

5

■ ■ série livros didáticos informática ufrgs ■ ■

teoria da computação:
máquinas universais
e computabilidade

■ ■ 3ª edição

■ ■ tiarajú asmuz diverio
■ ■ paulo blauth menezes



→ os autores

Tiarajú Asmuz Diverio é doutor e mestre em ciência da computação pelo PGCC da UFRGS, e licenciado em matemática pela UFRGS. É professor titular do Departamento de Informática Teórica do Instituto de Informática da UFRGS, tendo mais de 35 anos de experiência no meio acadêmico; é o professor responsável pela disciplina de Teoria da Computação do Instituto de Informática nos últimos 20 anos.

Paulo Blauth Menezes é doutor em matemática pelo IST/Universidade Técnica de Lisboa, Portugal, mestre em ciência da computação e licenciado em matemática pela UFRGS. É professor titular do Departamento de Informática Teórica do Instituto de Informática da UFRGS, tendo mais de 35 anos de experiência no meio acadêmico.



D618t Diverio, Tiarajú Asmuz.
Teoria da computação [recurso eletrônico] : máquinas universais e computabilidade / Tiarajú Asmuz Diverio, Paulo Blauth Menezes. – 3. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2011.

(Livros Didáticos Informática UFRGS ; 5).

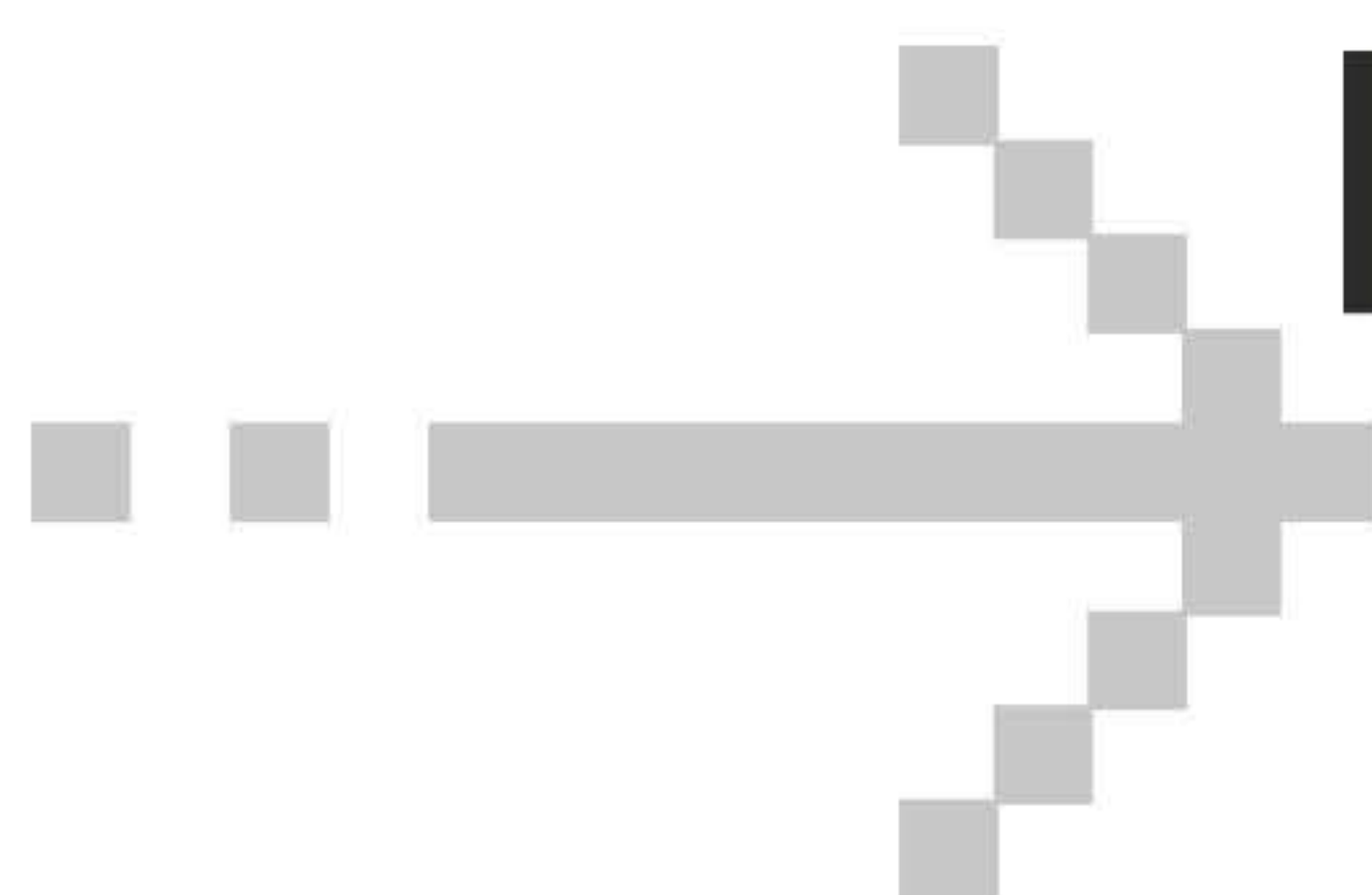
Editado também como livro impresso em 2011.
ISBN 978-85-7780-831-1

1. Ciência da computação – Teoria da computação.
I. Menezes, Paulo Blauth. II. Título.

CDU 004

tiarajú asmuz diverio

paulo blauth menezes



**teoria da computação:
máquinas universais
e computabilidade**

■ ■ 3ª edição

Versão impressa
desta obra 2011



2011

© Artmed Editora SA, 2011

Capa e projeto gráfico interno: *Tatiana Sperhacke*

Imagem da capa: ©*iStockphoto.com/enot-poloskun*

Leitura final: *Renata Ramisch*

Editora sênior: *Arysinha Jacques Affonso*

Editora responsável por esta obra: *Júlia Angst Coelho*

Editoração eletrônica: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
ARTMED® EDITORA S.A.
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S. A.)
Av. Jerônimo de Ornelas, 670 - Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

SÃO PAULO
Av. Embaixador Macedo Soares, 10.735 - Pavilhão 5 - Cond. Espace Center
Vila Anastácio 05035-000 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Este livro é dedicado a:

Ana Elisa, Amanda, Rafael e Bidu – minha nova família
Tiarajú

Maria Fernanda, Maria Lúcia e Maria Luíza
Paulo

Esta página foi deixada em branco intencionalmente.



apresentação

A série *Livros Didáticos* do Instituto de Informática da Universidade Federal do Rio Grande do Sul tem como objetivo a publicação de material didático para disciplinas ministradas em cursos de graduação em computação e informática, ou seja, para os cursos de bacharelado em ciência da computação, de bacharelado em sistemas de informação, de engenharia da computação e de licenciatura em informática. A série é desenvolvida tendo em vista as Diretrizes Curriculares do MEC e é resultante da experiência dos professores do Instituto de Informática e dos colaboradores externos no ensino e na pesquisa.

Os primeiros títulos, *Fundamentos da matemática intervalar* e *Programando em Pascal XSC*, foram publicados em 1997 no âmbito do Projeto Aritmética Intervalar Paralela (ArInPar), financiados pelo ProTeM - CC CNPq/Fase II. Essas primeiras experiências serviram de base para os volumes subsequentes, os quais se caracterizam como livros-texto para disciplinas dos cursos de computação e informática.

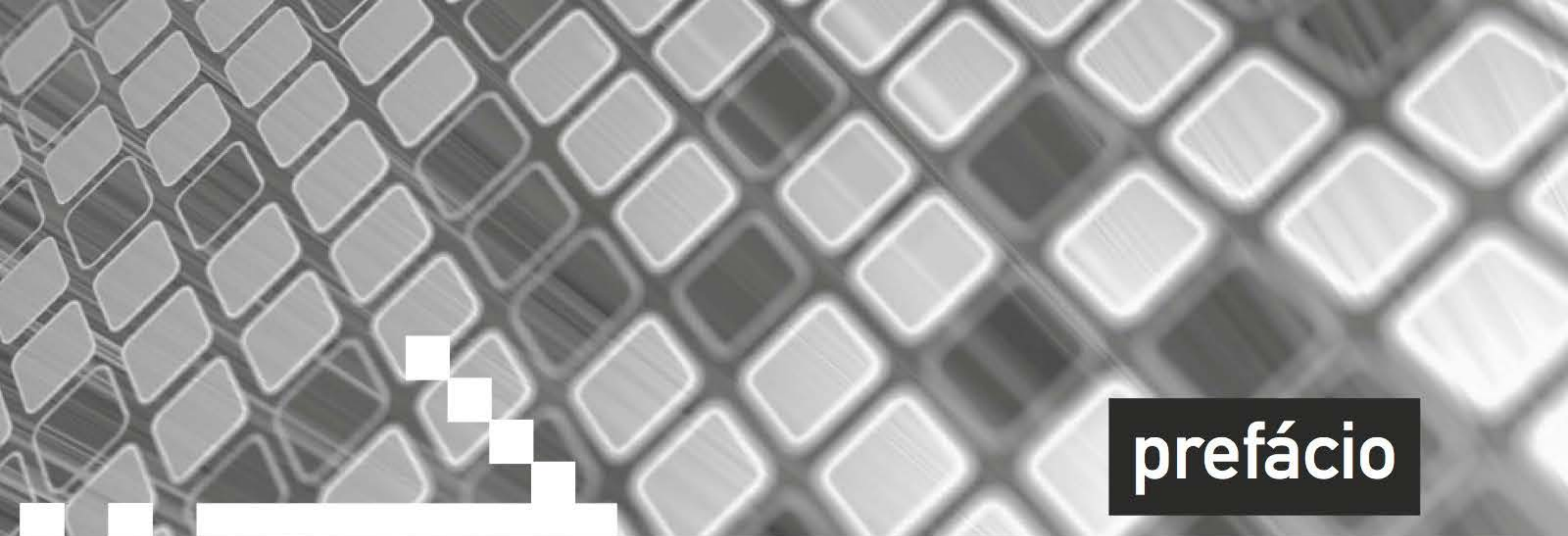
Em seus títulos mais recentes, a série *Livros didáticos* tem contado com a colaboração de professores externos que, em parceria com professores do Instituto, estão desenvolvendo livros de alta qualidade e valor didático. Hoje a série está aberta a qualquer autor de reconhecida capacidade.

O sucesso da experiência com esses livros, aliado à responsabilidade que cabe ao Instituto na formação de professores e pesquisadores em computação e informática, conduziu à ampliação da abrangência e à institucionalização da série.

Em 2008, um importante passo foi dado para a consolidação e ampliação de todo o trabalho: a publicação dos livros pelo Grupo A, por meio da editora Bookman. Hoje são 20 os títulos publicados – uma lista completa encontra-se nas orelhas desta obra – ampliando a oferta aos leitores da série. Sempre com a preocupação em manter o nível compatível com a elevada qualidade do ensino e da pesquisa desenvolvidos no âmbito do Instituto de Informática da UFRGS e no Brasil.

*Prof. Paulo Blauth Menezes
Comissão Editorial da Série Livros Didáticos
Instituto de Informática da UFRGS*

Esta página foi deixada em branco intencionalmente.



prefácio

Teoria da Computação: Máquinas Universais e Computabilidade pretende servir como um livro-texto para disciplinas dos cursos de bacharelado em ciência da computação e em sistemas de informação, engenharia da computação e licenciatura em computação, de acordo com a Diretrizes Curriculares do MEC. O enfoque adotado neste livro não é histórico (cronológico), mas didático, visando à construção dos conceitos de teoria da computação e ao desenvolvimento do raciocínio abstrato.

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 nos cursos de bacharelado em ciência da computação e de engenharia da computação da Universidade Federal do Rio Grande do Sul. O livro é destinado, principalmente, para um primeiro curso de teoria da computação, é autocontido e pode ser adotado como livro-texto. Ele possui também um texto simples, exemplos detalhados e exercícios em níveis crescentes de raciocínio.

A carga horária total recomendada é de 60 horas/aula e é suposto, como pré-requisito, matemática discreta. Em particular, a matéria matemática discreta é tratada em publicações específicas desta mesma série:

Matemática Discreta para Computação e Informática, v. 16 – Paulo Blauth Menezes

Aprendendo Matemática Discreta com Exercícios, v. 19 – Paulo Blauth Menezes, Laura Vieira Toscani & Javier García López

Baseado na experiência de 10 anos de utilização deste livro, nesta terceira edição, o texto foi completamente reestruturado em capítulos menores e com objetivos mais específicos. Foram incluídos novos exemplos e alguns detalhamentos visando a complementar e aprofundar conceitos fundamentais de teoria da computação. E, ainda, os exercícios foram complementados com questões objetivas desenvolvidas e testadas.

Comparando esta edição com a anterior, os seis capítulos foram redistribuídos em 10 capítulos, como segue:

- o capítulo inicial permaneceu com a mesma estrutura;
- o segundo capítulo foi desmembrado em dois capítulos, sendo que um aborda os conceitos fundamentais e outro apresenta o algoritmo de verificação de equivalência forte de programas monolíticos;
- o terceiro capítulo foi reestruturado e subdividido em quatro capítulos, um para a máquina Norma, um para a máquina de Turing, um para máquinas universais e hipótese de Church e outro que apresenta os demais modelos de máquinas universais estudados;
- no quarto capítulo, das funções recursivas, optou-se por uma reformulação conceitual, na qual se aprofundou nos formalismos de Kleene e de Church (cálculo Lambda) em detrimento de outros formalismos;
- no quinto capítulo, houve uma reformulação da forma como os conteúdos são explorados, resultando em um texto mais direto e objetivo, deixando mais evidente a técnica de investigação da solucionabilidade de um problema;
- no capítulo final, as conclusões proporcionam uma reflexão abrangente sobre abstração, formalização e demonstração da computabilidade e dos limites do que pode ser efetivamente resolvido em um computador.

Tal reestruturação, somada à nova diagramação, propiciou um texto de melhor qualidade de leitura e visualização, facilitando o aprendizado e a consulta.

A seguir é apresentado um resumo de cada capítulo desta nova edição:

Capítulo 1 – Introdução. Inicia com uma breve história do surgimento e do desenvolvimento dos conceitos, formalismos e resultados nos quais a teoria da computação é baseada. Em seguida, é apresentada a abordagem geral adotada nesta publicação. Por fim, são introduzidos alguns conceitos básicos fundamentais que são usados ao longo de todo o texto.

Capítulo 2 – Programas, máquinas e computações. Introduz a formalização das noções de programa e de máquina, os quais são usados para construir as definições de computação e função computada. Em particular, são estudados três tipos de programas: monolítico, iterativo e recursivo. Define-se o que é computável em uma máquina e, baseadas em função computada, são apresentadas noções de simulação e de equivalência de programas: programas fortemente equivalentes, programas equivalentes (em uma máquina).

Capítulo 3 – Verificação da equivalência forte de programas. São estudadas formas de se determinar se dois programas, sob determinadas condições, são ou não fortemente equivalentes. Mostra-se a existência de um algoritmo para decidir a equivalência forte entre programas monolíticos. E, como todo o programa iterativo possui um programa monolítico fortemente equivalente, o mesmo pode ser afirmado para programas iterativos. Porém, até o momento, não é conhecido se o problema da equivalência forte de programas recursivos é solucionável. O algoritmo apresentado foi inicialmente desenvolvido por R. Bird.

Capítulo 4 – Máquina de registradores – Norma. São introduzidas as máquinas de registradores e, em especial, a máquina Norma, que possui três tipos de instruções (**ad**, **sub** e **zero**) sobre registradores naturais. São apresentadas também evidências internas de que Norma é uma máquina universal. É especialmente interessante observar como uma máquina tão simples e rudimentar pode ser tão poderosa.

Capítulo 5 – Máquina de Turing. Estuda a máquina de Turing, um dos modelos mais utilizado como formalização de algoritmo. Trata-se de um modelo que formaliza a ideia de uma pessoa que realiza cálculos. Duas das abordagens da máquina de Turing são apresentadas: reconhecimento de linguagens e processamento de funções. Também são apresentadas algumas evidências de que se trata de uma máquina universal.

Capítulo 6 – Máquinas universais e hipótese de Church. Caracteriza o conceito da máquina que simula todas as outras máquinas e que permite que todo problema computável seja nela implementado. É apresentada a tese de Church, a qual supõe que o limite máximo do que é computável pode ser expresso como um algoritmo implementado em uma máquina de Turing. Como a noção de algoritmo é intuitiva, a tese não é demonstrável, sendo aceita (suposta) como verdadeira. Corroboraram para isso algumas evidências externas e internas, como a verificação de que as máquinas de Turing e Norma são equivalentes e a constatação de que as modificações sobre a máquina de Turing apresentadas nesse capítulo não ampliam a classe das funções computadas. Como ilustração, é mostrada a equivalência entre os formalismos máquina de Turing e máquina Norma, por meio da simulação recíproca de programas da máquina de Turing na máquina Norma e de programas da máquina Norma na máquina de Turing.

Capítulo 7 – Outros modelos de máquinas universais. Apresenta os formalismos da máquina de Post, da máquina com pilhas e do autômato com pilhas. O primeiro é baseado em uma estrutura de dados tipo fila, e os demais constituem uma família de máquinas com poder ou capacidade computacional distintos. A característica de duas pilhas é suficiente para atingir um poder computacional equivalente da máquina de Turing. É apresentado, ainda, o formalismo de máquinas não determinísticas, que produzem resultados interessantes comparando-as com suas versões determinísticas.

Capítulo 8 – Funções recursivas. Apresenta o estudo de dois formalismos denotacionais (funcionais) do tipo recursivo (permite a definição de uma função em termos de si mesma) denominados: funções recursivas de Kleene e cálculo Lambda. O desenvolvimento de tais formalismos tem como objetivo formalizar a noção intuitiva de função computável. O fato de que as classes das funções Turing-computáveis, as funções recursivas de Kleene e as funções Lambda-computáveis são iguais reforça a tese de Church em termos de credibilidade, fortalecendo-a como hipótese para toda a ciência da computação clássica.

Capítulo 9 – Computabilidade. Apresenta técnicas de investigação da solucionabilidade de problemas, iniciando por um apanhado geral histórico da questão, mostrando dificuldades e resultados. Passa-se para a formalização da questão, que inicia com a definição das classes

de solucionabilidade, de problemas de decisão e de uma codificação bijetora de programas nos naturais. Como primeiro problema estudado, se aborda o problema da autoaplicação, que é utilizado em uma máquina de redução para demonstrar o problema da parada. Usando as propriedades da solucionabilidade, mostra-se que o problema de detecção universal de *loops* (problema da não parada – complemento do problema da parada) é não computável. Ilustra-se, ainda, pelo problema da correspondência de Post, a técnica de investigação da solucionabilidade.

Capítulo 10 – Conclusões. Apresenta um resumo dos conceitos fundamentais desenvolvidos ao longo do livro, visando a uma reflexão abrangente sobre abstração, formalização e investigação de solucionabilidade e dos limites do que pode ser efetivamente resolvido em um computador, do que é computável.

Observa-se que algumas questões correlatas ao estudo de teoria da computação muitas vezes são desenvolvidas no escopo da teoria da computação como:

- Para um dado problema ou algoritmo, quais os recursos necessários (esforço exigido) para computá-lo? Eventualmente, tais recursos podem ser muito grandes, tornando a tarefa impraticável?
- No universo dos problemas com solução, podem-se estabelecer classes ou hierarquias de problemas nas quais certas propriedades são verificadas sempre, como por exemplo a existência de um algoritmo que possa ser aplicado a todos os problemas da classe; operações fechadas sobre problemas/algoritmos da classe (compondo problemas/algoritmos potencialmente mais complexos a partir de problemas mais simples)?

Entretanto, com o objetivo de manter o foco deste livro, preferiu-se não diversificar nem abrir muito o tema. Estas perguntas são tratadas em publicações específicas desta série, as quais complementam este livro, respectivamente:

Complexidade de Algoritmos, v. 13 – Laira Vieira Toscani & Paulo A. S. Veloso
Linguagens Formais e Autômatos, v. 3 – Paulo Blauth Menezes

Em particular, a notação e os formalismos adotados são consistentes com os dos livros *Linguagens Formais e Autômatos* e *Matemática Discreta para Computação e Informática*.

Os autores agradecem:

- aos colegas do Departamento de Informática Teórica do Instituto de Informática da UFRGS, onde este livro foi elaborado e, em especial, aos professores Laira Vieira Toscani, Leila Ribeiro e Vanderlei M. Rodrigues (*In Memoriam*);
- aos bolsistas Caroline Cintra, Thiago Moesch, Ingrid Mito e Aline Malanovicz pelo trabalho de edição e correção do texto (primeira edição);
- à Maria Lúcia Menezes pelas figuras e ilustrações;
- ao CNPq, à CAPES e à FAPERGS pelos auxílios financeiros aos projetos de pesquisa que viabilizaram a realização deste e de outros trabalhos;
- aos nossos amigos e colegas;
- ao Instituto de Informática da UFRGS, que tem apoiado a série *Livros Didáticos*.

Em especial, para esta terceira edição, os autores agradecem às pessoas a seguir, as quais devem se considerar nominalmente citadas:

- a todos os colegas professores que têm adotado nosso livro em suas aulas e que nos remeteram sugestões e contribuições diversas. Espera-se ter suprido as expectativas;
- aos nossos alunos e ex-alunos, que sempre foram críticos e ajudaram no aperfeiçoamento do texto e dos exercícios.

Por fim, um agradecimento muito especial a nossa querida mestra, colega e parceira, a professora Laira Vieira Toscani. Espera-se que essa parceria dê novos frutos, como novos livros.

Tiarajú Asmuz Diverio
Paulo Blauth Menezes
2011

Esta página foi deixada em branco intencionalmente.

sumário

1 → introdução 21

1.1	notas históricas.....	22
1.2	abordagem	25
1.3	conceitos básicos	25
1.4	conclusões.....	29
1.5	exercícios.....	30

2 → programas, máquinas e computações 33

2.1	programas.....	35
2.1.1	monolítico	37
2.1.2	programa iterativo	40
2.1.3	programa recursivo	43
2.2	máquinas	45
2.3	computações e funções computadas	48
2.3.1	computação.....	48
2.3.2	função computada.....	54

2.4	equivalências de programas e máquinas	57
2.4.1	equivalência forte de programas.....	58
2.4.2	equivalência de programas	65
2.4.3	equivalência de máquinas	66
2.5	conclusões.....	67
2.6	exercícios.....	68

3 → verificação da equivalência forte de programas**79**

3.1	máquina de traços	81
3.2	instruções rotuladas compostas	84
3.3	equivalência forte de programas monolíticos.....	88
3.4	conclusões.....	96
3.5	exercícios.....	96

4 → máquinas de registradores – Norma**103**

4.1	codificação de conjuntos estruturados	105
4.2	definição da máquina Norma	108
4.3	máquina Norma como máquina universal	110
4.3.1	operações e testes.....	110
4.3.2	valores numéricos	115
4.3.3	dados estruturados	117
4.3.4	endereçamento indireto e recursão.....	120
4.3.5	cadeias de caracteres	121
4.4	conclusões.....	121
4.5	exercícios.....	122

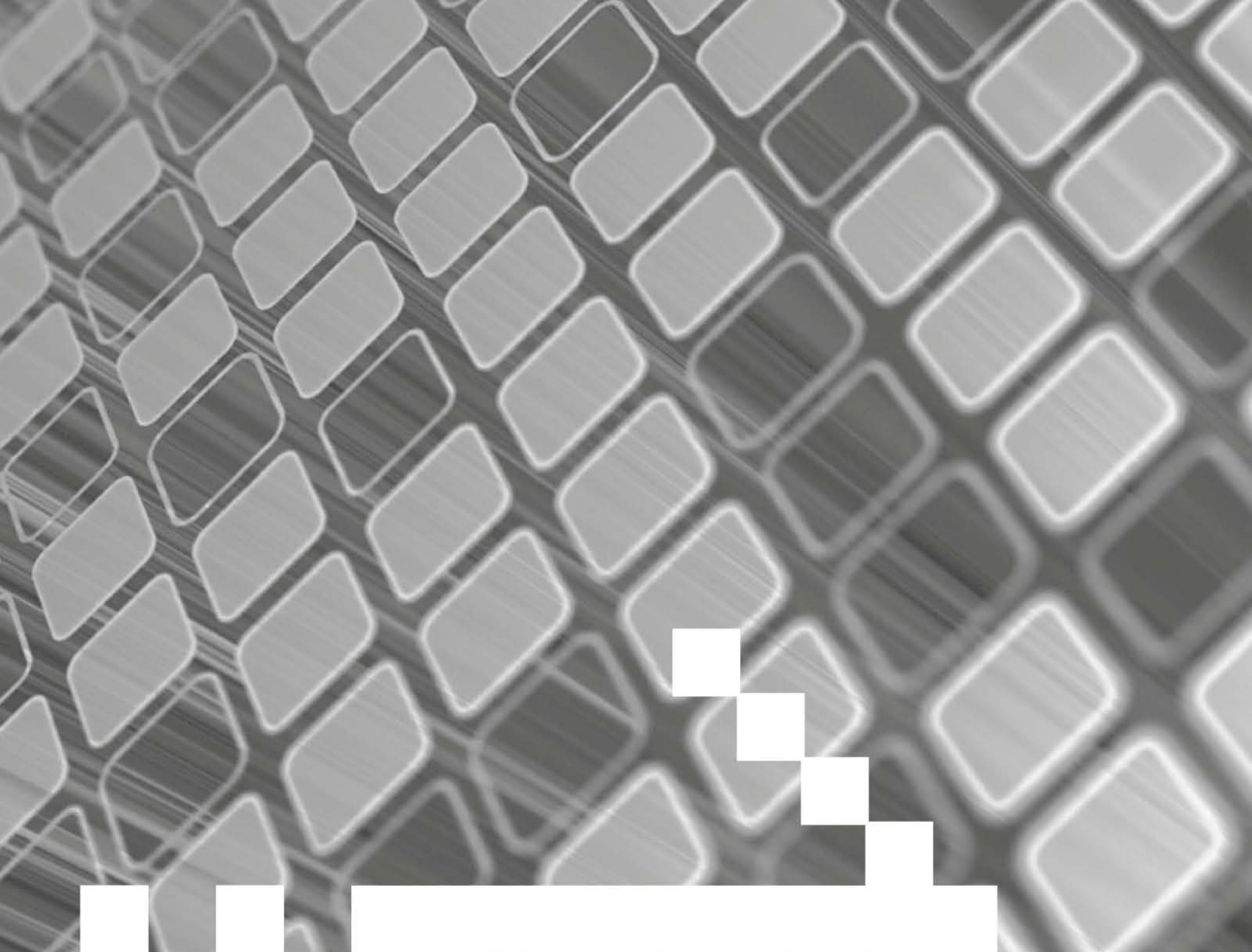
5	→ máquina de Turing	131
5.1	noção intuitiva	133
5.2	noção como máquina	133
5.3	modelo formal	134
5.4	máquinas de Turing como reconhecedores de linguagens	137
5.5	máquinas de Turing como processadores de funções	143
5.6	conclusões.....	149
5.7	exercícios.....	150
6	→ máquinas universais e hipótese de Church	157
6.1	equivalência entre as máquinas de Turing e Norma.....	159
6.2	modificações sobre a máquina de Turing	172
6.2.1	máquina de Turing não determinística	172
6.2.2	máquina de Turing com fita infinita à esquerda e à direita.....	173
6.2.3	máquina de Turing com múltiplas fitas	173
6.2.4	outras modificações sobre a máquina de Turing	174
6.3	hipótese de Church	175
6.4	conclusões.....	175
6.5	exercícios.....	176
7	→ outros modelos de máquinas universais	181
7.1	máquina de Post	183
7.2	máquina com pilhas.....	189
7.3	autômato com pilhas.....	196

7.4	máquinas não determinísticas.....	201
7.5	hierarquia de classes de máquinas	202
7.6	conclusões.....	205
7.7	exercícios.....	205

 → **funções recursivas** **213**

8.1	funções recursivas de Kleene	215
8.1.1	substituição composicional	216
8.1.2	recursão primitiva	217
8.1.3	minimização	218
8.1.4	função recursiva parcial e total	221
8.2	cálculo Lambda	223
8.2.1	aspectos gerais do cálculo Lambda	223
8.2.2	linguagem Lambda	224
8.2.3	variável livre e substituição	226
8.2.4	cálculo Lambda	229
8.2.5	tipos de dados básicos	231
8.2.6	recursão e ponto fixo	234
8.2.7	cálculo Lambda e computabilidade.....	235
8.3	funções recursivas e ciência da computação.....	237
8.3.1	importância das funções recursivas.....	237
8.3.2	linguagem de programação funcional	238
8.4	conclusões.....	240
8.5	exercícios.....	240

9 →	computabilidade	245
9.1	classes de solucionabilidade de problemas.....	250
9.2	problemas de decisão	251
9.3	codificação de programas	252
9.4	problema da autoaplicação	253
9.5	máquina de redução	256
9.6	problema da parada	258
9.7	outros problemas de decisão.....	259
9.8	problema da correspondência de Post	261
9.9	propriedades da solucionabilidade	266
9.10	conclusões.....	269
9.11	exercícios.....	269
10 →	conclusões	275
10.1	resumo dos principais conceitos	277
10.2	contribuições da teoria da computação	279
→	leituras sugeridas	281
→	índice	283



capítulo

1

introdução

- ■ A ciência da computação é o conhecimento sistematizado da computação. Sua origem é milenar, tendo se desenvolvido em diferentes regiões e épocas. A teoria da computação é a base fundamental da ciência da computação. Desenvolvida independentemente do estudo do computador (*hardware* e *software*) como se conhece hoje, objetiva responder algumas perguntas:
 - O que é uma solução computável?*
 - Quais são os limites do que pode ser computado?*
 - Existem problemas sem solução computacional?*Este capítulo começa com um breve histórico do surgimento e desenvolvimento da teoria da computação, introduz alguns dos principais conceitos e resultados sobre os quais todo o estudo é construído e apresenta a abordagem adotada nesta publicação.



