \geq 0 \}$$

A Máquina com Pilhas:

$$Pilhas\_Triplo\_Bal = (\{a, b, c\}, D)$$

onde D é como ilustrado na Figura 3.29, é tal que:

$ACEITA(Pilhas\_Triplo\_Bal) = Triplo\_Bal$

$REJEITA(Pilhas\_Triplo\_Bal) = \Sigma^* - Triplo\_Bal$

e, portanto, $LOOP(Pilhas\_Triplo\_Bal) = \varnothing$

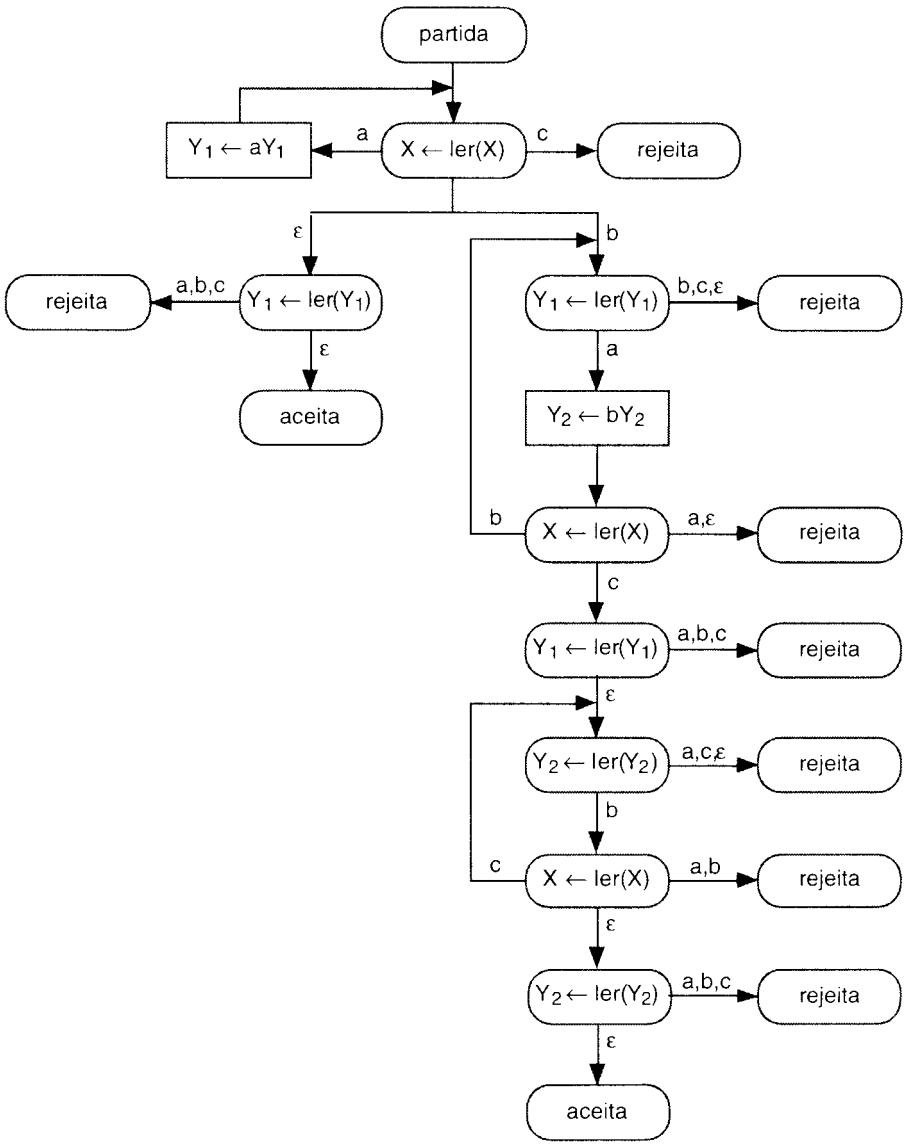


Figura 3.28 Diagrama de Fluxo da Máquina com Pilhas – Triplo Balanceamento

Note-se que foram usadas duas pilhas. O algoritmo lê o prefixo de símbolos a e empilha em Y_1. Após, para cada símbolo b em X deve existir um correspondente a em Y_1. Adicionalmente, empilha a subpalavra de símbolos b em Y_2. Por fim, para cada c em X, deve existir um correspondente b em Y_2. ❑

EXEMPLO 3.24 Máquina com Pilhas – Prefixo.

Considere a seguinte linguagem:

Prefixo_aaa = {w | w ∈ {a, b}* e w possui a subpalavra aaa como prefixo}

Por exemplo, aaabab, aaaaaaab e aaa são palavras de Prefixo_aaa. A Máquina com Pilhas:

Pilhas_Prefixo_aaa = ({a, b}, D)

onde D é como ilustrado na Figura 3.29, é tal que:

ACEITA(Pilhas_Prefixo_aaa) = Prefixo_aaa
REJEITA(Pilhas_Prefixo_aaa) = Σ^* - Prefixo_aaa

e, portanto, LOOP(Pilhas_Prefixo_aaa) = ∅

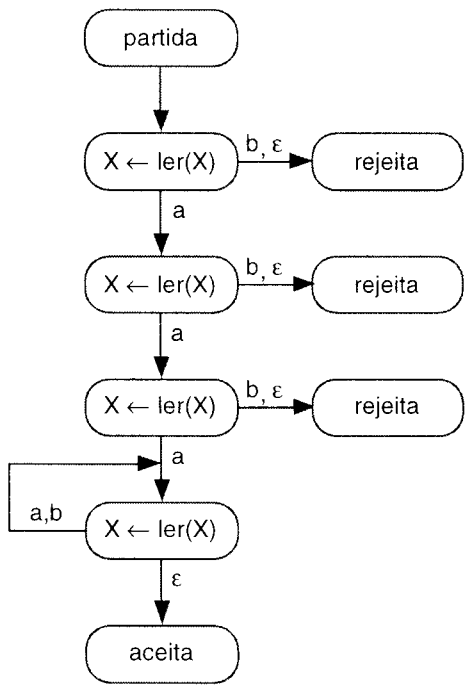


Figura 3.29 Diagrama de Fluxo da Máquina com Pilhas – Prefixo

Note-se que *não* foram usadas pilhas. O algoritmo lê o prefixo aaa executando três leituras. Após, qualquer sufixo é aceito. ❑

3.5.3 Autômato com Duas Pilhas

O *Autômato com Duas Pilhas* ou simplesmente *Autômato com Pilhas* é uma Máquina Universal similar à Máquina com Duas Pilhas. A principal diferença é que o programa é especificado, usando a noção de estados (de forma análoga à Máquina de Turing), e não como um diagrama de fluxos.

Diagramas de fluxos são úteis no desenvolvimento de algoritmos e na visualização da sua estruturação e computação. Já as máquinas com estados são mais indicadas para estudos teórico-formais pois facilitam provas e estudos de facilidades especiais como não-determinismo (será introduzido em 3.6.1 - Não-Determinismo). Adicionalmente, simuladores de modelos baseados em estados são, em geral, mais fáceis de serem implementados.

A abordagem que segue é muito usada no estudo das Linguagens Formais. Um Autômato com Pilhas é composto, basicamente, por quatro partes, como segue:

a) *Fita*. Dispositivo de entrada que contém a informação a ser processada;

b) *Pilhas*. Memórias auxiliares que podem ser usadas livremente para leitura e gravação;

c) *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma cabeça de fita e uma cabeça para cada pilha;

d) *Programa ou Função de Transição*. Comanda a leitura da fita, leitura e gravação das pilhas e define o estado da máquina.

Uma pilha é dividida em células, armazenando, cada uma, um símbolo do alfabeto auxiliar (pode ser igual ao alfabeto de entrada). Lembre-se que, em uma estrutura do tipo pilha, a leitura e a gravação são sempre na mesma extremidade (topo). Uma pilha não possui tamanho fixo e nem máximo, sendo seu tamanho corrente igual ao tamanho da palavra armazenada. Seu valor inicial é vazio.

A unidade de controle possui um número finito e predefinido de estados. Possui uma cabeça de fita e uma cabeça para cada pilha, como segue:

a) *Cabeça da Fita*. Unidade de leitura a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita. É possível testar se a entrada foi completamente lida;

b) *Cabeça da Pilha*. Unidade de leitura e gravação para cada pilha a qual move para cima ao gravar e para baixo ao ler um símbolo. Acessa um símbolo de cada vez, estando sempre posicionada no topo. A leitura exclui o símbolo lido (leitura destrutiva). É possível testar se a pilha está vazia.

O programa é uma função parcial que, dependendo do estado corrente, do símbolo lido da fita e do símbolo lido de cada pilha, determina o novo estado e o símbolo a ser gravado em cada pilha.

Definição 3.21 Autômato com Duas Pilhas.

Um *Autômato com Duas Pilhas* M ou simplesmente *Autômato com Pilhas* M é uma 6-upla:

$$M = (\Sigma, Q, \Pi, q_0, F, V)$$

onde:

- Σ *alfabeto de símbolos de entrada*;
- Q *conjunto de estados* possíveis do autômato o qual é finito;
- Π *função programa* ou *função de transição*:

  $\Pi: Q \times (\Sigma \cup \{\varepsilon, ?\}) \times (V \cup \{\varepsilon, ?\}) \times (V \cup \{\varepsilon, ?\}) \rightarrow Q \times (V \cup \{\varepsilon\}) \times (V \cup \{\varepsilon\})$

  a qual é uma função parcial;
- q_0 *estado inicial* do autômato tal que q_0 é elemento de Q;
- F *conjunto de estados finais* tal que F está contido em Q;
- V *alfabeto auxiliar*. ❑

As seguintes características da função programa devem ser consideradas:

- a função pode não ser total, ou seja, pode ser indefinida para alguns argumentos do conjunto de partida; a omissão do parâmetro de leitura, representada por "?", indica o teste da correspondente pilha vazia ou toda palavra de entrada lida;
- o símbolo ε na leitura da fita ou de alguma pilha indica que o autômato não lê nem move a cabeça. Note-se que pelo menos uma leitura deve ser realizada ou sobre a fita ou sobre alguma pilha;
- o símbolo ε na gravação indica que nenhuma gravação é realizada na pilha (e não move a cabeça).

Resumidamente, a função programa considera:

- estado corrente;
- símbolo lido da fita (pode ser omitido) ou teste se toda a palavra de entrada foi lida;
- símbolo lido de cada pilha (pode ser omitido) ou teste de pilha vazia;

para determinar:

- novo estado;
- símbolo gravado em cada pilha (pode ser omitido).

Por exemplo, $\Pi(p, ?, a, \varepsilon) = \{(q, \varepsilon, b)\}$ indica que:

se:	então:
• no estado p	• assume o estado q
• a entrada foi completamente lida (na fita)	• não grava na pilha 1
• o topo da pilha 1 contém o símbolo a	• grava o símbolo b no topo da pilha 2
• não lê da pilha 2	

Analogamente à Máquina de Turing, a função programa de um Autômato com Pilhas pode ser representada como um grafo direto, como na Figura 3.30.

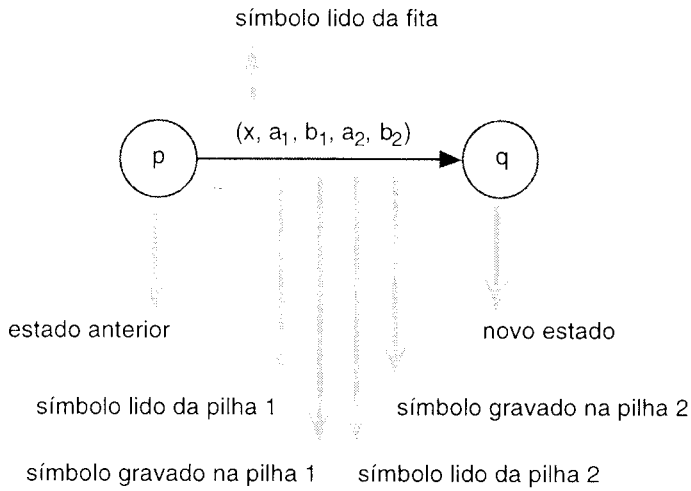


Figura 3.30 Representação da função programa como um grafo

O processamento de um Autômato com Pilhas, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada. Entretanto, é possível que um Autômato com Pilhas nunca atinja uma condição de parada. Nesse caso, fica processando indefinidamente (ciclo ou *loop* infinito). Um exemplo simples de ciclo infinito é um programa que empilha e desempilha um mesmo símbolo indefinidamente, sem ler da fita. As condições de parada são as seguintes:

a) *Estado Final.* O autômato assume um estado final: o autômato pára e a palavra de entrada é aceita;

b) *Função Indefinida.* A função programa é indefinida para o argumento: o autômato pára e a palavra de entrada é rejeitada.

EXEMPLO 3.25 Autômato com Pilhas – Duplo Balanceamento.

Considere a seguinte linguagem introduzida no Exemplo 3.16:

$$\text{Duplo\_Bal} = \{a^n b^n \mid n \geq 0\}$$

O Autômato com Pilhas

$$\text{A2P\_Duplo\_Bal} = (\{a, b\}, \{q_0, q_1, q_f\}, \Pi, q_0, \{q_f\}, \{B\})$$

onde Π é como abaixo, é tal que ACEITA(A2P_Duplo_Bal) = Duplo_Bal (o conjunto LOOP(A2P_Duplo_Bal) é vazio?):

$\Pi(q_0, a, \varepsilon, \varepsilon) = (q_0, B, \varepsilon)$
$\Pi(q_0, b, B, \varepsilon) = (q_1, \varepsilon, \varepsilon)$
$\Pi(q_0, ?, ?, ?) = (q_f, \varepsilon, \varepsilon)$
$\Pi(q_1, b, B, \varepsilon) = (q_1, \varepsilon, \varepsilon)$
$\Pi(q_1, ?, ? ?) = (q_f, \varepsilon, \varepsilon)$

O autômato pode ser representado pelo grafo ilustrado na Figura 3.31. No estado q_0, para cada símbolo a lido da fita, é armazenado um símbolo B na pilha 1. No estado q_1, é realizado um batimento, verificando se, para cada símbolo b da fita, existe um correspondente B na pilha 1. O algoritmo somente aceita se, ao terminar de ler toda a palavra de entrada, as pilhas estiverem vazias. Compare com o Exemplo 3.22 para Máquina com Pilhas.

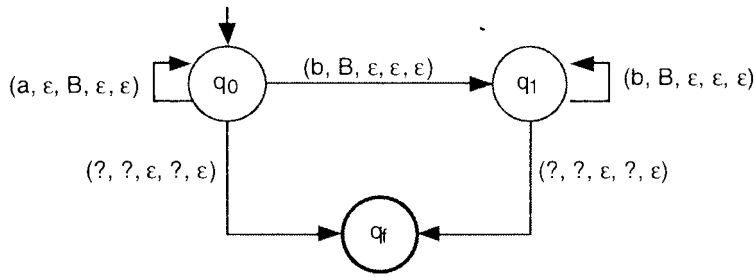


Figura 3.31 Grafo do Autômato com Pilhas - Duplo Balanceamento

Note-se que o algoritmo de exemplo acima não usa a pilha 2. Entretanto, no caso de um triplo balanceamento, é necessário o uso das duas pilhas, como ilustrado no exemplo que segue.

EXEMPLO 3.26 Autômato com Pilhas – Triplo Balanceamento.

Considere a seguinte linguagem introduzida no Exemplo 3.17:

$$\text{Triplo\_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$$

O Autômato com Pilhas

$$\text{A2P\_Triplo\_Bal} = (\{a, b\}, \{q_0, q_1, q_2, q_f\}, \Pi, q_0, \{q_f\}, \{B, C\})$$

ilustrado na Figura 3.32, é tal que ACEITA(A2P_Triplo_Bal) = Triplo_Bal (e sempre pára).

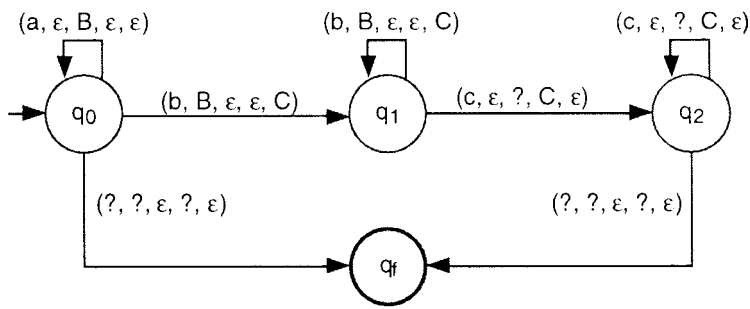


Figura 3.32 Grafo do Autômato com Pilhas - Triplo Balanceamento

No estado q_0, para cada símbolo a lido da fita, é armazenado um símbolo B na pilha 1. No estado q_1, é realizado um batimento, verificando se, para cada símbolo b da fita, existe um correspondente B na pilha 1, bem como é armazenado na pilha 2 um símbolo C. Por fim, no estado q_2, é realizado um batimento, verificando se, para cada símbolo c da fita, existe um correspondente C na pilha 2. O algoritmo somente aceita se ao terminar de ler toda a palavra de entrada as pilhas estiverem vazias. Compare com o Exemplo 3.23 para Máquina com Pilhas. ❑

A seguir é enunciado o teorema que prova que a Classe dos Autômatos com Duas Pilhas é equivalente à Classe das Máquinas de Turing, reforçando a evidência de que ambas são máquinas universais (bem como Post e Norma). Resumidamente, a prova é como abaixo (a prova formal é omitida). Note-se que a mesma idéia básica pode ser usada para provar que a Classe de Máquinas com Pilhas é equivalente à Classe das Máquinas de Turing:

a) *Máquina de Turing* $\leq$ *Autômato com Duas Pilhas.* A estrutura de fita da Máquina de Turing é simulada usando as duas pilhas como segue: a pilha 1 simula o conteúdo da fita à esquerda da cabeça da fita, e a pilha 2, o conteúdo à direita;

b) *Autômato com Duas Pilhas*$\leq$ *Máquina de Turing.* A fita e as duas pilhas do Autômato com Duas Pilhas são simuladas, usando a fita da Máquina de Turing, como segue:

- a palavra de entrada corresponde às primeiras posições da fita da Máquina de Turing;
- a pilha 1 corresponde às células ímpares da fita, após a palavra de entrada;
- analogamente, a pilha 2 corresponde às células pares da fita, após a palavra de entrada.

Teorema 3.22 Máquina de Turing $\times$ Autômato com Duas Pilhas.

O formalismo Máquina de Turing pode ser simulado pelo formalismo Autômato com Duas Pilhas e vice-versa. Logo, são formalismos equivalentes. ❑

3.6 Modificações sobre as Máquinas Universais

Um reforço importante para considerar as máquinas universais introduzidas como os mais gerais dispositivos de computação é o fato de serem equivalentes às diversas versões modificadas do modelo básico, as quais, supostamente, aumentam o poder computacional. Como ilustração, as seguintes modificações são introduzidas:

- máquinas universais não-deterministas;
- Máquina de Turing com fita infinita à esquerda e à direita;
- Máquina de Turing com múltiplas fitas;
- Máquina de Turing com fitas multidimensionais;
- Máquina de Turing com múltiplas cabeças de fita.

As modificações sobre a Máquina de Turing podem ser generalizadas, com variações, para as demais máquinas universais introduzidas.

3.6.1 Não-Determinismo

O não-determinismo é uma importante generalização dos modelos de máquinas. Assim, no caso da Máquina de Turing, para o mesmo estado corrente e símbolo lido, diversas alternativas são possíveis. Cada alternativa é percorrida de forma totalmente independente. Isto significa que as alterações de conteúdo na fita realizadas em um caminho não modificam o conteúdo da mesma nos demais caminhos alternativos. A mesma idéia é válida para a variável X da Máquina de Post ou para as pilhas do Autômato com Pilhas.

Genericamente, a facilidade de *não-determinismo* é interpretada como segue para uma máquina de estados (como seria para uma máquina baseada em diagrama de fluxos?): a máquina, ao processar uma entrada, tem como resultado um conjunto de novos estados. Ou seja, assume um conjunto de estados alternativos, como se houvesse uma multiplicação da unidade de controle, uma para cada alternativa, processando independentemente, sem compartilhar recursos com as demais. Assim, o processamento de um caminho não influi no estado geral e nem no símbolo lido dos demais caminhos alternativos.

Para uma máquina M não-determinística, uma palavra w pertence a ACEITA(M) se existe pelo menos um caminho alternativo que aceita a palavra. Caso contrário, se todas as alternativas rejeitam a entrada, então w pertence a REJEITA(M). Se nenhum caminho aceita a palavra e pelo menos um fica em *loop*, w pertence a LOOP(M).

EXEMPLO 3.27 Autômato com Pilhas Não-Determinístico - Palavra & Reversa

Considere a seguinte linguagem:

$$\text{Palavra\_Reversa} = \{ ww^r \mid w \text{ pertence a } \{a, b\}^* \}$$

A linguagem Palavra_Reversa contém todas as palavras sobre o alfabeto {a, b} tais que a primeira metade é igual à segunda metade, mas invertida ("espelhada"). Por exemplo, as seguintes palavras pertencem à linguagem:

ε	abaaba|abaaba
abbbba	bbbbbbbbbba|abbbbbbbbbb

Este tipo de linguagem pode ser considerada como uma generalização do duplo balanceamento. Mas não deve ser confundida com as palíndromas (palavras que

possuem a mesma leitura da esquerda para a direita e vice-versa), pois estas podem ter comprimento ímpar.

O Autômato com Pilhas ilustrado na Figura 3.33 é tal que ACEITA(APN_Palavra_Reversa) = Palavra_Reversa (o autômato pode entrar em *loop* infinito para alguma entrada?). É não-determinístico devido às duas alternativas de movimentos a partir de q_0, para os mesmo símbolos lidos da fita de entrada (ciclo em q_0 e desvio para q_1). Adicionalmente, o alfabeto auxiliar é igual ao de entrada. Em q_0, é empilhado (pilha 1) o reverso do prefixo. A cada símbolo lido, se existe o correspondente símbolo no topo da pilha 1, então:

- o autômato permanece no estado q_0 e continua empilhando o reverso da entrada (pois não existe controle se já identificou toda a primeira metade);
- ocorre um movimento não-determinista para q_1 (e, portanto, o autômato inicia uma alternativa), o qual verifica se o sufixo da palavra é igual ao conteúdo da pilha 1 até então empilhado. ❑

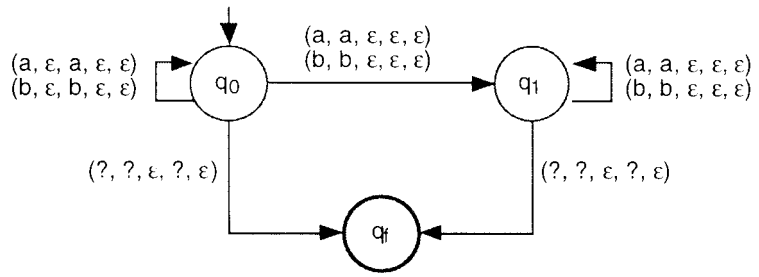


Figura 3.33 Grafo do Autômato com Pilhas Não-Determinístico - palavra e sua reversa

O seguinte Teorema não será demonstrado.

Teorema 3.23 Máquina de Turing × Máquinas Não-Determinísticas.

O formalismo Máquina de Turing pode ser simulado pelos seguintes formalismos e vice-versa.

a) Máquina de Turing Não-Determinística;

b) Máquina de Post Não-Determinística;

c) Autômato com Pilhas Não-Determinístico. ❑

Assim, embora o não-determinismo seja, aparentemente, um significativo acréscimo às máquinas já conhecidas, na realidade não aumenta o poder computacional das mesmas. Ou seja, para cada Máquina de Turing, Máquina de Post ou Autômato com Pilhas Não-Determinístico, é possível construir uma Máquina de Turing (ou Máquina de Post ou Autômato com Pilhas) determinística equivalente, que realiza o mesmo processamento. A recíproca também é verdadeira.

Uma importante aplicação do não-determinismo (como descrito acima) nos sistemas de computadores atuais é o estudo dos sistemas concorrentes, em especial dos conceitos de multiprogramação e multiprocessamento. *Multiprocessamento* é a existência de duas ou mais unidades de processamento realizando computações simultâneas. *Multiprogramação* é um conceito lógico, normalmente implementado em nível de sistema operacional, que permite manter ativo mais de um programa ao mesmo tempo. A principal diferença é que o multiprocessamento apresenta um paralelismo físico, implementado em hardware, e a multiprogramação é uma facilidade aparente ("virtual") implementada em *software*. Esses dois conceitos, combinados ou não, podem ser estudados, em termos do poder computacional, usando a facilidade do não-determinismo. Isto significa que o uso de multiprocessamento, ou de multiprogramação, embora resultem em computadores mais eficientes e flexíveis, não aumentam o seu poder computacional. É importante destacar que o conceito de não-determinismo não deve ser confundido com o de concorrência. Entretanto, a clara diferenciação destes conceitos não é objetivo desta publicação.

3.6.2 Máquina de Turing com Fita Infinita à Esquerda e à Direita

A modificação da definição básica da Máquina de Turing, permitindo que a fita seja infinita dos dois lados, não aumenta o seu poder computacional. Na realidade, a fita infinita à esquerda e à direita é facilmente simulável por uma fita tradicional, como ilustrado na Figura 3.34. Assim, as células pares representam a parte direita da fita e as ímpares a parte esquerda. O símbolo ✪ é usado para controlar a fronteira entre as partes esquerda e direita.

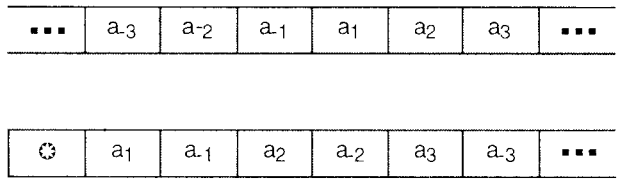


Figura 3.34 Simulação de uma fita infinita dos dois lados

3.6.3 Máquina de Turing com Múltiplas Fitas

A *Máquina de Turing com Múltiplas Fitas* possui k fitas infinitas à esquerda e à direita e k correspondentes cabeças de fita. O processamento é realizado como segue:

- depende do estado corrente da máquina e do símbolo lido em cada uma das fitas;

- grava um novo símbolo em cada uma das fitas, move cada uma das cabeças independentemente para a esquerda ou para a direita, e a máquina assume um (único) novo estado.

Inicialmente, a palavra de entrada é armazenada na primeira fita, ficando as demais com valor branco. A simulação da Máquina de Turing com múltiplas fitas é ilustrada na Figura 3.35 para o caso k = 3 (três fitas/cabeças de fita). As três fitas são simuladas em uma única fita, modificando os alfabetos de entrada e auxiliar. Assim, cada símbolo contido em uma célula é uma 6-upla, sendo 3 componentes para representar as células de cada uma das 3 fitas, e as demais 3 componentes reservadas para marcar a posição corrente das cabeças de cada fita (representadas na Figura 3.35 pelo símbolo ▲).

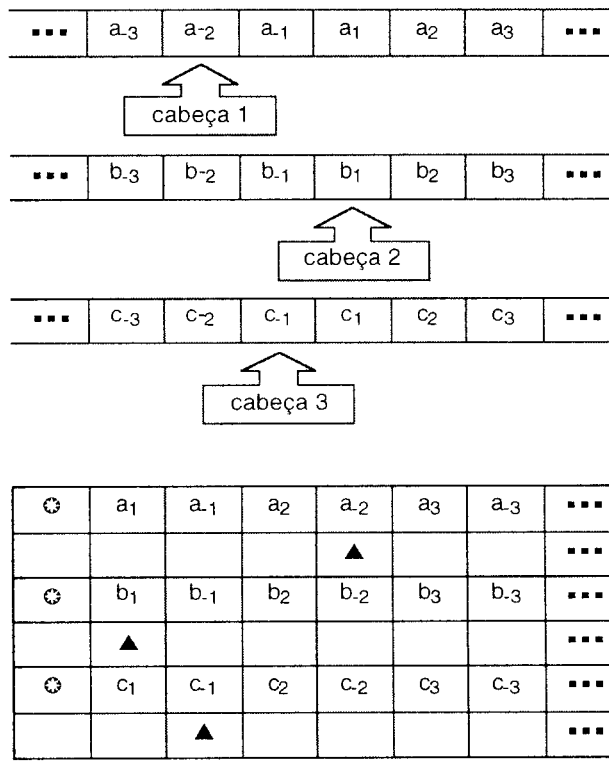


Figura 3.35 Simulação de 3 fitas infinitas dos dois lados

3.6.4 Outras modificações sobre a Máquina de Turing

A seguir são apresentadas, resumidamente, modificações adicionais que podem ser realizadas sobre o modelo básico da Máquina de Turing, as quais não aumentam o seu poder computacional. Ou seja, é possível construir Máquinas de Turing tradicionais que simulam cada uma das modificações sugeridas.

a) *Máquina de Turing Multidimensional.* Neste modelo, a fita tradicional é substituída por uma estrutura do tipo arranjo k-dimensional, infinita em todas as 2k direções;

b) *Máquina de Turing com Múltiplas Cabeças.* A Máquina de Turing com esta modificação possui k cabeças de leitura e gravação sobre uma única fita. Cada cabeça possui movimento independente. Assim, o processamento depende do estado corrente e do símbolo lido em cada uma das cabeças;

c) *Combinações.* Combinações de algumas ou de todas as modificações apresentadas. A combinação de algumas ou de todas as modificações apresentadas não aumenta o poder computacional da Máquina de Turing. Por exemplo, uma Máquina de Turing Não-Determinística com Múltiplas Fitas e Múltiplas Cabeças pode ser simulada por uma Máquina de Turing tradicional.

3.7 Hierarquia de Classes de Máquinas

Todos os modelos formais de máquinas introduzidos para representar algoritmos, bem como suas diversas modificações, são equivalentes à Máquina de Turing, ou seja, são máquinas universais. Lembre-se que, no caso do Autômato com Pilhas, são necessárias pelo menos duas pilhas para ser considerada uma Máquina Universal. Assim, os diversos modelos, bem como suas modificações, podem ser usados indistintamente, dependendo do caso, para facilitar construções ou provas.

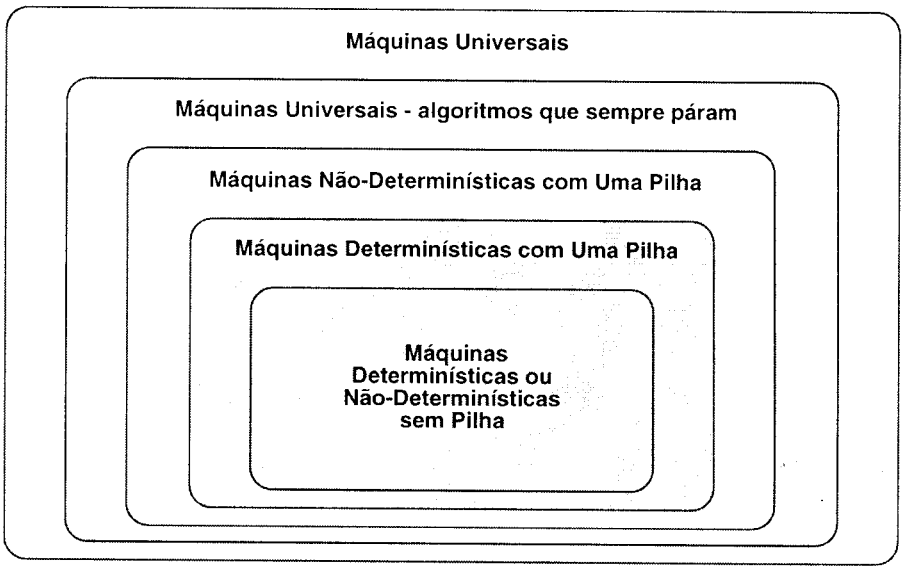


Figura 3.36 Hierarquia de Classes de Máquinas

A Figura 3.36 ilustra a *Hierarquia de Classes de Máquinas* estudadas em relação a seus poderes computacionais. Na hierarquia ilustrada, quanto mais interna for uma classe, menor é o seu poder computacional.

À Hierarquia de Classes de Máquinas corresponde uma *Hierarquia de Classes de Linguagens*, ilustrada na Figura 3.37, a qual é como segue:

a) *Linguagens Regulares*. Correspondem à Classe das Máquinas sem Pilha. São exemplos de linguagens desta classe: alguns editores de texto e protocolos de comunicação. São linguagens muito simples onde, por exemplo, não é possível fazer qualquer tipo de balanceamento de tamanho não-predefinido. A principal característica desta classe é que o tempo de reconhecimento de uma palavra é diretamente proporcional ao comprimento da entrada. Adicionalmente, qualquer algoritmo especificado usando este formalismo é igualmente eficiente, em termos de tempo de processamento;

b) *Linguagens Livres do Contexto Determinísticas*. Correspondem à Classe dos Autômatos com Uma Pilha Determinísticos. São linguagens mais complexas que as Regulares, mas ainda muito simples, onde, por exemplo, não é possível reconhecer a seguinte linguagem (Exemplo 3.27):

   Palavra_Reversa = $\{ww^r \mid w \text{ pertence a } \{a, b\}^*\}$

   O tempo de reconhecimento é diretamente proporcional ao dobro do tamanho da entrada;

c) *Linguagens Livres do Contexto*. Correspondem à Classe dos Autômatos com Uma Pilha Não-Determinísticos. Constituem uma classe de fundamental importância, pois incluem linguagens de programação como Algol e Pascal. Entretanto, algumas linguagens muito simples não pertencem a esta classe de linguagens como (Exemplo 3.17):

   Triplo_Bal = $\{a^n b^n c^n \mid n \geq 0\}$

   Palavra_Palavra = $\{ww \mid w \text{ é palavra sobre os símbolos } a \text{ e } b\}$

   Os melhores algoritmo de reconhecimento conhecidos possuem tempo de processamento proporcional ao tamanho da entrada elevado ao cubo;

d) *Linguagens Recursivas*. Correspondem à classe de todas as linguagens que podem ser reconhecidas mecanicamente (pela Máquina Universal) e para as quais existe um algoritmo de reconhecimento que sempre pára para qualquer entrada. Inclui a grande maioria das linguagens aplicadas. Reconhecedores de linguagens recursivas podem ser muito ineficientes, tanto em termos de tempo de processamento como de recursos de memória;

e) *Linguagens Enumeráveis Recursivamente*. Correspondem à Classe das Máquinas Universais. Portanto, corresponde à classe de todas as linguagens que podem ser reconhecidas mecanicamente.

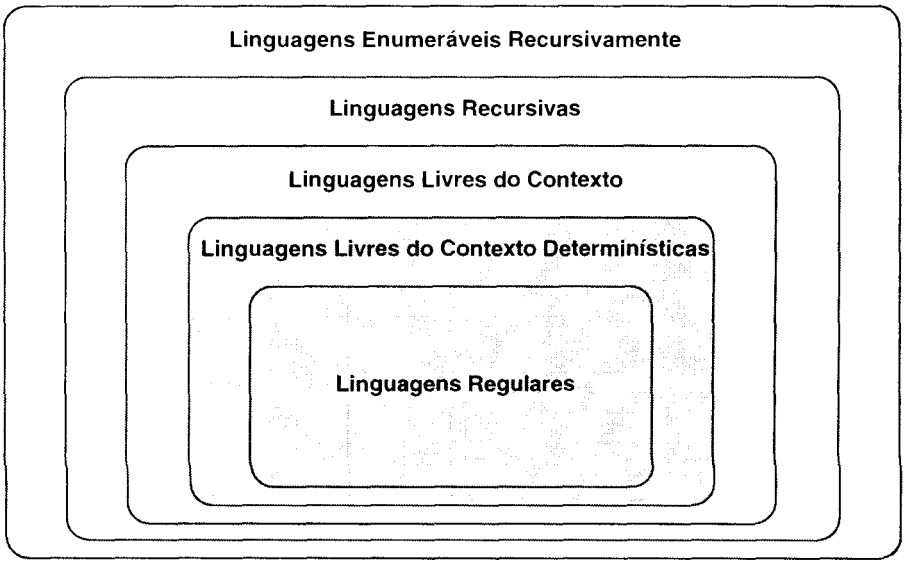


Figura 3.37 Hierarquia de Classes de Linguagens

3.8 Hipótese de Church

Turing propôs um modelo abstrato de computação, conhecido como Máquina de Turing, com o objetivo de explorar os limites da capacidade de expressar soluções de problemas. Trata-se, portanto, de uma proposta de definição formal da noção intuitiva de algoritmo. Diversos outros trabalhos, como *Máquina de Post* e *Funções Recursivas* (Kleene - 1936), bem como trabalhos mais recentes como Máquina Norma e Autômato com Pilhas, resultaram em conceitos equivalentes ao de Turing. As Funções Recursivas serão introduzidas no Capítulo 4. O fato de todos esses trabalhos independentes gerarem o mesmo resultado em termos de capacidade de expressar computabilidade é um forte reforço no que é conhecido como *Hipótese de Church* ou *Hipótese de Turing-Church*:

> *"A capacidade de computação representada pela Máquina de Turing é o limite máximo que pode ser atingido por qualquer dispositivo de computação"*

Em outras palavras, a Hipótese de Church afirma que qualquer outra forma de expressar algoritmos terá, no máximo, a mesma capacidade computacional da Máquina de Turing. Como a noção de algoritmo ou função computável é intuitiva, a Hipótese de Church não é demonstrável.

3.9 Exercícios

Exercício 3.1 Qual a importância do estudo da Máquina de Turing na Ciência da Computação?

Exercício 3.2 Faça um quadro comparativo entre os modelos Máquina de Turing, Máquina de Post e Autômato com Duas Pilhas, destacando suas características e seus principais aspectos funcionais.

Exercício 3.3 Sobre não-determinismo:

a) O que é e quais suas principais características?

b) Qual o seu efeito, em termos de poder computacional, nas máquinas apresentadas?

Exercício 3.4 Sobre a Hipótese de Church:

a) Por que não é demonstrável?

b) Qual seu significado e importância na Teoria da Computação?

Exercício 3.5 Desenvolva Máquinas de Turing, determinísticas ou não, que aceitem as seguintes linguagens

a) $L_1 = \varnothing$

b) $L_2 = \{\varepsilon\}$

c) $L_3 = \{w \mid w$ tem o mesmo número de símbolos a e b$\}$

d) $L_4 = \{w \mid$ o décimo símbolo da direita para a esquerda é a$\}$

e) $L_5 = \{waw \mid w$ é palavra de $\{a, b\}^*\}$

f) $L_6 = \{ww \mid w$ é palavra de $\{a, b\}^*\}$

g) $L_7 = \{ww^r \mid w$ é palavra de $\{a, b\}^*\}$

h) $L_8 = \{www \mid w$ é palavra de $\{a, b\}^*\}$

I) $L_9 = \{w \mid w = a^1 b^2 a^3 b^4 \ldots a^{n-1} b^n$ e n é número natural par$\}$

j) $L_{10} = \{w \mid w = a^n b^n$ ou $w = b^n a^n\}$

k) $L_{11} = \{w \mid w = a^i b^j c^k$, onde ou i = j ou j = k$\}$

Exercício 3.6 Desenvolva Máquinas de Post, determinísticas ou não, que aceitem as linguagens dadas no Exercício 3.5.

Exercício 3.7 Desenvolva Autômatos com Duas Pilhas ou Máquinas com Pilhas, determinísticas ou não, que aceitem as linguagens dadas no Exercício 3.5.

Exercício 3.8 Seja a expressão booleana (EB) definida indutivamente como segue:

i) v e f são EB;
ii) Se p e q são EB, então p e q e p ou q também são EB.

Assuma que o conetivo e tem prioridade sobre o ou:

a) Construa uma Máquina de Turing sobre $\Sigma = \{v, f, e, ou\}$ tal que:
ACEITA(M) = {EB | EB é v}
REJEITA(M) = {EB | EB é f}
LOOP(M) = {w | w não é EB}

b) Construa uma Máquina de Post sobre Σ conforme definido no item a)

c) Construa uma Máquina de Turing não-determinística ou Autômato com Duas Pilhas não-determinístico, sobre Σ, conforme definido no item a)

Exercício 3.9 Na demonstração do Teorema 3.18, é sugerido o desenvolvimento de um diagrama de fluxos da Máquina de Post que realize rotação no conteúdo de X, fazendo com que o último símbolo de X passe a ser o primeiro, ou seja, se o conteúdo da variável X é $a_1 a_2 \ldots a_{n-1} a_n$ passa a ser $a_n a_1 a_2 \ldots a_{n-1}$. Desenvolva uma Máquina de Post que processe esta rotação.

Exercício 3.10 Prove que qualquer Máquina de Turing pode ser simulada por uma Máquina de Turing com somente dois estados.

Sugestão: modifique o alfabeto auxiliar, introduzindo novos símbolos.

Exercício 3.11 Prove que o poder computacional da Classe das Máquinas com n Pilhas ($n > 2$) é equivalente ao da Classe das Máquinas com Duas Pilhas.

Exercício 3.12 Desenvolva um programa em computador que simule qualquer Máquina de Turing. A entrada para o simulador deve ser a função programa e a saída o estado final da máquina simulada, o conteúdo da fita e o número de movimentos da cabeça da fita.

Exercício 3.13 Como é possível ampliar o exercício anterior, introduzindo a facilidade de não-determinismo, usando uma linguagem de programação não-concorrente?

Exercício 3.14 Mostre como as instruções abaixo podem ser construídas como macros em Norma:

a) r_1: se A < 2 então vá_para r_2 senão vá_para r_3

b) r_1: se div(A, B) então vá_para r_2 senão vá_para r_3

Exercício 3.15 Desenvolva os programas, em Norma, que realizam as operações e testes abaixo:

a) A:= B - C

b) A:= B / C

c) Divisão inteira de B por C

d) Fatorial;

e) Potência;

f) Se $A < B$

g) Se $A \leq B$

h) Se $A \leq 0$

Exercício 3.16 Desenvolva os programas, em Norma, que realizam as operações abaixo nos inteiros (utilize a representação sinal-magnitude introduzida no Exemplo 3.11):

a) $A := B + C$

b) $A := B \times C$

c) $A := B - C$

d) $A := B / C$

Exercício 3.17 Desenvolva os programas, em Norma, que realizam as operações abaixo nos números racionais (utilize a representação de racionais introduzida no Exemplo 3.12):

a) $A := B + C$

b) $A := B \times C$

c) $A := B - C$

d) $A := B / C$

Exercício 3.18 Seja a *Máquina NormaNeg*, a qual é, em todos os aspectos, idêntica a Norma, exceto pelo fato de poder armazenar números negativos (inteiros) em seus registradores. Prove que, para cada fluxograma NormaNeg, pode-se definir um fluxograma Norma equivalente sem o uso de registradores extras.

Sugestão: utilize a representação de inteiros baseada na função codificação e desenvolva os programas para as instruções de Norma que simulam as de NormaNeg.

Exercício 3.19 Qual a diferença fundamental entre as Classes das Linguagens Recursivas e das Linguagens Enumeráveis Recursivamente? Qual a importância de se distinguir estas duas classes?

Exercício 3.20 Demonstre que a Classe das Linguagens Recursivas é fechada para as operações de união, intersecção e diferença. Demonstre inicialmente para a operação sobre duas linguagens recursivas. Após, amplie a demonstração para n linguagens recursivas (demonstre por indução em n).

Exercício 3.21 Dê a Máquina de Post MP, sobre o alfabeto {a, b}, tal que:

ACEITA(MP) = {w | w tem o mesmo número de símbolos a e b}
REJEITA(MP)= {w | w cuja diferença entre o número de símbolos a e b é 1}
LOOP(MP)= $\{a, b\}^* - (ACEITA(MP) \cup REJEITA(MP))$

Por exemplo:

$ab \in ACEITA(MP)$
$aba \in REJEITA(MP)$
$aaba \in LOOP(MP)$.

Exercício 3.22 Desenvolva Máquinas de Turing que processem as funções:

a) Subtração, definida por:

m - n, se m > n
zero, caso contrário

b) Fatorial de n, para $n \in \mathbb{N}$

Exercício 3.23 Codifique o fluxograma ilustrado na Figura 3.38 como um número natural, usando a codificação de programas introduzida no Exemplo 3.2.

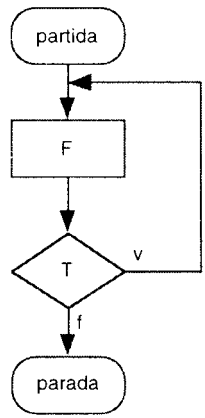


Figura 3.38 Fluxograma

Exercício 3.24 Faça uma comparação entre o poder computacional das classes de máquinas de Turing, de Post e dos Autômatos com Duas Pilhas e, então, responda:

a) Dê suas principais características;

b) Quais classes de máquinas são equivalentes? Não é necessário demonstrar. Apenas justifique sua resposta;

c) Diga se o programa de cada classe de máquina é parcial ou total. Justifique sua resposta.

Exercício 3.25 Elabore um Autômato com Duas Pilhas sobre o alfabeto de entrada {1}. Suponha que as palavras de entrada são números naturais representados em unário, como no Exemplo 3.20, onde, por exemplo, 3 é denotado por 111. O autômato deve aceitar os naturais pares (e rejeitar os ímpares).

Exercício 3.26 Relativamente à Hipótese de Church, justifique:

a) Quais as implicações da Hipótese de Church?

b) Por que ela é chamada de Hipótese de Church ao invés de Teorema de Church?

Exercício 3.27 Considere a Máquina de Turing cuja função programa Π é como na Figura 3.39.

a) Verifique qual o estado final após a computação para as seguintes palavras:
   ab
   aba
   aaba

b) A linguagem aceita é enumerável recursivamente ou recursiva?

Π	⌾	a	b	A	B	ß
q_0	(q_0, ⌾, D)	(q_1, A, D)	(q_0, b, D)	(q_0, A, D)	(q_0, B, D)	(q_4, ß, E)
q_1		(q_1, a, D)	(q_1, b, D)		(q_2, B, E)	(q_2, ß, E)
q_2	(q_5, ⌾, D)	(q_2, a, E)	(q_3, B, E)	(q_2, A, E)	(q_2, B, E)	
q_3	(q_0, ⌾, D)	(q_3, a, E)	(q_3, b, E)	(q_0, A, D)		
q_4	(q_7, ⌾, D)		(q_6, b, D)	(q_4, A, E)	(q_4, B, E)	
q_5		(q_6, a, D)		(q_5, A, D)	(q_5, B, D)	
q_6	(q_6, ⌾, D)	(q_6, a, D)	(q_6, b, D)	(q_6, A, D)	(q_6, B, D)	(q_6, ß, E)
q_7						

Figura 3.39 Tabela de transições da Máquina de Turing

Exercício 3.28 Elabore uma Máquina de Turing MT_Palíndroma (determinística ou não) que sempre pára para qualquer entrada e que reconhece todas as palíndromas (palavras que possuem a mesma leitura da esquerda para a direita e vice-versa) sobre o alfabeto {a, b}. Por exemplo:

aba, abba, babab ∈ ACEITA(MT_Palíndroma)
abab ∈ REJEITA(MT_Palíndroma)

Exercício 3.29 Elabore uma Máquina de Turing M sobre o alfabeto {a, b} tal que:

ACEITA(M) = {w | w inicia com a e, após cada a, existe pelo menos um b}
LOOP(M) = {w | w ∉ ACEITA(MP) e existe pelo menos um b entre dois a}
REJEITA(M)= {a, b}* - (ACEITA(M) ∪ LOOP(M))

Por exemplo:

ab, abbab ∈ ACEITA(MP)
b, aba, abbaba ∈ REJEITA(MP)

Exercício 3.30 Elabore uma Máquina de Post P sobre o alfabeto {a, b} tal que:

ACEITA(P) = $\{a^n b^n \mid m > n \geq 0\}$
REJEITA(P)= {a, b}* - ACEITA(P)

Exercício 3.31 Codifique os programas da Figura 3.40 usando a codificação de programas monolíticos.