Este capítulo inicia com uma breve história do surgimento e do desenvolvimento dos conceitos, formalismos e resultados 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**

A ciência da computação é o conhecimento sistematizado da computação. Sua origem é milenar, tendo se desenvolvido em diversas regiões e momentos ao longo da história da humanidade, com destaque para culturas como Mesopotâmia, Egito, Grécia, Babilônia, Índia, China, Roma e Império Asteca. O interesse atual possui duas ênfases: fundamentos e modelos

computacionais (ênfase teórica) e projeto de sistemas computacionais (ênfase prática), aplicando a teoria à prática. Assim, a ênfase teórica independe de instrumentos ou máquinas de computação e suas tecnologias e, portanto, independe de computadores (*hardware* e *software*) como são conhecidos hoje. Ou seja, computador é apenas uma instância tecnológica de alguns conhecimentos da ênfase prática da ciência da computação.

Ao longo da história da ciência da computação, algumas perguntas se destacaram nos estudos desenvolvidos, como:

- O que é uma solução computável?
- Quais são os limites do que pode ser computado?
- Existem problemas sem solução computacional?

Tais perguntas caracterizam em grande parte o escopo da *teoria da computação*. Assim, a teoria da computação estuda modelos de computação suficientemente genéricos para especificar qualquer *função computável*, bem como explora os limites do que pode ser computado.

Antecipando algumas respostas, existem problemas não computáveis! De fato, o cardinal (quantidade) dos problemas computáveis é muito inferior ao dos não computáveis. Informalmente, no universo dos problemas, “poucos” problemas são computáveis, o que contradiz a intuição comum.

No início do século XX, ocorreu um grande impulso nas pesquisas, com o objetivo de definir um modelo computacional suficientemente genérico, capaz de implementar qualquer *função computável* base para viabilizar a exploração dos limites do que pode ser computado.

Um importante marco da teoria da computação é o trabalho do matemático alemão David Hilbert (1862~1943), denominado de *problema de decisão* (também conhecido como *Entscheidungsproblem*), proposto como um dos desafios para o século XX e formalizado em 1928 como segue:

encontrar um algoritmo que recebe como entrada a descrição de uma linguagem formal e uma sentença nesta linguagem e tem como saída “verdadeiro” ou “falso” dependendo se a sentença de entrada é verdadeira ou falsa.

Este problema é frequentemente identificado com o problema de decisão da lógica de primeira ordem (no qual a quantificação é restrita às variáveis que denotam elementos de conjuntos), ou seja, determinar algoritmicamente se uma sentença na lógica de primeira ordem é válida ou não. A procura de Hilbert por tal procedimento se justifica pelo fato de que ele acreditava que todo problema bem definido poderia ser resolvido. De fato, até 1930, ele acreditava que não existia problema sem solução.

Em 1931, o matemático austríaco naturalizado americano Kurt Gödel (1906~1978) publicou o trabalho denominado de *incompleteness theorem* (*teorema da não completude*), demonstrando que qualquer sistema axiomático que define a multiplicação e a adição no conjunto dos números naturais não pode ser simultaneamente completo e consistente: se é consistente, existem proposições que não podem ser verificadas nesta axiomatização (incompleto); se for completo, não poderá validar a si mesmo, (inconsistente) (Gödel, 1965). Uma característica importante do trabalho de Gödel é o uso dos números naturais para codificar fórmulas

lógicas, ou seja, uma enumeração das mesmas. Possivelmente, foi o primeiro a identificar um formalismo para definir a noção de *procedimento efetivo*.

Entretanto, para responder o problema de decisão de Hilbert, era necessário formalizar a definição de algoritmo, o que foi feito em 1936 independentemente pelo matemático americano Alonzo Church (1903~1995) e pelo matemático britânico Alan Turing (1912~1954).

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

- cálculo Lambda (Church, 1936);
- funções recursivas (Kleene, 1936).

De fato, a equivalência de ambos os formalismos foi verificada por Kleene (1956), levando Church a sugerir o que é conhecido como a *tese de Church*:

Tais formalismos são caracterizações tão gerais da noção do efetivamente computável quanto consistentes com o entendimento intuitivo usual (Church, 1936).

A tese de Church não é demonstrável, pois é fundamentada em uma noção intuitiva (não formal) de o que é efetivamente computável (algoritmo). Como todas as evidências indicavam ser verdadeira, foi assumida como uma hipótese para toda a teoria da computação, razão pela qual também é conhecida como *hipótese de Church*.

Independentemente de Church, Alan Turing propôs, em 1936, 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 do efetivamente computável. A *máquina de Turing* é um formalismo muito simples, universalmente conhecido e provavelmente o mais usado como modelo teórico de computação. A intenção do modelo foi simular, tanto quanto possível, as atitudes humanas relacionadas à computação. O resultado foi uma fundamentação teórica para o desenvolvimento do computador como se conhece hoje.

Desde então, muitos outros formalismos foram propostos, os quais comprovadamente possuem (no máximo) o mesmo poder computacional das funções recursivas ou do cálculo Lambda, como, por exemplo:

- máquina de Turing (1936);
- máquina Norma (1976)
- sistema canônico de Post (1943);
- algoritmo de Markov e linguagem Snobol (1954);
- máquina de registradores (1963);
- RASP (*Random Access Stored Programs* - 1964).

Assim, define-se *programa* como sendo um *procedimento efetivo*, ou seja, que pode ser descrito usando qualquer formalismo equivalente aos 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 da teoria da computação combinando abordagens históricas com abordagens próximas dos sistemas de 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.

Assim, busca-se uma forma de tratar a teoria da computação que efetivamente contribua na formação de profissionais em computação, propiciando um equilíbrio entre o lado teórico-formal, necessário para o adequado desenvolvimento desta e de outras matérias correlatas, e o lado aplicado, permitindo uma correta visualização de como usar os conceitos e resultados estudados.

Parte desta abordagem voltada para os sistemas computadores atuais é inspirada, entre outros, no trabalho de Richard Bird (1976) em *Programs and Machines - An Introduction to the Theory of Computation*.

1.3

→ conceitos básicos

Linguagem é um conceito fundamental no estudo da teoria da computação, pois se trata de uma forma precisa de expressar problemas, permitindo um desenvolvimento formal adequado ao estudo da computabilidade. Mais precisamente, em capítulos subsequentes, 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 define linguagem como:

*o uso da palavra articulada ou escrita como meio
de expressão e comunicação entre pessoas (Ferreira, 1984).*

Entretanto, esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria baseada em linguagens.

De fato, *linguagem* é um dos conceitos mais fundamentais em computação e informática. Entretanto, para definir linguagem é necessário antes introduzir os conceitos de *alfabeto* e de *palavra* ou *cadeia de caracteres*.

As definições que seguem são construídas usando como base a noção de *símbolo* ou *caractere*. Portanto, esta é uma entidade abstrata básica, não sendo definida formalmente. Letras e dígitos são exemplos de símbolos frequentemente 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:

$\{ a, b, c \}$
 $\emptyset$ (conjunto vazio)

b Os seguintes conjuntos *não* são exemplos de alfabetos (por quê?):

$\mathbf{N}$ (conjunto dos números naturais)
 $\{ a, b, aa, ab, ba, bb, aaa, \dots \}$ □

exemplo 1.2 – Alfabeto: linguagem de programação

O alfabeto de uma linguagem de programação como Pascal é o conjunto de todos os símbolos usados na construção de programas, incluindo:

- letras;
- dígitos;
- caracteres especiais como ">", "/", etc.;
- espaço ou "branco". □

Ao longo de todo o texto, é usado, com frequência, um alfabeto binário (no caso, o alfabeto $\{ a, b \}$) pois, além de manter uma perfeita analogia com a representação interna dos computadores reais (o domínio de valores de um *bit* é binário), a manipulação de poucos símbolos simplifica as diversas abordagens desenvolvidas.

definição 1.2 – Cadeia de símbolos, palavra

Seja Σ um alfabeto. Então:

- a** Uma *cadeia de símbolos* sobre Σ é uma sequência de zero ou mais símbolos (de Σ) justapostos;
- b** Uma *palavra* sobre Σ é uma cadeia finita de símbolos de Σ . □

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

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

definição 1.3 – Comprimento, tamanho de uma palavra

O *comprimento* ou *tamanho* de uma palavra w , representado por $|w|$, é o número de símbolos que compõem a palavra. Portanto, para um dado alfabeto Σ , comprimento é uma função com domínio em Σ^* e codomínio em $\mathbf{N}$. □

exemplo 1.3 – Comprimento de uma palavra

a $|abcb| = 4$

b $|\epsilon| = 0$ □

definição 1.4 – Prefixo, sufixo, subpalavra

Um *prefixo* (respectivamente, *sufixo*) de uma palavra é qualquer sequência inicial (respectivamente, final) de símbolos da palavra.

Uma *subpalavra* de uma palavra é qualquer sequência de símbolos contíguos da palavra. □

exemplo 1.4 – Palavra, prefixo, sufixo, subpalavra

a **abcb** é uma palavra sobre o alfabeto $\{a, b, c\}$

b Em relação à palavra **abcb**, vale que:

$\epsilon, a, ab, abc, abcb$ são todos os prefixos;

$\epsilon, b, cb, bcb, abcb$ são todos os sufixos;

c Qualquer prefixo ou sufixo de uma palavra é uma subpalavra;

d Para a palavra **aa**, o conjunto de todos os prefixos, de todos os sufixos e de todas as subpalavras é o mesmo:

$$\{\epsilon, a, aa\}$$

□

exemplo 1.5 – Palavra: linguagem de programação

Em uma linguagem de programação como Pascal, uma palavra é um programa. □

definição 1.5 – Conjunto de todas as palavras

Seja Σ um alfabeto. Então o *conjunto de todas as palavras* Σ^* é indutivamente definido como segue:

a *Base de indução.*

$$\epsilon \in \Sigma^*$$

para qualquer $x \in \Sigma$, vale $x \in \Sigma^*$

b *Passo de indução.*

Se **u** e **v** são palavras de Σ^* ,

então a justaposição **u v** é uma palavra de Σ^* □

Portanto, uma definição alternativa para *palavra* sobre um alfabeto Σ é qualquer elemento **w** de Σ^* , ou seja:

$$w \in \Sigma^*$$

exemplo 1.6 – Conjunto de todas as palavras

Se $\Sigma = \{a, b\}$, então:

$$\Sigma^+ = \{a, b, aa, ab, ba, bb, aaa, \dots\}$$

$$\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$$

□

definição 1.6 – Linguagem formal

Uma *linguagem formal*, ou simplesmente *linguagem* L , sobre um alfabeto Σ é um conjunto de palavras sobre Σ , ou seja:

$$L \subseteq \Sigma^* \quad \square$$

exemplo 1.7 – Linguagem formal

- a** O conjunto vazio $\emptyset$ e o conjunto formado pela palavra vazia $\{\epsilon\}$ são linguagens sobre qualquer alfabeto. Obviamente, vale que:

$$\emptyset \neq \{\epsilon\}$$

- b** Os conjuntos Σ^* e Σ^+ são linguagens sobre um alfabeto Σ qualquer. Obviamente, vale que:

$$\Sigma^* \neq \Sigma^+$$

- c** Suponha o alfabeto $\Sigma = \{a, b\}$. Então, 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, são palavras desta linguagem:

$$\epsilon, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, \dots \quad \square$$

exemplo 1.8 – Conjunto de todas as linguagens sobre um alfabeto

O conjunto de todas as linguagens sobre um alfabeto Σ é o conjunto das partes de Σ^* , ou seja:

$$2^{\Sigma^*} \quad \square$$

exemplo 1.9 – Linguagem formal: linguagem de programação

Uma linguagem de programação como Pascal é formalmente definida pelo conjunto de todos os programas (palavras) da linguagem.

definição 1.7 – 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 as seguintes propriedades (suponha v , w , t palavras quaisquer):

- a** *Associativa*.

$$v(wt) = (vw)t$$
- b** *Elemento neutro*.

$$\epsilon w = w = w \epsilon$$

□

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 $vw t$.

exemplo 1.10 – Concatenação de palavras

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

$$v w = baaaabb$$

$$v \varepsilon = v = baaaa$$

□

definição 1.8 – 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):

$$w^n \quad \text{onde } n \text{ é o número de concatenações sucessivas}$$

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

$$w^0 = \varepsilon$$

$$w^n = w w^{n-1}, \text{ para } n > 0$$

□

exemplo 1.11 – Concatenação sucessiva

Sejam w uma palavra e a um símbolo. Então:

$$w^3 = w w w$$

$$w^1 = w$$

$$a^5 = aaaaa$$

$$a^n = aaa...a \quad (\text{o símbolo } a \text{ repetido } n \text{ vezes})$$

□

1.4

→ conclusões

A origem do estudo da computação é milenar e se desenvolveu em diversas épocas e culturas, objetivando responder questões como:

- O que é uma solução computável?
- Quais são os limites do que pode ser computado?
- Existem problemas sem solução computacional?

Foi a partir do século XX que importantes contribuições ocorreram, com destaque para os trabalhos de Hilbert (problema de decisão), de Church (cálculo Lambda e hipótese de Church) e de Turing (máquina de Turing e problema da parada). Assim, o estudo da teoria da computação independe do estudo do computador (*hardware* e *software*) como se conhece hoje. Por outro lado, é uma base fundamental para qualquer estudo atual em ciência da computação.

Neste capítulo, também, foram introduzidos alguns dos principais conceitos sobre os quais todo o estudo é construído, como alfabeto, palavra e linguagem formal.

1.5

→ exercícios

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

exercício 1.2 Em que se consistia o problema de Hilbert – *Entscheidungsproblem* e por que ele é sem solução?

exercício 1.3 Qual a importância da tese de Church e por que ela não é demonstrável?

exercício 1.4 Marque os conjuntos que são alfabetos:

- | | | |
|----------|--|-----|
| a | Conjunto dos números inteiros | [] |
| 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 1.5 Apresente os possíveis prefixos e sufixos de cada uma das seguintes palavras:

- | | |
|----------|--------------|
| a | teoria |
| b | universidade |
| c | aaa |
| d | abccba |
| e | abcabc |

exercício 1.6 Exemplifique, comprovando ou negando as seguintes propriedades algébricas da operação de concatenação de palavras:

- | | |
|----------|---|
| a | Total (suponha a operação definida sobre uma determinada linguagem); |
| b | Comutativa; |
| c | Elemento neutro; |
| d | Associativa; |
| e | Elemento absorvente, genericamente definida como segue: para uma dada operação binária $\oplus$ sobre um conjunto A , afirma-se que a operação $\oplus$ possui <i>elemento absorvente</i> se existe a $\in$ A tal que, para qualquer x $\in$ A vale que: |

$$\mathbf{a} \oplus \mathbf{x} = \mathbf{x} \oplus \mathbf{a} = \mathbf{a}$$

- | | |
|----------|--|
| f | Elemento inverso, genericamente definido como segue: para uma dada operação binária $\oplus$ sobre um conjunto A , afirma-se que a operação $\oplus$ possui <i>elemento inverso</i> se possui elemento neutro ε e, para qualquer a $\in$ A , existe a tal que: |
|----------|--|

$$\mathbf{a} \oplus \mathbf{a} = \mathbf{a} \oplus \mathbf{a} = \varepsilon$$

exercício 1.7 Quando se pode dizer que a estrutura algébrica da operação de concatenação sobre uma linguagem é análoga à estrutura da operação de adição nos naturais?

exercício 1.8 Analise as seguintes afirmações:

- I. O conjunto dos números primos é um alfabeto;
- II. O conjunto das partes de $\{a, b, c\}$ não é um alfabeto;
- III. O conjunto dos algarismos romanos se constitui em um alfabeto.

Marque a alternativa correta:

- ☐ **a** Apenas I e III estão corretas;
- ☐ **b** Apenas I está correta;
- ☐ **c** Todas as afirmativas estão corretas;
- ☐ **d** Todas as afirmativas estão erradas;
- ☐ **e** Nenhuma das alternativas anteriores está correta.

exercício 1.9 Todos os possíveis prefixos da palavra **abccba** são:

- ☐ **a** $\{ a, ab, abc, abcc, abccb, abccba \}$
- ☐ **b** $\{ a, ba, cba, ccba, bccba, abccba \}$
- ☐ **c** $\{ a, ab, abc, abccb, abccba \}$
- ☐ **d** $\{ a, ba, cba, ccba, abccba \}$
- ☐ **e** Nenhuma das alternativas anteriores está correta.

exercício 1.10 Em relação à palavra **abccba**, marque a alternativa correta:

- ☐ **a** Os conjuntos de sufixos e de prefixos são o mesmo;
- ☐ **b** O alfabeto sobre o qual é gerada essa palavra só pode ser $\Sigma = \{a, b, c\}$;
- ☐ **c** Existem exatamente três subpalavras que são prefixos e sufixos ao mesmo tempo;
- ☐ **d** Existem exatamente duas subpalavras que são prefixos e sufixos ao mesmo tempo;
- ☐ **e** Nenhuma das afirmativas anteriores está correta.

Termos-chaves

ciência da computação, p. 22

funções recursivas, p.24

hipótese de Church, p. 24

cálculo Lambda, p.24

máquina de Turing, p. 24

máquinas universais, p. 25

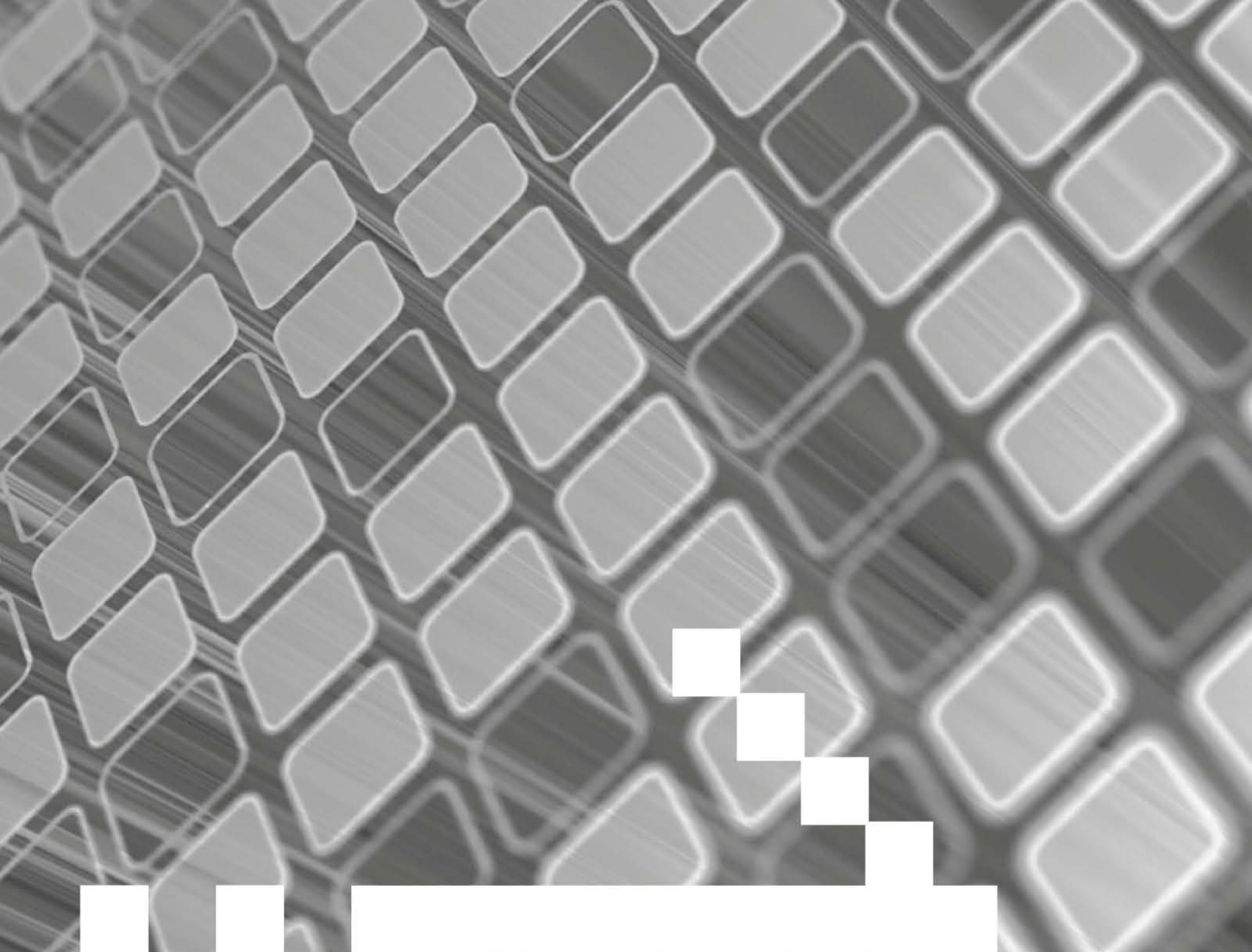
problema de decisão, p. 23

procedimento efetivo, p. 24

teorema da não completude, p. 23

teoria da computação, p. 22

tese de Church, p. 24

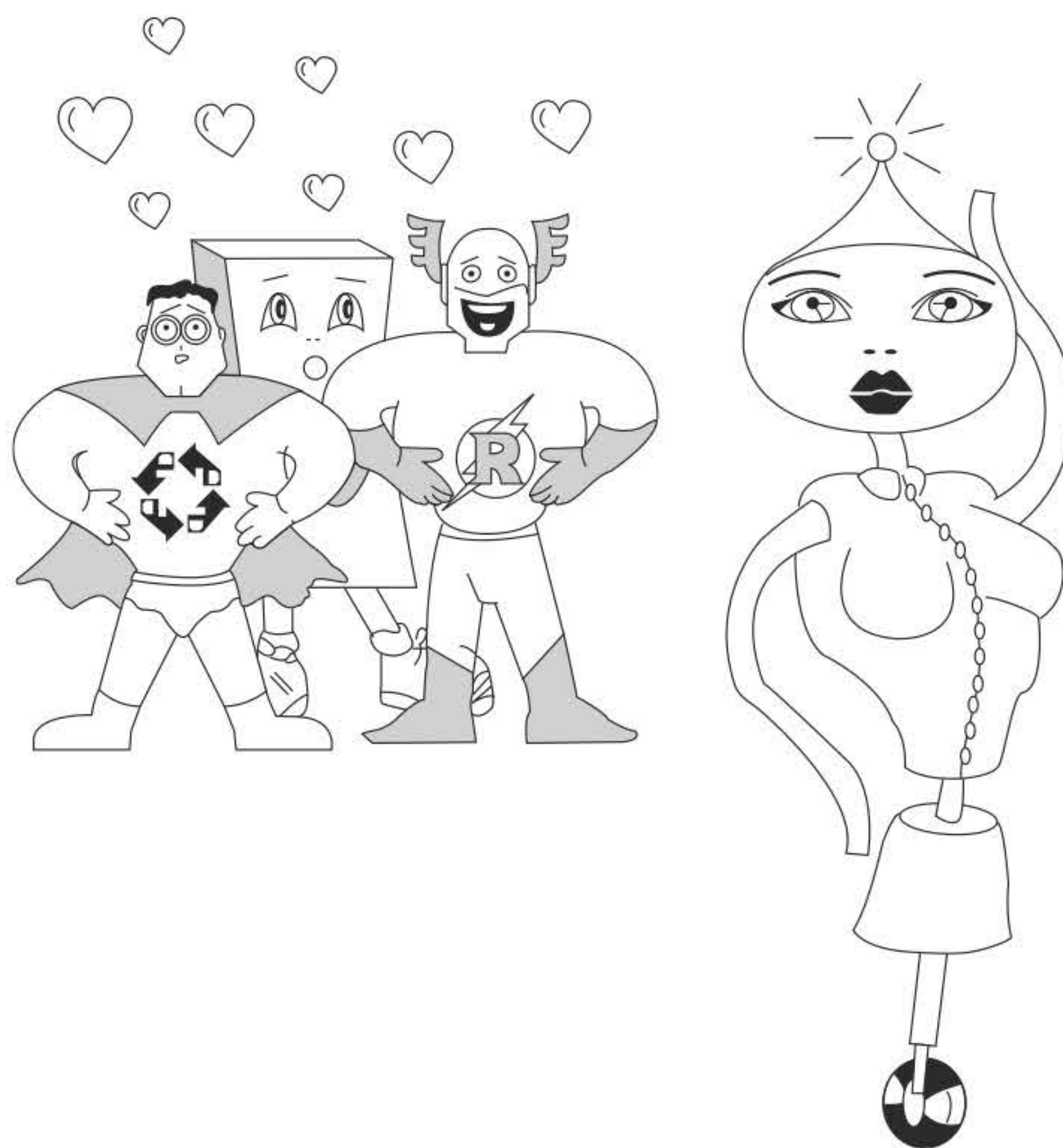


capítulo

2

programas, máquinas e computações

■ ■ Programa (estrutura algorítmica) e máquina (semântica para as operações e testes do programa) são conceitos usados para construir a noção de computação (o histórico da execução do programa em uma máquina) e de função computada. Três tipos de programas são estudados: monolítico, iterativo e recursivo. Duas relações de equivalência são estudadas, ambas baseadas na função computada, refletindo uma noção de "programas que fazem a mesma coisa": programas fortemente equivalentes (programas equivalentes em qualquer máquina) e programas equivalentes em uma máquina. A equivalência forte de programas induz uma hierarquia de tipos de programas: recursivos são mais gerais que os monolíticos os quais, por sua vez, são mais gerais que os iterativos. Analogamente, é estabelecida noções de equivalência de máquinas, ou seja, se uma máquina simula a outra e vice-versa.



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 é baseada 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 his-

tórico 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 para).

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

- *programas fortemente equivalentes* : 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.

Em relação aos programas fortemente equivalentes , 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.

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 sequencial*. A execução da operação ou teste subsequente somente pode ser realizada após o encerramento da execução da operação ou teste anterior;
- b** *Composição não determinística ou escolha*. A próxima operação ou teste a ser executado é uma escolha entre diversas alternativas de operações ou testes compostos (em oposição

à determinística, na qual, para as mesmas condições, a próxima operação ou teste a ser executado é sempre o mesmo).

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

Portanto, na modelagem de sistemas em geral, as três formas de composição devem ser consideradas.

No caso específico da *teoria da computação*, as composições sequencial e não determinística são especialmente importantes, e é nestas que o estudo é centrado. Como curiosidade, destaca-se que a composição concorrente pode ser simulada usando a técnica de *intercalação* ou *concorrência falsa*. Na intercalação, uma concorrência (verdadeira) é representada como composições não determinísticas de todas as combinações de composição sequenciais possíveis, garantindo que a ordem de execução é irrelevante.

A semântica do *não determinismo* adotada em todo o texto que segue é a usual para a *teoria da computação*, para as *linguagens formais*, bem como para os demais estudos correlatos, e objetiva determinar a capacidade de resolver problemas. 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.

Esta semântica do não determinismo, detalhada ao longo do texto, difere, por exemplo, da usualmente adotada no estudo dos *modelos para concorrência* (e consequente aplicações como, por exemplo, no estudo dos *sistemas operacionais*), e pode causar alguma confusão com a semântica da concorrência.

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.*

$F, F_1, F_2, F_3, \dots, G, G_1, G_2, G_3, \dots, H, H_1, H_2, H_3, \dots$

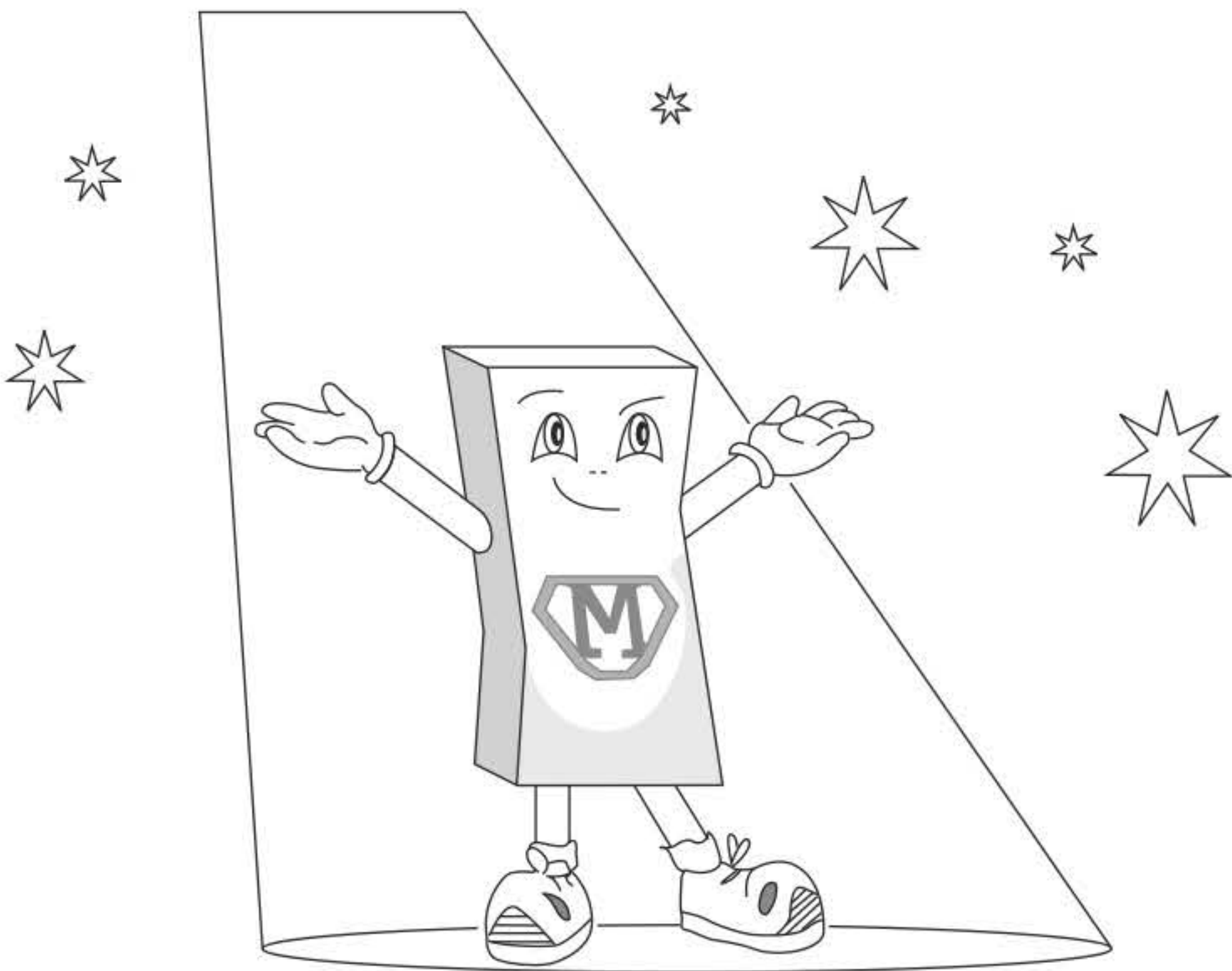
- b** *Identificadores de testes.*

$T, T_1, T_2, T_3, \dots$

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, são supostos conhecimentos básicos de programas.



2.1.1 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 por meio 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.

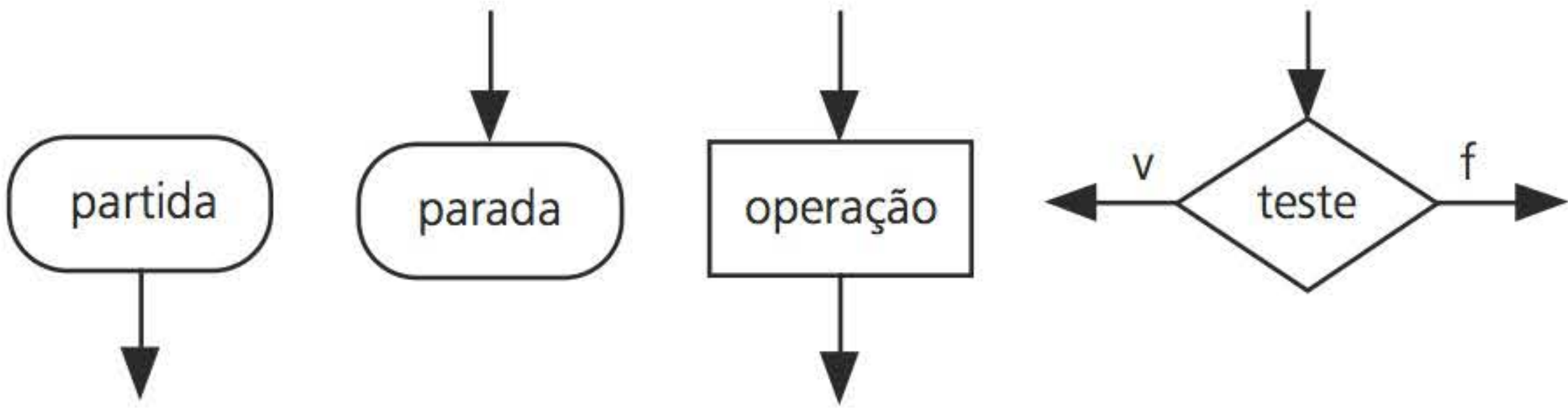


figura 2.1 Componentes elementares de um fluxograma.

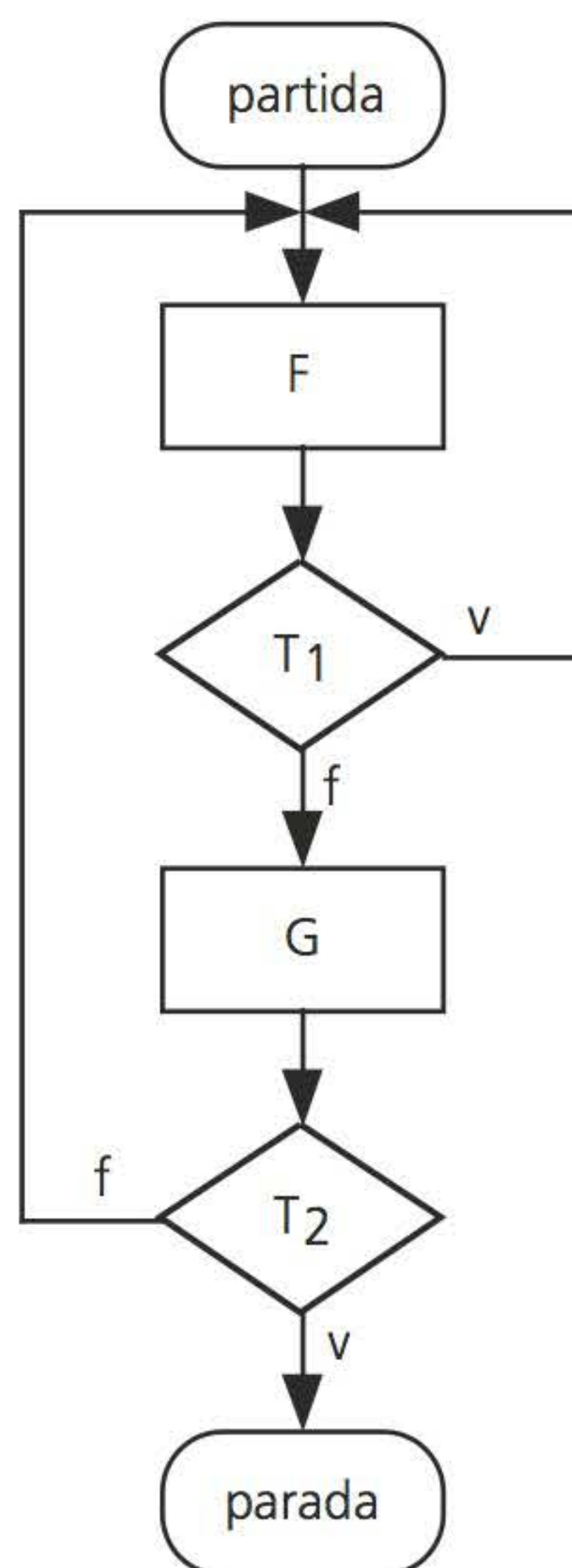


figura 2.2 Fluxograma.

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 subsequente;
- b** *Teste*. Determina um desvio condicional, ou seja, que depende da avaliação de um teste para determinar qual a instrução subsequente.

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, etiqueta, instrução rotulada, instrução etiquetada

- a** Um *rótulo* ou *etiqueta* é uma palavra sobre o alfabeto de letras ou dígitos;
- b** Uma *instrução rotulada* ou *instrução etiquetada* é uma cadeia de caracteres de uma das duas formas a seguir (suponha que **F** e **T** são identificadores de operação e de teste, respectivamente e que r_1 , r_2 e r_3 são rótulos):
 - b.1) *Operação*.
 r_1 : faça **F** vá_para r_2 ou
 r_1 : faça ✓ vá_para r_2
 - b.2) *Teste*.
 r_1 : se **T** então vá_para r_2 senão vá_para r_3 □

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 monolíticos

Um *programa monolítico* **P** é um par ordenado:

$$\mathbf{P} = (\mathbf{I}, \mathbf{r})$$

onde:

- a** $\mathbf{I}$ é um conjunto finito de *instruções rotuladas*, sendo que não existem duas instruções diferentes com um mesmo rótulo;
- b** $\mathbf{r}$ é um rótulo distinguido denominado *rótulo inicial* que identifica a instrução rotulada inicial em $\mathbf{I}$;
- c** um rótulo é dito *rótulo final* se é referenciado por pelo menos uma instrução de $\mathbf{I}$, mas não rotula qualquer instrução de $\mathbf{I}$. □

Portanto, em um programa monolítico, o conjunto de instruções rotuladas nunca é vazio, pois sempre existe pelo menos a instrução etiquetada 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** $P_2 = (\{ r_1: \text{faça } \checkmark \text{ vá_para } r_2 \}, r_1)$ 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 sequencial 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 muitas 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 nos quais existe uma grande liberdade para definir desvios incondicionais, ocasionando o que vulgarmente é conhecido como “quebras de lógica”. A ideia 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* e inspiraram uma nova geração de linguagens de programação, como Pascal.

Programas iterativos são baseados em três mecanismos de composição (sequenciais) de programas, os quais podem ser encontrados com variações em um grande número de linguagens de alto nível, como, por exemplo, Algol, Pascal, C e Java. Esses mecanismos de composição são:

- *sequencial*: 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**.

Um mecanismo alternativo ao *enquanto* é a seguinte composição:

- *até*: análoga à 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 sequencial*. Se **V** e **W** são programas iterativos, então a composição sequencial denotada por

$$\mathbf{V;W}$$

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

$$(\text{se } \mathbf{T} \text{ então } \mathbf{V} \text{ senão } \mathbf{W})$$

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

$$\text{enquanto } \mathbf{T} \text{ faça } (\mathbf{V})$$

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, até que o resultado do teste for o valor **verdadeiro** (ou enquanto o resultado do teste for o valor **falso**). Caso contrário, a iteração termina. $\square$

Assim, em relação à composição sequencial, tem-se que:

$P_1; P_2; P_3; \dots; P_n$

denota o programa cujo efeito é a execução na ordem $P_1, P_2, \dots, P_n$ (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 ou nas chaves da linguagem C.

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, o programa **V** pode não ser executado (em que condições isto ocorre?).

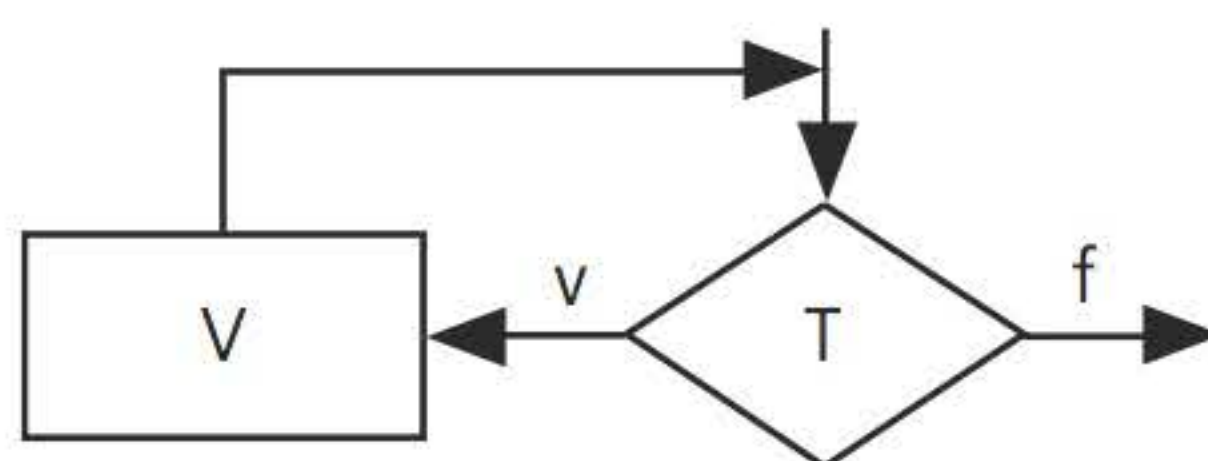


figura 2.4 Fluxograma que simula a composição enquanto.

exemplo 2.5 – Programa iterativo

- a** A operação vazia $\checkmark$ 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 indentação objetiva facilitar a identificação visual da estrutura do programa. $\square$


```
(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, em linguagens de programação que admitem 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.

Para o que segue, suponha um conjunto de *identificadores de sub-rotinas* os quais são denotados por:

$$R, R_1, R_2, R_3, \dots$$

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 $\checkmark$ 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 sequencial*. Se D_1 e D_2 são expressões de sub-rotinas, então a composição sequencial 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:

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

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:

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

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

- 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. □

A expressão inicial $\checkmark$ 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,

$$P \text{ é } \checkmark$$

denota o programa recursivo P que faz nada. Analogamente, se o programa, é da forma:

$$P \text{ é } E_0 \text{ onde}$$

também é usual omitir a palavra *onde*, ou seja,

$$P \text{ é } E_0$$

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 autorreferenciam indiretamente. □

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

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. Um programa é simplesmente uma sequência de caracteres, uma palavra, e portanto, sem significado semântico. Assim, 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. Ou seja, um programa é simplesmente uma sequência de símbolos, uma palavra, e portanto, sem significado semântico. A natureza das operações e dos testes é especificada na definição de máquina, a qual permite dar uma interpretação semântica do programa.

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 as funções de entrada e de saída, ou seja, o armazenamento ou recuperação de informações na estrutura de memória; e o significado (dar semântica) aos identificadores das operações e testes. 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. Observe que, para cada identificador de operação ou teste definido em uma máquina, existe somente uma função associada.

definição 2.7 – Máquina

Uma *máquina* é uma 7-upla ordenada:

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

na qual:

- a** V , um conjunto de *valores de memória*;
- b** X , um conjunto de *valores de entrada*;
- c** Y , um conjunto de *valores de saída*;
- d** π_X , uma *função de entrada* tal que:

$$\pi_X: X \rightarrow V$$

- e** π_Y , uma *função de saída* tal que:

$$\pi_Y: V \rightarrow Y$$

- f** Π_F , um conjunto de *interpretações de operações* onde, para cada identificador de operação F interpretado por M , existe uma única função:

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

- g** Π_T , um conjunto de *interpretações de testes* tal que, para cada identificador de teste T interpretado por M , existe uma única função:

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

□

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.

exemplo 2.7 – Máquina de dois registradores

Suponha uma especificação de uma máquina com duas posições de memória, denominadas registradores **a** e **b**, os quais assumem valores em $\mathbf{N}$, 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, a entrada é constituída de um único valor armazenado em **a**, zerando **b**, e a saída retorna o valor de **b**.

Dois registradores com valores no conjunto dos números naturais $\mathbf{N}$ podem ser definidos pelo produto cartesiano $\mathbf{N}^2$ onde os registradores **a** e **b** são representados pela primeira e segunda componentes, respectivamente. Então a máquina na figura 2.7 implementa a especificação acima.

Note que as operações e os testes são definidos para certas posições de memória.

□

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

onde:

$\mathbf{N}^2$ é conjunto de valores de memória

$\mathbf{N}$ é conjunto de valores de entrada, bem como o de saída

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

$$\text{armazena_a}(n) = (n, 0)$$

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

$$\text{retorna_b}(n, m) = m$$

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

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

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

$$\text{adiciona_b}(n, m) = (n, m+1)$$

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

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

figura 2.7 Máquina com dois registradores.

Afirma-se que $\mathbf{P}$ é um *programa para a máquina* $\mathbf{M}$ se cada identificador de teste e operação em $\mathbf{P}$ tiver uma correspondente função de teste e operação em $\mathbf{M}$, 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 $\mathbf{M} = (\mathbf{V}, \mathbf{X}, \mathbf{Y}, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e $\mathbf{P}$ um programa onde $\mathbf{P}_F$ e $\mathbf{P}_T$ são os conjuntos de identificadores de operações e testes de $\mathbf{P}$, respectivamente. $\mathbf{P}$ é um *programa para a máquina* $\mathbf{M}$ se, e somente, se:

- para qualquer $F \in \mathbf{P}_F$, existe uma única função $\pi_F: \mathbf{V} \rightarrow \mathbf{V}$ em Π_F ;
- para qualquer $T \in \mathbf{P}_T$, existe uma única função $\pi_T: \mathbf{V} \rightarrow \{\text{verdadeiro}, \text{falso}\}$ em Π_T .

Adicionalmente, a operação vazia $\checkmark$ sempre é interpretada em qualquer máquina. □

Portanto, para cada identificador de operação F de $\mathbf{P}_F$ (respectivamente, teste T de $\mathbf{P}_T$) existe, na máquina $\mathbf{M}$, 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** (figura 2.8) e recursivo **rec_b←a** (figura 2.9), são programas para a máquina **dois_reg** (figura 2.7) e atribuem o valor armazenado em **a** ao registrador **b**. □

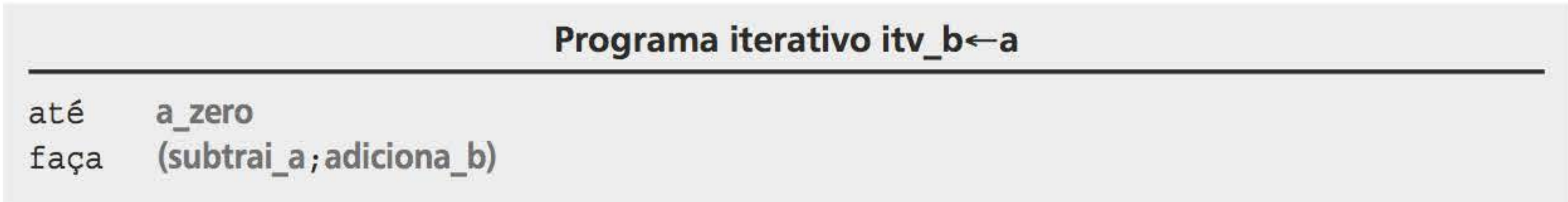


figura 2.8 Programa iterativo para a máquina de dois registradores.

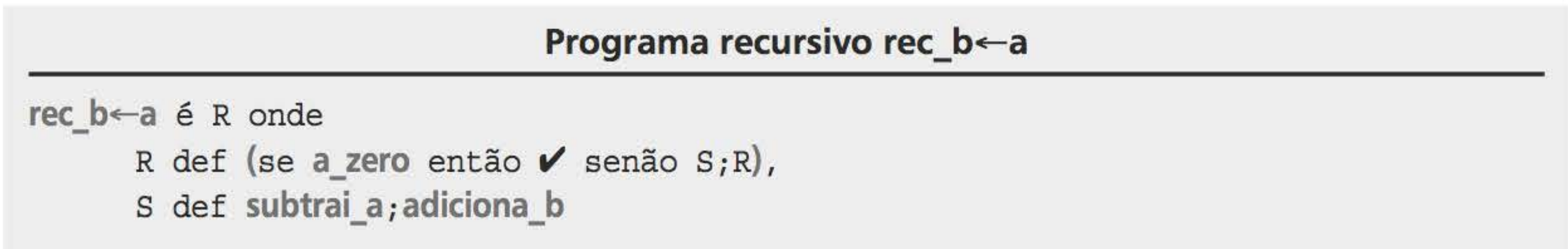


figura 2.9 Programa recursivo para a máquina de dois registradores.

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 ordenados onde:

- cada par ordenado reflete uma configuração ou descrição instantânea da máquina para o programa, ou seja, a instrução a ser executada e o valor corrente da memória;
- a cadeia de pares é a sequência de configurações possíveis a partir da configuração inicial (instrução inicial e valor de memória considerado).

definição 2.9 – Computação de programa monolítico em uma máquina

Sejam $\mathbf{M} = (\mathbf{V}, \mathbf{X}, \mathbf{Y}, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e $\mathbf{P} = (\mathbf{I}, \mathbf{r})$ um programa monolítico para $\mathbf{M}$ onde $\mathbf{L}$ é o seu correspondente conjunto de rótulos. Uma *computação do programa monolítico $\mathbf{P}$ na máquina $\mathbf{M}$* , para uma dada entrada $\mathbf{x} \in \mathbf{X}$, é uma cadeia (finita ou infinita) de *configurações* ou *descrições instantâneas* em $\mathbf{L} \times \mathbf{V}$:

$$(s_0, v_0)(s_1, v_1)(s_2, v_2) \dots$$

A *configuração inicial* (s_0, v_0) é tal que $s_0 = \mathbf{r}$ é o rótulo inicial do programa $\mathbf{P}$, $v_0 = \pi_X(\mathbf{x})$ é o valor inicial de memória.

Para cada configuração (s_k, v_k) da cadeia, onde $k \in \{0, 1, 2, \dots\}$, tem-se que (suponha que $\mathbf{F}$ é um identificador de operação, $\mathbf{T}$ é um identificador de teste e $\mathbf{r}', \mathbf{r}''$ são rótulos de $\mathbf{L}$):

a Operação.

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

$$s_k: \text{faça } \mathbf{F} \text{ vá_para } \mathbf{r}'$$

então $(s_{k+1}, v_{k+1}) = (\mathbf{r}', \pi_F(v_k))$ é a configuração subsequente de (s_k, v_k) na cadeia, representando a execução de $\mathbf{F}$ e a transferência para a próxima instrução;

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

$$s_k: \text{faça } \checkmark \text{ vá_para } \mathbf{r}'$$

então $(s_{k+1}, v_{k+1}) = (\mathbf{r}', v_k)$ é configuração subsequente de (s_k, v_k) na cadeia, representando uma transferência incondicional;

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

$$s_k: \text{se } \mathbf{T} \text{ então vá_para } \mathbf{r}' \text{ senão vá_para } \mathbf{r}''$$

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

$$\begin{aligned} s_{k+1} &= \mathbf{r}' && \text{se } \pi_T(v_k) = \text{verdadeiro} \\ s_{k+1} &= \mathbf{r}'' && \text{se } \pi_T(v_k) = \text{falso} \end{aligned}$$

representando uma transferência condicional, conforme o valor resultante do teste;

Uma *computação* é dita *finita* ou *infinita*, se a cadeia que a define é finita ou infinita, respectivamente. Sendo finita, o rótulo da última configuração da cadeia é um rótulo final. $\square$

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.

Assim, uma computação de um programa em uma máquina para uma dada entrada é uma sequência de configurações a qual define (dá semântica) passo a passo a execução do programa na máquina. Cada configuração resume as informações para determinar a configuração subsequente.

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 a entrada **3**, o que corresponde ao 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 dois 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 dois 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 dois 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 dois registradores.

Observe que, em relação ao programa monolítico para uma dada máquina, uma configuração ou descrição instantânea contém toda a informação necessária para realizar os processamentos subsequentes a partir deste instante, ou seja, o conteúdo atual da memória (consequência de processamentos passados) e a próxima instrução a ser executada.

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

- cada par ordenado reflete uma *configuração* ou *descrição instantânea* da máquina para o programa, ou seja, a expressão de sub-rotina a ser executada e o valor corrente da memória;
- a cadeia de pares é a sequência de configurações possíveis a partir da configuração inicial (expressão de sub-rotina inicial e valor de memória considerado).

definição 2.10 – Computação de programa recursivo em uma máquina

Sejam $\mathbf{M} = (\mathbf{V}, \mathbf{X}, \mathbf{Y}, \pi_{\mathbf{X}}, \pi_{\mathbf{Y}}, \Pi_{\mathbf{F}}, \Pi_{\mathbf{T}})$ uma máquina e $\mathbf{P}$ um programa recursivo para $\mathbf{M}$ tal que:

$$\mathbf{P} \text{ é } \mathbf{E}_0 \text{ onde } R_1 \text{ def } \mathbf{E}_1, R_2 \text{ def } \mathbf{E}_2, \dots, R_n \text{ def } \mathbf{E}_n$$

Uma *computação do programa recursivo P na máquina M* é uma cadeia (finita ou infinita) de *configurações* ou *descrições instantâneas* da forma:

$$(\mathbf{D}_0, \mathbf{v}_0)(\mathbf{D}_1, \mathbf{v}_1)(\mathbf{D}_2, \mathbf{v}_2)\dots$$

A *configuração inicial* $(\mathbf{D}_0, \mathbf{v}_0)$ é tal que $\mathbf{D}_0 = \mathbf{E}_0; \checkmark$, $\mathbf{v}_0 = \pi_{\mathbf{X}}(\mathbf{x})$ é o valor inicial de memória.

Para cada par $(\mathbf{D}_k, \mathbf{v}_k)$ da cadeia, onde $k \in \{0, 1, 2, \dots\}$ (sendo $\mathbf{D}_k$ uma sequência), tem-se que (suponha que $\mathbf{F}$ é um identificador de operação, $\mathbf{T}$ é um identificador de teste e $\mathbf{C}, \mathbf{C}_1, \mathbf{C}_2$ são expressões de sub-rotina):

Caso 1. Se $\mathbf{D}_k$ é uma expressão de sub-rotina da forma:

$$\mathbf{D}_k = \checkmark; \mathbf{C}$$

então, após a execução de $\checkmark$, $(\mathbf{D}_{k+1}, \mathbf{v}_{k+1}) = (\mathbf{C}, \mathbf{v}_k)$ é configuração subsequente de $(\mathbf{D}_k, \mathbf{v}_k)$ na cadeia, ou seja, $\checkmark$ é executado e resta $\mathbf{C}$ para ser executada;

Caso 2. Se $\mathbf{D}_k$ é uma expressão de sub-rotina da forma:

$$\mathbf{D}_k = \mathbf{F}; \mathbf{C}$$

então, após a execução de $\mathbf{F}$, $(\mathbf{D}_{k+1}, \mathbf{v}_{k+1}) = (\mathbf{C}, \pi_{\mathbf{F}}(\mathbf{v}_k))$ é configuração subsequente de $(\mathbf{D}_k, \mathbf{v}_k)$, ou seja, $\mathbf{F}$ é executada e resta $\mathbf{C}$ para ser executada;

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

$$D_k = R_i; C$$

então, após a chamada da sub-rotina R_i , $(D_{k+1}, v_{k+1}) = (E_i; C, v_k)$ é configuração subsequente de (D_k, v_k) na cadeia. Observe que, na configuração subsequente, primeiro é executada a expressão E_i da definição da sub-rotina R_i . Assim, a chamada da sub-rotina R_i é executada, a execução vai para E_i e, depois, é executado C ;

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 configuração subsequente de (D_k, v_k) na cadeia (a máquina interpreta a composição sequencial como associativa). Assim, $(C_1; (C_2; C))$ é executada da esquerda para a direita;

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

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

então (D_{k+1}, v_{k+1}) é configuração subsequente de (D_k, v_k) na cadeia caracterizando o desvio condicional como segue:

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

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?);
- um teste ou referência a uma sub-rotina não alteram o valor da memória;
- em uma computação finita, na configuração final (último par da cadeia) e apenas nesta, a expressão é ✓;
- 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:

$$(R; \checkmark, v_0) (R; \checkmark, v_0) (R; \checkmark, v_0) \dots$$

□

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:

$$\text{id}_A: A \rightarrow A$$

tal que, para qualquer $a \in A$, tem-se que $\text{id}_A(a) = a$.

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. □

um_reg = (N, N, N, id_N, id_N, { ad, sub }, { zero })

onde:

N corresponde, simultaneamente, aos conjuntos de valores de memória, entrada e saída

id_N: **N** → **N** é a função de entrada e de saída

ad: **N** → **N** é interpretação tal que, $\forall n \in \mathbf{N}, \text{ad}(n) = n+1$

sub: **N** → **N** é interpretação tal que, $\forall n \in \mathbf{N}$:

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

zero: **N** → { verdadeiro, falso } é interpretação tal que, $\forall n \in \mathbf{N}$:

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

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 sequencial
((R;ad;ad);✓, 2)	subtraiu 1 da memória
(R;(ad;ad);✓, 2)	caso 4, composição sequencial
((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 sequencial
((R;ad;ad);(ad;ad);✓, 1)	subtraiu 1 da memória
(R;(ad;ad);(ad;ad);✓, 1)	caso 4, composição sequencial
((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 sequencial
((R;ad;ad);(ad;ad);(ad;ad);✓, 0)	subtraiu 1 da memória
(R;(ad;ad);(ad;ad);(ad;ad);✓, 0)	caso 4, composição sequencial
((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 sequencial
((ad;(ad;ad);(ad;ad);✓), 1)	adicionou 1 na memória
(ad;((ad;ad);(ad;ad);✓), 1)	caso 4, composição sequencial
((ad;ad);(ad;ad);✓), 2)	adicionou 1 na memória
(ad;(ad;(ad;ad);✓), 2)	caso 4, composição sequencial
((ad;(ad;ad);✓), 3)	adicionou 1 na memória
(ad;((ad;ad);✓), 3)	caso 4, composição sequencial
((ad;ad);✓), 4)	adicionou 1 na memória
(ad;((ad);✓), 4)	caso 4, composição sequencial
((ad);✓), 5)	adicionou 1 na memória
(ad;(✓), 5)	caso 4, composição sequencial
((✓), 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 para;
- 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 $\mathbf{M} = (\mathbf{V}, \mathbf{X}, \mathbf{Y}, \pi_{\mathbf{X}}, \pi_{\mathbf{Y}}, \Pi_{\mathbf{F}}, \Pi_{\mathbf{T}})$ uma máquina e $\mathbf{P}$ um programa monolítico para $\mathbf{M}$. A função computada pelo programa monolítico $\mathbf{P}$ na máquina $\mathbf{M}$ denotada por:

$$\langle \mathbf{P}, \mathbf{M} \rangle: \mathbf{X} \rightarrow \mathbf{Y}$$

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

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

é uma computação finita de $\mathbf{P}$ em $\mathbf{M}$, onde o valor inicial da memória é dado pela função de entrada, ou seja, $v_0 = \pi_{\mathbf{X}}(\mathbf{x})$. Neste caso, a imagem de $\mathbf{x}$ é dada pela função de saída aplicada ao valor da memória da configuração final da computação, ou seja:

$$\langle \mathbf{P}, \mathbf{M} \rangle(\mathbf{x}) = \pi_{\mathbf{Y}}(v_n) \quad \square$$

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,

$$\langle \text{mon_b} \leftarrow \text{a}, \text{dois_reg} \rangle: \mathbf{N} \rightarrow \mathbf{N}$$

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

$$\langle \text{mon_b} \leftarrow \text{a}, \text{dois_reg} \rangle(n) = n$$

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

- $\pi_X(3) = (3, 0)$
- a correspondente computação é representada na figura 2.11
- $\langle \text{mon_b} \leftarrow a, \text{dois_reg} \rangle(3) = \pi_Y(0, 3) = 3$

Portanto, $\langle \text{mon_b} \leftarrow a, \text{dois_reg} \rangle$ 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). Em relação à função computada $\langle \text{comp_infinita}, \text{dois_reg} \rangle: \mathbf{N} \rightarrow \mathbf{N}$, 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 é análoga à de um monolítico:

- a computação inicia na expressão inicial concatenada com a expressão vazia e 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 para;
- 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 $\mathbf{M} = (\mathbf{V}, \mathbf{X}, \mathbf{Y}, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina e **P** um programa recursivo para **M**. A função computada pelo programa recursivo **P** na máquina **M** denotada por:

$$\langle \mathbf{P}, \mathbf{M} \rangle: \mathbf{X} \rightarrow \mathbf{Y}$$

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

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

onde:

$$\begin{aligned} D_0 &= E_0; \checkmark, \text{ sendo } E_0 \text{ expressão inicial de } \mathbf{P} \\ v_0 &= \pi_X(x) \\ D_n &= \checkmark \end{aligned}$$

Neste caso, tem-se que:

$$\langle \mathbf{P}, \mathbf{M} \rangle(x) = \pi_Y(v_n)$$

□

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:

$$\langle \text{qq_máquina}, M \rangle: X \rightarrow Y$$

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

exemplo 2.16 – Função computada por programa recursivo na máquina de um registrador

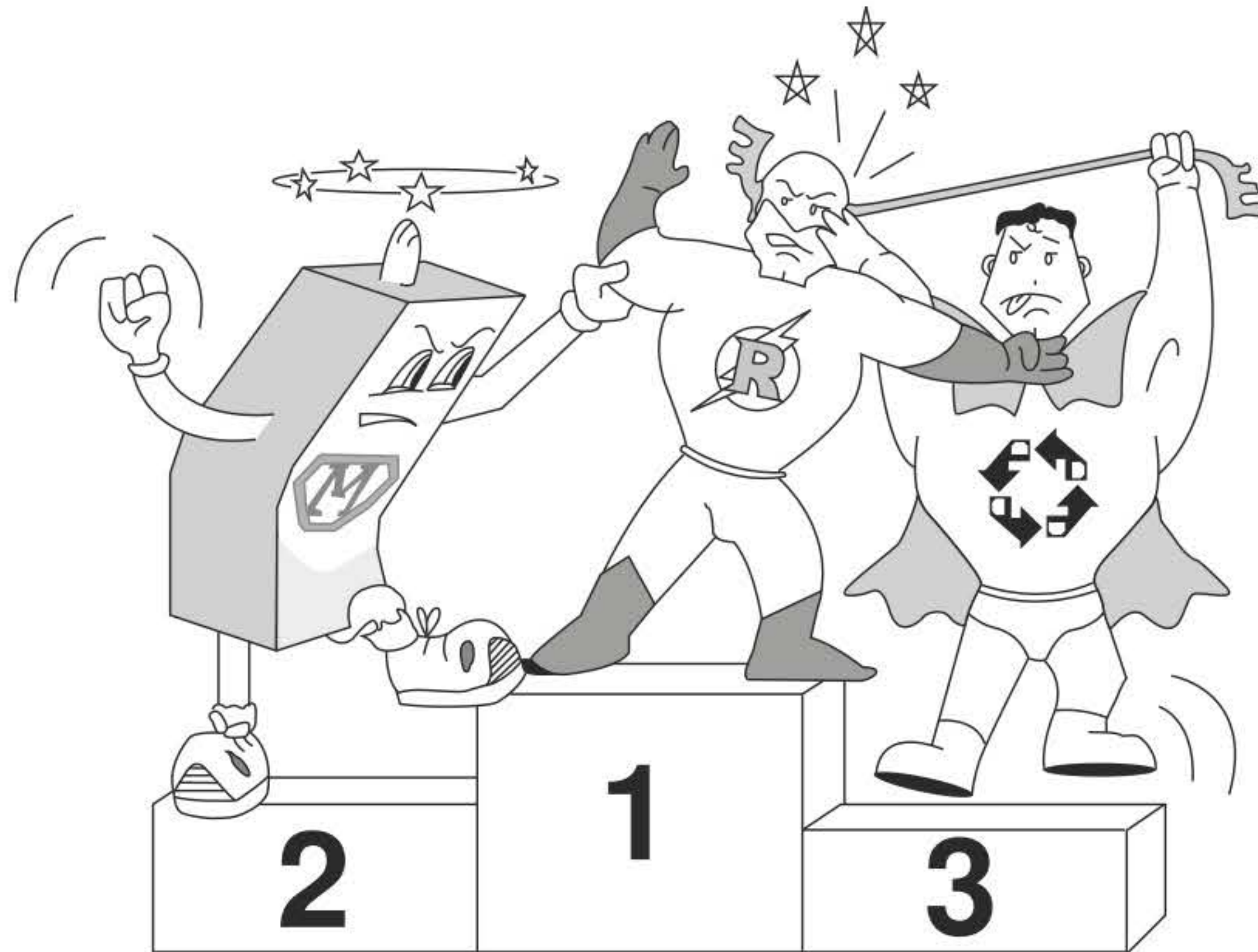
Considere o programa recursivo **duplica** (figura 2.16) para a máquina **um_reg** (figura 2.15). A correspondente função computada é:

$$\langle \text{duplica}, \text{um_reg} \rangle: \mathbb{N} \rightarrow \mathbb{N}$$

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

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

$\square$



2.4

→ equivalências de programas e máquinas

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

- a** *Relação de equivalência forte de programas.* Um par de programas pertence à relação se as correspondentes funções computadas coincidem para *qualquer* máquina;
- b** *Relação de 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 de 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 de 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 de 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:

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

2.4.1 equivalência forte de programas

definição 2.13 – Relação de equivalência forte de programas, programas fortemente equivalentes .

Sejam P e Q dois programas arbitrários, não necessariamente do mesmo tipo. Então o par (P, Q) está na *relação de 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

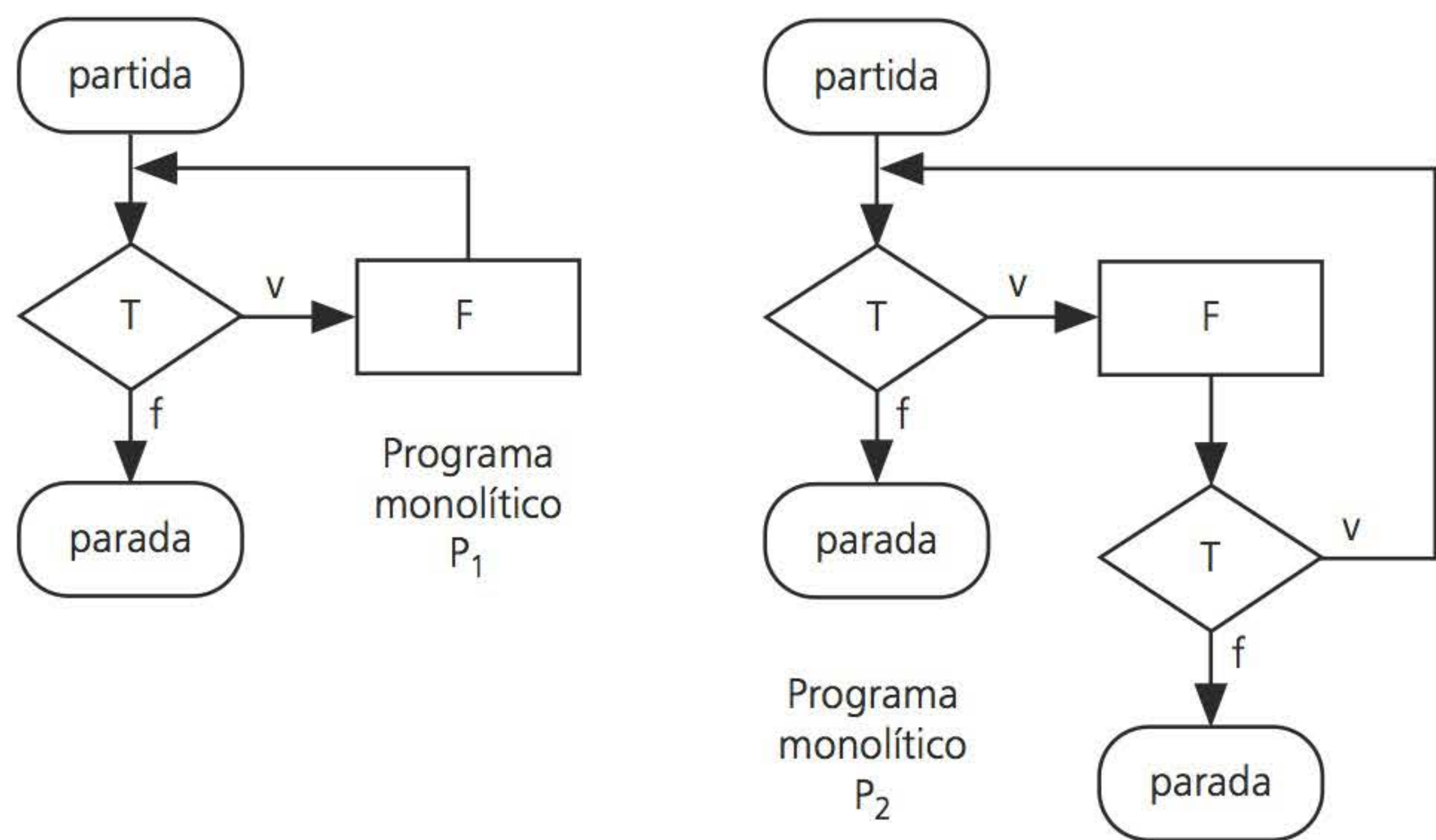
$$\langle P, M \rangle = \langle Q, M \rangle$$

Neste caso, P e Q são ditos *programas fortemente equivalentes*. □

É fácil verificar que a relação de 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 fortemente equivalentes

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 P_4 (abaixo) são todos fortemente equivalentes .



Programa iterativo P3
enquanto T faça (F)
Programa recursivo P4
P ₄ é R onde R def (se T então F;R senão ✓)

figura 2.18 Programas fortemente equivalentes .

Como ilustração, no que segue, é verificado que, de fato, os programas monolítico **P₁** e o recursivo **P₄** são fortemente equivalentes . O programa **P₁** reescrito na forma de instruções rotuladas é representado na figura 2.19.

Programa monolítico P1
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**, π_X , π_Y , Π_F , Π_T) uma máquina arbitrária e **x** ∈ **X** tal que $\pi_X(\mathbf{x}) = \mathbf{v}$. Então, se **⟨P₁, M⟩** é definida para **x**, a correspondente computação é dada pela cadeia:

$$(1, \mathbf{v})(2, \mathbf{v})(1, \pi_F(\mathbf{v}))(2, \pi_F(\mathbf{v}))(1, \pi_F^2(\mathbf{v}))(2, \pi_F^2(\mathbf{v}))\dots(1, \pi_F^n(\mathbf{v}))(3, \pi_F^n(\mathbf{v}))$$

supondo que n é o menor natural tal que $\pi_T(\pi_F^n(v)) = \text{falso}$. 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; ✓, π_F(v))
((se T então F; R senão ✓); ✓, π_F(v))
(F; R; ✓, π_F(v))

(R; ✓, π_F^2(v))
((se T então F; R senão ✓); ✓, π_F^2(v))
(F; R; ✓, π_F^2(v))

...
(R; ✓, π_F^n(v))
((se T então F; R senão ✓); ✓, π_F^n(v))
(✓; ✓, π_F^n(v))
(✓, π_F^n(v))