Programa Monolítico M_1

```
1:    faça F vá_para 2
2:    se T então vá_para 3 senão vá_para 5
3:    faça G vá_para 4
4:    se T então vá_para 1 senão vá_para 0
5:    faça F vá_para 6
6:    se T então vá_para 7 senão vá_para 2
7:    faça G vá_para 8
8:    se T então vá_para 6 senão vá_para 0
```

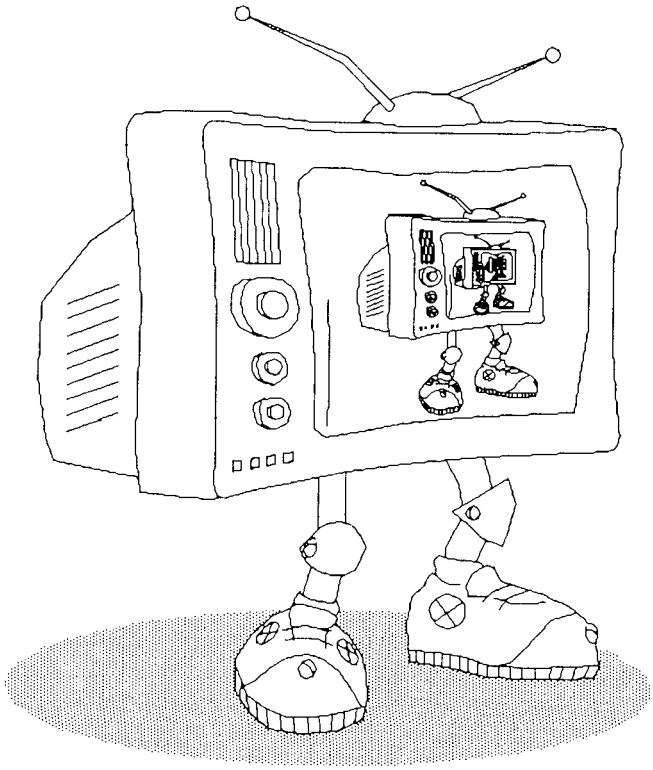
Programa Monolítico M_2

```
1:    faça F vá_para 2
2:    se T então vá_para 3 senão vá_para 1
3:    faça G vá_para 4
4:    se T então vá_para 1 senão vá_para 0
```

Figura 3.40 Programas monolíticos

Exercício 3.32 Como, sem perda de generalidade, pode-se supor que a função programa de uma Máquina de Turing seja total?

Exercício 3.33 Modifique a prova do Teorema 3.16 - Máquina Norma ≤ Máquina de Turing de tal forma que Norma simule as condições ACEITA e REJEITA de parada da Máquina de Turing.



4 Funções Recursivas

Os formalismos usados para especificar algoritmos podem ser classificados nos seguintes tipos:

a) *Operacional*. Define-se uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado. Uma máquina abstrata deve ser suficientemente simples para não permitir dúvidas sobre a sua computação;

b) *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;

c) *Denotacional*. Também é denominado formalismo *Funcional*. Em geral, trata-se de uma função construída a partir de funções elementares de forma composicional (horizontalmente) no sentido em que o algoritmo denotado pela função pode ser determinado em termos de suas funções componentes.

Até o momento, os formalismos estudados como Máquina Norma e Máquina de Turing são do tipo operacional.

O formalismo axiomático mais usual para o estudo de computabilidade é a Gramática. Em termos de poder computacional, a gramática é equivalente às Máquinas Universais. Dependendo de restrições feitas na definição das gramáticas é possível estabelecer uma hierarquia, conhecida como Hierarquia de Chomsky ([MEN98]). De fato, as seguintes equivalências podem ser estabelecidas:

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

Gramáticas e formalismos axiomáticos, em geral, não são objetivo desta publicação mas podem ser encontrados em [MEN98].

Neste capítulo, é estudado um formalismo denotacional (funcional) denominado *Funções Recursivas Parciais*, introduzidas por Kleene (1936), as quais, como o próprio nome indica, são funções parciais definidas recursivamente.

Assim como Turing, Kleene tinha como objetivo formalizar a noção intuitiva de função computável. Quando foi provado que a Classe das Funções Turing-Computáveis era igual à Classe das Funções Recursivas Parciais, a Hipótese de Church cresceu significativamente em termos de credibilidade.

De fato, verifica-se que a composição de três funções naturais simples:

- constante zero;
- sucessor;
- projeção;

juntamente com recursão e minimização, constitui uma forma compacta e natural para definir muitas funções e suficientemente poderosa para descrever toda função intuitivamente computável. Recursão e minimização constituem uma forma especial de compor funções as quais são formalmente introduzidas.

Entretanto, a associação entre os computadores e programas atuais com as funções recursivas introduzidas por Kleene não é tão evidente, o que dificulta uma abordagem mais didática. Assim, optou-se por introduzir, adicionalmente, um outro formalismo baseado no conceito de recursão, inspirado em [BIR76]. Ambas as abordagens são equivalentes. A equivalência é verificada através das máquinas universais. Assim, pode-se afirmar que as duas abordagens fornecem as seguintes visões sobre recursão:

- visão histórica ou clássica, na qual muitos estudos foram e são baseados e que é universalmente aceita e conhecida;
- visão revista, mais adequada aos atuais sistemas computacionais.

4.1 Linguagem Lambda

Inicialmente, é apresentada a notação conhecida como *Linguagem Lambda* (*λ-Linguagem*, introduzida por Alonzo Church em 1941) no *Cálculo Lambda* (*λ-Cálculo*), cujo principal objetivo é evitar ambigüidades de notação.

4.1.1 Funções e Funcionais

Lembre-se que uma *função parcial* é uma relação $f \subseteq A \times B$ onde cada elemento do *domínio* (conjunto A) está relacionado com, no máximo, um elemento do *contradomínio* (conjunto B). O seguinte é usual (suponha uma função $f \subseteq A \times B$):

- $f \subseteq A \times B$ é denotada por $f: A \to B$
- o *tipo* de f é $A \to B$
- $(a, b) \in f$ é denotado por $f(a) = b$

Como motivação, considere as três seguintes funções parciais, definidas usando uma notação matemática muito comum, onde as variáveis x e y têm seus valores em $\mathbb{N}$:

$f(x) = x^3 + 4$
$g(x, y) = (x^2, y - x)$
$h(f, x) = f(f(x))$

A função f possui somente um argumento, uma variável. Portanto, o tipo de f é como segue:

$$f: \mathbb{N} \to \mathbb{N}$$

A função g possui dois argumentos (um par de valores naturais), produzindo outro par de valores naturais, e, portanto, seu tipo é como segue (supondo que o operador × tem precedência sobre →):

$$g: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \quad \text{ou} \quad g: \mathbb{N}^2 \to \mathbb{N}^2$$

Já a função h é um exemplo de função que possui funções como argumentos. Assim, o tipo do argumento de h é composto pelo produto cartesiano do tipo de f com o tipo de x, ou seja:

$$h: (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$$

Definição 4.1 Funcional.

Funcional é uma função que possui uma ou mais funções como argumentos. ❑

Portanto, a função h acima é um funcional. Funcionais ocorrem com freqüência em programação, como exemplificado a seguir.

EXEMPLO 4.1 Programas × Funcionais.

Considere um programa que receba como parâmetro um par constituído por um arranjo A de números naturais e um número natural n e que calcula o máximo entre A(1), A(2),..., A(n). O arranjo A pode ser visto como uma função como segue:

$$A: \mathbb{N} \to \mathbb{N}$$

Assim, a função computada pelo programa pode ser dada pela seguinte função:

$$f(A, n) = \max\{A(i) \mid 1 \leq i \leq n\}$$

cujo tipo é como segue (o mesmo tipo da função h acima):

$$f: (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$$ ❑

Observação 4.2 Funcionais × Funções com mais de um Argumento.

Funcionais podem ser usados para substituir funções com mais de um argumento. Portanto, funções com mais de um argumento podem ser vistas como funcionais com um argumento, o que se constitui num dispositivo útil na simplificação de notação e facilita a manipulação de expressões. ❑

EXEMPLO 4.2 Funcionais × Funções com mais de um Argumento.

Considere a seguinte função:

$$g'(x)(y) = (x^2, y - x)$$

A função g' é um funcional, onde cada natural x define a função g'(x) cujo tipo é:

$$g'(x): \mathbb{N} \to \mathbb{N}^2$$

Ou seja, uma vez fixado um natural x, g' define uma função tal que, para cada natural y, associa-o ao par de naturais $(x^2, y - x)$. Logo, o tipo de g' é como segue:

$$g': \mathbb{N} \to (\mathbb{N} \to \mathbb{N}^2)$$

Embora o tipo de $g': \mathbb{N} \to (\mathbb{N} \to \mathbb{N}^2)$ seja diferente do tipo da função $g: \mathbb{N}^2 \to \mathbb{N}^2$, as duas são "iguais nos seus efeitos". Para verificar que, de fato, os tipos são diferentes, tem-se que:

$g: \mathbb{N}^2 \to \mathbb{N}^2$ significa que $g \subseteq (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$

$g': \mathbb{N} \to (\mathbb{N} \to \mathbb{N}^2)$ significa que $g' \subseteq \mathbb{N} \times (\mathbb{N} \times (\mathbb{N} \times \mathbb{N}))$

Como o produto cartesiano é não-associativo, tem-se que os tipos são diferentes. Entretanto, é fácil verificar que são *isomorfos* (existe uma bijeção), razão pela qual se afirma que, em termos práticos, seus "efeitos são iguais" (a menos de isomorfismo). ❑

EXEMPLO 4.3 Funcionais × Funções com mais de um Argumento.

Considere a seguinte função:

$$h'(f)(x) = f(f(x))$$

O tipo de h' é:

$$h': (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$

Analogamente ao exemplo anterior, embora o tipo de $h': (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ seja diferente do tipo da função $h: (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$, estes são isomorfos. Portanto, pode-se afirmar que as duas funções são "iguais nos seus efeitos". ❑

4.1.2 Motivação e Introdução

Considere novamente a função f: $\mathbb{N} \rightarrow \mathbb{N}$ tal que:

$$f(x) = x^3 + 4$$

É importante reparar que o que efetivamente está sendo definido não é a função f propriamente dita, mas sim f(x), ou seja, o resultado da aplicação de f ao parâmetro x (o qual é suposto assumir valores nos naturais). Claramente $f(x) = x^3 + 4$ não é uma função mas uma equação (pois $x^3 + 4 - f(x) = 0$).

Uma forma de definir funções com mais rigor é usando a *Linguagem Lambda* a qual é introduzida inicialmente de forma informal. Duas construções destacam-se na Linguagem Lambda:

a) *Abstração Lambda*. Permite abstrair a definição da função. Por exemplo, a função f acima é denotada pelo seguinte *termo lambda*:

$$\lambda x.x^3 + 4$$

que pode ser interpretado como segue:

função tal que, para um argumento arbitrário x resulta em $x^3 + 4$

b) *Aplicação Lambda*. Determina o valor da função aplicada a um dado parâmetro. Por exemplo, a função f acima aplicada ao parâmetro 2 é denotada pelo seguinte termo lambda:

$$(\lambda x.x^3 + 4)(2)$$

que pode ser interpretado como segue:

aplicação da função $\lambda x.x^3 + 4$ *ao valor 2*

Em geral, termos (palavras) da Linguagem Lambda são denotados por letras maiúsculas M, N, P,... No caso acima, supondo que:

M denota $x^3 + 4$
N denota $\lambda x.x^3 + 4$
P denota 2

então o termo $(\lambda x.x^3 + 4)(2)$ pode ser denotado como segue:

$(\lambda x.M)(P)$
$(N)(P)$

A semântica de uma aplicação está relacionada com a denominada *Regra de Redução Beta* (*Regra de Redução β*) e é tal que:

$$(\lambda x.M)(P) = [P/x]M$$

o que significa:

substituição de x por P em M

Por exemplo:

$(\lambda x.x^3 + 4)(2) =$
$= [2/x]x^3 + 4 =$
$= 2^3 + 4 =$
$= 12$

4.1.3 Termo e Linguagem Lambda

A seguir, é formalizado o conceito de Linguagem Lambda, a qual é constituída de termos (palavras ou expressões) lambda. As definições que seguem são ditas "não-puras" pois incluem constantes.

Definição 4.3 Termo Lambda.

Sejam X e C conjuntos contáveis de *variáveis* e *constantes*, respectivamente. Então um *Termo Lambda*, *Expressão Lambda* ou *Palavra Lambda* sobre X e C é indutivamente definido como segue:

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

b) Toda constante $c \in C$ é um termo lambda;

c) Se M e N são termos lambda e x é uma variável, então:
   c.1) (M N) é um termo lambda;
   c.2) (λx.M) é um termo lambda.

Um termo da forma (λx.M) é usualmente denominado de *abstração lambda*. ❑

Portanto, um termo lambda não possui qualquer tipo de nomeação. Para nomear um termo, é usado o sinal de igualdade como exemplificado a seguir onde o termo lambda $(\lambda x.x^3)$ é nomeado pelo identificador cubo:

$$\text{cubo} = (\lambda x.x^3)$$

Note-se que, na expressão acima, x é uma variável, e o expoente 3 é uma constante.

Relativamente ao uso de parênteses, as seguintes simplificações de notação são adotadas:

a) *Associatividade à Esquerda*. Parênteses podem ser eliminados, respeitando a associativa à esquerda, ou seja, para os termos M, N e P, tem-se que:

$$M\,N\,P \quad \text{é uma notação simplificada de} \quad ((M\,N)\,P)$$

b) *Escopo de uma Variável*. Parênteses podem ser eliminados, respeitando o escopo de uma variável em uma abstração, ou seja, para os termos M, N e P e para a variável x, tem-se que:

$$\lambda x.M\,N\,P \quad \text{é uma notação simplificada de} \quad (\lambda x.(M\,N\,P))$$

Note-se que os identificadores usados não são importantes. Por exemplo, os seguintes termos lambda denotam a função adição equivalentemente:

$$\lambda(x, y).x + y \quad \text{e} \quad \lambda(r, s).r + s$$

Entretanto, algumas regras simples devem ser observadas, como no caso em que os identificadores devem ser diferentes. Um contra-exemplo é o seguinte, o qual não é considerado um termo lambda bem formado:

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

Definição 4.4 Linguagem Lambda.

Sejam X e C conjuntos contáveis de variáveis e constantes, respectivamente. Uma *Linguagem Lambda* Λ sobre X e C é o conjunto de todos os termos lambda sobre estes conjuntos. ❑

EXEMPLO 4.4 Termos Lambda.

As seguintes funções parciais:

$f(x) = x^3 + 4$ $\quad g(x, y) = (x^2, y - x)$ $\quad h(f, x) = f(f(x))$

$g'(x)(y) = (x^2, y - x)$ $\quad h'(f)(x) = f(f(x))$

são denotadas pelos seguintes termos na Linguagem Lambda, respectivamente:

$f = \lambda x.x^3 + 4$ $\quad g = \lambda(x, y).(x^2, y - x)$ $\quad h = \lambda(f, x).f(f(x))$

$g' = \lambda x.\lambda y.(x^2, y - x)$ $\quad h' = \lambda f.\lambda x.f(f(x))$

❑

Observação 4.5 Linguagens Tipadas e Não-Tipadas.

A Linguagem Lambda inspirou diretamente a linguagem de programação *LISP*. Analogamente à definição original da Linguagem Lambda, LISP é uma linguagem sem tipos. Ou seja, programas e dados são não-distingüíveis. Até hoje são discutidas as vantagens e desvantagens das *linguagens tipadas* e *linguagens não-tipadas*, ou seja se:

- tipos devem constituir uma esquema básico do conhecimento;
- entidades devem ser organizadas em subconjuntos com propriedades uniformes a partir de um universo não-tipado.

De qualquer forma, após a introdução da linguagem Algol ([MAL69]), onde variáveis são associadas a tipos quando de suas declarações, permitindo verificações sintáticas e semânticas de suas instâncias no programa em tempo de compilação, foi então, que tipos passaram a ser considerados como uma facilidade essencial para o desenvolvimento de programas. ❑

A associação de tipo a um termo lambda é como em uma definição tradicional de função. A definição formal de termos lambda tipados é omitida.

EXEMPLO 4.5 Termos Lambda Tipados.

As seguintes funções parciais, onde as variáveis x e y têm seus valores em $\mathbb{N}$:

$f(x) = x^3 + 4$ $\quad g(x, y) = (x^2, y - x)$ $\quad h(f, x) = f(f(x))$

$g'(x)(y) = (x^2, y - x)$ $\quad h'(f)(x) = f(f(x))$

são denotadas pelos seguintes termos tipados na Linguagem Lambda:

$f = \lambda x.x^3 + 4: \mathbb{N} \to \mathbb{N}$

$g = \lambda(x, y).(x^2, y - x): \mathbb{N}^2 \to \mathbb{N}^2$

$g' = \lambda x.\lambda y.(x^2, y - x): \mathbb{N} \to (\mathbb{N} \to \mathbb{N}^2)$

$h = \lambda(f, x).f(f(x)): (\mathbb{N} \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$

$h' = \lambda f.\lambda x.f(f(x)): (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ ❑

4.1.4 Semântica de um Termo Lambda

A semântica de um termo lambda está relacionada com os conceitos de variável livre, substituição de variável livre e regra de redução beta, introduzidos a seguir.

Definição 4.6 Variável Ligada, Variável Livre.

Uma variável em um termo lambda é denominada:

a) *Variável Ligada*, se está dentro do escopo de um abstração lambda;

b) *Variável Livre*, caso contrário. ❑

EXEMPLO 4.6 Variável Ligada, Variável Livre.

a) No termo $\lambda x.x^k$, as variáveis x e k são ditas ligada e livre, respectivamente.

b) No termo $\lambda k.\lambda x.x^k$, as variáveis x e k são ditas ligadas. ❑

Definição 4.7 Substituição de Variável Livre.

Sejam x uma variável e M, P termos lambda. A *Substituição de uma Variável Livre* x por P em M denotada por:

$$[P/x]M$$

é simplesmente a substituição de todas as ocorrências de x em M pelo termo P. ❑

EXEMPLO 4.7 Substituição de Variável Livre.

A substituição da variável livre x por 5 em $\lambda k.x^k$ é como segue:

$$[5/x]\lambda k.x^k = \lambda k.5^k$$ ❑

Definição 4.8 Regra de Redução Beta.

Sejam x uma variável e M, P termos lambda. A *Regra de Redução Beta* (*Regra de Redução β*) de um termo $(\lambda x.M)(P)$ é dada pela substituição de x por P em M, ou seja:

$$[P/x]M$$ ❑

Note-se que x é variável ligada em $\lambda x.M$, mas é livre em M, o que garante a coerência da definição acima.

Definição 4.9 Semântica de um Termo Lambda.

A *Semântica de um Termo Lambda* dado por uma função aplicada a um parâmetro é dada pelas aplicações sucessivas possíveis da regra de redução beta. ❑

EXEMPLO 4.8 Regra de Redução Beta, Semântica de um Termo Lambda.

A semântica do termo $(\lambda k.(\lambda x.x^k)(5))(2)$ é dada pela sucessiva aplicação da regra de redução beta como segue:

$(\lambda k.(\lambda x.x^k)(5))(2) =$	aplicação da regra de redução beta em k
$= [2/k](\lambda x.x^k)(5) =$	substituição da variável livre k
$= (\lambda x.x^2)(5) =$	aplicação da regra de redução beta em x
$= [5/x](x^2) =$	substituição da variável livre x
$= 5^2 =$	
$= 25$	

❑

4.2 Funções Recursivas de Kleene

As funções recursivas parciais propostas por Kleene são funções construídas sobre funções básicas, usando três tipos de construções denominadas de composição, recursão e minimização, as quais são apresentadas individualmente antes da introdução do conceito de função recursiva parcial.

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 (e, portanto, também com Linguagens Enumeráveis Recursivamente);
- Funções Recursivas Totais e Funções Turing-Computáveis Totais (e, portanto, com Linguagens Recursivas).

4.2.1 Composição

A composição definida a seguir generaliza o conceito usual de composição de funções.

Definição 4.10 Composição de Funções.

Sejam $g, f_1, f_2, \ldots, f_k$ funções parciais tais que:

$g = \lambda(y_1, y_2, \ldots, y_k).g(y_1, y_2, \ldots, y_k)$: $\mathbb{N}^k \rightarrow \mathbb{N}$

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

A função parcial h tal que:

$$h = \lambda(x_1, x_2, \ldots, x_n).h(x_1, x_2, \ldots, x_n): \mathbb{N}^n \rightarrow \mathbb{N}$$

é a *Composição de Funções* definida a partir de $g, f_1, f_2, \ldots, f_k$ como segue:

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

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

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

A composição de funções generalizada também é denominada de *Substituição de Funções* pois h é obtida a partir da substituição dos y_i em g por $f_i(x_1,x_2,...x_n)$, para todo $i \in \{1, 2, ..., k\}$.

EXEMPLO 4.9 Composição de Funções - Funções Constantes.

Suponha as seguintes funções:

$zero = \lambda x.0: \mathbb{N} \rightarrow \mathbb{N}$	função constante zero
$sucessor = \lambda x.x + 1: \mathbb{N} \rightarrow \mathbb{N}$	função sucessor
$adição = \lambda(x, y).x + y: \mathbb{N}^2 \rightarrow \mathbb{N}$	função adição

As seguintes funções são definidas, usando composição de funções:

$um = \lambda x.sucessor(zero(x)): \mathbb{N} \rightarrow \mathbb{N}$	função constante um
$dois = \lambda x.sucessor(um(x)): \mathbb{N} \rightarrow \mathbb{N}$	função constante dois
$três = \lambda x.adição(um(x), dois(x)): \mathbb{N} \rightarrow \mathbb{N}$	função constante três

Sugere-se, como exercício, verificar se, usando as mesmas funções zero, sucessor e adição, existem outras formas de definir equivalentemente as funções um, dois e três. ❑

4.2.2 Recursão

O conceito de recursão é de fundamental importância na Ciência da Computação. Atualmente, não só grande parte das linguagens de programação possuem facilidades de recursão como, também, a arquitetura de muitos computadores modernos possui estruturas para tratar eficientemente recursão.

Definição 4.11 Recursão.

Sejam f e g funções parciais tais que:

$f = \lambda(x_1, x_2,..., x_n).f(x_1, x_2,..., x_n): \mathbb{N}^n \rightarrow \mathbb{N}$

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

A função parcial h tal que:

$$h = \lambda(x_1,x_2,...x_n, y).h(x_1,x_2,...x_n, y): \mathbb{N}^{n+1} \rightarrow \mathbb{N}$$

é definida por *Recursão* a partir de f e g como segue:

$h(x_1, x_2,..., x_n, 0) = f(x_1, x_2,..., x_n)$

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

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

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

EXEMPLO 4.10 Recursão - Adição.

Suponha as seguintes funções:

$id = \lambda x.x: \mathbb{N} \rightarrow \mathbb{N}$ função identidade

$sucessor = \lambda x.x + 1: \mathbb{N} \rightarrow \mathbb{N}$ função sucessor

$proj3_3 = \lambda(x, y, z).z: \mathbb{N}^3 \rightarrow \mathbb{N}$ função projeção da 3ª componente da tripla

A função adição nos naturais tal que:

$$adição = \lambda(x, y).x + y: \mathbb{N}^2 \rightarrow \mathbb{N}$$

é definida, usando recursão, como segue:

$adição(x, 0) = id(x)$

$adição(x, y + 1) = proj3_3(x, y, sucessor(adição(x, y)))$

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

$adição(3, 2) =$
$= proj3_3(3, 2, sucessor(adição(3, 1))) =$
$= proj3_3(3, 2, sucessor(proj3_3(3, 1, sucessor(adição(3, 0))))) =$
$= proj3_3(3, 2, sucessor(proj3_3(3, 1, sucessor(id(3))))) =$
$= proj3_3(3, 2, sucessor(proj3_3(3, 1, sucessor(3)))) =$
$= proj3_3(3, 2, sucessor(proj3_3(3, 1, 4))) =$
$= proj3_3(3, 2, sucessor(4)) =$
$= proj3_3(3, 2, 5) =$
$= 5$

Note-se que, nas instâncias de $proj3_3(x, y, sucessor(adição(x, y)))$, somente a componente $sucessor(adição(x, y))$ é importante na lógica apresentada. A função $proj3_3$, bem como as componentes x e y, estão presentes somente para satisfazer a definição de recursão. De fato, funções do tipo "projeção" são fundamentais em recursão, como será visto adiante. ❑

4.2.3 Minimização

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

Definição 4.12 Minimização.