```

figura 2.20 Computação.

É importante que se considere a relação de 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 fortemente equivalentes têm a propriedade de que as mesmas operações são efetuadas na mesma ordem, independentemente do significado das mesmas (por que não diferentes testes ou operações ou, ainda, diferente ordem?);
- fornece subsídios para analisar a complexidade estrutural de programas. Por exemplo, analisando os programas monolíticos fortemente equivalentes 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 nos quais se verifica que:

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

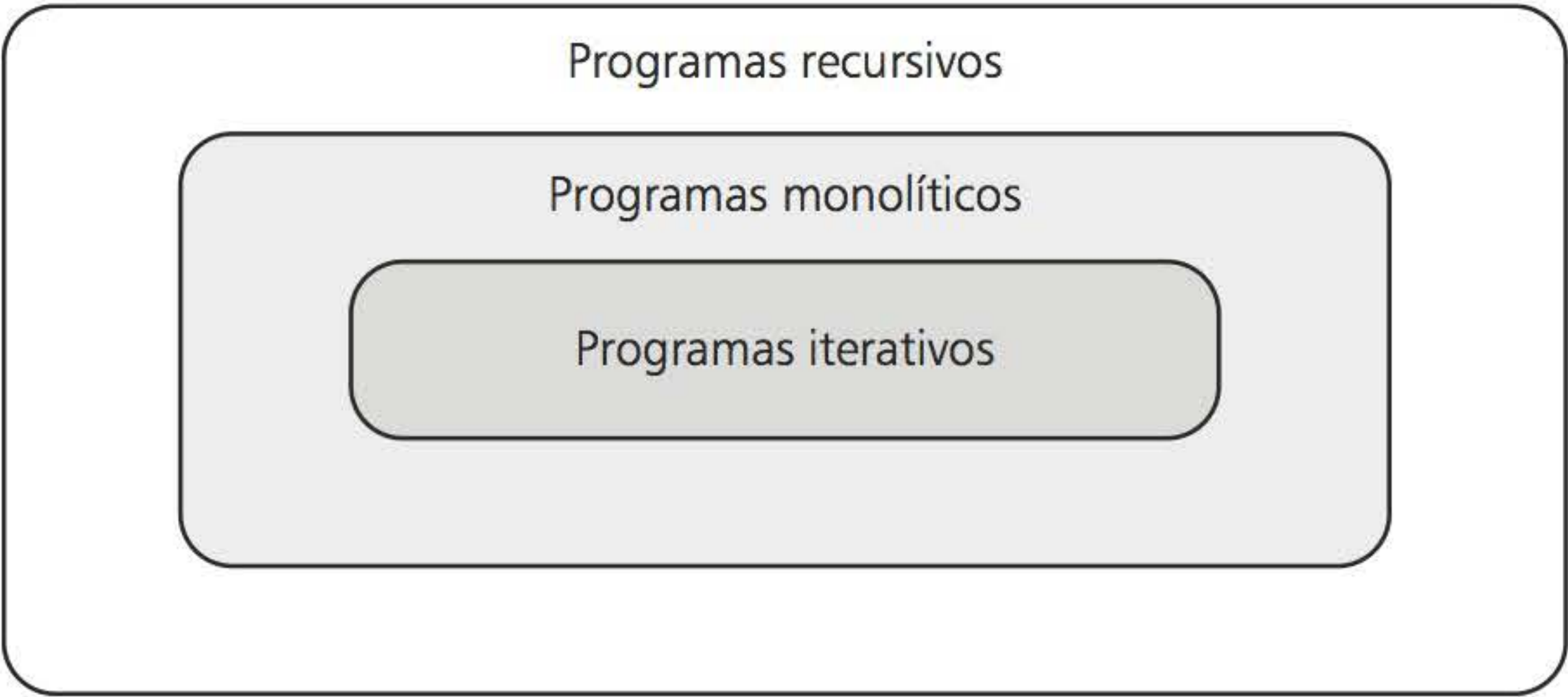


figura 2.21 Hierarquia induzida pela relação de equivalência forte de programas.

Entretanto, a inversa não necessariamente é verdadeira, ou seja, em relação à relação de 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, na qual as inclusões são próprias.

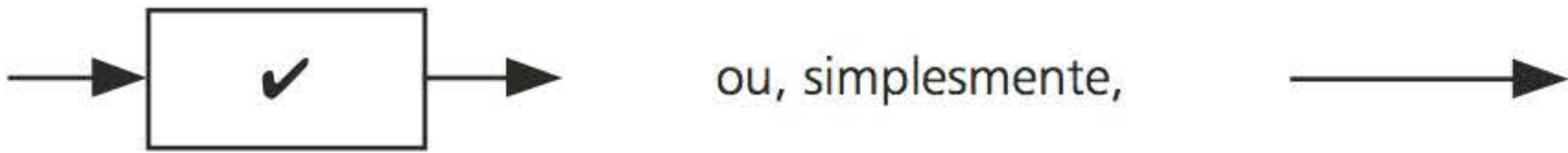
teorema 2.14 – Equivalência forte de programas: iterativo → monolítico

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:

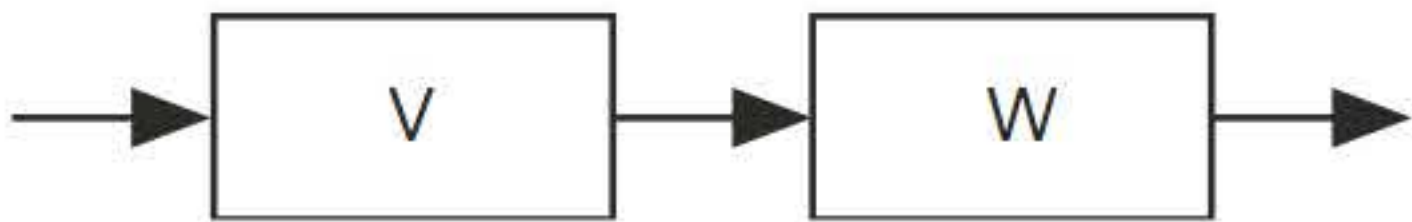


- b** Para cada identificador de operação F de P_i corresponde o seguinte fluxograma elementar:

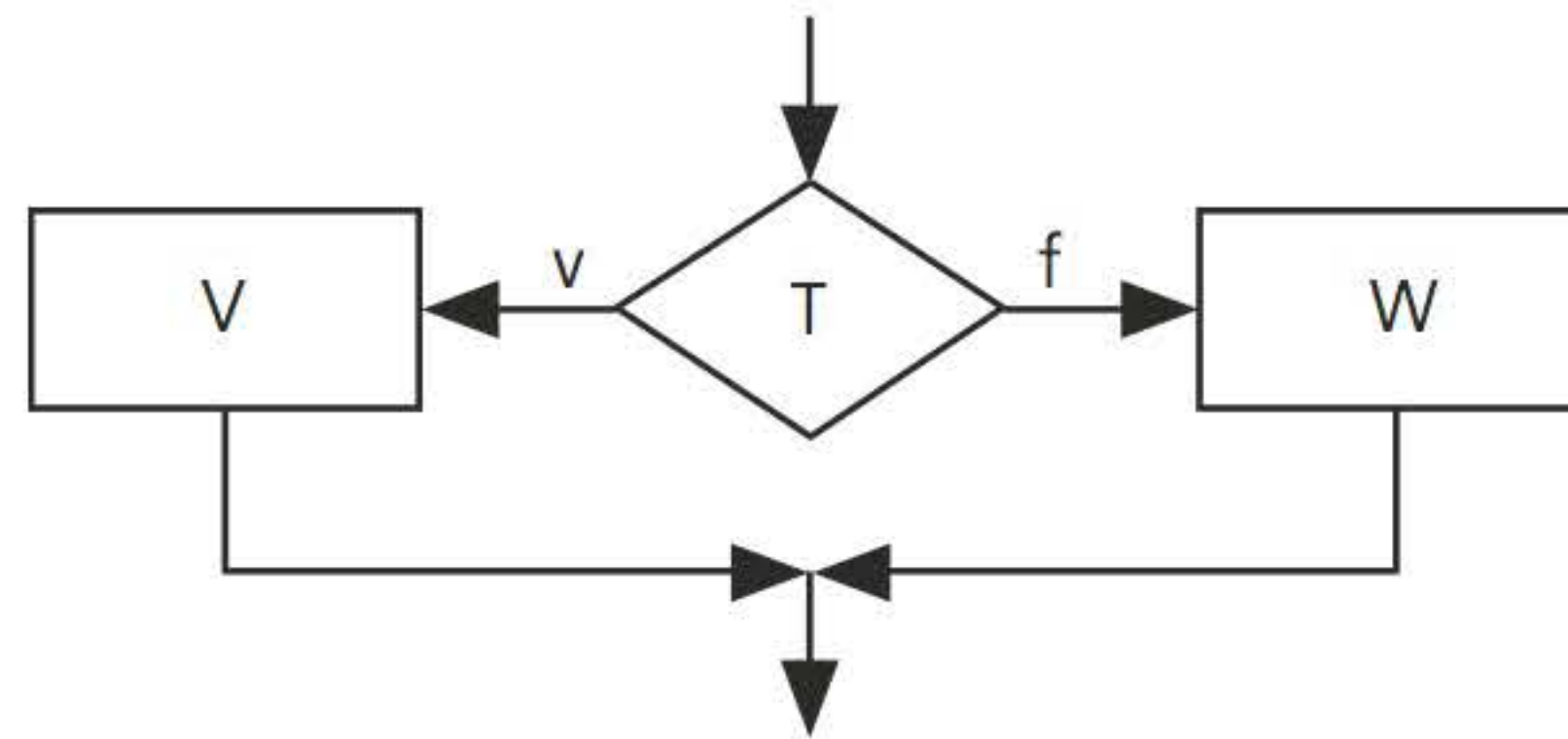


- c** Suponha que T é um identificador de teste e que V, W são programas iterativos usados na construção de P_i . Então, para cada um dos seguintes tipos de composição é apresentado o correspondente fluxograma de P_m :

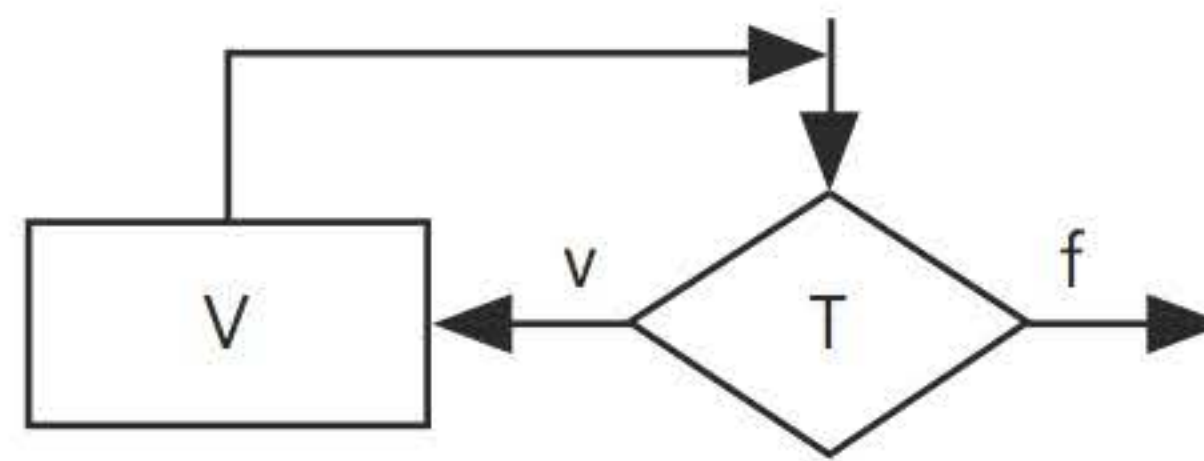
c.1) *Composição sequencial. $V;W$*



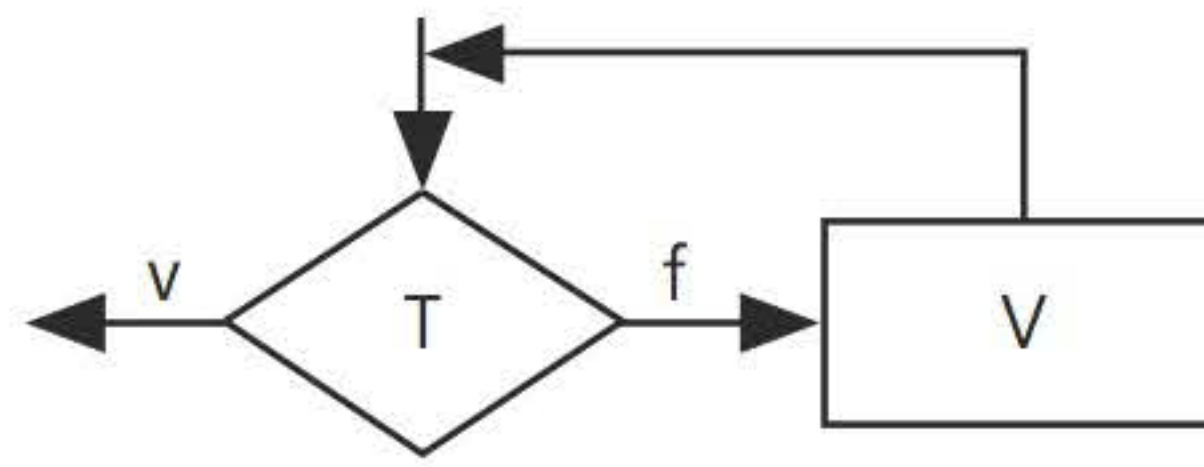
c.2) *Composição condicional. (se T então V senão W)*



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 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 $\rightarrow$ 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 $L = \{ r_1, r_2, \dots, r_n \}$ é o correspondente conjunto de rótulos. Suponha, sem perda de generalidade, que, em P_m , r_n é o *único* rótulo final (por que 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 $R_1 \text{ def } E_1, R_2 \text{ def } E_2, \dots, R_n \text{ def } \checkmark$

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

a *Operação.* Se a instrução rotulada r_k é da forma:

$r_k: \text{ faça } F \text{ vá_para } r_k'$

então E_k é a seguinte expressão de sub-rotinas:

$F; R_{k'}$

b *Teste.* Se a instrução rotulada r_k é da forma:

$$r_k: \text{ se } T \text{ então vá_para } r_k' \text{ senão vá_para } r_k''$$

então E_k é a seguinte expressão de sub-rotinas:

$$(\text{se } T \text{ então } R_k' \text{ senão } R_k'')$$

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

O seguinte corolário é consequência direta dos dois teoremas anteriores.

corolário 2.16 – Equivalência forte de programas: iterativo $\rightarrow$ 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 $\nrightarrow$ 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 contraexemplo, isto é, um programa recursivo que, para uma determinada máquina, não apresente programa monolítico equivalente.

Considere o programa recursivo **duplica** (figura 2.16) e a máquina **um_reg** (figura 2.15). A correspondente função computada $\langle \text{duplica}, \text{um_reg} \rangle: \mathbf{N} \rightarrow \mathbf{N}$ é tal que, para qualquer $n \in \mathbf{N}$:

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

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

$$\langle \text{duplica}, \text{um_reg} \rangle = \langle P_m, \text{um_reg} \rangle$$

Suponha que P_m é constituído de k operações **ad**. Suponha $n \in \mathbf{N}$ 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 . Nesse caso, o ciclo será executado indefinidamente.

Note que, com um único registrador, não é possível fazer o controle do ciclo e, ainda, utilizá-lo como acumulador. De fato, a cada iteração são somadas duas unidades no acumulador e o único teste para a máquina é zero, resultando em um ciclo executado indefinidamente.

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 fortemente equivalentes.

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

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 sequencial (e, neste caso, existe o correspondente programa monolítico).

co equivalente?). Note-se que, no contraexemplo usado na prova, a recursão de R não ocorre na última componente, como pode ser observado a seguir:

$R \text{ def } (\text{se } \mathbf{zero} \text{ então } \checkmark \text{ senão } \mathbf{sub}; R; \mathbf{ad}; \mathbf{ad})$

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

teorema 2.18 – Equivalência forte de programas: monolítico $\leftrightarrow$ 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.

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

$\langle \mathbf{par}, \mathbf{um_reg} \rangle(n) = 1$, se n é par;
 $\langle \mathbf{par}, \mathbf{um_reg} \rangle(n) = 0$, se n é ímpar.

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

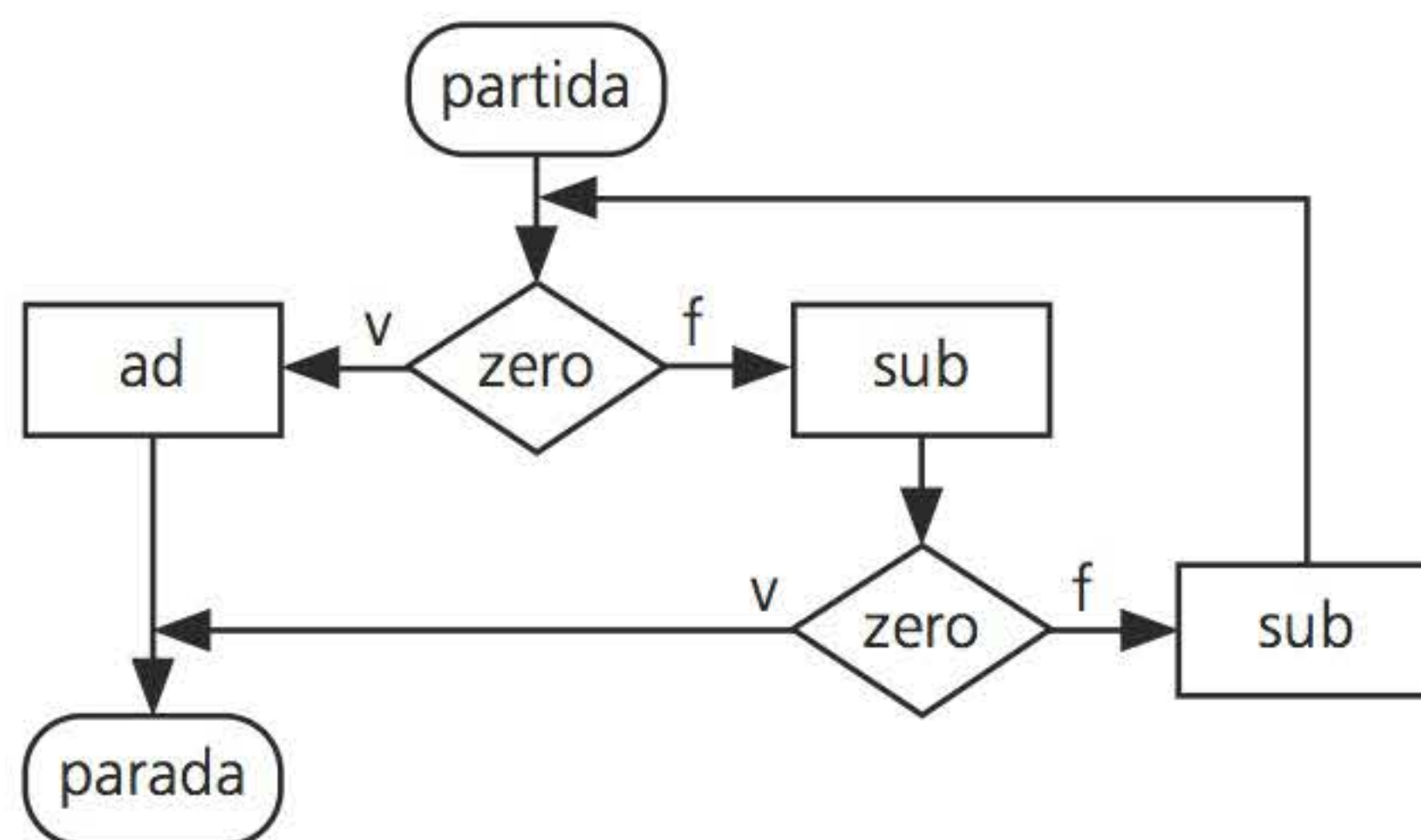


figura 2.22 Programa monolítico.

Suponha que existe um programa iterativo P_i que computa a mesma função, ou seja, que $\langle P_i, \mathbf{um_reg} \rangle: \mathbf{N} \rightarrow \mathbf{N}$ e:

$$\langle \mathbf{par}, \mathbf{um_reg} \rangle = \langle P_i, \mathbf{um_reg} \rangle$$

Suponha que P_i é constituído de k operações **sub**. Suponha $n \in \mathbf{N}$ 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 fortemente equivalentes.

Logo, não existe um programa iterativo fortemente equivalente ao programa monolítico **par**. $\square$

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, em qualquer máquina. É importante destacar que consideram a relação de 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 subsequentes), 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. $\square$

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 de 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 de 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*. $\square$

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 se prova, 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. Afirmar-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

$$\langle P, M \rangle = \langle Q, N \rangle \quad \square$$

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 necessário 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 por meio 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:

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

definição 2.22 – Relação de equivalência de máquinas

Sejam M e N duas máquinas arbitrárias. Então o par (M, N) está na relação de equivalência de máquina, se, e somente se:

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

2.5

→ conclusões

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, estudam-se três tipos de programas: monolítico, iterativo e recursivo. A partir dos conceitos de programa (uma estrutura algorítmica) e máquina (que dá semântica para as operações e testes do programa) são definidos: computação (a execução do programa em uma máquina) e função computada.

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

A relação de equivalência de programas fortemente equivalentes 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.

A relação entre os conceitos deste e demais capítulos é demonstrada na figura 2.23.

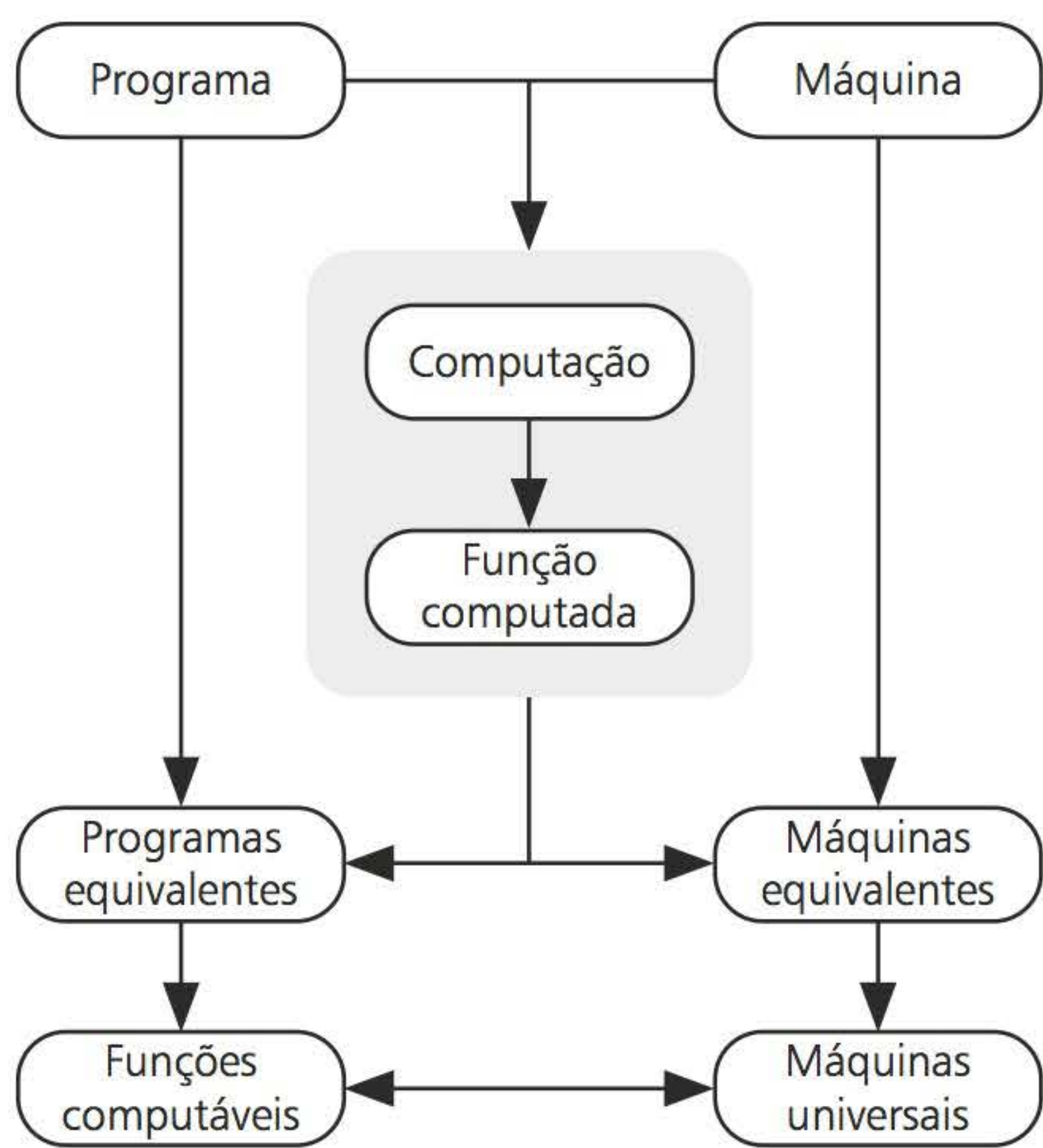


figura 2.23 Relação entre os conceitos deste e demais capítulos.

2.6

→ 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** $P_2 = (\{ r_1: \text{faça } \checkmark \text{ vá_para } r_2 \}, r_1)$
- b** Composição até (programa iterativo);
- c** Programa sem instrução alguma;
- d** Programa sem instrução de **parada**.

exercício 2.3 Em relação 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 $\checkmark$ constitui um programa iterativo?
- c** Por que se pode afirmar que:

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

exercício 2.4 Em relação à 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 $\langle \mathbf{W}, \mathbf{M} \rangle$ 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 Em relação 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 $\langle \mathbf{mon_b \leftarrow a}, \mathbf{dois_reg} \rangle$ 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 $\langle \text{comp_infinita}, \text{dois_reg} \rangle$ 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 $\langle \text{qq_máquina}, M \rangle$ é indefinida para qualquer entrada?

exercício 2.11 Traduza os programas monolíticos representados por fluxogramas em programas recursivos e simplifique se possível.

- a** representado na figura 2.24;
- b** representado na figura 2.25;
- c** representado na figura 2.26;
- d** representado na figura 2.27;
- e** representado na figura 2.28.

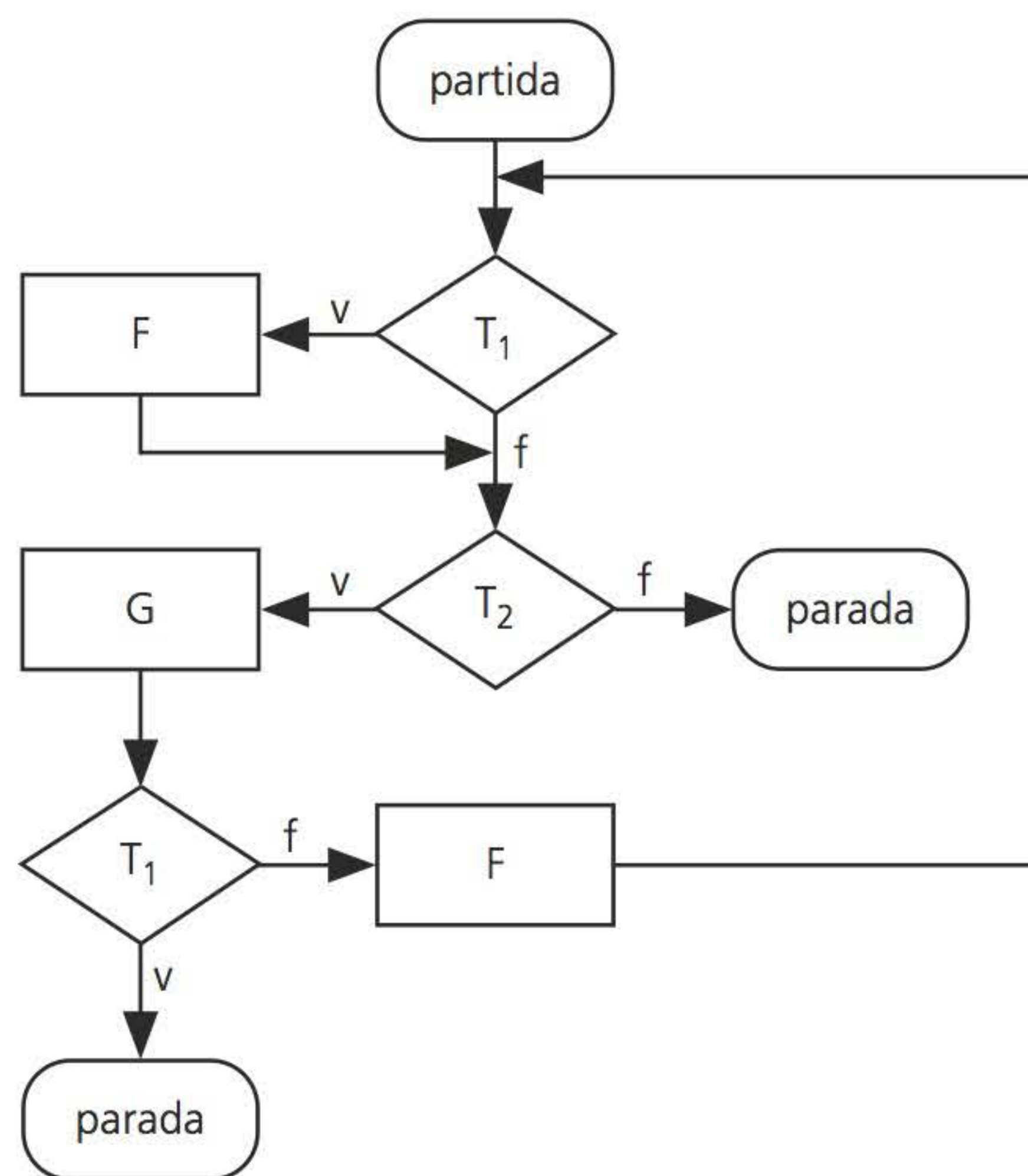


figura 2.24 Fluxograma 1.

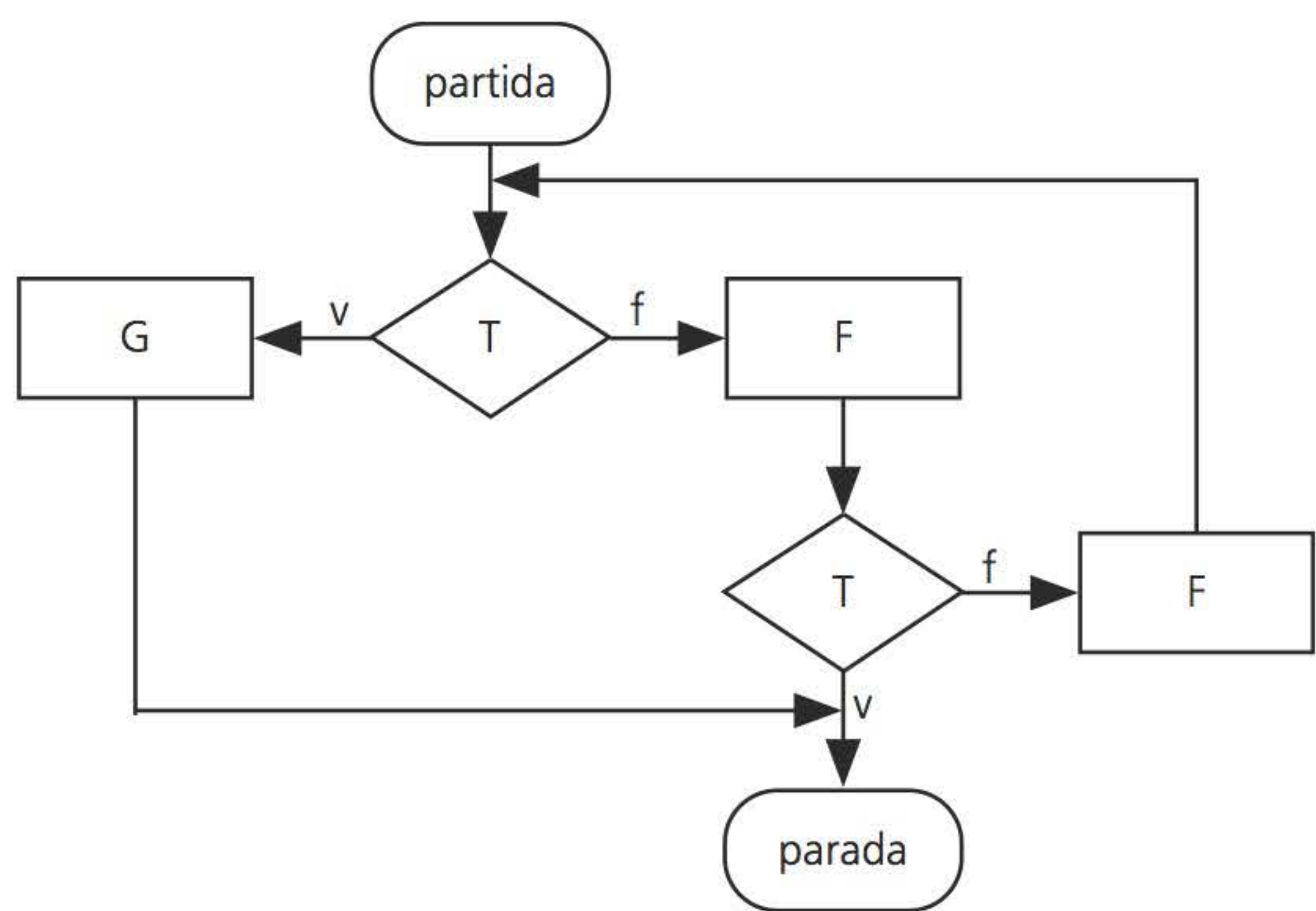


figura 2.25 Fluxograma 2.

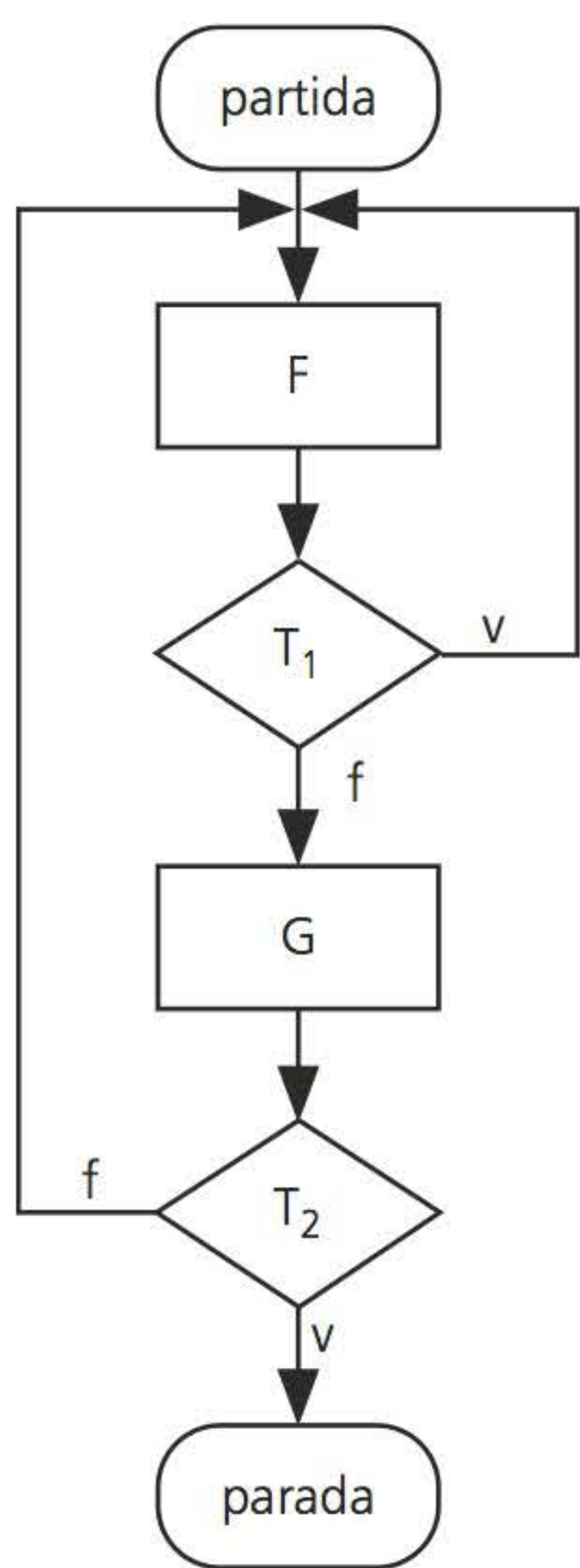


figura 2.26 Fluxograma 3.

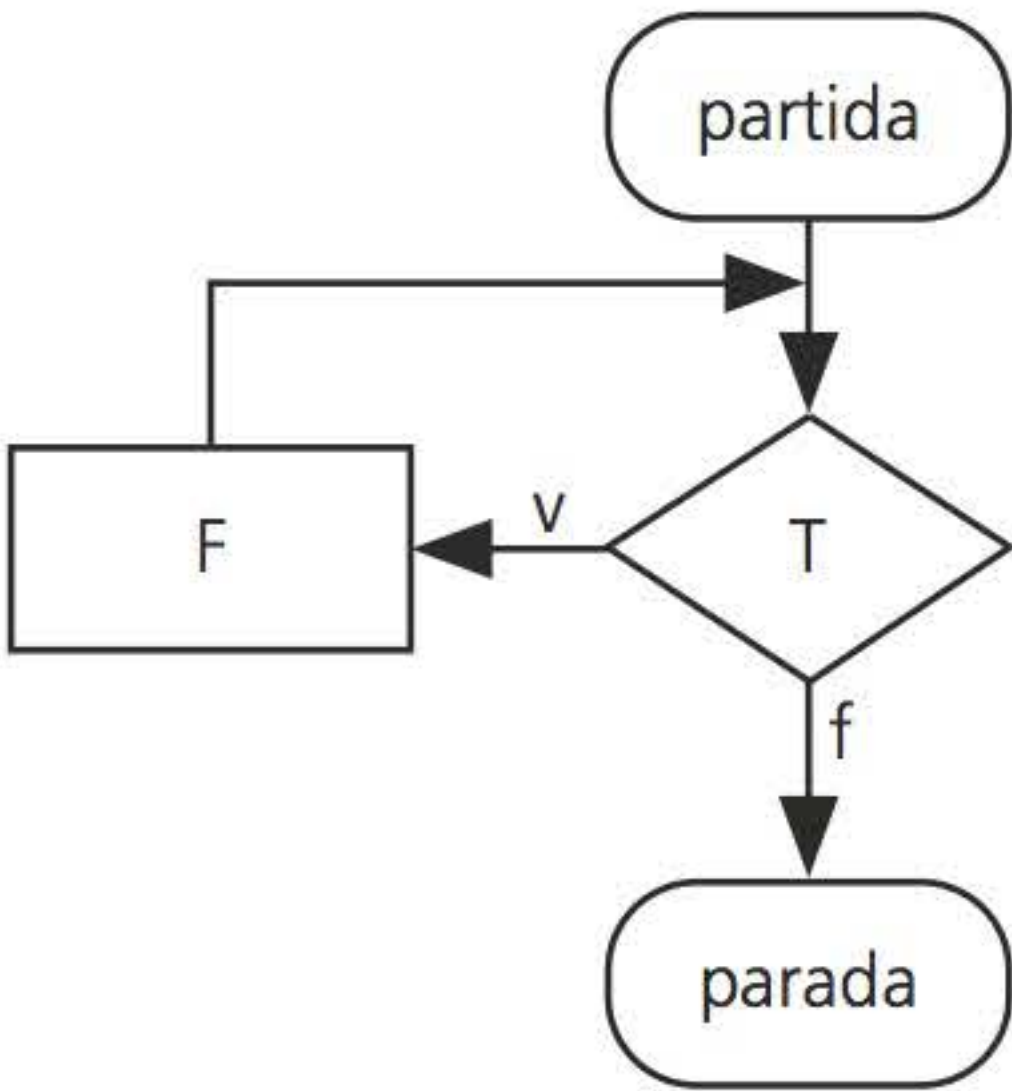


figura 2.27 Fluxograma 4.

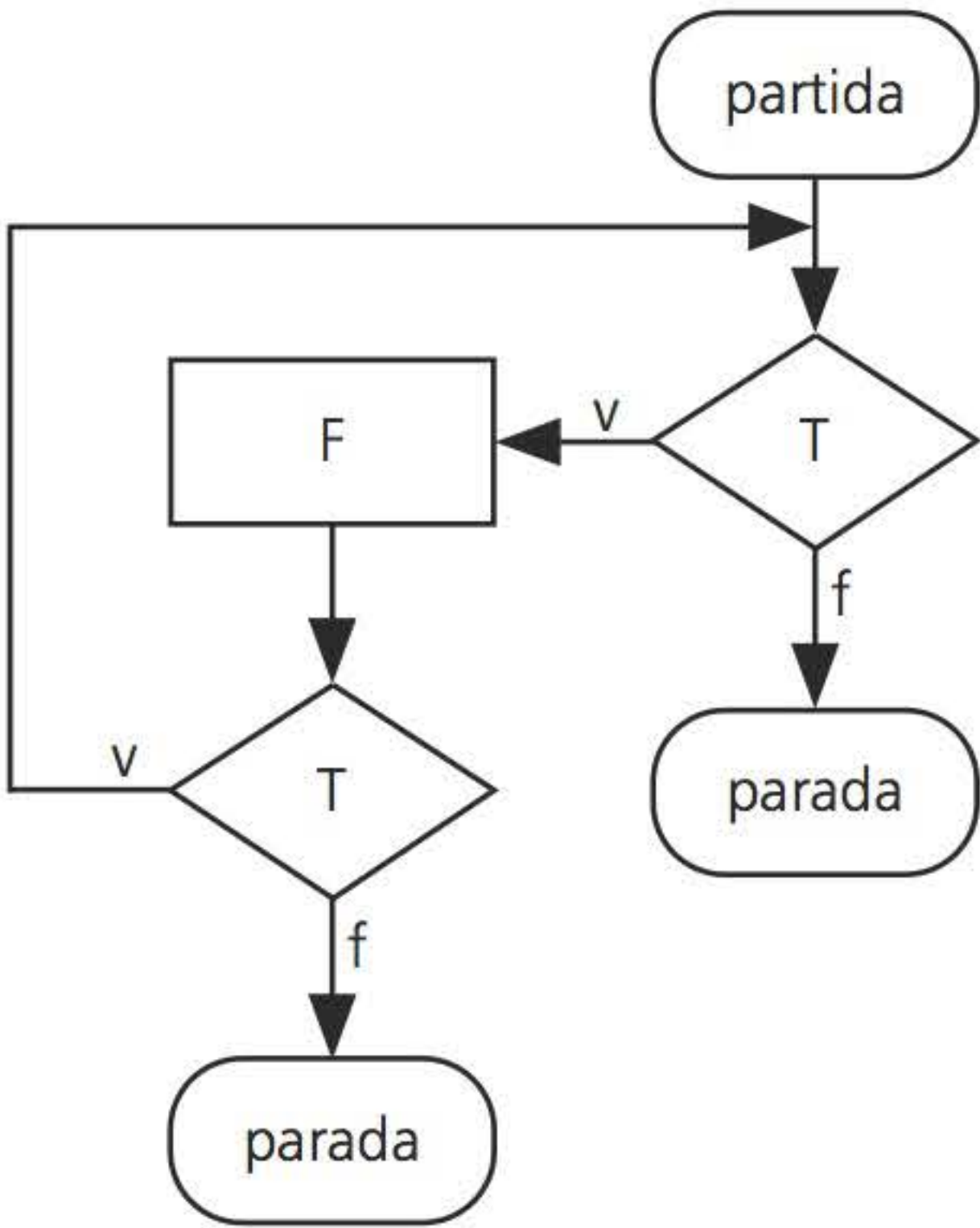


figura 2.28 Fluxograma 5.

exercício 2.12 Traduza o programa iterativo representado na figura 2.29 em programa monolítico, nas formas de:

- a** Fluxograma;
- b** Instruções rotuladas.