Seja f uma função parcial tal que:

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

A função parcial h tal que:

$$h = \lambda(x_1, x_2, \ldots, x_n).h(x_1, x_2, \ldots, x_n): \mathbb{N}^n \rightarrow \mathbb{N}$$

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

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

❑

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

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

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

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

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

EXEMPLO 4.11 Minimização, Recursão - Constante Zero.

Suponha a função constante $zero = \lambda x.0: \mathbb{N} \rightarrow \mathbb{N}$. Considere a seguinte função que identifica o número zero nos naturais:

$$const_{zero}: \rightarrow \mathbb{N}$$

Note-se que é uma função *sem* variáveis, ou seja, é uma *constante*. Sugere-se como exercício compará-la com a *função constante* zero e diferenciá-la da mesma. A constante $const_{zero}$ é definida usando minimização como segue:

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

De fato, o menor natural y tal que zero(y) = 0 é 0. ❑

EXEMPLO 4.12 Minimização, Recursão - Antecessor.

Suponha a constante zero $const_{zero}$, bem como a seguinte função de projeção:

$proj2_1 = \lambda(x, y).x: \mathbb{N}^2 \rightarrow \mathbb{N}$ função projeção da 1ª componente do par

A seguinte função antecessor nos naturais:

$$antecessor = \lambda x.antecessor(x): \mathbb{N} \rightarrow \mathbb{N}$$

pode ser definida, usando recursão (supondo que antecessor de 0 é 0), como segue:

$antecessor(0) = const_{zero}$

$antecessor(y + 1) = proj2_1(y, antecessor(y))$

Por exemplo, antecessor(2) é como segue:

antecessor(2) =
= $proj2_1$(1, antecessor(1)) =
= $proj2_1$(1, $proj2_1$(0, antecessor(0))) =
= $proj2_1$(1, $proj2_1$(0, $const_{zero}$)) =
= $proj2_1$(1, $proj2_1$(0, 0)) =
= $proj2_1$(1, 0) =
= 1

Repare que foi necessário usar uma função de "projeção". ❑

EXEMPLO 4.13 Minimização, Recursão - Subtração.

Suponha a constante zero $const_{zero}$, bem como as seguintes funções:

id = $\lambda x.x$: $\mathbb{N} \rightarrow \mathbb{N}$ função identidade

$proj3_3 = \lambda(x, y, z).z$: $\mathbb{N}^3 \rightarrow \mathbb{N}$ função projeção da 3ª componente da tripla

A seguinte função subtração nos naturais:

$$sub = \lambda(x, y).sub(x, y): \mathbb{N}^2 \rightarrow \mathbb{N}$$

pode ser definida usando recursão:

sub(x, 0) = id(x)
sub(x, y + 1) = $proj3_3$ antecessor(x, y, sub(x, y))

Por exemplo, sub(3, 2) é como segue:

sub(3, 2) =
= $proj3_3$ antecessor(3, 1, sub(3, 1)) =
= $proj3_3$ antecessor(3, 1, $proj3_3$ antecessor(3, 0, sub(3, 0))) =
= $proj3_3$ antecessor(3, 1, $proj3_3$ antecessor(3, 0, id(3))) =
= $proj3_3$ antecessor(3, 1, $proj3_3$ antecessor(3, 0, 3)) =
= $proj3_3$ antecessor(3, 1, 2) =
= 1

Novamente, foi necessário usar uma função de "projeção". Sugere-se como exercício, determinar o valor de sub(2, 3). ❑

4.2.4 Função Recursiva Parcial e Total

Funções recursivas parciais são definidas a partir de três funções básicas sobre o conjunto dos números naturais, como segue:

- constante zero;
- sucessor;
- projeção.

Mais precisamente, projeção não é uma função, mas uma família de funções, pois depende do número de componentes, bem como de qual componente que se deseja projetar. É interessante observar que, somente com estas três funções,

bem como com as construções de composição, recursão e minimização, é possível definir qualquer função intuitivamente computável.

Definição 4.13 Função Recursiva Parcial.

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

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

$\text{zero} = \lambda x.0: \mathbb{N} \to \mathbb{N}$ função constante zero

$\text{sucessor} = \lambda x.x + 1: \mathbb{N} \to \mathbb{N}$ função sucessor

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

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

$g = \lambda(y_1, y_2,\ldots, y_k).g(y_1, y_2,\ldots, y_k): \mathbb{N}^k \to \mathbb{N}$

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

então a seguinte função é recursiva parcial :

$h = \lambda(x_1, x_2,\ldots, x_n).h(x_1, x_2,\ldots, x_n): \mathbb{N}^n \to \mathbb{N}$

a qual é definida pela *composição de funções* a partir de g, f_1, f_2, ..., f_k como segue:

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

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

$f = \lambda(x_1, x_2,\ldots, x_n).f(x_1, x_2,\ldots, x_n): \mathbb{N}^n \to \mathbb{N}$

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

então a seguinte função é recursiva parcial :

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

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

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

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

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

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

então a seguinte função é recursiva parcial:

$h = \lambda(x_1, x_2,\ldots, x_n).h(x_1, x_2,\ldots, x_n): \mathbb{N}^n \to \mathbb{N}$

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

$h = \lambda(x_1,x_2,\ldots,x_n).\min\{y \mid f(x_1,\ldots,x_n, y) = 0\}: \mathbb{N}^n \to \mathbb{N}$ ❑

EXEMPLO 4.14 Funções Recursivas Parciais.

a) *Função Identidade*. A função identidade definida como segue:

$\text{id} = \lambda x.x: \mathbb{N} \to \mathbb{N}$

é recursiva parcial pois é uma função básica de projeção, ou seja:

$\text{id} = \text{proj}1_1$

b) As seguintes funções exemplificadas anteriormente são recursivas parciais:

adição = $\lambda(x, y).x + y$: $\mathbb{N}^2 \rightarrow \mathbb{N}$	função adição
sub = $\lambda(x, y)$.sub(x, y): $\mathbb{N}^2 \rightarrow \mathbb{N}$	função subtração
um = λx.sucessor(zero(x)): $\mathbb{N} \rightarrow \mathbb{N}$	função constante um
dois = λx.sucessor(um(x)): $\mathbb{N} \rightarrow \mathbb{N}$	função constante dois
três = λx.adição(um(x), dois(x)): $\mathbb{N} \rightarrow \mathbb{N}$	função constante três
$const_{zero}$: $\rightarrow \mathbb{N}$	valor constante zero
antecessor = λx.antecessor(x): $\mathbb{N} \rightarrow \mathbb{N}$	função antecessor

❑

Uma função recursiva total, como o próprio nome indica, nada mais é do que uma função recursiva parcial que é total.

Definição 4.14 Função Recursiva Total.

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

Para um completo entendimento da definição de uma função recursiva total, sugere-se revisar:

- Definição 4.10 - Composição de Funções
- Definição 4.11 - Recursão
- Definição 4.12 - Minimização

verificando as condições para que a função resultante destas definições seja total. Sugere-se, como exercício, verificar quais das funções do Exemplo 4.14 são totais.

O seguinte teorema (a demonstração é omitida) é mais um reforço para a Hipótese de Church.

Teorema 4.15 Funções Recursivas × Funções Turing-Computáveis.

As seguintes classes de funções são equivalentes:

a) Funções Recursivas Parciais e Funções Turing-Computáveis;

b) Funções Recursivas Totais e Funções Turing-Computáveis Totais. ❑

Conseqüentemente, a seguinte relação entre classes também pode ser estabelecida:

- Funções Recursivas Parciais e Linguagens Enumeráveis Recursivamente;
- Funções Recursivas Totais, Funções Turing-Computáveis Totais e Linguagens Recursivas.

4.3 Definições Recursivas de Bird

Nesta seção é apresentado o formalismo de R. Bird para o tratamento do conceito de recursão, através de *definições recursivas*. Inicialmente, são introduzidos alguns exemplos que ilustram o potencial destas definições. De fato, verifica-se que a Classe das Funções com Definição Recursiva é a mesma Classe das Funções Computáveis.

Nos exemplos que seguem, considere expressões da seguinte forma:

$$(p \rightarrow e_1, e_2)$$

Informalmente, o valor de tal expressão é e_1, se p é verdadeiro, e e_2, caso contrário. A definição formal é dada adiante.

EXEMPLO 4.15 Fatorial.

Considere a seguinte definição recursiva da função fatorial:

$$\text{fatorial} = \lambda x.(x = 0 \rightarrow 1, x \bullet \text{fatorial}(x - 1))$$

Por exemplo, fatorial(0) = 1. De fato:

$$\text{fatorial}(0) = (0 = 0 \rightarrow 1, 0 \bullet \text{fatorial}(0 - 1)) = 1$$

Observa-se que 0•fatorial(0 - 1) não tem valor definido, pois fatorial(0 - 1) é *indefinido*. Mesmo assim, como x = 0 é verdadeira, a função tem valor definido e igual a 1 (conforme o valor de uma expressão da forma $(p \rightarrow e_1, e_2)$ introduzido acima). ❑

EXEMPLO 4.16 Adição, Multiplicação e Potenciação.

Considere as seguintes definições recursivas das funções adição (adição), multiplicação (mult) e potenciação (pot):

$$\text{adição} = \lambda(x, y).(x = 0 \rightarrow y, \text{adição}(x - 1, y) + 1)$$
$$\text{mult} = \lambda(x, y).(x = 0 \rightarrow 0, \text{adição}(\text{mult}(x - 1, y), y))$$
$$\text{pot} = \lambda(x, y).(y = 0 \rightarrow 1, \text{mult}(\text{pot}(x, y - 1), x))$$

Observa-se que:

- a adição é definida em termos da função sucessor (sucessor = λy.y + 1)
- a multiplicação é definida em termos de adições de y
- a potenciação é definida em termos de multiplicações de x ❑

EXEMPLO 4.17 Divisão, Multiplicação.

Considere a definição da função multiplicação do exemplo anterior, juntamente com as seguintes definições recursivas das funções divisão inteira (div) de x por y (indefinida para y = 0) e multiplicação (m) desta por y:

$$\text{div} = \lambda(x, y).(x < y \rightarrow 0, \text{div}(x - y, y) + 1)$$
$$m = \lambda(x, y).\text{mult}(y, \text{div}(x, y))$$

Entretanto, para y = 0, m(x, 0) admite as seguintes interpretações:

- m(x, 0) = mult(0, div(x, 0)) = 0 portanto é definido;
- m(x, 0) = mult(0, div(x, 0)) é indefinido pois:
  div(x, 0) é indefinido;
  portanto o argumento (0, div(x, 0)) é indefinido;
  logo mult(0, div(x, 0)) é indefinido. ❑

Problemas de dupla interpretação (definido/indefinido) como o do exemplo acima são solucionados, usando regras de avaliação de funções recursivas, as quais são introduzidas em 4.3.2 - Semântica de uma Função Definida Recursivamente.

EXEMPLO 4.18 Minimização.

Seja a função h definida por minimização de $f = \lambda(x, y).f(x, y)$ como na Definição 4.12:

$$h = \lambda x.\min\{y \mid f(x, y) = 0 \text{ e, } \forall z \text{ tal que } z < y, f(x, z) \text{ é definida}\}$$

Para definir h considere a seguinte função k:

$$k = \lambda(x, z).(f(x, z) = 0 \rightarrow z, k(x, z + 1))$$

Então, h pode ser definida como segue:

$$h = \lambda x.k(x, 0)$$

Note que, de fato, h para o valor x é definida como o menor natural y tal que $f(x, y) = 0$ e garante que é possível determinar, em um tempo finito, se para qualquer valor z menor do que y, f(x, z) é diferente de zero. Sugere-se como exercício o caso geral para $h = \lambda(x_1, x_2, \ldots, x_n).h(x_1, x_2, \ldots, x_n)$, supondo $f = \lambda(x_1, x_2, \ldots, x_n, y).f(x_1, x_2, \ldots, x_n, y)$. ❑

4.3.1 Classe das Funções Definidas Recursivamente

A seguir, é apresentada formalmente a Classe das Funções Definidas Recursivamente. Inicialmente, são introduzidos os tipos necessários e, posteriormente, as definições de expressões predicativas e expressões numéricas, necessárias para o conceito de *definição recursiva*.

Definição 4.16 Tipo de Definição Recursiva.

Um *Tipo* de uma definição recursiva pode ser como segue (suponha $n \geq 1$):

$\mathbb{N}$
$\mathbb{B}$
$\mathbb{N}^n \rightarrow \mathbb{N}$
$\mathbb{N}^n \rightarrow \mathbb{B}$

Relativamente às *constantes* e *variáveis* dos tipos, assume-se que:

a) *Tipo* $\mathbb{N}$. As constantes são os números:

$$0, 1, 2, \ldots$$

As variáveis são denotadas pelos identificadores:

$$x, y, z, \ldots, x_1, y_1, z_1, \ldots$$

b) *Tipo* $\mathbb{B}$. As únicas as constantes são denotadas pelos identificadores:

$$\text{verdadeiro} \quad \text{e} \quad \text{falso}$$

os quais representam os valores-verdade *verdadeiro* e *falso*, respectivamente;

c) *Tipo* $\mathbb{N} \rightarrow \mathbb{B}$. A única constante é denotada por zero e representa a função constante a qual verifica se o argumento é nulo, ou seja:

$$\text{zero} = \lambda x.(x = 0): \mathbb{N} \rightarrow \mathbb{B}$$

d) *Tipo* $\mathbb{N} \rightarrow \mathbb{N}$. As únicas constantes são denotadas pelos identificadores sucessor e antecessor, os quais denotam as funções *sucessor* e *antecessor*, respectivamente, ou seja:

$$\text{sucessor} = \lambda x.(x + 1): \mathbb{N} \rightarrow \mathbb{N}$$
$$\text{antecessor} = \lambda x.(x = 0 \rightarrow 0, x - 1): \mathbb{N} \rightarrow \mathbb{N}$$

e) *Tipo* $\mathbb{N}^n \rightarrow \mathbb{N}$ ou $\mathbb{N}^n \rightarrow \mathbb{B}$. As variáveis são denotadas pelos identificadores:

$$f, g, h, \ldots, f_1, g_1, h_1, \ldots$$ ❑

Dois tipos de expressões são consideradas: predicativas e numéricas. Note que as definições destas expressões apresentadas a seguir referenciam-se mutuamente. Neste contexto, uma expressão predicativa ou numérica é dita *sobre* um tipo $\mathbb{X} \in \{\mathbb{N}, \mathbb{B}\}$, quando:

- é do tipo $\mathbb{X}$ ou
- seu contradomínio é do tipo $\mathbb{X}$, ou seja, é do tipo $\mathbb{N}^n \rightarrow \mathbb{X}$

Definição 4.17 Expressão Numérica.

Uma *Expressão Numérica* é sobre $\mathbb{N}$ e é indutivamente definida como segue:

a) *Expressões Numéricas Básicas*. Cada constante ou variável do tipo $\mathbb{N}$ constitui uma expressão numérica;

b) *Composição*. Se $e_1, e_2, \ldots, e_n$ são expressões numéricas e f é uma variável do tipo $\mathbb{N}^n \rightarrow \mathbb{N}$, então é expressão numérica:

$$f(e_1, e_2, \ldots, e_n)$$

c) *Composição Numérica Condicional*. Se p é uma expressão predicativa e e_1, e_2 são expressões numéricas, então a seguinte composição condicional é expressão numérica:

$$(p \rightarrow e_1, e_2)$$ ❑

Definição 4.18 Expressão Predicativa.

Uma *Expressão Predicativa* é sobre $\mathbb{B}$ e é indutivamente definida como segue:

a) *Expressões Predicativas Básicas.* As constantes verdadeiro e falso são, por si mesmas, expressões predicativas;

b) *Composição com Expressão Numérica.* Se e é uma expressão numérica, então é uma expressão predicativa:

$$\text{zero(e)}$$

c) *Composição Predicativa Condicional.* Se p_1, p_2 e p_3 são expressões predicativas, então é uma expressão predicativa:

$$(p_1 \rightarrow p_2, p_3)$$ ❑

Por simplicidade, uma expressão predicativa ou numérica é denominada simplesmente de *expressão.*

Definição 4.19 Definição Recursiva.

Sejam $e_1, e_2, \ldots, e_k$ expressões tais que:

a) Para cada $s \in \{1, 2, \ldots, k\}$, e_s é sobre o tipo $\mathbb{X}_s$, onde $\mathbb{X}_s$ é $\mathbb{N}$ ou $\mathbb{B}$;

b) As variáveis que aparecem em e_s são elementos do conjunto:

$$\{x_{s1}, x_{s2}, \ldots, x_{sns}, f_1, f_2, \ldots, f_k\}$$

onde:

x_{sr} é do tipo $\mathbb{N}$, para cada $r \in \{1, 2, \ldots, n_s\}$
f_s é do tipo $\mathbb{N}^{ns} \rightarrow \mathbb{X}_s$, para cada $s \in \{1, 2, \ldots, k\}$

Uma *Definição Recursiva* de $f_1, f_2, \ldots, f_k$ é dada por:

$f_1 = \lambda(x_{11}, x_{12}, \ldots, x_{1n1}).f_1(x_{11}, x_{12}, \ldots, x_{1n1}) = e_1$
$f_2 = \lambda(x_{21}, x_{22}, \ldots, x_{2n2}).f_2(x_{21}, x_{22}, \ldots, x_{2n2}) = e_2$
...
$f_k = \lambda(x_{k1}, x_{k2}, \ldots, x_{knk}).f_k(x_{k1}, x_{k2}, \ldots, x_{knk}) = e_k$ ❑

Portanto, uma definição recursiva pode definir mais de uma função simultaneamente. É interessante reparar que, de certa forma, as definições de expressão numérica e expressão predicativa acima são definidas simultaneamente (referenciam-se mutuamente).

EXEMPLO 4.19 Definição Recursiva.

A definição:

$$f = \lambda x.(\text{zero}(x) \rightarrow \text{sucessor}(x), \text{sucessor}(f(\text{antecessor}(x))))$$

é a definição recursiva (formal) de:

$$f = \lambda x.(x = 0 \rightarrow x + 1, f(x - 1) + 1)$$

Sugere-se como exercício revisar as funções introduzidas nos Exemplos 4.15 a 4.18, garantindo que são definições recursivas (formais). ❑

4.3.2 Semântica de uma Função Definida Recursivamente

No Exemplo 4.17, foi introduzida uma operação de multiplicação a qual admitia duas interpretações, uma definida e outra indefinida. De fato, para que se seja capaz de determinar exatamente como funções são descritas por definições recursivas, é necessário especificar uma regra de avaliação (semântica) para associar valores de funções com os argumentos dados. Entre muitas possíveis regras, duas são descritas, a saber:

- *regra-valor*, onde os valores dos argumentos são avaliados *antes* de serem requeridos;
- *regra-nome*, onde os valores são avaliados *se e quando* são requeridos.

Portanto, em uma regra-valor, os argumentos são avaliados antes da avaliação da função que o referencia, enquanto que, em uma regra-nome, a avaliação dos argumentos é postergada até o último momento possível. Em algumas linguagens de programação como Algol, as avaliações regra-valor e regra-nome possuem paralelo na passagem de argumentos "por valor" (*by value*) ou "por nome" (*by name*), respectivamente.

EXEMPLO 4.20 Regra-Valor × Regra-Nome.

Considere a seguinte função:

$$f = \lambda x.(x = 0 \rightarrow 0, f(x - 1, f(x, y)))$$

Suponha que é desejado avaliar f(1,0). Assim, tem-se que:

$$f(1, 0) = (1 = 0 \rightarrow 0, f(1 - 1, f(1, 0)))$$

Como 1 = 0 é falso, a avaliação continua em f(1 - 1, f(1, 0)). Logo:

a) *Regra-Valor*. Inicialmente, os argumentos 1 - 1 e f(1, 0) são avaliados, para então avaliar f(1 - 1, f(1, 0)). Entretanto, a avaliação do argumento f(1, 0) implica uma recursão sem fim, o que significa que f(1, 0) é *indefinido*;

b) *Regra-Nome*. Os argumentos 1 - 1 e f(1, 0) não são avaliados, mas sim substituídos, resultando em:

$$(1 - 1 = 0 \rightarrow 0, f((1 - 1) - 1, f((1 - 1), f(1,0))))$$

Somente, então, é feita a avaliação de 1 - 1 = 0 resultando em verdadeiro, determinando que a avaliação resulta no valor 0. Portanto, pela regra-nome, f(1,0) *é definido* (e resulta em 0). ❑

O exemplo acima ilustra um fato importante sobre as regras de avaliação: o processo de avaliação é efetuado pela manipulação de expressões e não necessariamente pelas quantidades abstratas que as expressões representam.

Suponha que, para um tipo $\mathbb{X}$, C($\mathbb{X}$) denota o *conjunto de todas as constantes* de $\mathbb{X}$. Ambas as regras permitem avaliar uma expressão e dada por f = $\lambda(x_1$,

$x_2,\ldots, x_n).f(x_1, x_2,\ldots, x_n)$ para um argumento $(a_1, a_2,\ldots,a_n) \in C(\mathbb{X}_1)\times C(\mathbb{X}_2)\times\ldots\times C(\mathbb{X}_n)$. Para tal, é suposto que cada constante de função c: $\mathbb{X}_1 \to \mathbb{X}_2$ é interpretada como uma função do tipo $C(\mathbb{X}_1)\times C(\mathbb{X}_2)$.

Antes de definir formalmente as regras de avaliação regra-valor e regra-nome, é necessário introduzir o conceito de expressão livre.

Definição 4.20 Expressão Livre.

Uma expressão numérica ou predicativa e é dita uma *Expressão Livre* se contém somente constantes, excetuando-se, eventualmente, variáveis do tipo $\mathbb{N}^n \to \mathbb{N}$ ou $\mathbb{N}^n \to \mathbb{B}$. ⌟

Portanto, uma expressão livre não contém variáveis, excetuando-se, eventualmente, variáveis identificadas por f, g, h,..., f_1, g_1, h_1,... . É importante não confundir os conceitos expressão livre e variável livre (Linguagem Lambda). Assim, sugere-se como exercício comparar e diferenciar as duas definições.

Definição 4.21 Avaliação Regra-Valor.

Suponha uma definição recursiva de $f_1, f_2,\ldots,f_k$ dada por:

$f_1 = \lambda(x_{11}, x_{12},\ldots, x_{1n1}).f_1(x_{11}, x_{12},\ldots, x_{1n1}) = e_1$
$f_2 = \lambda(x_{21}, x_{22},\ldots, x_{2n2}).f_2(x_{21}, x_{22},\ldots, x_{2n2}) = e_2$
...
$f_k = \lambda(x_{k1}, x_{k2},\ldots, x_{knk}).f_k(x_{k1}, x_{k2},\ldots, x_{knk}) = e_k$

Uma *Avaliação Regra-Valor* é definida como segue:

Caso 1. Para cada $s \in \{1, 2,\ldots, k\}$, a avaliação de f_s para o argumento $(a_{s1}, a_{s2},\ldots,a_{sns})$, onde $a_{s1}, a_{s2},\ldots,a_{sns}$ são expressões livres, para todo $r \in \{1, 2,\ldots, n_s\}$, consiste na substituição de todas as ocorrências de x_{sr} por a_{sr} na expressão e_s. Após, é realizada a avaliação de cada expressão livre de acordo com os demais casos;

Caso 2. Se a expressão livre e é da forma $(p \to e_1, e_2)$ onde e_1, e_2 são expressões, então é realizada a avaliação p. Se o resultado é verdadeiro, então a avaliação de e é dada pela avaliação de e_1; caso contrário, e é dada pela avaliação da avaliação de e_2. Se o processo de avaliação de p não termina, então o valor de e é *indefinido*;

Caso 3. Se a expressão livre e é da forma $c(e_1, e_2,\ldots, e_k)$, onde c é uma constante de função, então é realizada a avaliação de $e_1, e_2,\ldots, e_k$ na seqüência. Se o processo termina, resultando nas expressões constantes $b_1, b_2,\ldots, b_k$, então o valor de e é dado por $c(b_1, b_2,\ldots, b_k)$; caso contrário, o valor de e é *indefinido*;

Caso 4. Se a expressão livre e é da forma $f(e_1, e_2,\ldots, e_k)$, então é realizada a avaliação de $e_1, e_2,\ldots, e_k$ na seqüência. Se o processo termina, resultando nas expressões constantes $b_1, b_2,\ldots, b_k$, então o valor de e é dado por $f(b_1, b_2,\ldots, b_k)$ (*caso 1*); caso contrário, o valor de e é *indefinido*;

Caso 5. Se a expressão livre e não corresponde a nenhuma das formas acima, então o valor de e é a própria expressão e. ❑

Definição 4.22 Avaliação Regra-Nome.