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

figura 2.29 Programa iterativo.

exercício 2.13 Traduza o programa recursivo representado na figura 2.30 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 2.30 Programa recursivo.

exercício 2.14 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 à máquina M_2 ?

exercício 2.15 Suponha que, na definição de uma máquina, sejam admitidas 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 2.16 Traduza os programas iterativos W_1 e W_2 definidos na figura 2.31 e na figura 2.32 para programas recursivos.

Programa iterativo W1

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

figura 2.31 Programa iterativo.

Programa iterativo W2

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

figura 2.32 Programa iterativo.

exercício 2.17 Sobre computação de programa recursivo em uma máquina, marque a afirmação *incorreta*:

- ☐ **a** É um histórico das instruções executadas e os correspondentes valores de memória;
- ☐ **b** Um teste e referência a uma sub-rotina não alteram o valor corrente da memória;
- ☐ **c** Em uma computação finita, a expressão (✓) ocorre no último par da cadeia e não ocorre em qualquer outro par;
- ☐ **d** Para um dado valor inicial de memória, a correspondente cadeia de computação é única, ou seja, a computação é não determinística;
- ☐ **e** Em uma computação infinita, expressão alguma da cadeia é (✓).

exercício 2.18 Qual das alternativas abaixo representa a hierarquia induzida pela relação de equivalência forte de programas?

- ☐ **a** Programas recursivos $\leq$ Programas monolíticos $\leq$ Programas iterativos;
- ☐ **b** Programas monolíticos $\leq$ Programas recursivos $\leq$ Programas iterativos;
- ☐ **c** Programas iterativos $\leq$ Programas recursivos $\leq$ Programas monolíticos;
- ☐ **d** Programas monolíticos $\leq$ Programas iterativos $\leq$ Programas recursivos;
- ☐ **e** Programas iterativos $\leq$ Programas monolíticos $\leq$ Programas recursivos.

exercício 2.19 Em relação à equivalência de programa, marque a alternativa *errada*:

- ☐ **a** Qualquer programa monolítico tem um fortemente equivalente recursivo;
- ☐ **b** Os programas recursivos podem simular qualquer programa iterativo;
- ☐ **c** Para todo programa iterativo, existe um monolítico fortemente equivalente;
- ☐ **d** Para todo programa recursivo, existe um iterativo fortemente equivalente.
- ☐ **e** Para todo programa recursivo, existe um equivalente monolítico na máquina Norma.

exercício 2.20 Dada uma máquina qualquer e considerando a análise dos diversos tipos de programa nesta máquina, marque a alternativa correta:

- ☐ **a** Qualquer programa iterativo pode ser traduzido para fluxograma;
- ☐ **b** Qualquer fluxograma pode ser traduzido para programa iterativo;
- ☐ **c** Qualquer programa recursivo “faz alguma coisa”, já que a operação vazia não constitui um programa recursivo;
- ☐ **d** Dado um programa monolítico, não necessariamente existe um correspondente programa recursivo fortemente equivalente;
- ☐ **e** Qualquer programa iterativo que possuir composição enquanto, denotada por “enquanto X faça Y”, não possui um correspondente programa recursivo fortemente equivalente.

exercício 2.21 Marque a alternativa correta:

- ☐ **a** Desvios incondicionais são permitidos em um programa iterativo;

- ☐ **b** Um programa descreve completamente uma computação, visto que contém a natureza das operações ou dos testes;
- ☐ **c** Para todo programa monolítico, existe necessariamente um programa iterativo fortemente equivalente ;
- ☐ **d** Duas máquinas são equivalentes se uma pode simular a outra e vice-versa;
- ☐ **e** Dois programas são fortemente equivalentes se suas funções computadas coincidem para uma determinada máquina.

exercício 2.22 Sobre equivalência forte de programas, analise as afirmações abaixo:

- I. Dois programas são fortemente equivalentes se, e somente se, os dois são do mesmo tipo e suas funções computadas são iguais;
- II. Nem todo programa recursivo possui um monolítico fortemente equivalente pelo fato de que o primeiro é mais genérico que o segundo;
- III. As funções computadas por programas fortemente equivalentes possuem a propriedade de que as mesmas operações podem ser efetuadas em qualquer ordem, independente do significado das mesmas, pois a saída é a mesma.

Marque a alternativa correta:

- ☐ **a** Apenas I;
- ☐ **b** Apenas II;
- ☐ **c** Apenas III;
- ☐ **d** Apenas II e III;
- ☐ **e** I, II e III

exercício 2.23 Analise as afirmações abaixo:

- I. A palavra “Elèveador” pode ser formada a partir do alfabeto $\Sigma = \{a, d, e, l, o, r, v, w\}$;
- II. Sejam **P** e **Q**, dois programas. Se para uma máquina **M**, $\langle P, M \rangle \neq \langle Q, M \rangle$, então os programas não são equivalentes nessa máquina;
- III. Um programa pode ser representado por meio de um fluxograma que contenha ✓ como instrução de parada;
- IV. Cabe à máquina suprir o significado, ou seja, dar semântica, aos identificadores das operações e dos testes de um programa.

Marque a alternativa correta:

- ☐ **a** Apenas II e III estão incorretas;
- ☐ **b** Apenas IV e II estão corretas;
- ☐ **c** Ou I e IV estão corretas ou III está correta;

- ☐ **d** Ou I e II estão corretas ou III e IV estão incorretas;
- ☐ **e** Ou I está correta ou a III está correta.

exercício 2.24 No contexto da equivalência forte, marque a alternativa correta:

- ☐ **a** Para qualquer programa monolítico, existe um programa iterativo equivalente;
- ☐ **b** Para qualquer programa recursivo, existe um programa monolítico equivalente;
- ☐ **c** Para qualquer programa recursivo, existe um programa iterativo equivalente;
- ☐ **d** Para qualquer programa monolítico, existe um programa recursivo equivalente;
- ☐ **e** Nenhuma das alternativas anteriores está correta.

exercício 2.25 A relação de equivalência forte entre programas permite:

- ☐ **a** Identificar programas iguais em diferentes classes de equivalência;
- ☐ **b** Identificar programas iguais cujas funções computadas são iguais;
- ☐ **c** Afirmar que todo programa monolítico tem um iterativo fortemente equivalente ;
- ☐ **d** Identificar diferentes programas em uma mesma classe de equivalência;
- ☐ **e** Afirmar que funções computadas por programas fortemente equivalentes têm a propriedade de que as mesmas operações são efetuadas numa ordem muito semelhante.

exercício 2.26 Sobre a definição de programas recursivos, analise as seguintes afirmações:

- I. O programa tem, em sua definição, uma expressão inicial, indicando qual a primeira rotina a ser computada.
- II. Um programa recursivo é capaz de representar um conjunto de problemas maior do que o programa iterativo é capaz.
- III. A instrução representa um teste em um programa recursivo: "R def se T então ✓ senão F;"
- IV. Um programa recursivo termina quando há uma chamada de sub-rotina que não está definida.
- V. A representação de um programa recursivo pode ser reduzida, substituindo-se as chamadas de sub-rotinas pelas expressões executadas por estas, desde que essas sub-rotinas não sejam testes.

Marque a alternativa correta:

- ☐ **a** Apenas I e III estão corretas;
- ☐ **b** Apenas I e V estão corretas;
- ☐ **c** Apenas I, III e V estão corretas;
- ☐ **d** Apenas I, II e V estão corretas;
- ☐ **e** Apenas II, III e IV estão corretas.

exercício 2.27 Sobre a relação de equivalência forte de programas, marque a alternativa errada:

- ☐ **a** As funções computadas por programas fortemente equivalentes têm a propriedade de que as mesmas operações são efetuadas na mesma ordem, independentemente do significado dos mesmos;
- ☐ **b** Analisando programas fortemente equivalentes, é possível dizer qual é o “mais otimizado” estruturalmente;
- ☐ **c** Como para qualquer programa iterativo existe um monolítico fortemente equivalente, a classe de programas monolíticos tem maior poder computacional que os iterativos;
- ☐ **d** Para qualquer programa iterativo, existe um programa recursivo fortemente equivalente;
- ☐ **e** Permite identificar diferentes programas cujas funções computadas coincidem, para qualquer máquina.

exercício 2.28 Sobre a equivalência de programas e máquinas, marque a alternativa correta:

- ☐ **a** A relação de equivalência entre duas máquinas ocorre quando ao menos uma delas é capaz de simular todas as funções da outra.
- ☐ **b** A relação de equivalência forte entre um programa monolítico P_m e outro iterativo P_i pode ser verificada, transformando-se cada um deles em programas recursivos fortemente equivalentes ($P_{r,m} = P_m$ e $P_{r,i} = P_i$) e, então, aplicando-se um método conhecido para testar equivalência entre $P_{r,m}$ e $P_{r,i}$, a fim de provar a relação entre todos ($P_{r,m} = P_m = P_{r,i} = P_i$).
- ☐ **c** Não existe qualquer método capaz de verificar a equivalência forte entre dois programas iterativos;
- ☐ **d** Se dois programas P e Q são equivalentes em uma máquina de traços M , então P e Q são fortemente equivalentes ;
- ☐ **e** A simulação forte entre máquinas deve ser feita usando programas iguais;

exercício 2.29 Com relação à equivalência de programas, marque a alternativa que descreve corretamente a igualdade de funções parciais, tendo-se duas funções parciais $f, g: X \rightarrow Y$ para cada $x \in X$:

- ☐ **a** Ambas as funções são definidas e $f(x) = g(x)$;

- b** Ou ambas são definidas e $f(x)=g(x)$ ou ambas $f(x)$ e $g(x)$ são indefinidas;
- c** Ou ambas são definidas e $f(x)=g(x)$ ou pelo menos uma delas, $f(x)$ ou $g(x)$, é indefinida;
- d** Funções parciais iguais são encontradas apenas em programas iterativos, pois são os únicos que admitem funções;
- e** Nenhuma das alternativas anteriores está correta.

exercício 2.30 Sobre equivalência de programas e máquinas, analise as seguintes afirmações:

- I. Se um programa P_1 é equivalente a um programa P_2 em uma máquina de traços, então eles são fortemente equivalentes ;
- II. Seja R um programa recursivo, M um programa monolítico e I um programa iterativo. Não é verdade que M é mais geral que I , que, por sua vez, é mais geral que R ;
- III. Sabendo que a máquina M_1 pode simular a máquina M_2 , então M_1 é equivalente a M_2 .

Marque a alternativa correta:

- a** Apenas I está correta;
- b** Apenas I e II estão corretas;
- c** Apenas I e III estão corretas;
- d** Apenas III está correta;
- e** Apenas II e III estão corretas.

Termos-chaves

computação, p. 48

função computada, p. 54

instruções rotuladas, p. 39

máquina, p. 45

programa, p. 35

programa iterativo, p. 41

programa monolítico, p.37

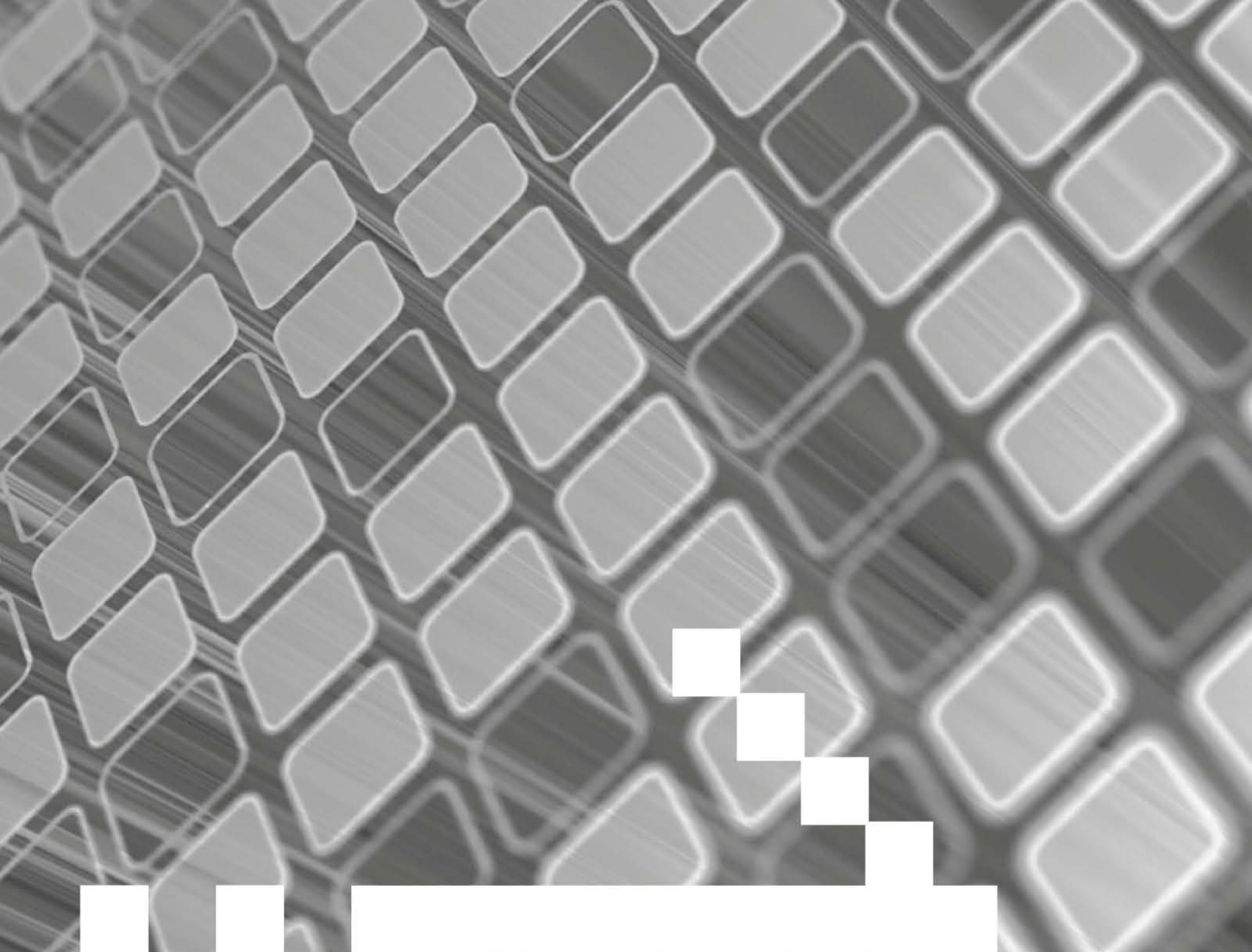
programa recursivo, p. 44

programas fortemente equivalentes, p. 58

relação de equivalência de máquinas, p. 57

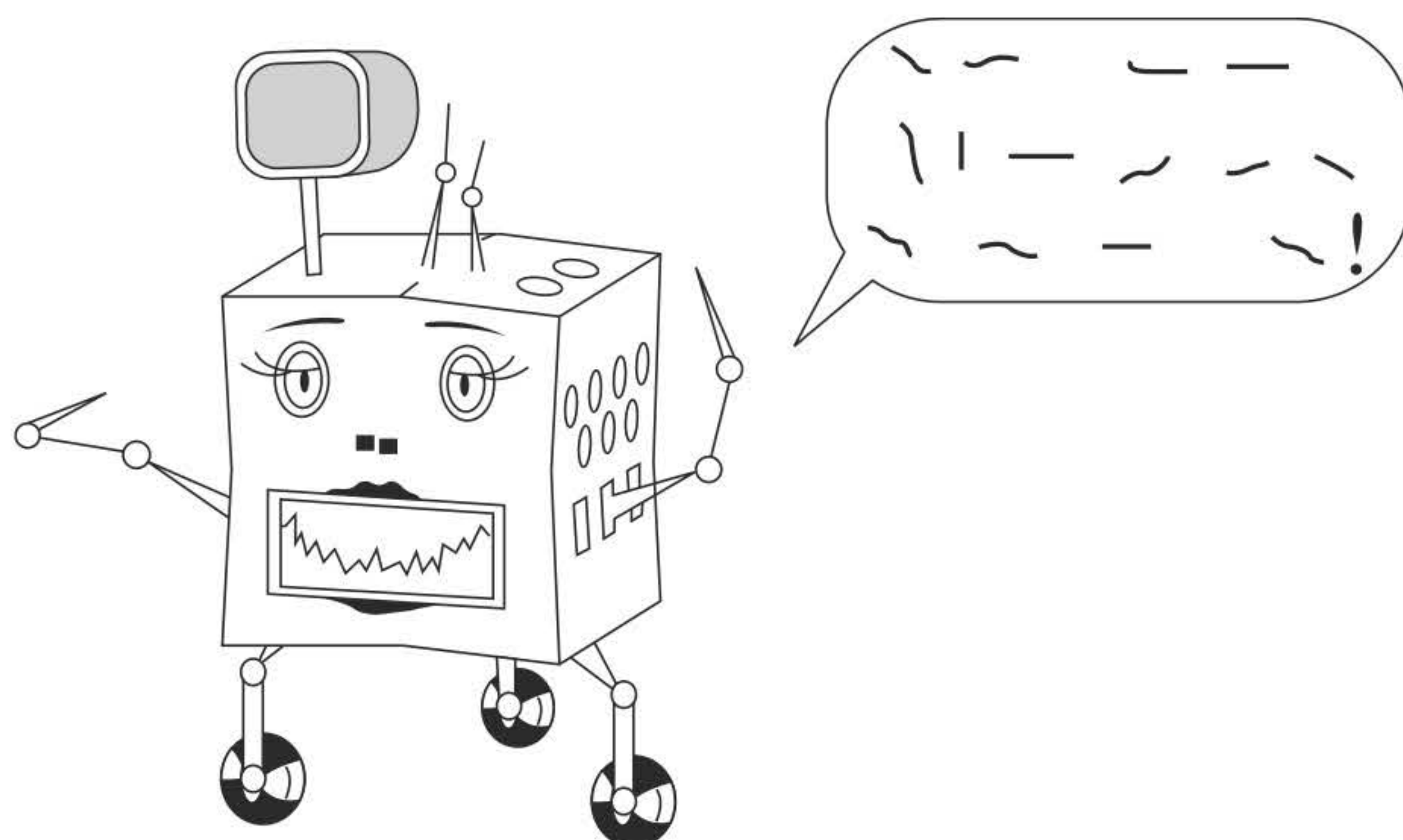
relação de equivalência de programas, p. 57

relação de equivalência forte, p. 57



verificação da equivalência forte de programas

- ■ Estuda formas e condições para determinar se dois programas são fortemente equivalentes (equivalentes em qualquer máquina). Um algoritmo para decidir a equivalência forte entre programas monolíticos é apresentado. E, como todo o programa iterativo possui um programa monolítico fortemente equivalente, o mesmo pode ser afirmado para programas iterativos. Entretanto, até o momento, não é conhecido se existe um algoritmo análogo para programas recursivos. A equivalência forte de programas monolíticos é construída usando a máquina de traços (produz um rastro ou histórico das operações do programa) e instruções rotuladas compostas (forma alternativa de descrever programas monolíticos).



Neste capítulo, são estudadas formas de se determinar se dois programas, sob determinadas condições, são ou não fortemente equivalentes. Uma vez que programas iterativos e monolíticos podem ser transformados em programas recursivos fortemente equivalentes, 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 se sabe se o problema da equivalência forte de programas recursivos é solucionável.

Na falta de um algoritmo genérico para programas recursivos, procuram-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 fortemente equivalente, o mesmo pode ser afirmado para programas iterativos.

A verificação de que dois programas monolíticos são fortemente equivalentes é baseada no trabalho de R. Bird (1976) e 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 fortemente equivalentes 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):

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

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

3.1

→ máquina de traços

Uma máquina de traços não executa as operações propriamente ditas, 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 fortemente equivalentes. A noção de traço também é importante no estudo dos modelos de concorrência e da semântica formal, os quais não são objetivos desta publicação.

definição 3.1 – 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 definido sobre o alfabeto de identificadores de operações $Op = \{ F, G, \dots \}$. Corresponde ao conjunto de valores de memória, de entrada e de saída;
- id_{Op^*} função identidade em Op^* . Corresponde às funções de entrada e de saída;
- Π_F conjunto de interpretações de operações onde, para cada identificador de operação F de Op , a interpretação $\pi_F: Op^* \rightarrow Op^*$ 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 :

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

□

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 3.2 – Função induzida por um traço em uma máquina

Sejam $M = (V, X, Y, \pi_X, \pi_Y, \Pi_F, \Pi_T)$ uma máquina, $Op = \{ F, G, H, \dots \}$ o conjunto de operações interpretadas em Π_F e $w = FG\dots H$ 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):

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

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

$$[w_x, M] = \pi_H \circ \dots \circ \pi_G \circ \pi_F \circ \pi_X(x) \quad \square$$

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 3.3 – Equivalência forte de programas $\leftrightarrow$ equivalência de programas em máquinas de traços

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

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

→ **Prova:**

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

($\leftarrow$) A prova que segue é por absurdo e 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 fortemente equivalentes. Então existe uma máquina $N = (V, X, Y, \pi_{X_N}, \pi_{Y_N}, \Pi_{F_N}, \Pi_{T_N})$ onde $Op = \{ F, G, H, \dots \}$ 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 de 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 teste correspondente 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 , tem-se que:

$$\begin{array}{ll} \langle R, M \rangle(\varepsilon) & \text{resulta em um traço (M é uma máquina de traços)} \\ [\langle R, M \rangle(\varepsilon), N] & \text{é a função induzida por } \langle R, M \rangle(\varepsilon) \text{ em } N \text{ para a entrada } x \end{array}$$

Neste contexto, prova-se que:

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

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

$$\begin{aligned} \langle P, N \rangle(x) \neq \langle Q, N \rangle(x) &\Rightarrow \\ \Rightarrow \pi_{Y_N} \circ [\langle P, M \rangle(\varepsilon), N] &\neq \pi_{Y_N} \circ [\langle Q, M \rangle(\varepsilon), N] \Rightarrow \\ \Rightarrow [\langle P, M \rangle(\varepsilon), N] &\neq [\langle Q, M \rangle(\varepsilon), N] \Rightarrow \\ \Rightarrow \langle P, M \rangle(\varepsilon) &\neq \langle Q, M \rangle(\varepsilon) \end{aligned} \quad \begin{array}{l} \pi_{Y_N} \text{ é função} \\ \text{(por quê?)} \end{array}$$

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 fortemente equivalentes.

Logo, **P** e **Q** são fortemente equivalentes.

A prova que, de fato, para qualquer programa monolítico $R = (I, r_0)$, tem-se que $\langle R, N \rangle(x) = \pi_{Y_N} \circ [\langle R, M \rangle(\varepsilon), N]$ é por indução no 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 $\pi_{Y_N} \circ [\langle R, M \rangle(\varepsilon), N]$, respectivamente.

a Base de indução. Suponha $n = 0$. Então:

$$\begin{aligned} r_0 &= m_0 \\ v_0 &= \pi_{X_N}(x) = [\varepsilon, N] = [w_0, N] \end{aligned}$$

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

$$\begin{aligned} r_n &= m_n \\ v_n &= [w_n, N] \end{aligned}$$

c Passo de indução. Existem dois casos possíveis:

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

$$\begin{aligned} r_{k+1} &= m_{k+1} = r' \\ v_{k+1} &= \pi_{F_N}(v_k) = \pi_{F_N}([w_k, N]) = [w_k F, N] = [w_{k+1}, N] = w_{k+1} = w_k F \end{aligned}$$

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

$$\begin{aligned} r_{k+1} &= m_{k+1} \text{ pois } T_M(w_k) = T_N([w_k, N]) = T_N(v_k) \\ v_{k+1} &= v_k = [w_k, N] = [w_{k+1}, N] = w_{k+1} = w_k F \end{aligned}$$

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

A justificativa do seguinte corolário é sugerida como exercício:

corolário 3.4 – Equivalência forte de programas $\leftrightarrow$ 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**, $\langle P, M \rangle(\varepsilon) = \langle Q, M \rangle(\varepsilon)$ □

3.2

→ 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 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 3.5 – *Instrução rotulada composta*, instrução etiquetada composta

Uma *instrução rotulada composta* ou *instrução etiquetada composta* é uma sequência 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* ou *etiquetas sucessoras* de r_1
- r_1 é dito *rótulo antecessor* ou *etiqueta antecessora* de r_2 e r_3 □

definição 3.6 – Programa monolítico com instruções rotuladas compostas

Um *programa monolítico com instruções rotuladas compostas* ou *programa monolítico com instruções etiquetadas compostas* **P** é um par ordenado:

$$\mathbf{P} = (\mathbf{I}, r)$$

no qual:

- a** $\mathbf{I}$ é um conjunto finito de instruções rotuladas compostas, sendo que não existem duas instruções diferentes com um mesmo rótulo;
- b** r é um rótulo distinguido denominado *rótulo inicial* ou *etiqueta inicial* que identifica a instrução rotulada inicial em $\mathbf{I}$;
- c** um rótulo é dito *rótulo final* ou *etiqueta final* se é referenciado por pelo menos uma instrução de $\mathbf{I}$, mas não rotula qualquer instrução de $\mathbf{I}$. □

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 3.1. 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:

$$r_1: (\mathbf{F}, r_2), (\mathbf{G}, r_3)$$

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

definição 3.7 – Algoritmo: fluxograma $\rightarrow$ 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 3.1, 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 3.1, a correspondente instrução rotulada composta é como segue:

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

b.3) *Parada*. Para uma parada como na figura 3.1, a correspondente instrução rotulada composta é como segue:

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

b.4) *Testes encadeados*. No caso de testes encadeados como na figura 3.1, 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 não é atingível, 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 3.1, 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.

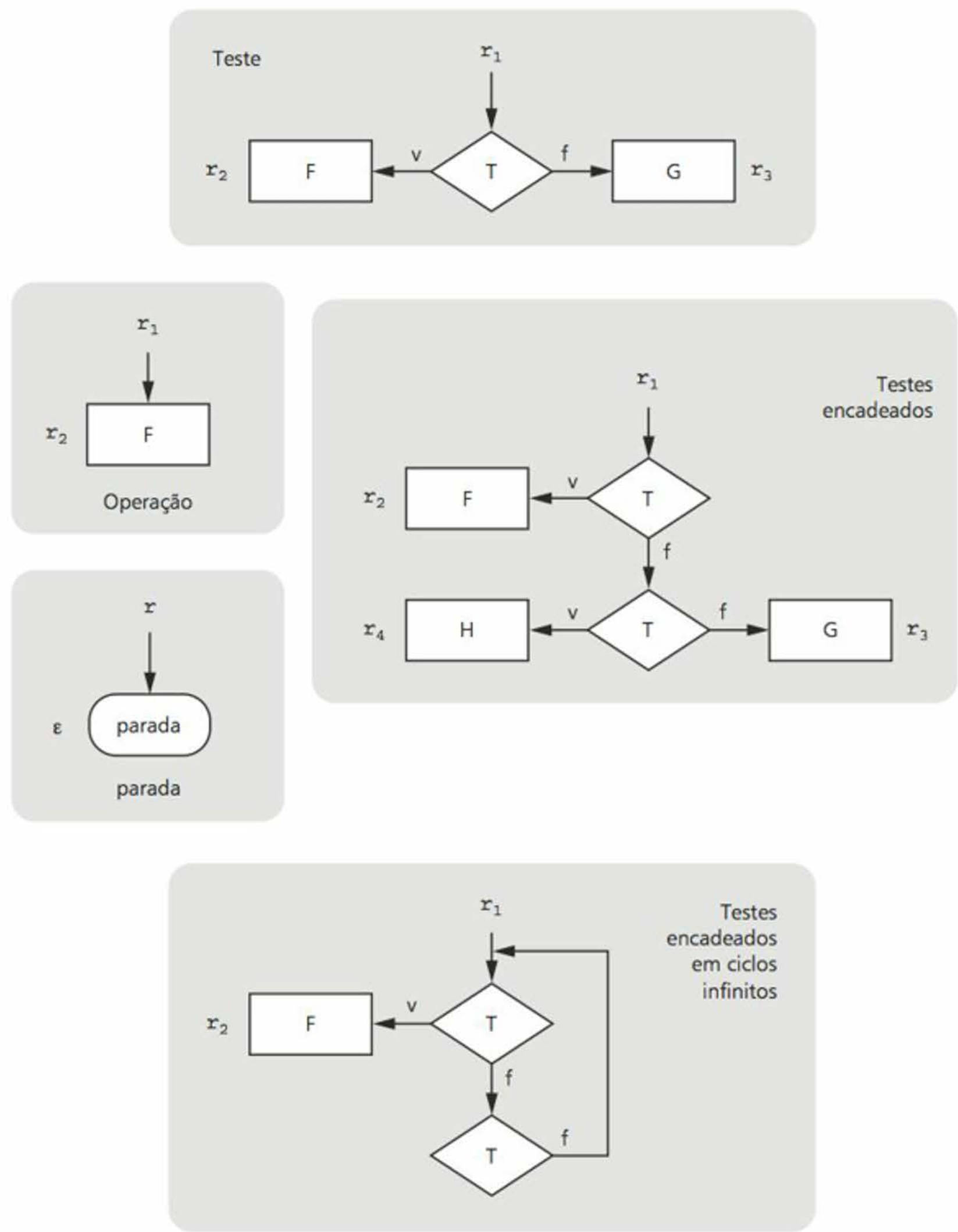


figura 3.1 Instruções rotuladas compostas.

exemplo 3.1 – Algoritmo: fluxograma → rotuladas compostas

Considere o programa monolítico especificado na forma de fluxograma na figura 3.2, cujos nós já estão rotulados. O correspondente programa com instruções rotuladas compostas é representado na figura 3.3, 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 o nó **parada**. Entretanto, somente um é representado no conjunto de instruções rotuladas compostas. O segundo caminho é impossível, pois se trata 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). □

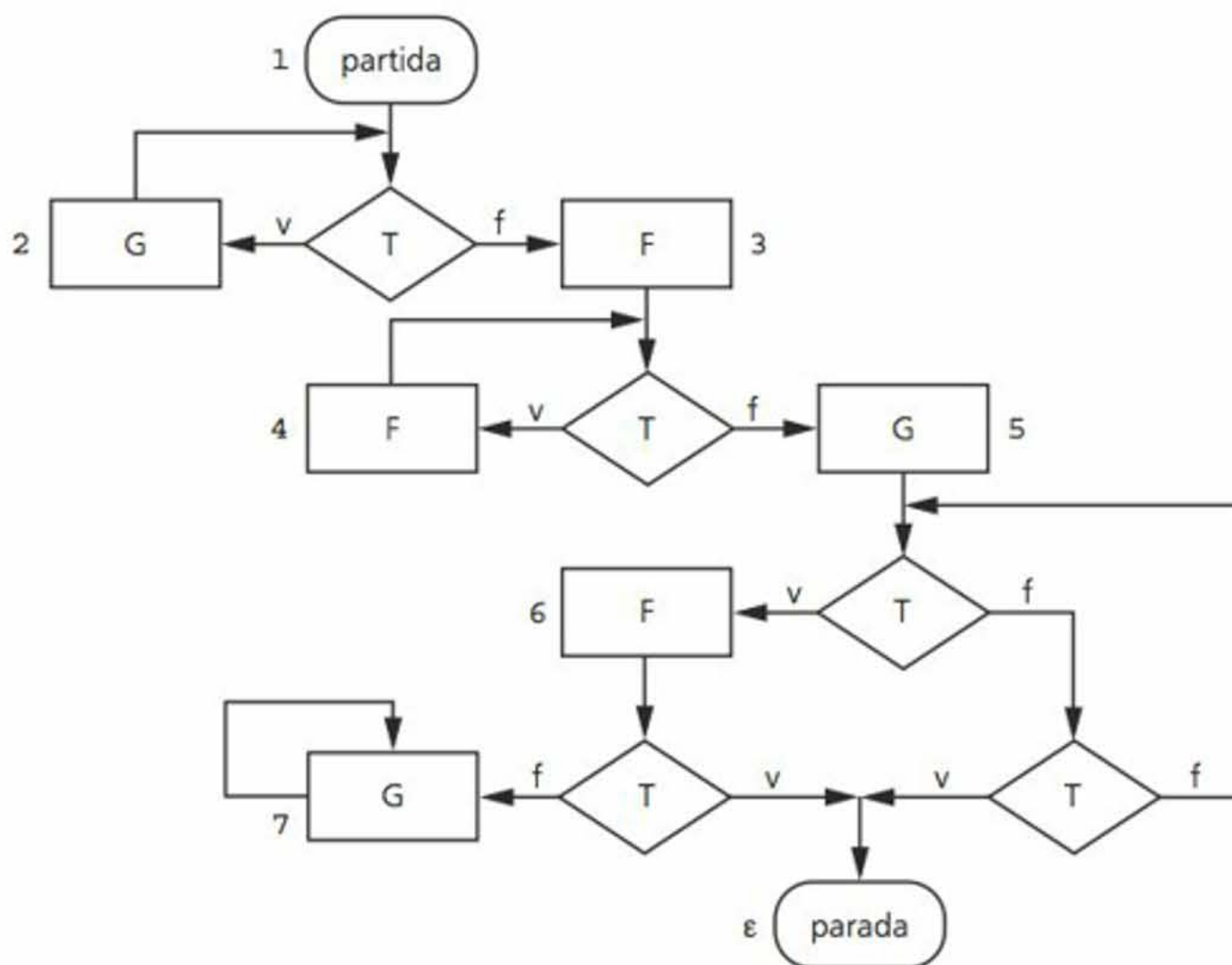


figura 3.2 Fluxograma Q com os nós rotulados.

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 3.3 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 fortemente equivalentes, excetuando-se para computações finitas, devido ao nó de **parada**.

lema 3.8 – 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:

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

→ **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. □

3.3

→ 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 3.8 - Equivalência forte: fluxograma → rotuladas compostas.

corolário 3.9 – Equivalência forte: união disjunta

Sejam $Q = (I_Q, q)$ e $R = (I_R, r)$ dois programas 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:

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

□

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

- *cadeia de conjunto*: sequência 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 fortemente equivalentes*: o algoritmo de verificação se P_q e P_r são fortemente equivalentes baseia-se em rótulos consistentes.

definição 3.10 – Cadeia de conjuntos, cadeia finita de conjuntos, limite de uma cadeia finita de conjuntos

Uma sequência de conjuntos $A_0 A_1 \dots$ 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:

$$\lim A_k = A_n$$

□

O lema a seguir fornece um algoritmo para determinar se existem ciclos infinitos em um conjunto de instruções rotuladas compostas. A ideia 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.

lema 3.11 – Identificação de ciclos infinitos em programa monolítico

Seja I é um conjunto de n instruções rotuladas compostas. Seja $A_0 A_1 \dots$ uma sequência de conjuntos de rótulos indutivamente definida como segue:

$$A_0 = \{ \varepsilon \}$$

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

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

$$(I, x) = (I, \omega) \quad \text{se, e somente se,} \quad x \notin \lim A_k \quad \square$$

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

exemplo 3.2 – Identificação de ciclos infinitos em programa monolítico

Considere o conjunto I de instruções rotuladas compostas representado na figura 3.3. A correspondente cadeia finita de conjuntos é como segue:

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

Logo:

$$\begin{aligned} \lim A_k &= \{ 1, 2, 3, 4, 5, 6, \varepsilon \} \\ (I, 7) &= (I, \omega), \text{ pois } 7 \notin \lim A_k \end{aligned} \quad \square$$

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

definição 3.12 – Algoritmo de simplificação de ciclos infinitos

Seja I um conjunto finito de instruções rotuladas compostas. O *algoritmo de simplificação de ciclos infinitos* é como segue:

- a** Determina-se a correspondente cadeia finita de conjuntos $A_0A_1\dots$ como no lema acima;
 - b** Para qualquer rótulo x de instrução de I tal que $x \notin \lim A_k$, tem-se que
 - a instrução rotulada por x é excluída
 - toda referência a pares da forma (F, x) em I é substituída por (ciclo, ω)
 - $I = I \cup \{ \omega: (\text{ciclo}, \omega), (\text{ciclo}, \omega) \}$
-

exemplo 3.3 – Simplificação de ciclos infinitos em programa monolítico

Considere o conjunto de instruções rotuladas compostas representado na figura 3.3, o qual corresponde ao fluxograma representado na figura 3.2. O correspondente conjunto de instruções rotuladas compostas simplificadas é representado na figura 3.4. □


```

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 3.4 Programa Q: Conjunto de instruções rotuladas compostas e simplificadas.

lema 3.13 – 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:   (F1, r1), (F2, r2)
s:   (G1, s1), (G2, s2)

```

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

- a** $F_1 = G_1$ e $F_2 = G_2$
- b** r_1 e s_1 são rótulos consistentes;
- c** r_2 e s_2 são rótulos consistentes. □

O lema acima induz a seguinte definição.

definição 3.14 – Rótulos fortemente equivalentes

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 fortemente equivalentes* se, e somente se:

- ou $r = s = \varepsilon$;
- ou r e s são ambos diferentes de ε e consistentes. □

teorema 3.15 – Determinação de rótulos fortemente equivalentes

Seja I um conjunto de n instruções compostas e simplificadas. Sejam r e s dois rótulos de instruções de I . Define-se, indutivamente, a sequência de conjuntos $B_0 B_1 \dots$ como segue:

$$B_0 = \{ (r, s) \}$$

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

Então $B_0 B_1 \dots$ é uma sequência de conjuntos que converge para o conjunto vazio e r, s são rótulos fortemente equivalentes 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 3.9 - Equivalência forte: união disjunta.

definição 3.16 – 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 simplificadas. 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 $P_q = P_r$ se, e somente se, $Q = R$;

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

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

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

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

Note-se que, no algoritmo acima, a construção dos conjuntos referida no teorema 3.15 - Determinação de rótulos fortemente equivalentes é implícita.

exemplo 3.4 – Algoritmo de equivalência forte de programas monolíticos

Considere os programas monolíticos Q e R especificados na forma de fluxograma representados na figura 3.2 e figura 3.5, respectivamente. Relativamente aos pré-requisitos do algoritmo, tem-se que:

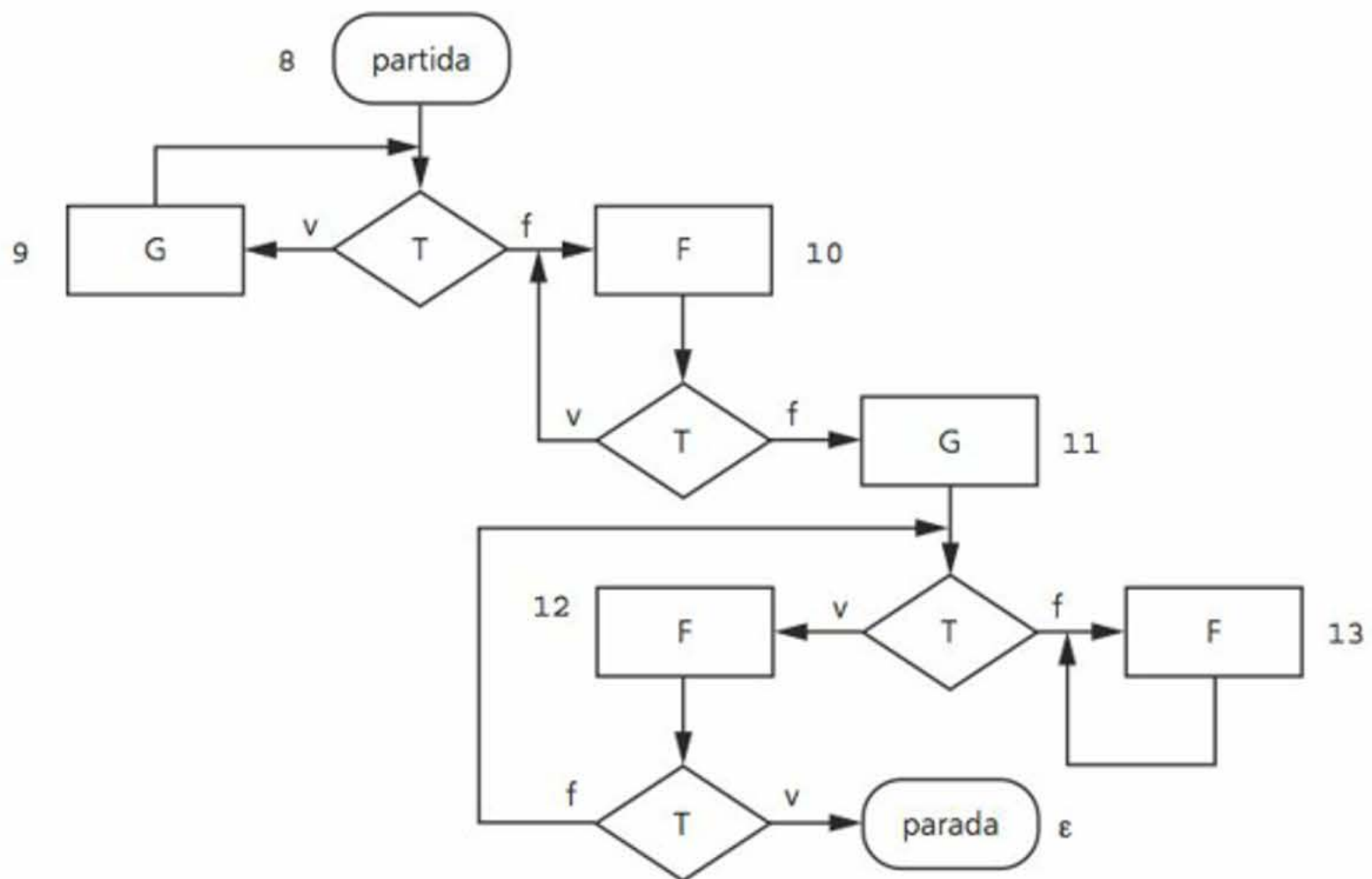


figura 3.5 Fluxograma R.

- a** A especificação do programa **Q** usando instruções rotuladas compostas, já simplificado, é representado na figura 3.4.
- 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.*

$$\begin{aligned}
 A_0 &= \{ \epsilon \} \\
 A_1 &= \{ 12, \epsilon \} \\
 A_2 &= \{ 11, 12, \epsilon \} \\
 A_3 &= \{ 10, 11, 12, \epsilon \} \\
 A_4 &= \{ 8, 9, 10, 11, 12, \epsilon \} \\
 A_5 &= \{ 8, 9, 10, 11, 12, \epsilon \}
 \end{aligned}$$

Portanto:

$$\begin{aligned}
 \lim A_k &= \{ 8, 9, 10, 11, 12, \epsilon \} \\
 (I_R, 13) &= (I, \omega), \text{ pois } 13 \notin \lim A_k
 \end{aligned}$$

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)
    
```

Em relação 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 fortemente equivalentes, então:

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

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

$$B_1 = \{ (2, 9), (3, 10) \}$$

$$B_2 = \{ (4, 10), (5, 11) \}$$

$$B_3 = \{ (6, 12), (\omega, \omega) \}$$

$$B_4 = \{ (\epsilon, \epsilon) \}$$

$$B_5 = \emptyset$$

Logo $(I, 1) = (I, 8)$ e, portanto, $Q = R$

pares de rótulos fortemente equivalentes

pares de rótulos fortemente equivalentes

pares de rótulos fortemente equivalentes

pares de rótulos fortemente equivalentes



exemplo 3.5 – Algoritmo de equivalência forte de programas monolíticos

Foi afirmado anteriormente que os programas monolíticos (fluxogramas) representados na figura 3.6 com os nós rotulados, são fortemente equivalentes. De fato, em relação aos pré-requisitos do algoritmo, tem-se que:

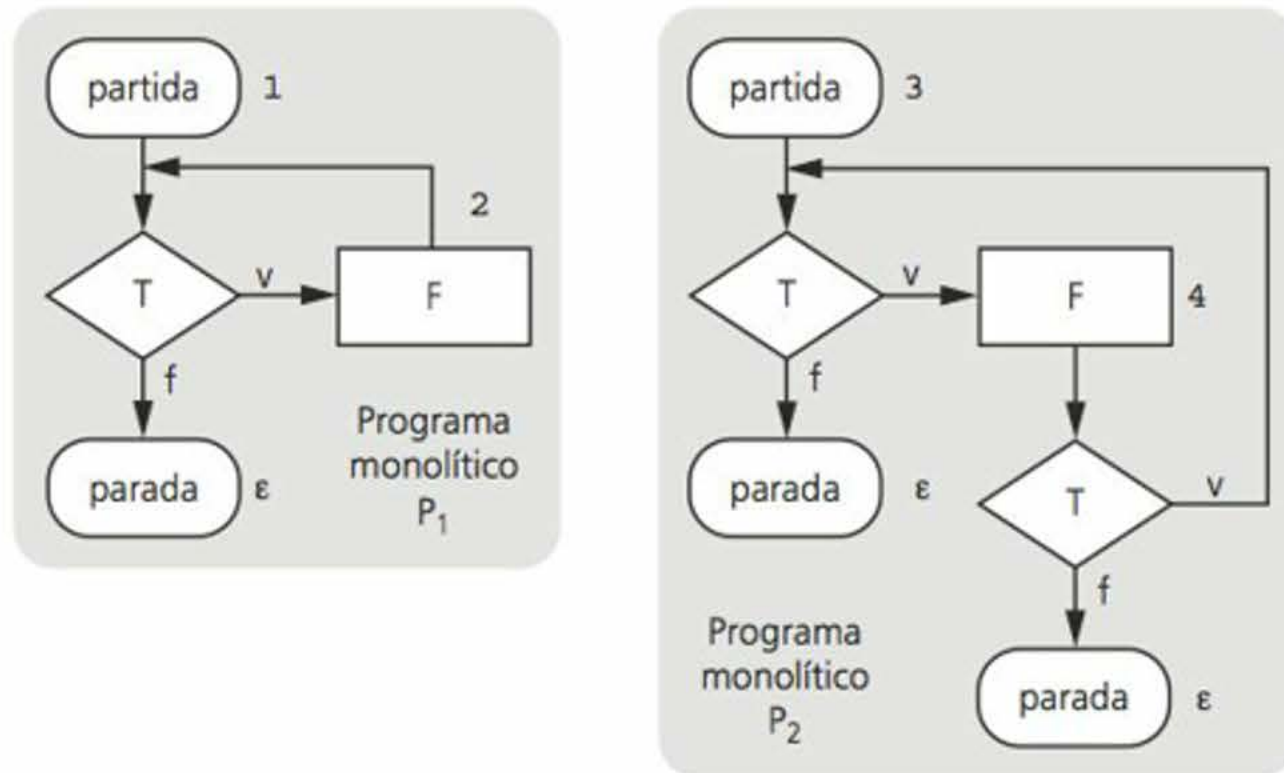


figura 3.6 Fluxogramas fortemente equivalentes.

- 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)
 4: (F, 4), (parada, ϵ)

Portanto, os correspondentes conjuntos de instruções rotuladas compostas são iguais, salvo os rótulos.

Com relação à aplicação do algoritmo, é fácil de verificar que $(I, 1) = (I, 3)$. Logo, também como afirmado anteriormente, a relação fortemente equivalente fornece subsídios para analisar a complexidade estrutural de programas. No caso, P_1 é estruturalmente “mais otimizado” que P_2 . □

3.4

→ conclusões

Neste capítulo, mostra-se a existência de um algoritmo para verificar se programas monolíticos (ou iterativos) são fortemente equivalentes. Entretanto, até o momento, não é conhecido se existe um algoritmo análogo para programas recursivos.

No próximo capítulo, é vista uma máquina de registradores, Norma e algumas versões simplificadas.

3.5

→ exercícios

exercício 3.1 Com relação aos seguintes corolários e teoremas:

- a** Corolário 3.4 - Equivalência forte de programas $\leftrightarrow$ equivalência de programas em máquinas de traços.
 - a.1) Justifique a afirmação de que a prova ($\rightarrow$) é imediata;
 - a.2) Esboce a prova ($\leftarrow$) para programas iterativo e recursivos;
- b** Justifique a afirmação de que a prova do Lema 3.8 - Equivalência Forte: Fluxograma $\rightarrow$ Rotuladas Compostas é imediata;
- c** Por que o Lema 3.8 - Equivalência Forte: Fluxograma $\rightarrow$ Rotuladas Compostas garante que $P_q \equiv P_r$ se, e somente se, $Q \equiv R$?
- d** Justifique o corolário 3.4 – equivalência forte de programas $\leftrightarrow$ equivalência de programas em máquinas de traços.

exercício 3.2 Mostre que os seguintes programas **P** e **Q**, representados na figura 3.7, são fortemente 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 3.7 Programas iterativo P (acima) e monolítico Q (abaixo).

exercício 3.3 Verifique se os programas monolíticos M_1 e M_2 , representados na figura 3.8, são fortemente equivalentes.

Programa monolítico M1	
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 M2	
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 3.8 Programas monolíticos M1 e M2.

exercício 3.4 Qual a importância da relação de equivalência forte de programas?

exercício 3.5 Verifique se os programas iterativos W_1 e W_2 , definidos na figura 3.9 e figura 3.10, respectivamente, são fortemente equivalentes.

Programa iterativo W1

```

enquanto T
  faça (F;(se T então ✓ senão G))

```

figura 3.9 Programa iterativo W1.

Programa iterativo W2

```

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

```

figura 3.10 Programa iterativo W2.

exercício 3.6 Traduza o programa monolítico da figura 3.11 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  $T_1$  então vá_para 1 senão vá_para 3
3:  faça G vá_para 4
4:  se  $T_2$  então vá_para 0 senão vá_para 1

```

figura 3.11 Programa monolítico dois_testes.

exercício 3.7 Adapte para o caso do programa monolítico da figura 3.11 os seguintes itens:

- a** Lema 3.11 - Identificação de ciclos infinitos em programa monolítico;
- b** Teorema 3.15 - Determinação de rótulos fortemente equivalentes;
- c** Definição 3.16 - Algoritmo de equivalência forte de programas monolíticos..

exercício 3.8 Generalize a definição abreviada de instruções rotuladas compostas para o caso de três testes distintos.

exercício 3.9 Traduza os fluxogramas da figura 3.12 e da figura 3.13 em instruções rotuladas compostas.

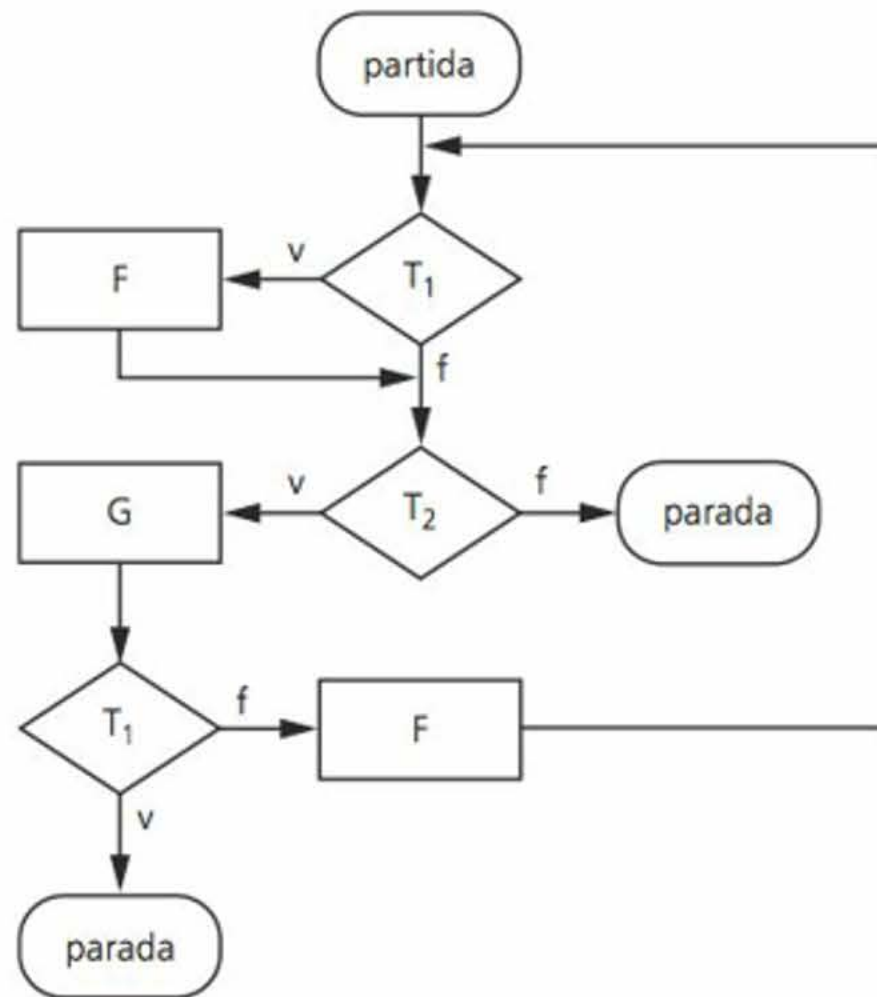


figura 3.12 Fluxograma.

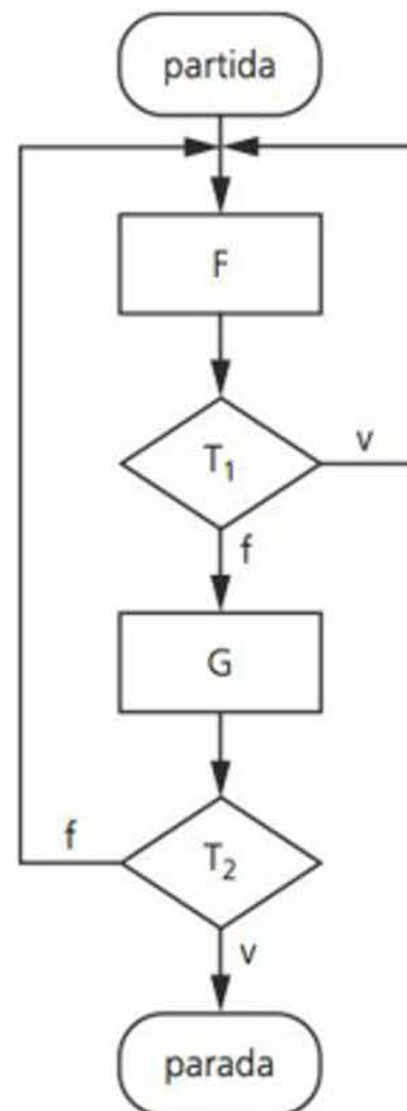


figura 3.13 Fluxograma.

exercício 3.10 Reescreva o programa monolítico da figura 3.14 na forma de instruções rotuladas compostas. Apresente o fluxograma com os nós identificados.

Observação: Atenção ao número de testes!

Programa monolítico	
1:	se T então vá_para 2 senão vá_para 3
2:	faça F vá_para 6
3:	se U então vá_para 5 senão vá_para 4
4:	faça G vá_para 0
5:	faça F vá_para 0
6:	se U então vá_para 4 senão vá_para 1

figura 3.14 Programa monolítico.

exercício 3.11 Reescreva o programa monolítico da figura 3.15 na forma de instruções rotuladas compostas. Apresente o fluxograma com os nós identificados.

Observação: Atenção ao número de testes!

Programa monolítico	
1:	faça F vá_para 2
2:	se T_1 então vá_para 3 senão vá_para 1
3:	faça G vá_para 4
4:	se T_2 então vá_para 1 senão vá_para 5
5:	faça H vá_para 6
6:	se T_2 então vá_para 0 senão vá_para 5

figura 3.15 Programa monolítico.

exercício 3.12 Reescreva o programa iterativo abaixo na forma de instruções rotuladas compostas. Apresente o fluxograma com os nós identificados.

Observação: Atenção ao número de testes!

```

enquanto  $T_1$  faça ( $F_1$ );
 $F_2$ ;
(se  $T_2$  então até  $T_1$  faça ( $F_3$ ;  $F_1$ ); ✓ senão ✓)

```


exercício 3.13 Reescreva o programa monolítico da figura 3.16 na forma de instruções rotuladas compostas. Apresente o fluxograma com os nós identificados.

Observação: Atenção ao número de testes!

Programa monolítico	
1:	se T_1 então vá_para 4 senão vá_para 3
2:	faça F_1 vá_para 1
3:	se T_1 então vá_para 2 senão vá_para 1

figura 3.16 Programa monolítico.