Uma *Avaliação Regra-Nome* possui os mesmos casos da regra-valor da Definição 4.21, excetuando-se o *caso 4*, dado a seguir:

Caso 4. Se a expressão livre e é da forma $f(e_1, e_2, \ldots, e_k)$, então o valor de e é dado por f, para o argumento $(e_1, e_2, \ldots, e_k)$, como no *caso 1.* ❑

EXEMPLO 4.21 Regra-Valor, Regra-Nome - Multiplicação.

Considere, novamente, a seguinte função de multiplicação:

$m = \lambda(x, y).mult(y, div(x, y))$

onde:

$mult = \lambda(x, y).(x = 0 \rightarrow 0, adição(mult(x - 1, y), y))$
$div = \lambda(x, y).(x < y \rightarrow 0, div(x - y, y) + 1)$

Para o argumento (1, 0), a avaliação de m para cada regra é como segue (por simplicidade, a definição formal da função x<y é omitida, bem como os detalhes de sua avaliação):

Regra-Valor.

caso 1: a substituição resulta em mult(0, div(1, 0))
caso 4: a avaliação de mult(0, div(1, 0)) necessita da avaliação de 0 e div(1, 0)
caso 5: a avaliação de 0 é a própria expressão 0
caso 1: a avaliação de div(1, 0) determinada pela substituição resulta em $(1 < 0 \rightarrow 0, div(1 - 0, 0) + 1)$
caso 2: a avaliação de $1 < 0$ é falso e determina a avaliação de div(1 - 0, 0)
caso 4: a avaliação de div(1 - 0, 0) necessita da avaliação de 1 - 0 e 0

Na seqüência, as avaliações de 1 - 0, 0 resultam em 1, 0, respectivamente, e determinam a avaliação de div(1, 0). Portanto, os casos 1-2-4 (detalhados acima) são repetidos indefinidamente.

Logo, pela regra-valor, m(1, 0) é *indefinido.*

Regra-Nome.

$m(1, 0) \Rightarrow$ caso 1: substituição
$\Rightarrow mult(0, div(1, 0)) \Rightarrow$ caso 4: substituição
$\Rightarrow (0 = 0 \rightarrow 0, adição(mult(0 - 1, div(1, 0)), div(1, 0))) \Rightarrow$ caso 2
$\Rightarrow 0$ caso 5: avaliação de 0 é 0
$\Rightarrow 0$

Logo, pela regra-nome, m(1, 0) *é definido* e resulta em 0. ❑

EXEMPLO 4.22 Regra-Valor, Regra-Nome.

Considere a definição recursiva de f_1, f_2, f_3 dada por:

$f_1 = \lambda x.f_2(f_3, x)$: $\mathbb{N} \rightarrow \mathbb{N}$

$f_2 = \lambda(g, x).g(x, g(x,0))$: $(\mathbb{N}^2 \to \mathbb{N}) \times \mathbb{N} \to \mathbb{N}$
$f_3 = \lambda(x, y).(x = 0 \to y, f_1(\text{antecessor}(x)))$: $\mathbb{N}^2 \to \mathbb{N}$

As avaliações de $f_1(1)$ por ambas as regras produzem zero como resultado, como pode ser verificado na Figura 4.1 e na Figura 4.2. ❑

Regra-Valor - $f_1(1)$

$f_1(1) \Rightarrow$	caso 1
$\Rightarrow f_2(f_3, 1) \Rightarrow$	caso 4
$\Rightarrow f_3(1, f_3(1, 0)) \Rightarrow$	caso 1
$\Rightarrow f_3(1, (1 = 0 \to 0, f_1(\text{antecessor}(1)))) \Rightarrow$	casos 2 e 3
$\Rightarrow f_3(1, f_1(0)) \Rightarrow$	caso 1
$\Rightarrow f_3(1, f_2(f_3, 0)) \Rightarrow$	casos 5 e 1
$\Rightarrow f_3(1, f_3(0, f_3(0, 0))) \Rightarrow$	caso 1
$\Rightarrow f_3(1, f_3(0, (0 = 0 \to 0, f_1(\text{antecessor}(0))))) \Rightarrow$	casos 2 e 5
$\Rightarrow f_3(1, f_3(0, 0)) \Rightarrow$	casos 1, 2 e 5 (como acima)
$\Rightarrow f_3(1, 0) \Rightarrow$	casos detalhados acima
$\Rightarrow 0$	

Figura 4.1 Avaliação por regra-valor

Regra-Nome - $f_1(1)$

$f_1(1) \Rightarrow$	caso 1
$\Rightarrow f_2(f_3, 1) \Rightarrow$	caso 4
$\Rightarrow f_3(1, f_3(1, 0)) \Rightarrow$	caso 4
$\Rightarrow (1 = 0 \to f_3(1, 0), f_1(\text{antecessor}(1))) \Rightarrow$	caso 2
$\Rightarrow f_1(\text{antecessor}(1)) \Rightarrow$	caso 4
$\Rightarrow f_2(f_3, \text{antecessor}(1)) \Rightarrow$	caso 4
$\Rightarrow f_3(\text{antecessor}(1), f_3(\text{antecessor}(1), 0)) \Rightarrow$	caso 4
$\Rightarrow (\text{antecessor}(1) = 0 \to$ $f_3(\text{antecessor}(1), 0), f_1(\text{antecessor}(\text{antecessor}(1)))) \Rightarrow$	caso 3
$\Rightarrow (0 = 0 \to f_3(\text{antecessor}(1), 0), f_1(\text{antecessor}(\text{antecessor}(1)))) \Rightarrow$	caso 2
$\Rightarrow f_3(\text{antecessor}(1), 0) \Rightarrow$	caso 4
$\Rightarrow (\text{antecessor}(1) = 0 \to 0, f_1(\text{antecessor}(\text{antecessor}(1)))) \Rightarrow$	caso 3
$\Rightarrow (0 = 0 \to 0, f_1(\text{antecessor}(\text{antecessor}(1)))) \Rightarrow$	caso 2
$\Rightarrow 0 \Rightarrow$	caso 5
$\Rightarrow 0$	

Figura 4.2 Avaliação por regra-nome

A diferença essencial entre regra-valor e regra-nome pode ser resumida como segue:

a) *Regra-Nome.* A avaliação das subexpressões é retardada até que os seus valores sejam realmente requeridos para continuar a avaliação. Deste modo,

subexpressões que não possuem valores definidos podem jamais serem avaliadas;

b) *Regra-Valor*. Exige a avaliação de todas subexpressões, podendo não terminar ou falhar em terminar, devido a uma dessas subexpressões falhar em ter um valor definido.

Assim, as duas regras nunca produzem resultados conflitantes. Entretanto, a regra-valor pode retornar um valor quando a regra-nome não retorna. Portanto:

- uma função definida recursivamente, juntamente com a regra-valor ou regra-nome, induz uma função, sem ambigüidades;
- entretanto, como exemplificado, para uma mesma definição recursiva, as funções induzidas por regra-valor e regra-nome não necessariamente coincidem.

Definição 4.23 Função Regra-Valor, Função Regra-Nome.

Uma *Função Regra-Valor* ou uma *Função Regra-Nome* é a função induzida por uma definição recursiva, considerando a avaliação regra-valor ou a avaliação regra-nome, respectivamente. ❑

Existe uma forma de construir definições recursivas de tal maneira que as correspondentes funções regra-nome e regra-valor coincidem. Tal resultado é usado para o teorema a seguir, cuja demonstração é omitida (ver [BIR76]).

Teorema 4.24 Funções Definidas Recursivamente ↔ Funções Computáveis.

As Classes das Funções Regra-Valor e Regra-Nome são equivalentes à Classe das Funções Turing-Computáveis. ❑

4.3.3 Tradução de Programas em Definições Recursivas

No que segue, é introduzido, de maneira informal, como um programa monolítico para uma máquina pode ser traduzido em definições recursivas de tal forma que as funções induzidas regra-valor e regra-nome coincidem com a função computada pelo programa na máquina.

Seja $P = (I, r_1)$ um programa monolítico onde $L = \{r_1, r_2, \ldots, r_n\}$ é o correspondente conjunto de rótulos. Suponha, sem perda de generalidade, que r_n é o único rótulo final. Seja $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina tal que P é um programa para M. Suponha a seguinte definição recursiva:

$f_Y = \lambda x.f_Y(x) = \pi_Y(f_1(\pi_X(x)))$
$f_1 = \lambda v.f_1(v) = e_1$
$f_2 = \lambda v.f_2(v) = e_2$
...
$f_n = \lambda v.f_n(v) = e_n$

onde:

v é uma variável do tipo V (conjunto de valores de memória de M)

$e_n = v$

e, para cada instrução rotulada por r_k em P, para $k \in \{1, 2, \ldots, n-1\}$, tem-se que (suponha que F é um identificador de operação, T é um identificador de teste e r_s, r_t são rótulos de L):

a) *Operação.*

a.1) Se r_k é o rótulo de uma operação em P da forma:

r_k: faça F vá_para r_s

então $e_k = f_{r_s}(\pi_F(v))$

a.2) Se r_k é o rótulo de uma operação da forma:

r_k: faça ✔ vá_para r_s

então $e_k = f_{r_s}(v)$

b) *Teste.* Se r_k é o rótulo de um teste da forma:

r_k: se T então vá_para r_s senão vá_para r_t

então $e_k = (\pi_T(v) \rightarrow f_{r_s}(v), f_{r_t}(v))$

Então a função computada $\langle P, M\rangle$ é tal que, para qualquer $x \in X$:

$$\langle P, M\rangle(x) = f_Y(x)$$

De fato, quando a regra-valor é usada, a avaliação passo-a-passo de $f_Y(x)$ é exatamente a mesma seqüência da computação de P em M para a entrada x. Quando a regra-nome é usada, a avaliação de $f_Y(x)$ procede em uma ordem diferente, mas o resultado final é o mesmo.

EXEMPLO 4.23 Programa Monolítico→ Definição Recursiva.

Considere o programa monolítico P_1 na Figura 4.3 para a Máquina Norma. A correspondente definição recursiva é como segue:

$f_Y = \lambda x.f_Y(x) = sai(f_1(ent(x)))$

$f_1 = \lambda(x, y).f_1(x, y) = (x = 0 \rightarrow f_4(x, y), f_2(x, y))$

$f_2 = \lambda(x, y).f_2(x, y) = f_3(x - 1, y)$

$f_3 = \lambda(x, y).f_3(x, y) = f_1(x, y + 1)$

$f_4 = \lambda(x, y).f_4(x, y) = (x, y)$

Programa Monolítico P_1

```
1:    se X = 0 então vá_para 4 senão vá_para 2
2:    faça X := X - 1 vá_para 3
3:    faça Y:= Y+1 vá_para 1
```

Figura 4.3 Programa monolítico para a Máquina Norma

No caso específico da Máquina Norma (com somente dois registradores X e Y), as seguintes simplificações podem ser realizadas:

- substituir ent(x) por (x, 0) na definição de f_Y, obtendo:

$$f_Y = \lambda x.f_Y(x) = sai(f_1(x, 0))$$

- como a função sai é tal que sai = $\lambda(x, y).y$, pode-se remover a função de saída sai da definição de f_Y, desde que a definição de f_4 seja alterada como segue (pois somente o valor do registrador Y deve ser considerado para a saída):

$$f_4 = \lambda(x, y).f_4(x, y) = y$$

Tais simplificações resultam na seguinte definição recursiva:

$f_Y = \lambda x.f_Y(x) = f_1(x, 0)$
$f_1 = \lambda(x, y).f_1(x, y) = (x = 0 \rightarrow f_4(x, y), f_2(x, y))$
$f_2 = \lambda(x, y).f_2(x, y) = f_3(x - 1, y)$
$f_3 = \lambda(x, y).f_3(x, y) = f_1(x, y + 1)$
$f_4 = \lambda(x, y).f_4(x, y) = y$

Para verificar que as funções induzidas regra-valor e regra-nome coincidem, observe que cada argumento na definição recursiva é constituído de funções constantes aplicadas a variáveis do tipo $\mathbb{N}$. Visto que Norma, como todas as demais máquinas introduzidas, define funções constantes totais e que cada argumento de função sempre possui um valor definido, esse valor é sempre avaliado, ou antes da chamada da função (regra-valor), ou em algum estágio mais tarde (regra-nome). Portanto, ambas as regras induzem a mesma função.❑

EXEMPLO 4.24 Programa Monolítico→ Definição Recursiva.

Suponha que, na Máquina Norma, a operação X:= X - 1 seja indefinida quando o conteúdo do registrador X é zero. Considere o programa monolítico P_2 na Figura 4.4 para tal Máquina Norma. Neste caso, a correspondente definição recursiva é como segue:

$f_Y = \lambda x.f_Y(x) = f_1(x, 0)$
$f_1 = \lambda(x, y).f_1(x, y) = f_2(x - 1, y)$
$f_2 = \lambda(x, y).f_2(x, y) = f_3(x, y + 1)$
$f_3 = \lambda(x, y).f_3(x, y) = y$

Programa Monolítico P_2

```
1:    se X:= X - 1 vá_para 2
2:    faça Y:= Y + 1 vá_para 3
```

Figura 4.4 Programa monolítico para a Máquina Norma modificada

A avaliação de $f_Y(0)$ pela regra-valor resulta em *indefinido*, visto que foi suposto 0 - 1 é não-definido. Por outro lado, a avaliação de $f_Y(0)$ pela regra-nome resulta

em 1. Na realidade, tal situação não ocorre, pois a definição de máquina exige que as interpretações de operações ou testes sejam funções totais. ❑

Retornando à simplificação do Exemplo 4.23, pode-se substituir f_2, f_3 e f_4 pelas suas correspondentes expressões resultando na seguinte definição recursiva:

$f_Y = \lambda x.f_Y(x) = f_1(x, 0)$

$f_1 = \lambda(x, y).f_1(x, y) = (x = 0 \rightarrow y, f_1(x - 1, y + 1))$

Observa-se, mais uma vez, que tal simplificação é possível pois ambas as regras de avaliação proporcionam o mesmo resultado.

4.4 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) completamente diferente da operacional. Comparativamente com as máquinas universais, possuem um poder de expressão significativo no sentido em que uma simples função pode representar um algoritmo consideravelmente complexo.

Em algumas universidades (principalmente na Europa), linguagens baseadas em funções recursivas são usadas como uma primeira linguagem de programação, principalmente em cursos técnicos como Engenharias. A grande vantagem é que um aluno com algum conhecimento de funções matemáticas e sem qualquer conhecimento prévio de computação pode desenvolver programas com razoáveis níveis de complexidade em pouco tempo (em alguns casos, em minutos).

De qualquer forma, quase a totalidade das linguagens de programação modernas como Pascal ou C possuem recursão como um construtor básico de programas.

Adicionalmente, a arquitetura da maioria dos atuais computadores possui facilidades para implementar recursão. Conseqüentemente, o processamento de recursão possui, em geral, bons níveis de eficiência.

Correntemente, uma das principais ênfases dos estudos teóricos-formais referente aos formalismos denotacionais baseados em recursão é a questão da composição concorrente, onde existem questões em aberto ou com soluções nem sempre satisfatórias. Entretanto, como afirmado anteriormente, concorrência não é ênfase desta publicação.

4.5 Exercícios

Exercício 4.1 Suponha que x, y e z têm seus valores em $\mathbb{N}$ e que o tipo de função g é g: $\mathbb{N}^2 \to \mathbb{N}$. Para cada item abaixo, determine o tipo da função e descreva-a, usando a Linguagem Lambda:

a) f(g) (x, y) = (g (g(x, y), y), y)

b) f(g, x, y) = g(x, y)

c) f(g, x) = g (g (x, x), x)

d) f(x) (y) = g(x, y)

Exercício 4.2 Descreva, usando a Linguagem Lambda:

a) Função exponencial;

b) Composição simples de um dado conjunto de funções.

Exercício 4.3 Compare as funções recursivas (Kleene) parciais e totais, bem com as definições recursivas (Bird), destacando:

a) Diferenças conceituais e a interpretação dessas diferenças;

b) Relação entre essas classes de funções;

c) Importância de cada classe no estudo da computabilidade.

Exercício 4.4 Usando as funções zero, sucessor e adição, verifique se existem outras formas de definir equivalentemente as funções um, dois e três apresentadas no Exemplo 4.9.

Exercício 4.5 Compare e diferencie a constante $\text{const}_{\text{zero}}$ (Exemplo 4.11) e a função constante zero.

Exercício 4.6 Determine o valor de sub(2, 3) para a função recursiva do Exemplo 4.13.

Exercício 4.7 Quais das funções do Exemplo 4.14 são totais?

Exercício 4.8 Desenvolva funções recursivas totais sobre $\mathbb{N}$ para as seguintes operações:

a) Multiplicação;

b) Quadrado (n^2);

c) Fatorial (n!).

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

Exercício 4.9 As funções recursivas de Kleene são definidas sobre $\mathbb{N}$. Como poderia ser uma função recursiva para tratar palavras? Analogamente para tratar as definições recursivas de Bird?

Exercício 4.10 Usando a solução do exercício acima, demonstre que as funções que seguem são recursivas (Kleene). Em cada caso, verifique se é total ou parcial (suponha $\Sigma = \{a, b\}$):

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

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

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

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

e) antecede = $\lambda(x, y)$.antecede(x, y): $(\Sigma^*)^2 \rightarrow \Sigma^*$ tal que retorna:

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

Exercício 4.11 *Função de Ackernmann*. A função de Ackernmann é um importante exemplo no estudo das funções recursivas e *recursivas primitivas* (não introduzidas nesta publicação). A função de Ackernmann é sobre $\mathbb{N}$ e é tal que:

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

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

b) Calcule, passo a passo:

$Ack(0, 0) = 1$
$Ack(2, 1) = 1$

c) Defina, formalmente, a função recursiva de um argumento Ack(x, 2).

Exercício 4.12 Determine se as funções regra-valor e regra-nome induzidas pela seguinte definição recursiva coincidem:

$$f = \lambda x.f(x, y) = (x = 0 \rightarrow 1, y = 0 \rightarrow f(x, 1), f(y - 1, f(x, y - 1)))$$

Exercício 4.13 Construa a correspondente definição recursiva para a função computada pelo programa na Figura 4.5, descrito em uma linguagem tipo Pascal.

```
programa:
início
        entrada x;
        saída y;
        z := x;
        y := 0;
        até   x = 0
        faça  início
              x := x - 1;
              até   z = 0
              faça  início
                    z := z - 1;
                    y := Y + 1
                    fim;
              z := y;
              fim;
fim.
```

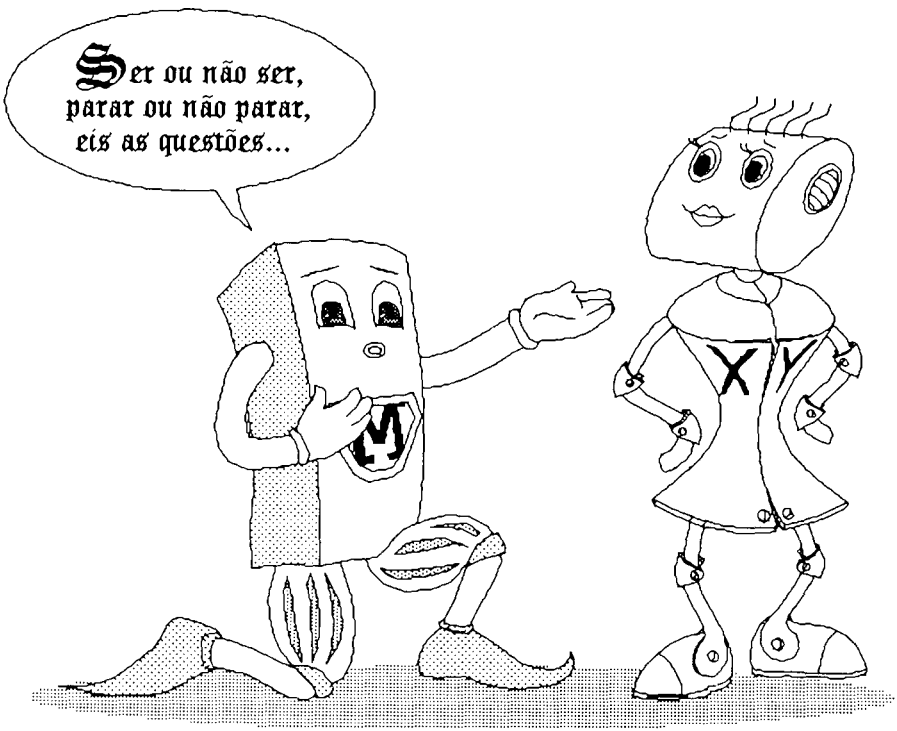
Figura 4.5 Programa em uma linguagem tipo Pascal

Exercício 4.14 Descreva a seguinte função usando a Linguagem Lambda, bem como dê a sua definição recursiva:

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

Exercício 4.15 Um número natural $n > 1$ é dito um *número perfeito* se for igual à soma de seus divisores, incluindo 1, mas excluindo n. Construa uma definição recursiva para a função:

perfeito(x) = (x + 1)-ésimo número perfeito



5 Computabilidade

O objetivo do estudo da solucionabilidade de problemas é o de investigar a existência ou não de algoritmos que solucionem determinada classe de problemas. Ou seja, investigar os limites da computabilidade e, conseqüentemente, os limites do que pode efetivamente ser implementado em um computador.

Em particular, o estudo da solucionabilidade objetiva evitar a pesquisa de soluções inexistentes. Para exemplificar a importância de tal estudo, em 1901, Hilbert formulou uma lista de problemas a serem resolvidos pelas futuras gerações de matemáticos. O décimo problema dessa lista consiste na existência ou não de um algoritmo que determine se uma equação polinomial qualquer, com coeficientes inteiros, possui solução nos inteiros. Somente em 1970, Matijasevic ([MAT70]) provou ser tal problema sem solução.

A abordagem apresentada concentra-se nos problemas com respostas binárias do tipo sim ou não, os quais serão referidos simplesmente como *problemas sim/não* ou *problemas de decisão*. A vantagem de tal abordagem é que a verificação da solucionabilidade de um problema pode ser tratada como a verificação se determinada linguagem é recursiva, associando as condições de ACEITA/REJEITA de uma Máquina Universal às respostas sim/não, respectivamente. Conseqüentemente, tem-se que:

a Classe dos Problemas Solucionáveis é equivalente à Classe das Linguagens Recursivas

Na prática, qualquer problema pode ser tratado equivalentemente como um problema (ou uma classe de problemas) sim/não.

Infelizmente, muitos dos problemas interessantes e importantes para a Ciência da Computação, bem como para as ciências em geral, são não-solucionáveis. Alguns exemplos de problemas não-solucionáveis de fundamental importância para a Ciência da Computação são os seguintes (introduzidos informalmente):

a) *Equivalência de Compiladores*. Não existe algoritmo genérico que sempre pare capaz de comparar quaisquer dois compiladores de linguagens livres do contexto (reconhecidas pelo formalismo Autômato com Uma Pilha Não-Determinístico, ver 3.7 - Hierarquia de Classes de Máquinas), como PASCAL, e verificar se são equivalentes, ou seja, se, de fato, reconhecem a mesma linguagem;

b) *Detector Universal de Loops*. Dados um programa e uma entrada quaisquer, não existe algoritmo genérico capaz de verificar se o programa vai parar ou não para a entrada. Este problema é universalmente conhecido como o *Problema da Parada*.

Alguns problemas não-solucionáveis são parcialmente solucionáveis, ou seja, existe um algoritmo capaz de responder sim, embora, eventualmente, possa ficar em *loop* infinito para uma resposta que deveria ser não. Como o leitor já deve ter intuído, problemas parcialmente solucionáveis são computáveis e, portanto, tem-se que:

a Classe dos Problemas Parcialmente Solucionáveis é equivalente à Classe das Linguagens Enumeráveis Recursivamente

É interessante comparar o cardinal da Classe dos Problemas Computáveis com o cardinal da Classe dos Problemas Não-Computáveis, pois tal relação fornece uma noção de grandeza. O seguinte pode ser estabelecido (tal estudo não é detalhado) sobre os cardinais das Classes dos Problemas:

- Computáveis é contável;
- Não-Computáveis é não-contável.

Assim, informalmente, pode-se afirmar que o cardinal da Classe dos Problemas Não-Computáveis é "muito maior" que a Classe dos Problemas Computáveis, ou seja, existem muito mais problemas não-computáveis do que computáveis.

O estudo da solucionabilidade de um problema é feito, em geral, usando o *Princípio da Redução*, o qual consiste, basicamente, na investigação da solucionabilidade de um problema a partir de outro, cuja classe de solucionabilidade é conhecida. O princípio da redução pode ser resumido como segue (veja a Figura 5.1):

a) Sejam A e B dois problemas de decisão. Suponha que é possível modificar ("reduzir") o problema A de tal forma que ele se porta como um caso do problema B;

b) Se A é não-solucionável (respectivamente, não-computável), então, como A é um caso de B, conclui-se que B também é não-solucionável (respectivamente, não-computável);

c) Se B é solucionável (respectivamente, parcialmente solucionável) então, como A é um caso de B, conclui-se que A também é solucionável (respectivamente, parcialmente solucionável).

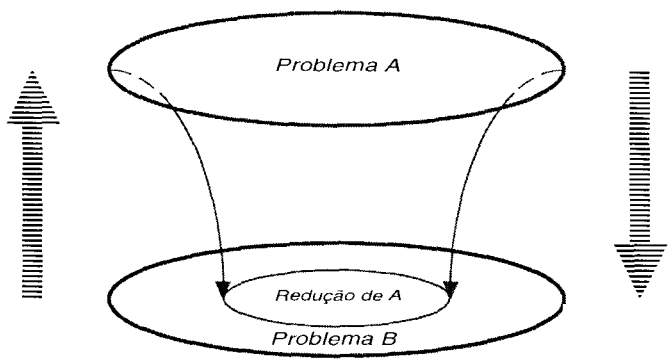


Figura 5.1 Princípio da Redução

5.1 Classes de Solucionabilidade de Problemas

O conjunto de todos os problemas pode ser particionado de diversas formas. Uma maneira consiste nas duas Classes ilustradas na Figura 5.2, induzidas pelas definições que seguem.

Definição 5.1 Problema Solucionável.

Um problema é dito *Solucionável* ou *Totalmente Solucionável* se existe um algoritmo (Máquina Universal) que solucione o problema tal que sempre pára para qualquer entrada, com uma resposta afirmativa (ACEITA) ou negativa (REJEITA). ❑

Definição 5.2 Problema Não-Solucionável.

Um problema é dito *Não-Solucionável* se *não* existe um algoritmo (Máquina Universal) que solucione o problema tal que sempre pára para qualquer entrada. ❑

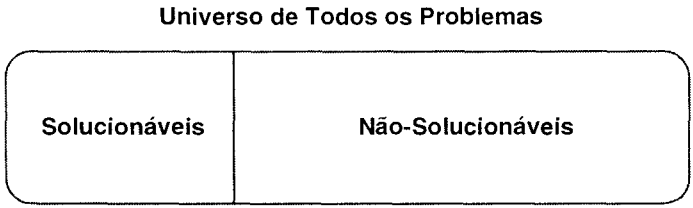


Figura 5.2 Particionamento do conjunto de todos os problemas em classes

Outra maneira de se particionar o conjunto de todos os problemas, consiste nas duas Classes ilustradas na Figura 5.3, induzidas pela definição que segue.

Definição 5.3 Problema Parcialmente Solucionável ou Computável.

Um problema é dito *Parcialmente Solucionável* ou *Computável* se existe um algoritmo (Máquina Universal) que solucione o problema tal que pare quando a resposta é afirmativa (ACEITA). Entretanto, quando a resposta esperada for negativa, o algoritmo pode parar (REJEITA) ou permanecer processando indefinidamente (LOOP). ❑

Definição 5.4 Problema Completamente Insolúvel ou Não-Computável.

Um problema é dito *Completamente Insolúvel* ou *Não-Computável* se *não* existe um algoritmo (Máquina Universal) que solucione o problema tal que pare quando a resposta é afirmativa (ACEITA). ❑

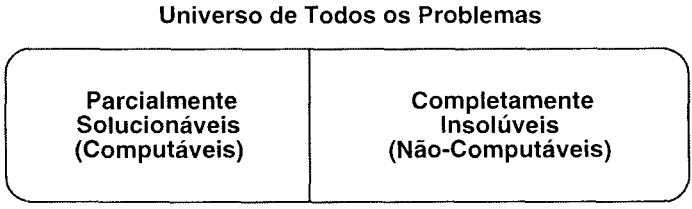


Figura 5.3 Particionamento do conjunto de todos os problemas em classes

É importante observar que alguns problemas não-solucionáveis são parcialmente solucionáveis. Relativamente ao relacionamento das classes de problemas, as seguintes conclusões podem ser estabelecidas (veja a Figura 5.4):

- a união das Classes Solucionáveis e Não-Solucionáveis é o Universo de Todos os Problemas;
- a união das Classes Parcialmente Solucionáveis e Completamente Insolúveis é o Universo de Todos os Problemas;
- a Classe dos Parcialmente Solucionáveis contém propriamente a Classe dos Solucionáveis e parte da Classe dos Não-Solucionáveis;
- todo problema solucionável é parcialmente solucionável;
- existem problemas não-solucionáveis que possuem solução parcial;
- os problemas completamente insolúveis não possuem solução total nem parcial.

Para qualquer algoritmo que solucione um problema parcialmente solucionável que é não-solucionável, sempre existe pelo menos uma palavra de entrada que faz com que o algoritmo fique em *loop*.

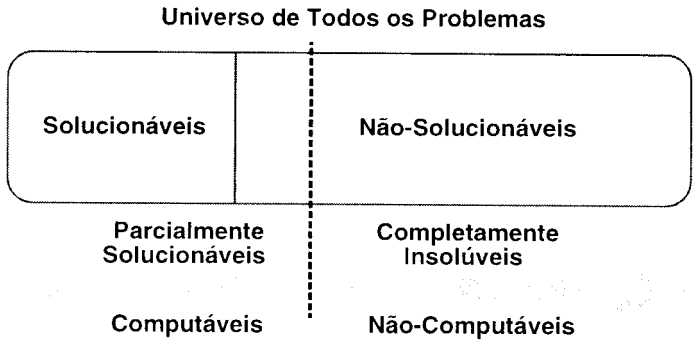


Figura 5.4 Relação entre as classes de problemas

5.2 Problemas de Decisão

Neste capítulo são tratados alguns problemas de decisão ou do tipo sim/não. A idéia básica é verificar se a função associada a eles é ou não computável em uma Máquina Universal. Como, pela Hipótese de Church, uma Máquina Universal é o dispositivo mais geral de computação, se a solução de um problema não puder ser expressa por uma Máquina Universal, então tal problema é completamente insolúvel.

A essência de um problema de decisão é dada pela seguinte idéia: dado um programa P para máquina universal M, decidir se a função computada ⟨P, M⟩ é total (ou seja, se a correspondente computação é finita). Lembre-se que, alguns dos formalismos estudados, como a Máquina de Turing, constituem, de fato, um programa para uma máquina.

Assim, não-solucionabilidade refere-se à inexistência de um método geral e efetivo para decidir se um programa para uma máquina universal pára para qualquer entrada. É importante reparar que o que está sendo discutido são métodos gerais. Portanto, é perfeitamente possível existir métodos específicos para programas particulares.

A existência de problemas não-solucionáveis é importante por diversas razões como, por exemplo:

- alguns desses problemas não-solucionáveis permitem estabelecer, por si sós, importantes resultados para a Ciência da Computação (como a inexistência de um detetor universal de *loops* referenciado anteriormente);
- demonstrar limitações da capacidade de expressar-se soluções através de programas.

Adicionalmente, a existência de um problema não-solucionável pode ser usada para verificar que outros problemas também o são. Isso é possível, usando o princípio da redução, o qual consiste em "reduzir" o problema que se está investigando em outro problema que já se saiba ser não-solucionável.

Observação 5.5 Solucionabilidade de Problemas × Problema de Reconhecimento de Linguagens.

A investigação da solucionabilidade de problemas em máquinas universais pode ser vista como um problema de reconhecimento de linguagens que segue o esquema abaixo:

a) O problema é reescrito como um problema de decisão, capaz de gerar respostas do tipo afirmativo/negativo (sim/não). Esta redefinição é simples para a maioria dos problemas gerais;

b) Os argumentos do problema sim/não são codificados como palavras de um alfabeto, gerando uma linguagem.

Assim, a questão da solucionabilidade de um problema pode ser traduzida como uma investigação se a linguagem gerada é recursiva (problema solucionável) ou enumerável recursivamente (problema parcialmente solucionável). ❑

5.3 Codificação de Programas

Alguns problem :s de decisão são definidos sobre programas ou máquinas. Assim é necessário estabelecer uma forma de codificar programas ou máquinas.

No Capítulo 3 - Máquinas Universais, foi mostrado como codificar um programa como um número natural. Como conseqüência, problemas de decisão sobre programas podem ser traduzidos em problemas de decisão sobre naturais. Uma desvantagem da função de codificação introduzida, denominada de código, é

que não é bijetora (mas é injetora). De fato, nem todo natural é codificação de algum programa. Tal problema pode ser contornado como exemplificado a seguir.

A abordagem é específica para programas monolíticos (para a Máquina Norma), mas pode facilmente ser generalizado para os demais tipos de programas e para máquinas universais.

EXEMPLO 5.1 Codificação Bijetora de Programas.

Seja $\mathbb{P}$ o conjunto de todos os programas do tipo que está sendo considerado (monolítico, iterativo ou recursivo). Considere a seqüência de inteiros $p_0, p_1, p_2, \ldots$ formada pelos códigos dos programas de $\mathbb{P}$ (ordenados pela relação "menor" sobre os naturais). Então:

$$\text{código\_bij} = \lambda p.\text{código\_bij}(p): \mathbb{P} \rightarrow \mathbb{N}$$

onde, para qualquer $P \in \mathbb{P}$, se o código de P é p_n, então (veja a Figura 5.5):

$$\text{código\_bij}(P) = n$$

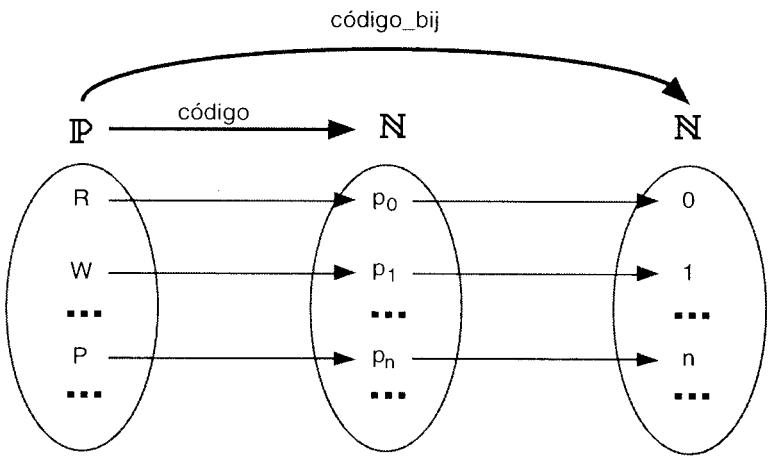


Figura 5.5 Função bijetora de codificação de programas

Claramente, cada programa é associado com um natural e vice-versa, ou seja, código_bij é uma função bijetora. Um algoritmo para determinar código_bij(P) é como segue:

- calcula p = código(P)
- verifica (usando decodificação - sugerido como exercício) quantos números naturais menores do que p são codificações de programas. Suponha que sejam n naturais que satisfaçam tal condição;
- então código_bij(P) = n + 1

A inversa da função código_bij (que, obviamente, também é bijetora):

$$\text{código\_bij}^{-1} = \lambda n.\text{código\_bij}^{-1}(n): \mathbb{N} \rightarrow \mathbb{P}$$

é tal que, para qualquer natural, retorna o correspondente programa. Um algoritmo para determinar código_bij^{-1}(n) é como segue:

- verifica (usando decodificação) os n primeiros números naturais que denotam codificações de programas. Suponha que p é o n-ésimo natural que satisfaz tal condição;
- assim, o n-ésimo programa será a decodificação de p. ❑

Por simplicidade, no texto que segue, a função código_bij será denominada simplesmente de código.

5.4 Problema da Auto Aplicação

O *Problema da Auto-Aplicação* é um problema não-solucionável (mas parcialmente solucionável), de natureza artificial, possuindo, por si só, aplicação restrita. Entretanto, é de fundamental importância, pois pode ser usado como base na demonstração de que outros problemas também são não-solucionáveis (ou parcialmente solucionáveis).

Definição 5.6 Problema da Auto-Aplicação.

O *Problema da Auto-Aplicação* é como segue (versão para a Máquina Norma):

Dado um programa monolítico arbitrário P para a Máquina Norma, decidir se a função computada ⟨P, Norma⟩ é definida para p, onde p é a codificação de P

❑

Em outras palavras, o Problema da Auto-Aplicação deve decidir, dado um programa monolítico arbitrário P para Norma, se a computação de P em Norma termina ou não, para a entrada p.

O Problema da Auto-Aplicação pode ser redefinido como uma linguagem, como segue:

L_{AA} = { p | ⟨P, Norma⟩(p) é definida,
P é programa monolítico para Norma e p = código(P) }

Assim, a questão da solucionabilidade do Problema da Auto-Aplicação é traduzida como uma investigação se a correspondente linguagem é recursiva (problema solucionável) ou enumerável recursivamente (problema parcialmente solucionável).

No que segue, para um programa P para a Máquina Norma e para uma entrada n, é suposto que:

n ∈ ACEITA(P) se, e somente se, ⟨P, Norma⟩(n) = 0
n ∈ REJEITA(P) se, e somente se, ⟨P, Norma⟩(n) = 1
n ∈ LOOP(P) se, e somente se, ⟨P, Norma⟩(n) é indefinida

Teorema 5.7 Problema da Auto-Aplicação é Parcialmente Solucionável.

A linguagem L_{AA} que traduz o Problema da Auto-Aplicação é enumerável recursivamente.

Prova:

Para provar que L_{AA} é enumerável recursivamente, é necessário mostrar que existe um programa monolítico Q para Norma, tal que:

$ACEITA(Q) = L_{AA}$
$REJEITA(Q) \cup LOOP(Q) = \mathbb{N} - L_{AA}$

Sejam P um programa monolítico qualquer para Norma e Q um programa monolítico para Norma capaz de simular qualquer outro programa.

O programa Q é tal que recebe como entrada p = código(P) e simula P para a entrada p (como este programa pode ser definido?). Adicionalmente:

- $p \in ACEITA(Q)$ se, e somente se, $\langle P, Norma \rangle(p)$ é definido, ou seja, a computação de P em Norma é finita;
- $p \in LOOP(Q)$ se, e somente se, a computação de P em Norma é infinita;
- $REJEITA(Q) = \varnothing$ (conseqüência dos dois casos acima).

Portanto:

$ACEITA(Q) = L_{AA}$
$REJEITA(Q) \cup LOOP(Q) = \mathbb{N} - L_{AA}$

Logo, L_{AA} é enumerável recursivamente e, portanto, o Problema da Auto-Aplicação é parcialmente solucionável. ⌟

Teorema 5.8 Problema da Auto-Aplicação é Não-Solucionável.

A linguagem L_{AA} que traduz o Problema da Auto-Aplicação *não* é recursiva.

Prova:

A demonstração que segue é por redução ao absurdo. Portanto, suponha que L_{AA} é recursiva. Então existe um programa monolítico Q para Norma como na Figura 5.6, tal que:

$ACEITA(Q) = L_{AA}$
$REJEITA(Q) = \mathbb{N} - L_{AA}$
$LOOP(Q) = \varnothing$ (conseqüência dos dois casos acima)

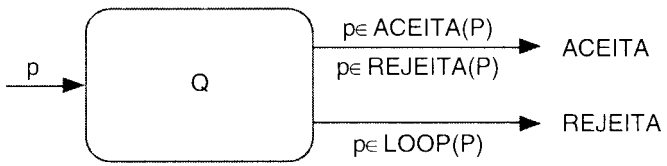


Figura 5.6 Programa para a Máquina Norma

Suponha o programa R como Q mas adicionando um trecho de programa que é executado ao final de cada computação finita de Q, como na Figura 5.7, com a seguinte função:

- antes de terminar a computação, testa o valor da saída de Q;
- se Q aceita ou rejeita, então R fica em *loop* infinito;
- se Q fica em *loop* infinito, R aceita.

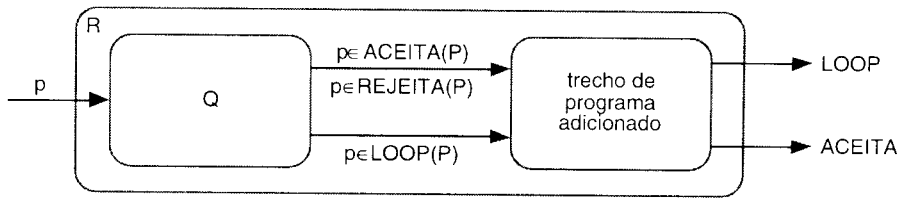


Figura 5.7 Programa modificado para a Máquina Norma

Assim, para a aplicação de R como entrada de R, tem-se que (suponha que r = código(R)):

a) R fica em *loop* infinito quando Q, ao simular R, aceita ou rejeita. Ou seja, R fica em *loop* infinito quando R pára;

b) R pára quando Q, ao simular R, fica em *loop* infinito. Ou seja, R pára quando R fica em *loop* infinito.

Assim, fica caracterizada a contradição.

Logo, L_{AA} *não* é recursiva e, portanto, o Problema da Auto-Aplicação é *não-solucionável*. ❑

5.5 Princípio da Redução

O *Princípio da Redução*, consiste, basicamente, na construção de um algoritmo de mapeamento entre as linguagens que traduzem os problemas. Assim, se a classe de uma dessas linguagens é conhecida, então se pode estabelecer algumas conclusões sobre a linguagem que se deseja investigar (e, portanto, sobre o grau de solucionabilidade do problema representado).

Nesta e nas demais seções deste capítulo, são supostas as seguintes generalizações:

- função código para qualquer Máquina Universal;
- Problema da Auto-Aplicação para qualquer Máquina Universal.

Assim, sempre que possível, as definições e resultados que seguem são independentes de formalismos.

Definição 5.9 Máquina de Redução.

Suponha dois problemas A e B e as correspondentes linguagens L_A e L_B. Uma *Máquina de Redução* R de L_A para L_B (sobre um alfabeto Σ) é tal que ($w \in \Sigma$):

a) Se $w \in L_A$, então $R(w) \in L_B$
b) Se $w \notin L_A$, então $R(w) \notin L_B$ ❑

Portanto, o mapeamento de linguagens é uma função computável total.

Teorema 5.10 Redução: Investigação da Solucionabilidade.

Suponha dois problemas A e B e as correspondentes linguagens L_A e L_B. Se existe uma máquina de redução R de L_A para L_B (sobre um alfabeto Σ), então os seguintes resultados podem ser estabelecidos:

a) Se L_B é recursiva, então L_A é recursiva;
b) Se L_B é enumerável recursivamente, então L_A é enumerável recursivamente;
c) Se L_A não é recursiva, então L_B não é recursiva;
d) Se L_A não é enumerável recursivamente, então L_B não é enumerável recursivamente.

Prova:

Seja R Máquina de Turing de Redução que sempre pára e que reduz L_A a L_B, como ilustrado na Figura 5.8.

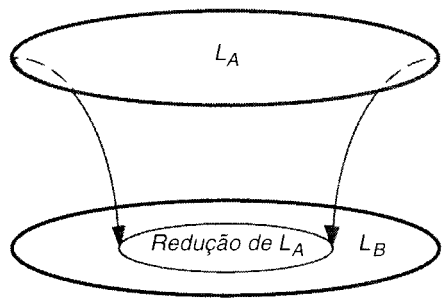


Figura 5.8 Redução

a) Suponha que L_B é uma linguagem recursiva. Então, existe M_B, Máquina Universal, que aceita L_B e sempre pára para qualquer entrada. Seja a Máquina Universal M definida como na Figura 5.9.

As seguintes conclusões podem ser estabelecidas:

- M sempre pára para qualquer entrada, pois R e M_B sempre param;
- se $w \in L_A$, então M aceita w, pois $R(w) \in L_B$
- se $w \notin L_A$, então M rejeita w, pois $R(w) \notin L_B$

Portanto, M aceita L_A e sempre pára para qualquer entrada. Logo, L_A é uma linguagem recursiva;

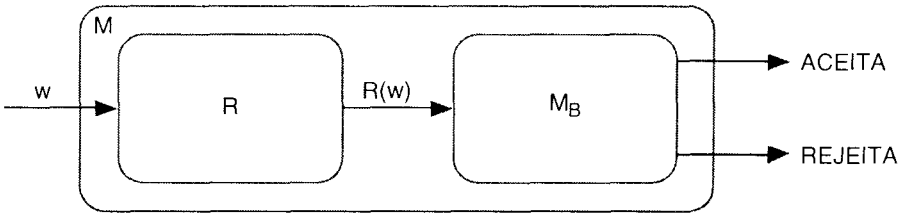


Figura 5.9 Máquina Universal

b) Suponha que L_B é uma linguagem enumerável recursivamente. Então, existe M_B, uma Máquina Universal, tal que:

$ACEITA(M_B) = L_B$

$REJEITA(M_B) \cup LOOP(M_B) = \Sigma^* - L_B$

Seja a Máquina Universal M definida como na Figura 5.10.

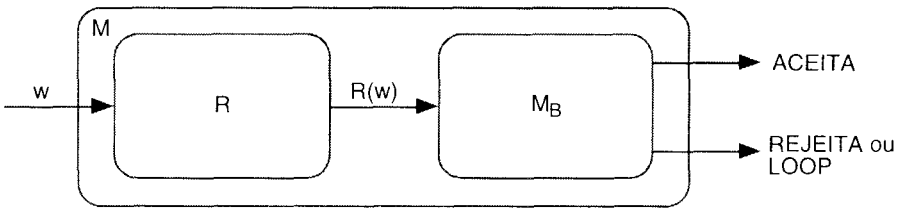


Figura 5.10 Máquina Universal

As seguintes conclusões podem ser estabelecidas:

- se $w \in L_A$, então M aceita w, pois $R(w) \in L_B$
- se $w \notin L_A$, então M rejeita ou fica em *loop* para a entrada w, pois M_B rejeita ou fica em *loop* para a entrada R(w).

Portanto, M aceita L_A, mas pode ficar em *loop* para entradas não-pertencentes a L_A. Logo, L_A é uma linguagem enumerável recursivamente;

c) e d) Por contraposição, as afirmações c) e d) são equivalentes às afirmações a) e b), respectivamente. Lembre-se que (suponha que p e q são proposições):

$$(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$$ ❑

5.6 Problema da Parada

Um dos mais importantes problemas não-solucionáveis é conhecido como o *Problema da Parada*.

Definição 5.11 Problema da Parada.

O *Problema da Parada* é como segue:

Dada uma Máquina Universal M qualquer e uma palavra w qualquer sobre o alfabeto de entrada, existe um algoritmo que verifique se M pára, aceitando ou rejeitando, ao processar a entrada w?

O Problema da Parada é um problema de decisão, do tipo sim/não e pode ser redefinido pela seguinte linguagem:

$$L_P = \{(m, w) \mid m = \text{código}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M)\}$$

O teorema a seguir mostra que a linguagem L_P não é recursiva, o que significa que o Problema da Parada é não-solucionável. A demonstração usa o princípio da redução.

Teorema 5.12 Problema da Parada é Não-Solucionável.

A seguinte linguagem que traduz o Problema da Parada não é recursiva:

$$L_P = \{(m, w) \mid m = \text{código}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M)\}$$

Prova:

A demonstração que segue é por redução ao absurdo e usa o princípio da redução. Suponha que L_P é recursiva. Então existe uma Máquina Universal M_P sobre o alfabeto Σ, tal que:

$\text{ACEITA}(M_P) = L_P$
$\text{REJEITA}(M_P) = \Sigma^* - L_P$
$\text{LOOP}(M_P) = \varnothing$ (conseqüência dos dois casos acima)

Suponha uma Máquina Universal R que, para qualquer entrada w, gera o par (w, w) (tal máquina pode ser facilmente definida). Seja M uma máquina como na Figura 5.11.

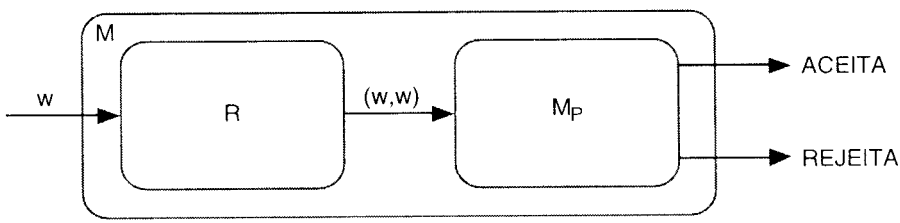


Figura 5.11 Máquina construída usando o princípio da redução

Claramente, o Problema da Auto-Aplicação foi reduzido ao Problema da Parada (Figura 5.12), pois:

- se $w \in L_{AA}$, então $R(w) = (w, w) \in L_P$
- se $w \notin L_{AA}$, então $R(w) = (w, w) \notin L_P$

Como é suposto que o Problema da Parada é solucionável, então, pelo Teorema 5.10 - Redução: Investigação da Solucionabilidade, o Problema da Auto-Aplicação

é solucionável, o que é um absurdo. Logo, é absurdo supor que o Problema da Parada é solucionável e, portanto, é não-solucionável. ❑

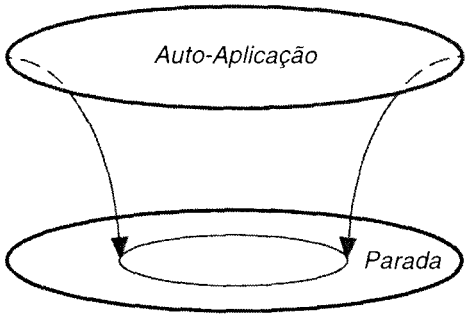


Figura 5.12 Redução

Analogamente ao problema da Auto-Aplicação, o Problema da Parada é parcialmente solucionável. A prova do teorema a seguir é omitida (é sugerida como exercício) e pode ser desenvolvida de duas formas:

- prova direta, de forma similar à prova de que o Problema da Auto-Aplicação é parcialmente solucionável;
- usando o princípio da redução.

Teorema 5.13 Problema da Parada é Parcialmente Solucionável.

A seguinte linguagem que traduz o Problema Parada é enumerável recursivamente:

$$L_P = \{(m, w) \mid m = \text{código}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M)\}$$ ❑

5.7 Outros Problemas de Decisão

Muitos problemas de decisão sobre Máquinas Universais são não-solucionáveis. Na realidade, é fácil definir um problema não-solucionável. Nesta seção, são apresentados, de forma resumida, os seguintes problemas não-solucionáveis:

- Problema da Parada da Palavra Vazia;
- Problema da Totalidade;
- Problema da Equivalência.

O *Problema da Parada da Palavra Vazia* é uma variação do Problema da Parada, restringindo a entrada à palavra vazia (ou ausência de entrada).

Definição 5.14 Problema da Parada da Palavra Vazia.

O *Problema da Parada da Palavra Vazia* é como segue:

Dada uma Máquina Universal M qualquer, existe um algoritmo que verifique se M pára, aceitando ou rejeitando, ao processar a entrada vazia?

❑

Teorema 5.15 Problema da Parada da Palavra Vazia é Não-Solucionável.

A seguinte linguagem que traduz o Problema da Parada da Palavra Vazia não é recursiva:

$$L_\varepsilon = \{ m \mid m = \text{código}(M) \text{ e } \varepsilon \in \text{ACEITA}(M) \cup \text{REJEITA}(M) \}$$

Prova:

A demonstração que segue usa o princípio da redução. Especificamente, a linguagem L_P que traduz o Problema da Parada (que não é recursiva) é reduzida à linguagem L_ε.

Sejam:

- T uma Máquina Universal qualquer, definida sobre o alfabeto Σ;
- w uma palavra qualquer sobre Σ;
- W uma Máquina Universal que recebe como entrada a palavra vazia e gera (saída) a palavra w;
- M uma Máquina Universal definida em termos de T e W como na Figura 5.13. Note que a construção de M a partir de T é o algoritmo de redução.

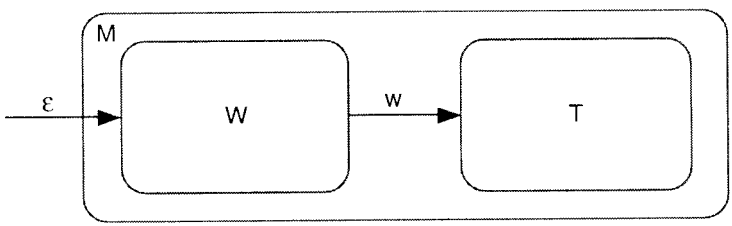


Figura 5.13 Máquina Universal

Assim, as seguintes conclusões podem ser estabelecidas:

- se T aceita a palavra w, então M aceita a palavra vazia;
- se T não aceita a palavra w (rejeita ou fica em *loop*), então M não aceita a palavra vazia (rejeita ou fica em *loop*).

Ou seja (suponha que t e m são os códigos de T e M, respectivamente):

- se $(t, w) \in L_P$, então $m \in L_\varepsilon$
- se $(t, w) \notin L_P$, então $m \notin L_\varepsilon$

Portanto, o Problema da Parada é reduzido ao Problema da Parada da Palavra Vazia (Figura 5.14).

Logo, o Problema da Parada da Palavra Vazia é não-solucionável. ❑

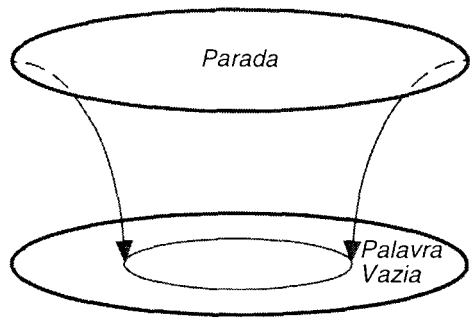


Figura 5.14 Redução

O *Problema da Totalidade* é uma variação do Problema da Parada, generalizado para toda entrada.

Definição 5.16 Problema da Totalidade.

O *Problema da Totalidade* é como segue:

Dada uma Máquina Universal M qualquer, existe um algoritmo que verifique se M pára, aceitando ou rejeitando, ao processar qualquer entrada?

❑

Teorema 5.17 Problema da Totalidade é Não-Solucionável.

A seguinte linguagem que traduz o Problema da Totalidade não é recursiva:

$$L_T = \{ m \mid m = \text{código}(M) \text{ e } LOOP(M) = \varnothing \}$$

Prova:

A demonstração que segue é feita por redução ao absurdo, usa o princípio da redução e é análoga à do Problema da Parada. Suponha que L_T é recursiva. Então existe uma Máquina Universal M_T, tal que sempre pára e $ACEITA(M_T) = L_T$.

Suponha uma Máquina Universal R que, para qualquer entrada (p, w), gera p (projeção da primeira componente do par). Seja M uma máquina como na Figura 5.15.

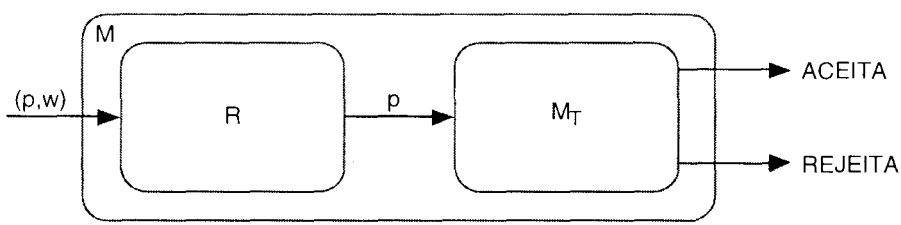


Figura 5.15 Máquina construída usando o princípio da redução

Claramente, o Problema da Parada foi reduzido ao Problema da Totalidade (Figura 5.16), pois:

- se $(p, w) \in L_P$, então $R((p, w)) = p \in L_T$;
- se $(p, w) \notin L_P$, então $R((p, w)) = p \notin L_T$.

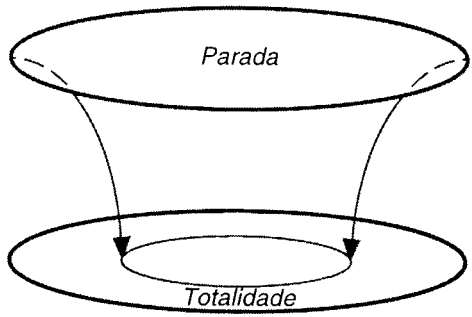


Figura 5.16 Redução

Como é suposto que o Problema da Totalidade é solucionável, então, pelo Teorema 5.10 - Redução: Investigação da Solucionabilidade, o Problema da Parada é recursivo, o que é um absurdo. Logo, é absurdo supor que o Problema da Totalidade é solucionável e, portanto, é não-solucionável. ❑

Definição 5.18 Problema da Equivalência.

O *Problema da Equivalência* é um problema de decisão (do tipo sim/não) que verifica a equivalência de duas máquinas universais. ❑

Teorema 5.19 Problema da Equivalência é Não-Solucionável.

A seguinte linguagem que traduz o Problema da Equivalência não é recursiva:

$$L_E = \{(m, p) \mid m = \text{código}(M), p = \text{código}(P),$$
$$\text{ACEITA}(M) = \text{ACEITA}(P) \text{ e } \text{REJEITA}(M) = \text{REJEITA}(P)\}$$

Prova:

A demonstração que segue usa o princípio da redução. Especificamente, a linguagem L_E que traduz o Problema da Parada da Palavra Vazia (que não é recursiva) é reduzida à linguagem L_E.

Sejam:

- T uma Máquina Universal qualquer;
- Vazia uma Máquina Universal que recebe como entrada qualquer palavra e sempre gera (saída) a palavra ε;
- Pára_Vazia uma Máquina Universal que sempre pára para a entrada vazia;
- M uma Máquina Universal definida em termos de T, Vazia e Pára_Vazia como na Figura 5.17.

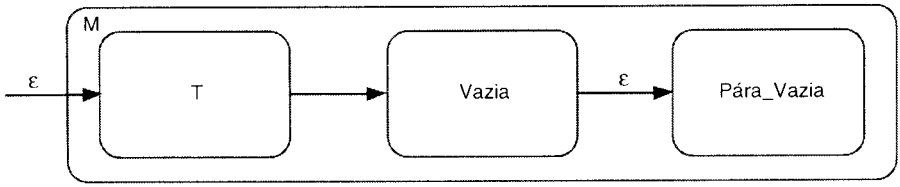


Figura 5.17 Máquina Universal

Assim, as seguintes conclusões podem ser estabelecidas (suponha que t e p são os códigos de T e Pára_Vazia, respectivamente):

- se $t \in L_\varepsilon$, então $(t, p) \in L_E$
- se $t \notin L_\varepsilon$, então $(t, p) \notin L_E$

Portanto, o Problema da Parada da Palavra Vazia é reduzido ao Problema da Equivalência (Figura 5.18).

Logo, o Problema da Equivalência é não-solucionável. ❑

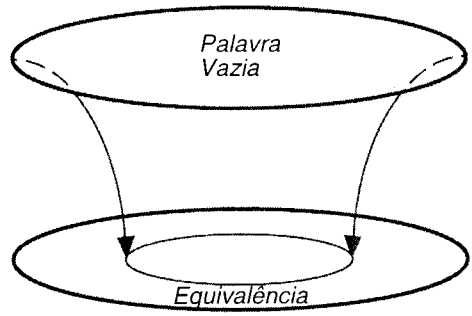


Figura 5.18 Redução

5.8 Problema da Correspondência de Post

O *Problema da Correspondência da Post* (o qual se prova ser não-solucionável) é de fundamental importância para a verificação da questão da solucionabilidade de diversas outras classes de problemas, principalmente no estudo das Linguagens Formais como, por exemplo, a equivalência de reconhecedores sintáticos de linguagens. O Problema da Correspondência de Post é definido sobre um *Sistema de Post*.

Definição 5.20 Sistema de Post.

Um *Sistema de Post* S definido sobre um alfabeto Σ é um conjunto finito e não-vazio de pares ordenados de palavras sobre Σ. ❑

Portanto, um Sistema de Post é um conjunto da seguinte forma, onde $n > 1$ e $x_i, y_i \in \Sigma^*$, para $i \in \{1, 2, \ldots, n\}$:

$$S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$

Uma solução para um Sistema de Post é uma seqüência não-vazia de números naturais com valores em $\{1, 2, \ldots, n\}$:

$$i_1, i_2, \ldots, i_k$$

tal que:

$$x_{i_1} x_{i_2} \ldots x_{i_k} = y_{i_1} y_{i_2} \ldots y_{i_k}$$

O *Problema da Correspondência de Post* é a investigação da existência de um algoritmo que analise qualquer Sistema de Post e determine se ele tem pelo menos uma solução. Demonstra-se, usando o princípio da redução, que esse problema é não-solucionável.

EXEMPLO 5.2 *Problema da Correspondência de Post.*

Seja o Sistema de Post sobre $\Sigma = \{a, b\}$ dado pelo seguinte conjunto:

$$S = \{(b, bbb), (babbb, ba), (ba, a)\}$$

Uma solução de S é:

$$2, 1, 1, 3 \quad \text{pois} \quad babbb\,b\,b\,ba = ba\,bbb\,bbb\,a$$ ❑

EXEMPLO 5.3 *Problema da Correspondência de Post.*

Seja o Sistema de Post sobre $\Sigma = \{a, b\}$ dado pelo seguinte conjunto:

$$S = \{(ab, abb), (b, ba), (b, bb)\}$$

S não tem solução pois, para qualquer par $(x_i, y_i) \in S$, tem-se que $|x_i| < |y_i|$. ❑

EXEMPLO 5.4 *Problema da Correspondência de Post.*

Seja o Sistema de Post sobre $\Sigma = \{a, b\}$ dado pelo seguinte conjunto:

$$S = \{(a, ba), (bba, aaa), (aab, b), (ab, bba)\}$$

S não tem solução pois, para qualquer par $(x_i, y_i) \in S$, o primeiro símbolo de x_i é diferente do primeiro símbolo de y_i. ❑

Teorema 5.21 Problema da Correspondência de Post é Não-Solucionável.

A seguinte linguagem que traduz o Problema da Correspondência de Post *não* é recursiva:

$L_{CP} = \{s \mid s = \text{código}(S) \text{ e } S \text{ é Sistema de Post com pelo menos uma solução}\}$

Prova:

A partir de uma Máquina de Post M qualquer sobre o alfabeto Σ e de uma palavra $w \in \Sigma^*$ qualquer, constrói-se um Sistema Normal de Post baseado na seqüência

de comandos executados por M para a entrada w, de tal forma que o Sistema Normal tenha solução se e somente se M pára para a entrada w. Portanto, o Problema da Parada é reduzido ao Problema da Correspondência de Post. Ou seja, a linguagem L_P que traduz o Problema da Parada (que não é recursiva) é reduzida à linguagem L_{CP}.

A idéia básica da construção é enumerar os comandos da máquina M e, para cada ação sobre a variável X, gerar um par do Sistema de Post.

Suponha que:

- $w \in \Sigma^*$ uma palavra qualquer, onde $w = a_1 a_2 \ldots a_n$
- o valor inicial de X é w
- as componentes elementares do diagrama de fluxos de M são enumeradas sobre $\{1, 2, \ldots, m\}$,

A relação entre os componentes elementares de M e os pares do correspondente Sistema de Post é como segue (veja a Figura 5.19):

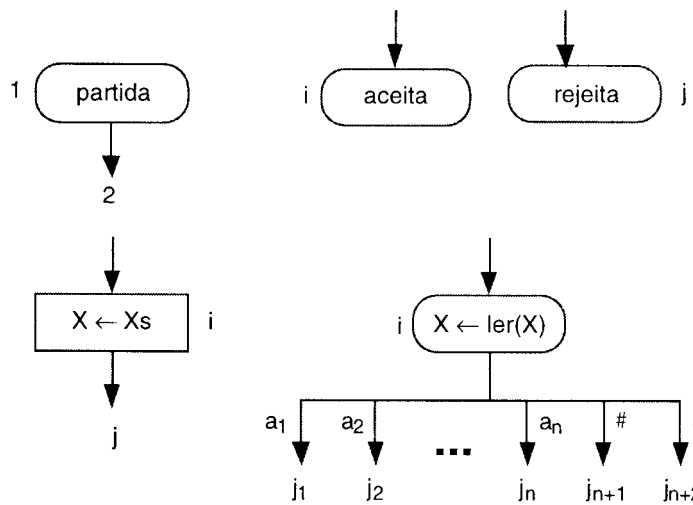


Figura 5.19 Enumeração das instruções de um diagrama de fluxos de uma Máquina de Post

a) *Partida*. Determina o seguinte par:

$$(1, 1\, a_1\, a_2 \ldots a_n\, 2)$$

b) *Desvio ou Teste*. Determina os seguintes pares:

$$(i\, a_1, j_1), (i\, a_2, j_2), \ldots, (i\, a_n, j_n), (i\, \#, j_{n+1}), (i\, \varepsilon, j_{n+2})$$

c) *Atribuição*. Determina o seguinte par:

$$(i, s\, j)$$

d) *Parada*. Determina o seguinte par:

$$(i, \varepsilon)$$

e) *Símbolo.* Cada símbolo $s \in \Sigma \cup \{\#\}$ determina o seguinte par

(s, s)

Deve-se reparar que, no par correspondente à partida, a segunda componente tem um número de instrução a mais que a primeira. Essa diferença só é compensada no par correspondente à instrução de aceita/rejeita. Assim, o Sistema Normal de Post somente tem solução se a máquina parar para a entrada w. Os demais detalhes são explicados através do exemplo que segue.

Portanto, é possível reduzir o Problema da Parada ao Problema da Correspondência de Post (Figura 5.20).

Logo, o Problema da Correspondência de Post é não-solucionável. ❑

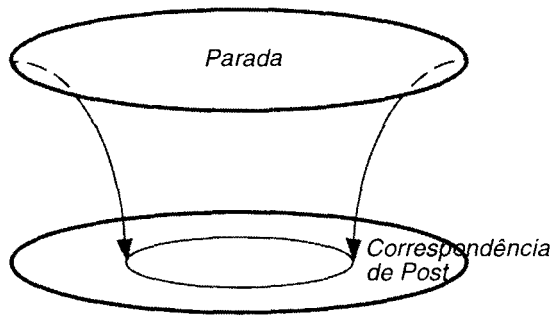


Figura 5.20 Redução

EXEMPLO 5.5 Máquina de Post → Sistema de Post.

Considere a Máquina de Post Post-Duplo_Bal na Figura 5.21 (introduzida anteriormente), a qual reconhece a linguagem Duplo_Bal = $\{a^n b^n \mid n \geq 0\}$, cujas instruções já estão enumeradas. Suponha a entrada w = ab. O Sistema de Post correspondente conforme o Teorema 5.21 resulta em 24 pares, os quais são como segue (os pares estão numerados para facilitar a sua identificação):

1: (1, 1ab2)	7: (4a, 5)	13: (6b, 7)	19: (10, ε)
2: (2, #3)	8: (4b, 6)	14: (6#, 8)	20: (11, ε)
3: (3a, 4)	9: (4#, 11)	15: (6ε, 12)	21: (12, ε)
4: (3b, 9)	10: (4ε, 11)	16: (7, b6)	22: (a, a)
5: (3#, 10)	11: (5, a4)	17: (8, #3)	23: (b, b)
6: (3ε, 9)	12: (6a, 12)	18: (9, ε)	24: (#, #)

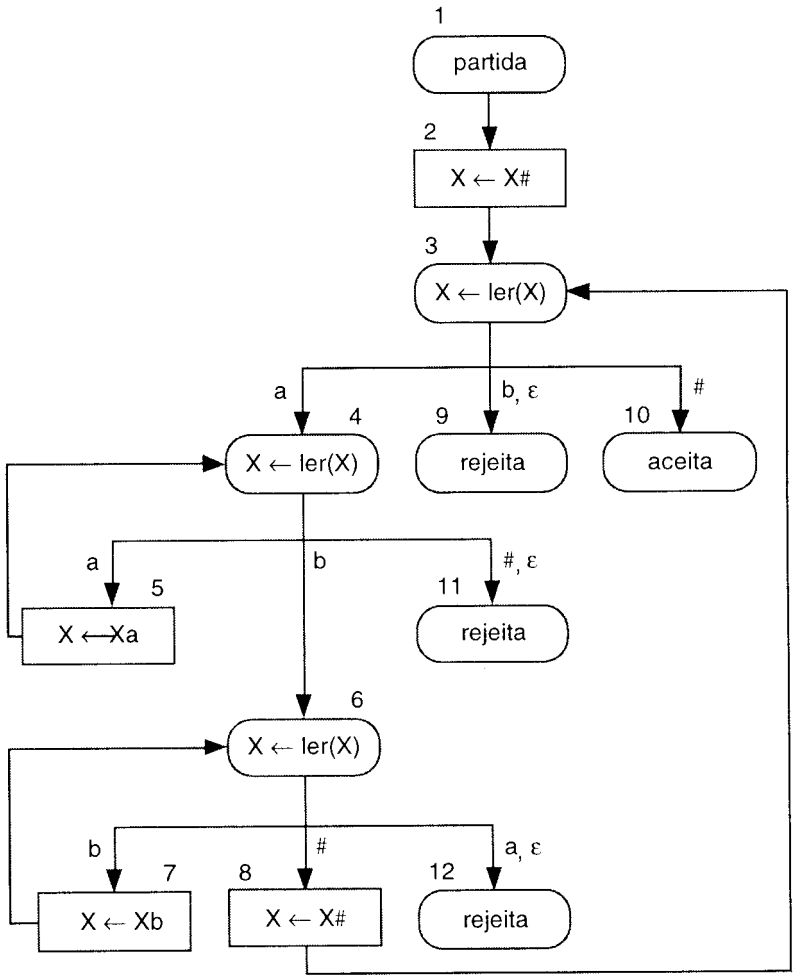


Figura 5.21 Máquina de Post com as instruções enumeradas

Uma solução para este sistema, a qual corresponde ao processamento da Máquina de Post Post-Duplo_Bal para a entrada w = ab, é a seqüência de pares:

1, 22, 23, 2, 22, 23, 24, 3, 23, 24, 8, 24, 14, 17, 24, 5, 19

o que resulta em:

1 a b 2 a b # 3 a b # 4 b # 6 # 8 # 3 # 10 =
= 1 a b 2 a b # 3 a b # 4 b # 6 # 8 # 3 # 10 ε

Repare que a informação contida entre dois números de instruções é o valor da variável X após o processamento do comando à esquerda e antes do comando à direita. Os pares 22, 23 e 24 possuem exatamente esse objetivo. ❑

5.9 Propriedades da Solucionabilidade

Inicialmente, são apresentados dois teoremas que estabelecem as seguintes propriedades:

- o complemento de uma linguagem recursiva é uma linguagem recursiva;
- uma linguagem é recursiva se, e somente se, a linguagem e seu complemento são enumeráveis recursivamente.

Como a questão da solucionabilidade pode ser traduzida na investigação se a linguagem correspondente é recursiva (problema solucionável) ou enumerável recursivamente (problema parcialmente solucionável), os resultados acima podem ser interpretados em termos de classes de problemas, como segue:

- o complemento de um problema solucionável é solucionável;
- um problema é solucionável se, e somente se, o problema e seu complemento são solucionáveis parcialmente.

Importantes resultados podem ser obtidos a partir destas propriedades como, por exemplo, sobre o Problema da Parada:

- o Problema da Parada é parcialmente solucionável;
- o Problema da Parada é não-solucionável;
- portanto, o *Problema da Não-Parada* é não-solucionável.

Uma conseqüência imediata deste resultado é a inexistência de um algoritmo genérico para identificar *loops* infinitos em sistemas.

Teorema 5.22 Complemento de uma Linguagem Recursiva é uma Linguagem Recursiva.

Se uma linguagem L sobre um alfabeto Σ qualquer é recursiva, então o seu complemento $\Sigma^* - L$ também é uma linguagem recursiva.

Prova:

Suponha L uma linguagem recursiva sobre Σ. Então existe M, Máquina Universal, que aceita a linguagem e sempre pára para qualquer entrada. Ou seja:

$ACEITA(M) = L$
$REJEITA(M) = \Sigma^* - L$
$LOOP(M) = \varnothing$

Seja M' uma Máquina Universal construída a partir de M, mas invertendo-se as condições de ACEITA por REJEITA e vice-versa (como a inversão pode ser implementada?). Portanto, M' aceita $\Sigma^* - L$ e sempre pára para qualquer entrada. Ou seja:

$ACEITA(M') = \Sigma^* - L$
$REJEITA(M') = L$
$LOOP(M') = \varnothing$

Logo $\Sigma^* - L$ é uma linguagem recursiva. ❑

Teorema 5.23 Linguagem Recursiva $\times$ Linguagem Enumerável Recursivamente.

Uma linguagem L sobre um alfabeto Σ qualquer é recursiva se, e somente se, L e $\Sigma^* - L$ são enumeráveis recursivamente.

Prova:

a) Suponha L uma linguagem recursiva sobre Σ. Então, como foi mostrado no Teorema 5.22, $\Sigma^* - L$ é recursiva. Como toda linguagem recursiva também é enumerável recursivamente, então L e $\Sigma^* - L$ são enumeráveis recursivamente;

b) Suponha L uma linguagem sobre Σ tal que L e $\Sigma^* - L$ são enumeráveis recursivamente. Então existem M_1 e M_2, Máquinas Universais tais que:

$ACEITA(M_1) = L$

$ACEITA(M_2) = \Sigma^* - L$

Seja M Máquina Universal não-determinística definida conforme esquema ilustrado na Figura 5.22 (como seria, detalhadamente, a definição de M?) Para qualquer palavra de entrada, M aceita-a se M_1 aceitá-la e M rejeita-a se M_2 aceitá-la. Portanto, claramente, M sempre pára. Logo, L é recursiva. ❑

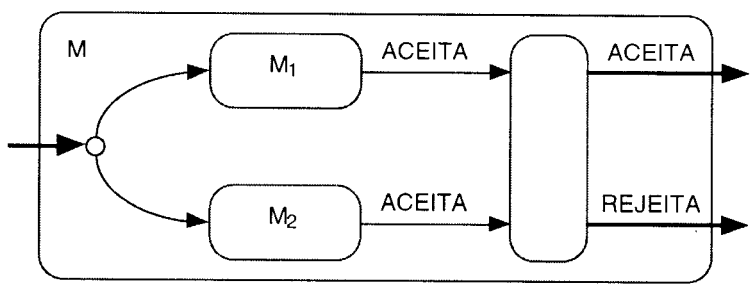


Figura 5.22 Máquina Universal não-determinística

5.10 Exercícios

Exercício 5.1 Qual a relação entre as seguintes classes de problemas:

a) Solucionáveis;

b) Parcialmente Solucionáveis (Computáveis);

c) Não-solucionável;

d) Completamente Insolúveis (Não-Computáveis).

Exercício 5.2 Descreva uma sistemática genérica para traduzir problemas em linguagens.

Exercício 5.3 Qual a importância do Problema da Auto-Aplicação no estudo da solucionabilidade de problemas?

Exercício 5.4 Quais as idéias básica do princípio da redução?

Exercício 5.5 Desenvolva um algoritmo de decodificação (função inversa da codificação).

Exercício 5.6 Esboce um programa que recebe como entrada p = código(P) e simula P para a entrada p.

Exercício 5.7 Demonstre que o *Problema da Parada da Palavra Constante* é não-solucionável.

Exercício 5.8 Escolha um dos problemas abaixo e mostre que ele é parcialmente solucionável:

a) Problema da Parada;

b) Problema da Parada da Palavra Vazia;

c) Problema da Totalidade;

d) Problema da Equivalência.

Exercício 5.9 O *Problema da Aceitação da Palavra* é definido como segue: dada uma Máquina Universal M qualquer e uma palavra w qualquer pertencente a Σ^*, M aceita w? Ou seja, investiga-se se uma palavra é aceita por uma Máquina Universal.

a) A linguagem correspondente a esse problema é conhecida como a *Linguagem Universal*. Qual é essa linguagem?

b) Prove que é um problema parcialmente solucionável;

c) Prove que é um problema não-solucionável.

Exercício 5.10 No estudo dos problemas do tipo sim/não, são válidas todas as operações lógicas como e, ou, se-então, negação, etc. Para os itens abaixo considere as operações e, ou e negação:

a) Interprete o significado dessas operações sobre problemas;

b) Qual o resultado ao aplicar essas operações sobre problemas solucionáveis? Por quê?

c) Idem para não-solucionáveis;

d) Idem para parcialmente solucionáveis.

Exercício 5.11 Suponha M uma Máquina Universal. Seja M' uma Máquina Universal construída a partir de M, mas invertendo-se as condições de ACEITA por REJEITA e vice-versa. Como essa inversão pode ser implementada? A resposta pode ser específica para qualquer dos formalismo estudados.

Exercício 5.12 Seja M uma Máquina Universal não-determinística definida conforme esquema ilustrado na Figura 5.22. Como seria a definição de M? A resposta pode ser específica para qualquer dos formalismo estudados.

Exercício 5.13 Sobre o Problemas de Correspondência de Post:

a) Encontre a menor solução para os seguintes Sistemas de Post:

$S_1 = \{(b, bbb), (babbb, ba), (ba, a)\}$

$S_2 = \{(\varepsilon, a), (a, b), (b, aa), (aa, ab), (ab, ba), (ba, bb), (bb, \varepsilon)\}$

b) Encontre uma solução para o seguinte Sistema de Post:

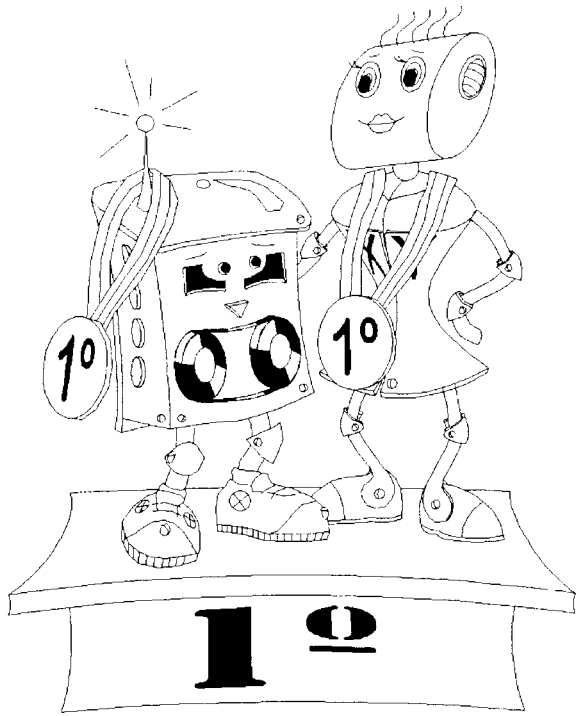
$S_3 = \{(aab, a), (ab, abb), (ab, bab), (ba, aab)\}$

Observação: a menor solução é uma seqüência de 66 índices.

c) Mostre que o seguinte Sistema de Post não tem solução:

$S_4 = \{(ba, bab), (abb, bb), (bab, abb)\}$

Exercício 5.14 Suponha S um Sistema de Post com solução. Desenvolva um algoritmo que determine a menor seqüência de índices que seja solução de S.



6 Conclusões

A abordagem adotada neste livro desenvolve os principais aspectos de Teoria da Computação combinando abordagens históricas com abordagens próximas dos sistemas computadores modernos. O objetivo dessa combinação é permitir um fácil entendimento e associação dos problemas abstratos com os problemas típicos da Ciência da Computação atual.

Este livro foi desenvolvido com o propósito de construir, de forma gradual, os diversos conceitos básicos de Teoria da Computação. O primeiro conteúdo tratado é a noção de procedimento efetivo ou de função computável. Para definir as funções computáveis, são descritos diversos formalismos e máquinas. Entre eles, destacam-se a Máquina de Turing e a Máquina Norma. O primeiro é um formalismo simples e usualmente adotado para desenvolver este tipo de estudo. O segundo, desenvolvido por Bird, possibilita a diferenciação entre programa e máquina, estando, por isso, bastante próximo da noção de computabilidade e dos computadores atuais. Conclui-se que eles, juntamente com outros formalismos

apresentados, são máquinas universais, ou seja, neles é possível representar qualquer função que seja computável. Adicionalmente, é desenvolvida a noção de função recursiva, formalismo equivalente às máquinas universais. Por fim, o livro trata da computabilidade e do estudo da solucionabilidade de problemas. Os problemas podem ser divididos em problemas solucionáveis (existe um algoritmo que resolva o problema, para qualquer entrada) e os não solucionáveis (não existe um algoritmo que sempre resolva o problema). Ou, alternativamente, podem ser divididos em problemas parcialmente computáveis (solucionáveis) e problemas completamente não-computáveis (insolúveis).

6.1 Resumo dos Principais Conceitos

O principal conceito estudado é o de computabilidade o qual é construído usando noções de programas, máquinas e computações. São três conceitos distintos mas diretamente relacionados, pois um programa para uma máquina pode induzir uma computação. Se ela for finita, então se define ainda a função computada por esse programa nessa máquina: ela descreve o que o programa faz.

A distinção entre programa e máquina é importante na Ciência da Computação, uma vez que o programa (ou algoritmo) independe da máquina e possui uma complexidade estrutural e computacional (quantidade de trabalho necessário para resolver o problema).

A partir do conceito de função computada, pode-se fazer comparações entre programas e entre máquinas e definir a equivalência dos mesmos. Se dois programas, em uma máquina, possuem a mesma função computada, ou seja, computam a mesma função, então eles são equivalentes. Se esses programas são equivalentes em qualquer máquina, então eles são equivalentes fortemente.

Utiliza-se o conceito de equivalência de programas para eliminar instruções desnecessárias e para otimizar o programa. Desta forma, pode-se dizer que um programa otimizado representa uma classe de programas que são equivalentes.

A comparação entre máquinas é feita em função da noção de simulação, que, por sua vez, utiliza-se do conceito de equivalência de programas: se uma máquina simula outra, é porque, para qualquer programa da outra máquina, pode-se encontrar um programa dessa que faça a mesma coisa. Se duas máquinas simulam-se mutuamente, é porque elas são equivalentes. Neste caso, ambas têm o mesmo poder computacional. Foi observado que nem sempre o fato de acrescentar instruções ou recursos a uma máquina aumenta o poder computacional da mesma. Muitas vezes, ocorre que essa nova instrução ou esse

novo recurso podem ser simulados pela máquina anterior. Portanto o seu poder computacional continua o mesmo.

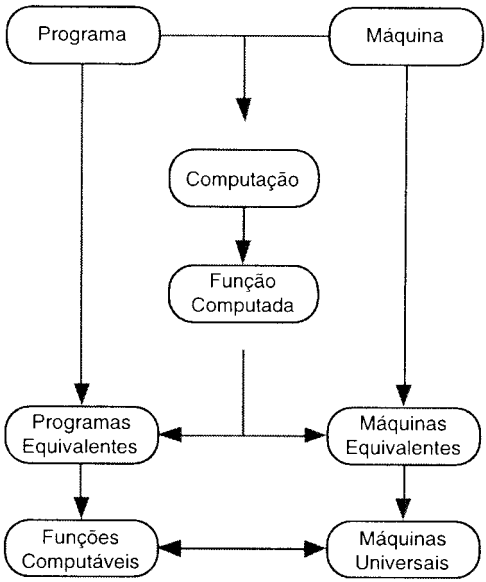


Figura 6.1 Conceitos básicos desenvolvidos

Neste ponto, pode-se pensar em suas situações limites:

- Qual a máquina mais poderosa?
- Qual o conjunto ou a classe de funções computáveis?

As respostas dessas perguntas são intuitivas, mas nem sempre são fáceis de serem verificadas. Diz-se que uma máquina é universal se toda função computada puder ser executada nela. E, segunda a Hipótese de Church, diz-se que uma função computada é aquela que pode ser processada numa Máquina de Turing ou equivalente. Assim, não se consegue demonstrar, tais afirmações devido à noção de algoritmo ser algo intuitivo. Entretanto, diversas evidências (muitas delas exploradas ao longo deste livro) fortalecem a Hipótese de Church.

Por outro lado, pode-se tratar a questão da solucionabilidade pelo ângulo dos problemas não-solucionáveis. Neste estudo, o princípio da redução é uma das principais ferramentas. A idéia básica é investigar a solucionabilidade (respectivamente, não-solucionabilidade) de um problema a partir de outro, cuja solucionabilidade (respectivamente, não-solucionabilidade) é conhecida.

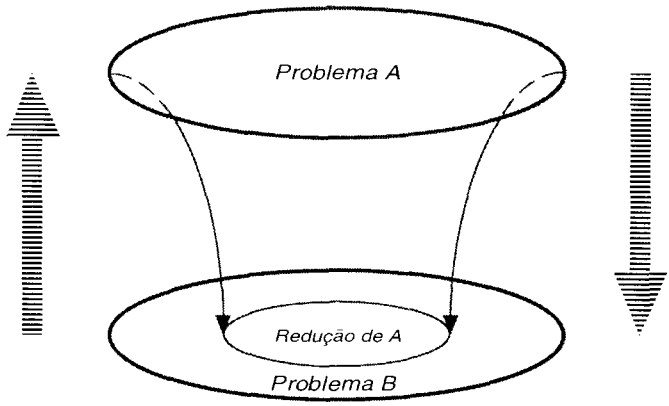


Figura 6.2 Princípio da Redução

6.2 Contribuições da Teoria da Computação

Teoria da Computação é básica e de fundamental importância para a Ciência da Computação. Ela não só proporciona um adequado embasamento teórico necessário para um correto e amplo entendimento da ciência envolvida na computação como, também, propicia o desenvolvimento de um raciocínio lógico e formal, cada vez mais necessário em todas as subáreas da computação. Além disso, introduz os conceitos fundamentais que são desenvolvidos em outras áreas como, por exemplo:

- a abordagem de reconhecimento de linguagens é a base de todo o estudo das Linguagens Formais, Semântica Formal, Compiladores e de todo o conjunto de disciplinas que tratam de Linguagens de Programação;
- a formalização de objetos através de suas características e propriedades possibilita que se identifiquem classes, as quais podem agrupar esses objetos. Os objetos herdam as propriedades da classe, sendo, portanto, instâncias dessa classe. Esses são alguns dos tópicos de Orientação a Objetos;
- a abordagem do processamento de funções lança a base para o desenvolvimento de algoritmos eficientes, não só na resolução dos problemas, mas também no estudo da otimização de algoritmos (programas);
- a complexidade estrutural diz respeito à estrutura de controle adotada e à otimização em termos do número de instruções e da eliminação de instruções desnecessárias. A complexidade computacional diz respeito à quantidade de trabalho envolvida na resolução do problema pelo algoritmo

(tempo), medida, muitas vezes, pela quantidade de trabalho que uma determinada instância do problema necessita para resolvê-lo. Também se pode considerar a quantidade de memória necessária (espaço) e, no caso de processamento paralelo e distribuído, o número de processadores necessários. Portanto, são temas abordados em Análise e Desenvolvimento de Algoritmos e Complexidade de Algoritmos Seqüenciais e Paralelos;

- o não determinismo é um conceito introduzido que tem grande impacto na formação dos bacharéis em Ciência da Computação, em Informática e em Engenharia da Computação pois é o primeiro tipo de composição não-seqüencial estudado. Inclusive, permite desenvolver o conceito de concorrência do tipo intercalação, fundamental para um correto entendimento da concorrência verdadeira (do tipo não-intercalação).

No campo cognitivo, Teoria da Computação proporciona mais um estágio na formação do raciocínio lógico, com destaque ao pensamento indutivo ou recursivo. Pelo aspecto da indução, parte-se de uma instância base, verifica-se a passagem para instâncias superiores, até construir o problema como um todo. Pelo aspecto da recursão, vê-se o problema como um todo e reduz-se o problema a subproblemas menores, até que se chega a um problema base, iniciando assim, o processo reverso, com a resolução de cada instância do problema.

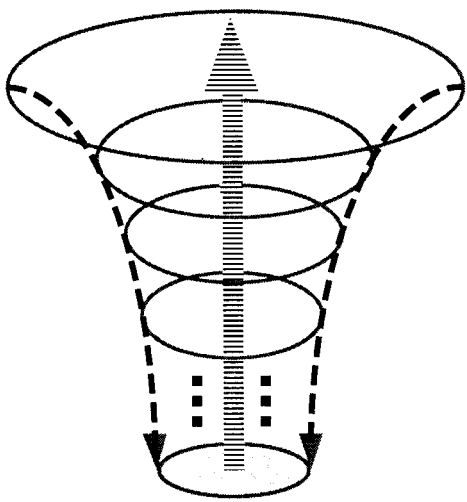


Figura 6.3 Princípio da Recursão

Outro importante ganho no campo cognitivo está no desenvolvimento de demonstrações e de suas técnicas. E aqui, o objetivo não é somente capacitar o estudante a desenvolver provas formais mas, também, informalmente. Assim, um raciocínio seguindo algumas regras simples e universalmente aplicáveis pode constituir uma prova precisa, mas que pode ser adequadamente seguida em uma argumentação informal.

Também é importante destacar o desenvolvimento da capacidade de abstração. Assim, propriedades abstratas podem ser especificadas e estudadas independentes de estruturas. Ou seja, permite tratar propriedades independentemente de implementação, o que constitui uma vantagem obvia para a Ciência da Computação. Um exemplo típico dessa capacitação e abordado neste livro é o uso do princípio da redução de problemas.

7 Bibliografia

[AHO92] AHO, A.; ULLMAN, J. **Foundations of Computer Science.** New York: Computer Science Press, 1992.

[ARB81] ARBIB, M.; KFOURI, A.; MOLL, R. **A Basis for Theoretical Computer Science**. New York: Springer Verlag, 1981.

[BIR76] BIRD, R. **Programs and Machines - an introduction to the theory of computation.** London: John-Wiley, 1976.

[BRA74] BRAINERD, W. S.; LANDWEBER, L. H. **Theory of Computation.** New York: Wiley, 1974.

[BRO93] BROOKSHEAR, J. G. **Teoría de la Computación** - Lenguajes formales, autómatas y complejidad. Wilmington: Addison-Weslwey Iberoamericana, 1993. 338p.

[COH97] COHEN, D. I. A. **Introduction to Computer Theory.** New York: John Wiley & Sons, 1997.634p.

[CHU36] CHURCH, A. A Unsolvable Problem in Elementary Number Theory. American Lournal of Mathematics. N.58, 1936.

[CLA90] CLARK, K.; COWELL, D. **Programs, machines and computation; an introduction to theory of computing.** London: McGraw-Hill, 1976.

[CLA97] CLARKE, A. C. 3001: A odisséia final. Rio de Janeiro: Novas Fronteiras, 1997. 297p. (3 Edição).

[DED1888] DEDEKIND, R. *Was sind und was sollen die Zahlen?* Braunschueig, 1888.

[ELL94] ELLIS, T. M. R.;PHILIPS, I.R; LAHEY, T.M. Fortran 90 - programming. Wokingham: Addison-Wesley, 1994. 825p.

[EVE63] EVEY, R. J. **The Theory and Applications of Pushdown Store Machines:** mathematical linguistics in machine translation. Cambridge: Harvard Computation Laboratory, 1963. Report NSF-10.

[FER84] FERREIRA, AURÉLIO B. H. Novo Dicionário da Língua Portuguesa. São Paulo: Editora Nova Fronteira, 1984.

[GOD65] GÖDEL, K. On formally undecidable propositions in Principia Mathematica and related sistems. In: The undecidable, M. Davis. Raven Press, 1965, p.4-38.

[HIL1900] HILBERT, D. Mathemayical Problems. Lecture delivered before the international Congress of Mathematicians at Paris in 1900. In: Mathematical Developments Arising from Hilbert Problems.American Mathematical Society. 1976, p.1-34.

[HOP79] HOPCROFT, J.; ULLMAN, J. **Introduction to Automata Theory, Languages and Computation**. Addison-Wesley, 1979.

[JEN74] JENSEN, K.; WIRTH, N. Pascal user manual and report. Berlin, Springer-Werlag, 1974.

[KLE56] KLEENE, S. C. Representation of Events in Nerve Nets and Finite Automata. In: Shannon, C. E., and McCarty, J. Automata Studies. Princeton Univ. Press, Princeton, 1956, p.3-42.

[KNU69] KNUTH, DONALD.The art of computer programming. v.1. Reading: Addison and Wesley, 1969.

[LUC79] LUCCHESI, C. et al. **Aspectos Teóricos da Computação.** Rio de Janeiro: Instituto de Matemática Pura e Aplicada, 1979. 292p. (projeto Euclides).

[MAL69] MALCOME-LAWES, D. J. **Programming Algol.** Oxford:Pergamon Press, 1969, 110p.

[MAN74] MANNA, Z. **Mathematical Theory of Computation.** New York: McGraw-Hill, 1974.

[MEN90] MENEZES, P. F. B.**Teoria da computação.** Porto Alegre: UFRGS, 1990.

[MEN98] MENEZES, P. F. B. **Linguagens Formais e Autômatos**. Porto Alegre: Sagra-Luzzatto, 1998. 165p. (*Série Livros Didáticos do Instituto de Informatica da UFRGS*, N.3.)

[MEN98a] MENEZES, P. F. B.; SERNADAS, A.; COSTA, J. Nonsequential automata semantics for a conccurent object-based language. In: US-Brazil Workshop on Formal Foundations Software Systems, 1, Rio de Janeiro, 1998. **Proceedings ...** Electronic Notes in Theoretical Computer Sciense, n.14, 1998. 29p. URL: http://www.elsevier.nl/locate/entcs/volume14.html

[MIN67] MINSKY, M. L. **Computation: finite and infinite machines**. Englewood Cliffs: Prentice Hall, 1967.

[OET61] OETTINGER, A. G. Automatic Syntactic Analysis and the Pushdown Store. In: SYMPOSIA IN APPLIED MATHEMATICS. v.12. **proceedings ...** Providence: American Mathematical Society, 1961. p.104-129.

[POS36] POST, E. Finite Combinatory Process: formulation I. **Journal of Symbolic Logic.** 1936. p.103-105.

[PLU83] PLUM, T. **Learning to program in C.** Englewood Clifs: Patience-Hall, v.1, 1983.

[SAG86] SAGASTUME, M; BAUM, G. **Problemas, Lenguajes y Algoritmos.** Campinas: EBAI/Editora da UNICAMP, 1986. 165p.

[SCH67] SCHÜTZEMBERGER, M. P. On context-free languages and pushdown automata. **Information and control,** v.6, p.246-264. 1967.

[SER93] SERNADAS, C. **Introdução à Teoria da Computação**. Lisboa: Editorial Presença, 1993.

[SIP97] SIPSER, M. **Introduction to the Theory of Computation**. Boston: PWS Publishing, 1997. 396p.

[SHO93] SHOENFIELD, J. R. Recursion Theory. **Lecture Notes in Logic.** v.1, Berlin: Spring-Verlag, 1993. 84p.

[TUR36] TURING, A. M. On computable numbers, with na aplication to the Entscheidungsproblem. **Proceedings**... London: Maythematical Society, 1936, p.230-265.

[UNI80] UNITED STATES DEPARTMENT OF DEFENSE. **The pregramming Language ADA.** Berlin: Springer-Verlag, 1980, 243p.

[WEG68] WEGNER, P. Programming languages, Information structures, and Machine organization. New York, McGraw-Hill, 1968, 401p.

[WIN95] WINSKEL, G.; NIELSEN, M. **Models for Currency, Handbook of Logic and Computer Science.** Oxford: Oxford University Press, v.4, 1995.

Índice Remissivo

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V