Termo-chave

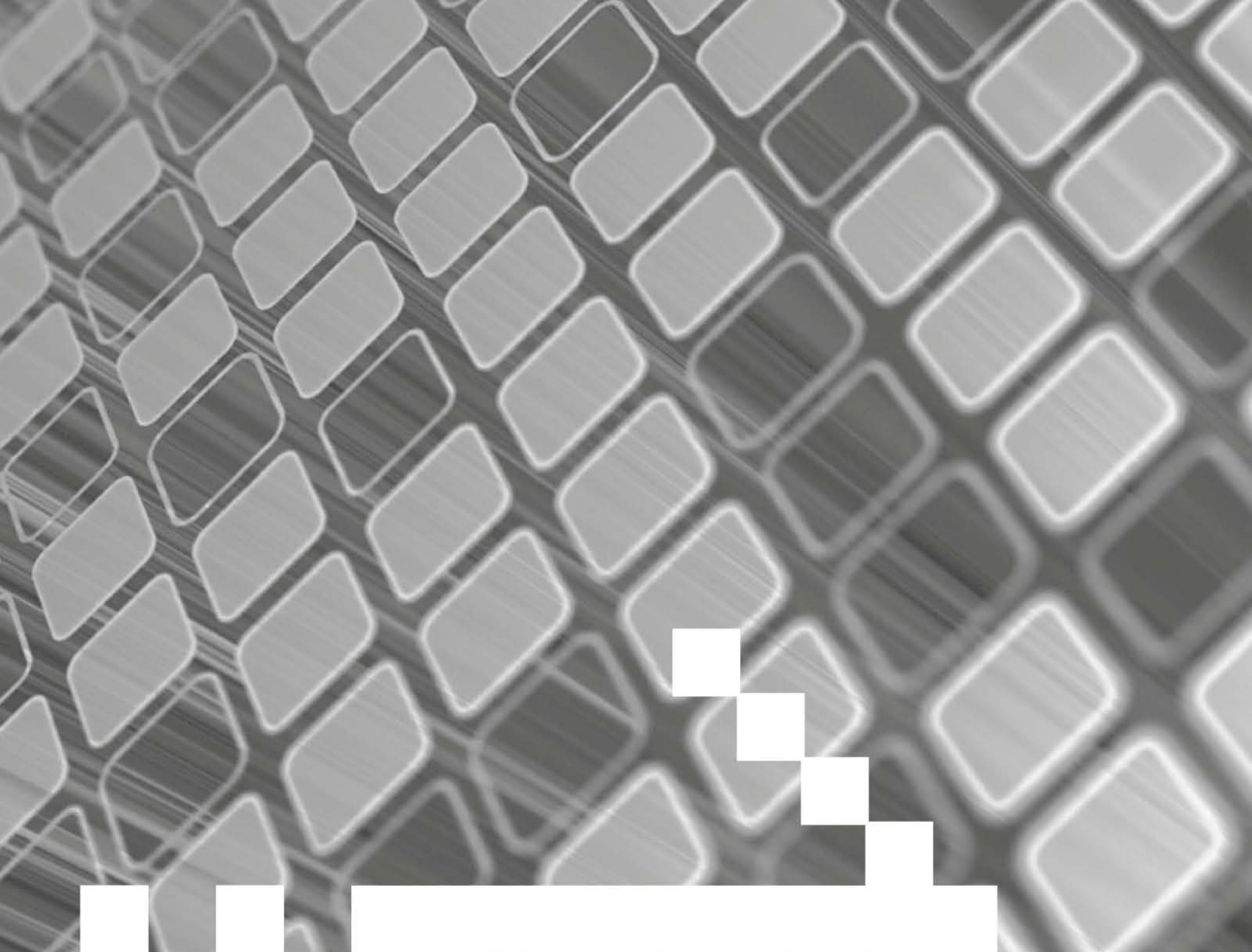
algoritmo de equivalência forte, p. 92

ciclos infinitos, p. 89

instrução rotulada composta, p. 84

máquina de traços, p. 80, 81

rótulos consistentes, p. 91



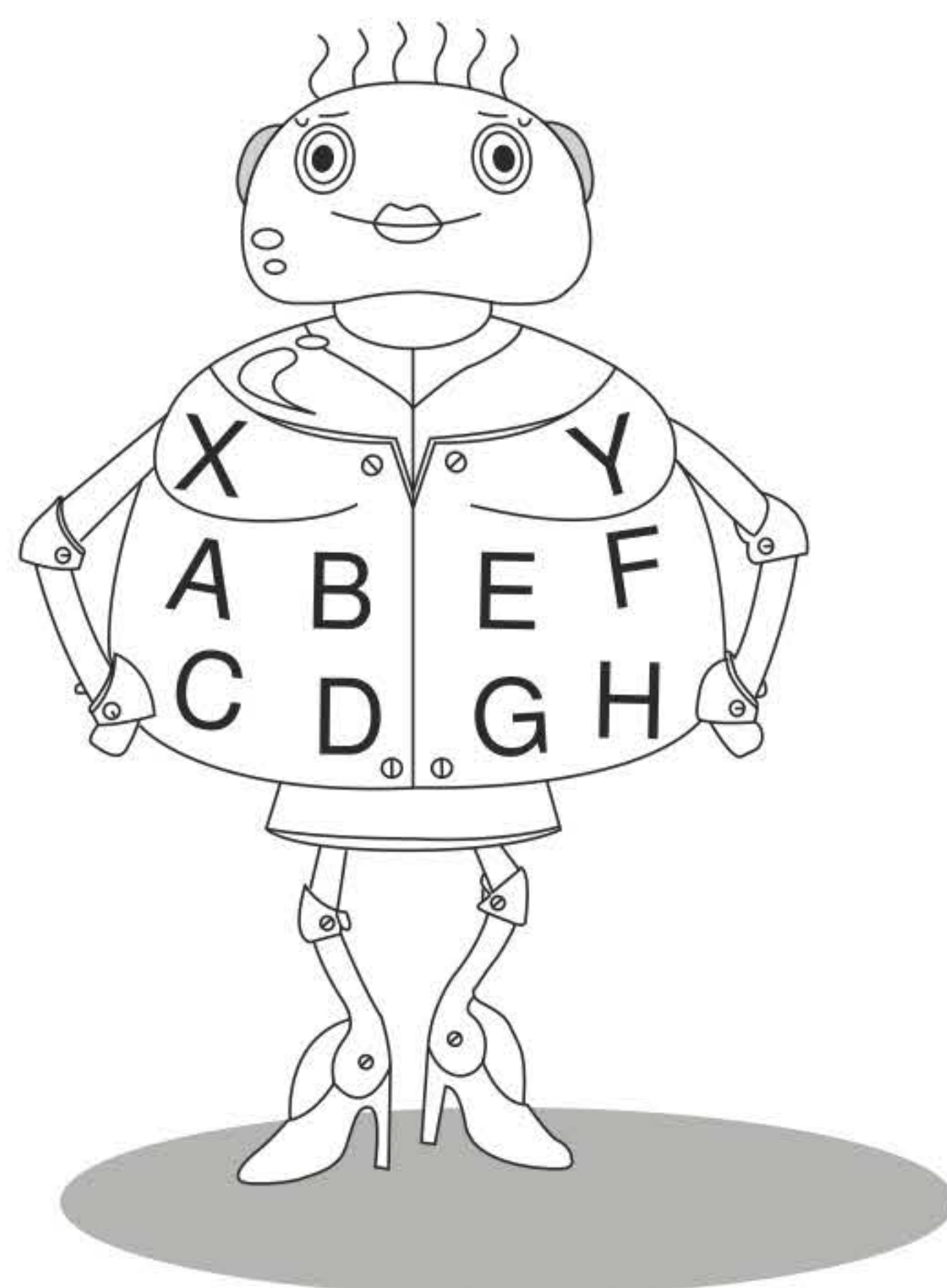
4

capítulo

máquinas de registradores – Norma

■ ■ Intuitivamente, um algoritmo tem descrição finita e não ambígua e consiste em passos discretos, executáveis mecanicamente em um tempo finito. O conceito de programa introduzido satisfaz essa noção intuitiva. Entretanto, é necessário definir a máquina a ser considerada. Se for possível representar qualquer algoritmo como um programa em tal máquina, então esta é uma máquina universal.

Com esse objetivo são introduzidas as máquinas de registradores. Em particular, é estudada a máquina Norma que possui três tipos de instruções/teste (ad, sub e zero) sobre registradores naturais, mostrando que uma máquina simples e rudimentar pode ser tão poderosa quanto qualquer outra.



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, o estudo da solucionabilidade de um problema é a investigação da existência de um algoritmo capaz de resolvê-lo. Entretanto, se a noção de algoritmo é intuitiva (não formal), qualquer afirmação sobre a não solucionabilidade de determinado problema é questionável.

Em relação 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 de espaço são “tanto quanto necessários”. O correto entendimento dessa observação é de fundamental importância para todo o estudo 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 $\mathbf{N}$) poderia ser equivalentemente considerado. Mesmo para os tipos cujos valores possuem descrição finita, uma grande variedade de tipos de dados 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, é necessário definir a máquina a ser considerada, a qual deve 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 fossem 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 máquina universal. As evidências de que uma máquina é, de fato, uma máquina 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.

Ao longo deste e dos próximos capítulos, a noção de máquina universal é construída. No que segue, é estudada a máquina universal denominada máquina Norma, proposta por Richard Bird (1976), a qual, resumidamente, possui um conjunto de registradores naturais e somente três tipos de instruções sobre os registradores: adição e subtração do valor um e 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 de que, de fato, trata-se de uma máquina universal.

4.1

→ codificação de conjuntos estruturados

Nesta seção, é considerado, de forma breve e por meio 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 ideia básica é definir uma função injetora:

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

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 4.1 – Número de Gödel

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

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

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 $c: \mathbf{N}^n \rightarrow \mathbf{N}$ definida a seguir é unívoca (suponha que $(x_1, x_2, \dots, x_n) \in \mathbf{N}^n$ e que o símbolo $\cdot$ denota a operação de multiplicação nos naturais):

$$c(x_1, x_2, \dots, x_n) = p_1^{x_1} \cdot p_2^{x_2} \cdot \dots \cdot p_n^{x_n}$$

Deve-se reparar que esta codificação, conhecida como *número de Gödel*, 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. $\square$

exemplo 4.2 – 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 $\{F_1, F_2, \dots, F_f\}$ e $\{T_1, T_2, \dots, T_t\}$ 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.

$$r_1: \text{faça } F_k \text{ vá_para } r_2$$

b Teste.

$$r_1: \text{se } T_k \text{ então vá_para } r_2 \text{ senão vá_para } r_3$$

Cada instrução rotulada pode ser denotada por uma quádrupla ordenada como segue:

a Operação.

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

b Teste.

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

onde a primeira componente identifica o tipo da instrução; a segunda, o k -ésimo teste ou operação; e as duas últimas são os rótulos sucessores (no caso de operação, são iguais).

Estas quádruplas são codificadas usando os quatro primeiros números primos, seguindo um raciocínio análogo ao do exemplo 4.1 – Número de Gödel. Desta forma, a codificação de P' é realizada em duas etapas:

- 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 conforme o exemplo 4.1 – Número de Gödel. $\square$

exemplo 4.3 – Codificação de programas monolíticos

Considere o programa monolítico **Q** da figura 4.1. Então, a correspondente codificação **c(i1, i2, i3, i4, i5, i6, i7, i8)** é o numero natural **q** dado por:

$$q = 2^{i1} \cdot 3^{i2} \cdot 5^{i3} \cdot 7^{i4} \cdot 11^{i5} \cdot 13^{i6} \cdot 17^{i7} \cdot 19^{i8}$$

sendo que (tomando **F** como **1**, **G** como **2** e o teste **T** como **1** na determinação de **k**, ou seja, o expoente do número primo **3**):

i1 = 2 ⁰ ·3 ¹ ·5 ² ·7 ²	obtido da quádrupla: (0, 1, 2, 2)
i2 = 2 ¹ ·3 ¹ ·5 ³ ·7 ⁵	(1, 1, 3, 5)
i3 = 2 ⁰ ·3 ² ·5 ⁴ ·7 ⁴	(0, 2, 4, 4)
i4 = 2 ¹ ·3 ¹ ·5 ¹ ·7 ⁰	(1, 1, 1, 0)
i5 = 2 ⁰ ·3 ¹ ·5 ⁶ ·7 ⁶	(0, 1, 6, 6)
i6 = 2 ¹ ·3 ¹ ·5 ⁷ ·7 ²	(1, 1, 7, 2)
i7 = 2 ⁰ ·3 ² ·5 ⁸ ·7 ⁸	(0, 2, 8, 8)
i8 = 2 ¹ ·3 ¹ ·5 ⁶ ·7 ⁰	(1, 1, 6, 0)

□

Programa monolítico Q	
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

figura 4.1 Programa monolítico Q.

Analogamente ao exemplo 4.1 – Número de Gödel, a codificação de programas monolíticos apresentada não é uma função bijetora (por quê?).

exemplo 4.4 – Decodificação de programas monolíticos

A decodificação de um número natural que denota um programa monolítico em seu correspondente programa é ilustrada. Suponha o número natural **p** decomposto em fatores primos como segue:

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

Portanto, o programa possui duas instruções rotuladas correspondentes aos expoentes dos números primos **2** e **3**, respectivamente, **150** e **105**. Decompondo-os em seus fatores primos, tem-se que:

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

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  $T_1$  então vá_para 2 senão vá_para 0
2:   faça  $F_1$  vá_para 1
```

Este exemplo ajuda a inferir que nem todo número natural corresponde a um programa codificado por esta função de codificação.

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

4.2

→ definição da máquina Norma

A *máquina Norma* (*Number Theoretic Register Machine*), proposta por Richard Bird (1976), é uma máquina de registradores. Máquinas de registradores são modelos propostos mais recentemente (em comparação com a máquina de Turing e com outros formalismos universalmente conhecidos) e são definidas de forma a lembrar a arquitetura básica dos computadores atuais. 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 um conjunto infinito (“tantos quantos forem necessários”) de registradores naturais como memória e três instruções sobre cada registrador:

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

No que se refere aos registradores naturais, entende-se que os registradores da máquina Norma podem assumir “qualquer valor natural tão grande quanto necessário”.

No texto que segue, $\mathbf{N}^\infty$ 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 $\mathbf{N}^\infty$:

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

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

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

- suponha a upla $(a_0, a_1, a_2, \dots)$ em $\mathbf{N}^\infty$;
- pode-se afirmar que, para cada $n \in \mathbf{N}$, a_n é a n -ésima componente da upla $(a_0, a_1, a_2, \dots)$;

- tal fato pode ser formalmente denotado como uma função $f: \mathbf{N} \rightarrow \mathbf{N}$ tal que, para qualquer $n \in \mathbf{N}$, $f(n) = a_n$;
- portanto, um elemento de $\mathbf{N}^\infty$ pode ser visto como uma função nos naturais, ou seja:

$$\mathbf{N}^\infty = \{ f: \mathbf{N} \rightarrow \mathbf{N} \mid f \text{ é função } \}$$

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

- função identidade de $\mathbf{N}$, ou seja, $\text{id}_{\mathbf{N}}: \mathbf{N} \rightarrow \mathbf{N}$ tal que, para qualquer $n \in \mathbf{N}$, tem-se que $\text{id}_{\mathbf{N}}(n) = n$;
- função constante 5, ou seja, $\text{const}_5: \mathbf{N} \rightarrow \mathbf{N}$ tal que, para qualquer $n \in \mathbf{N}$, tem-se que $\text{const}_5(n) = 5$.

definição 4.1 - Máquina Norma

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

$$\text{Norma} = (\mathbf{N}^\infty, \mathbf{N}, \mathbf{N}, \text{ent}, \text{sai}, \{ \text{ad}_K, \text{sub}_K \}, \{ \text{zero}_K \})$$

onde:

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

$$X, Y, A, B, \dots$$

- b** A função de entrada:

$$\text{ent}: \mathbf{N} \rightarrow \mathbf{N}^\infty$$

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:

$$\text{sai}: \mathbf{N}^\infty \rightarrow \mathbf{N}$$

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:

$$\text{ad}_K: \mathbf{N}^\infty \rightarrow \mathbf{N}^\infty$$

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

$$\text{sub}_K: \mathbf{N}^\infty \rightarrow \mathbf{N}^\infty$$

subtrai um da componente de memória 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:

$$\text{zero}_K: \mathbf{N}^\infty \rightarrow \{ \text{verdadeiro}, \text{falso} \}$$

resulta em **verdadeiro**, se a componente de memória correspondente ao registrador K for **zero** e, caso contrário, resulta em **falso**. $\square$

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

$K := K + 1$

$K := K - 1$

$K = 0$

4.3

→ 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. 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 de 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.

4.3.1 operações e testes

As seguintes operações e testes não definidos na máquina Norma são exemplificados a seguir (não necessariamente nesta ordem):

- atribuição de valor a um registrador;
- adição de dois registradores;
- atribuição do valor de um registrador a outro registrador;
- multiplicação de dois registradores;
- atribuição de um número primo a um registrador;
- teste se o valor de um registrador é menor que o valor de outro registrador;
- teste se o valor de um registrador é múltiplo do valor de outro registrador (ou teste de divisão inteira com resto **zero**);

- teste se o valor de um registrador é um número primo.

exemplo 4.5 – 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 monolítico:

```
1:   se A = 0 então vá_para 3 senão vá_para 2
2:   faça A := A - 1 vá_para 1
```

ou com seu equivalente 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 4.6 – 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 monolítico, exemplificado para **n = 3**:

```
1:   faça A := 0 vá_para 2
2:   faça A := A + 1 vá_para 3
3:   faça A := A + 1 vá_para 4
4:   faça A := A + 1 vá_para 5
```

ou pelo seu programa fortemente equivalente iterativo:

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



Apesar da máquina Norma só processar programas monolíticos, para facilitar a apresentação dos exemplos, os programas ilustrativos são apresentados na forma de programas iterativos. Lembre-se de que, a partir de um programa iterativo, sempre é possível obter o seu correspondente programa monolítico fortemente equivalente .

exemplo 4.7 – Adição de dois registradores

Uma macro correspondente à operação de adição do valor do registrador **B** ao de **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** ao de **A**, o registrador **B** é zerado. Para preservar o valor original de **B**, é necessário utilizar um registrador de trabalho.

exemplo 4.8 – Adição de dois registradores, preservando o conteúdo

Uma macro correspondente à operação de adição do valor do registrador **B** ao de **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:

$$A := A + B \text{ usando } C$$



É importante destacar que, ao executar na máquina Norma um programa **P** de outra máquina que contenha uma macro 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 4.9 - 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:

$$A := B \text{ usando } C$$

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 4.10 – Multiplicação de dois registradores

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

$$A := A \cdot B \text{ usando } C, D$$

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 4.11 – Teste se o valor de um registrador é menor que outro registrador

Uma macro de teste que verifica se o valor de um registrador **A** é menor que o valor de um registrador **B**, denotada por:

$$A < B \text{ usando } C, D, E$$

```

C := A usando E;
D := B usando E;
até C = 0
faça (se D = 0
      então falso
      senão C := C-1; D := D-1);
      (se D = 0
       então falso
       senão verdadeiro)

```



exemplo 4.12 – Teste se o valor da divisão inteira é zero

Uma macro de teste que verifica se o resto da divisão inteira do valor de um registrador **A** pelo valor de um registrador **B** é **zero**, denotada por:

$$\text{teste_mod}(A, B) \text{ usando } C, D, E, C', D', E'$$

```

C := A usando E; D := B usando E;
(se B = 0
 então falso
 senão (se A = 0
        então verdadeiro
        senão até C < B usando C', D', E'
              faça (C := C - D usando E);
        (se C = 0
         então verdadeiro
         senão falso)))

```



Por simplicidade, no texto que segue, na referência a uma macro definida anteriormente, são omitidos os registradores usados. Por exemplo:

teste_mod(A, B) usando C, D, E, C', D', E' é abreviada por **teste_mod(A, B)**

exemplo 4.13 – 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)

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

```
(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 o do exemplo 4.12 - Teste se o valor da divisão inteira é zero. □

exemplo 4.14 – 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)

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 “maior”;
- teste “maior ou igual”;
- teste “menor ou igual”.

4.3.2 valores numéricos

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

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

exemplo 4.15 – 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 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:

$$A := A + 1$$

é exemplificada na seguinte tabela da figura 4.2. Observando a tabela, percebe-se que primeiro é necessário verificar se **A₁** (sinal) é zero para então analisar **A₂** (magnitude). Assim, a operação pode ser realizada pelo seguinte programa iterativo (suponha **zero** com sinal positivo):

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


Analogamente, a operação inteira de subtração ($A := A - 1$) é exemplificada na tabela da figura 4.2 e pode ser realizada pelo seguinte programa iterativo:

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

Seguindo o mesmo raciocínio, para o teste se é zero ($A = 0$) basta verificar se A_2 (magnitude) é zero, como realizado pelo seguinte programa iterativo:

```
(se  A2 = 0
então verdadeiro
senão falso)
```



A	A ₁ (sinal)	A ₂ (mag.)	A + 1	A ₁ (sinal)	A ₂ (mag.)	A - 1	A ₁ (sinal)	A ₂ (mag.)
-2	1	2	-1	1	1	-3	1	3
-1	1	1	0	0	0	-2	1	2
0	0	0	1	0	1	-1	1	1
1	0	1	2	0	2	0	0	0
2	0	2	3	0	3	1	0	1

figura 4.2 Operações inteiras $A := A + 1$ e $A := A - 1$.

exemplo 4.16 - 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•d + b•c, b•d)

(a, b) - (c, d) = (a•d - b•c, b•d)

(a, b) × (c, d) = (a•c, b•d)

(a, b) ÷ (c, d) = (a•d, b•c) com c ≠ 0

(a, b) = (c, d) se, e somente se, a•d = b•c

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

4.3.3 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 4.17 - Arranjo unidimensional

Uma estrutura do tipo arranjo unidimensional da forma $A(1), A(2), \dots$, pode ser definida por um único registrador A , usando a codificação de n -uplas naturais como introduzido em 4.1 – Codificação de conjuntos estruturados. Exemplificando, se um arranjo unidimensional é tal que:

$$\begin{aligned} A(1) &= 5 \\ A(2) &= 2 \\ A(3) &= 0 \\ A(4) &= 3 \\ A(5) &= 1 \end{aligned}$$

então o valor a armazenada em A é o resultado de:

$$a = 2^5 \cdot 3^2 \cdot 5^0 \cdot 7^3 \cdot 11^1 \cdot 13^0 \cdot 17^0 \dots$$

Note-se que o arranjo não necessita ter tamanho máximo (número de posições indexáveis) predefinido. Lembre-se de 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 teste **teste_mod(A, C)** que retorna o valor **verdadeiro**, se o conteúdo do registrador C é um divisor do conteúdo do registrador A e **falso**, caso contrário, implementado no exemplo 4.12 - Teste se o valor da divisão inteira é zero;
- a seguinte macro (cuja definição é sugerida como exercício) denota a divisão de dois registradores:

$$A := \text{div}(A, C)$$

a *Indexação direta. As macros:*

$$\begin{aligned} &\text{ad}_{A(n)} \\ &\text{sub}_{A(n)} \\ &\text{zero}_{A(n)} \end{aligned}$$

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

Programa iterativo $\text{ad}_{A(n)}$

$C := p_n;$
 $A := A \cdot C$

Programa iterativo $\text{sub}_{A(n)}$

$C := p_n;$
(se $\text{teste_mod}(A, C)$
então $A := \text{div}(A, C)$
senão ✓)

Programa iterativo $\text{zero}_{A(n)}$

$C := p_n;$
(se $\text{teste_mod}(A, C)$
então falso
senão verdadeiro)

b *Indexação indireta.* As macros:
 $\text{ad}_{A(B)}$
 $\text{sub}_{A(B)}$
 $\text{zero}_{A(B)}$
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 $\text{ad}_{A(B)}$

$C := \text{primo}(B)$
 $A := A \cdot C$

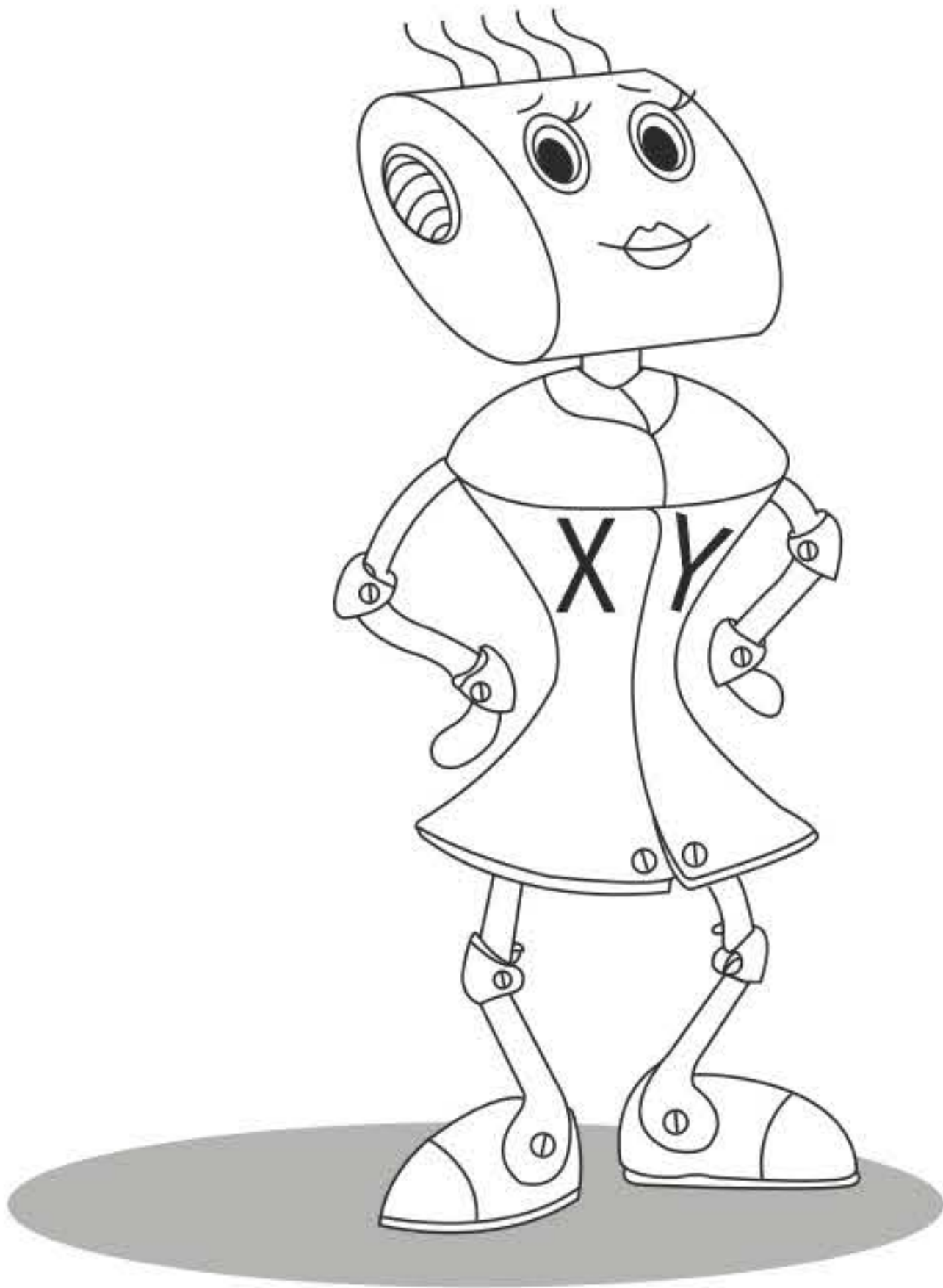
Programa iterativo $\text{sub}_{A(B)}$

$C := \text{primo}(B)$
(se $\text{teste_mod}(A, C)$
então $A := \text{div}(A, C)$
senão ✓)

Programa iterativo $\text{zero}_{A(B)}$

$C := \text{primo}(B)$
(se $\text{teste_mod}(A, C)$
então falso
senão verdadeiro)





observação 4.2 – Arranjo unidimensional $\times$ Norma com 2 registradores

Um resultado interessante (que será usado adiante) é que, usando a estrutura de arranjo unidimensional com indexação direta, como ilustrado no exemplo 4.17 - Arranjo unidimensional, 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 4.17 - Arranjo unidimensional:

$ad_{X(k)}$
 $sub_{X(k)}$
 $zero_{X(k)}$

onde **X(k)** denota a **k**-ésima posição do arranjo em **X**. □

exemplo 4.18 – Pilha

Estruturalmente, a principal característica de uma *pilha* é que o último valor gravado é o primeiro a ser lido, como ilustrado na figura 4.3. 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) 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). □

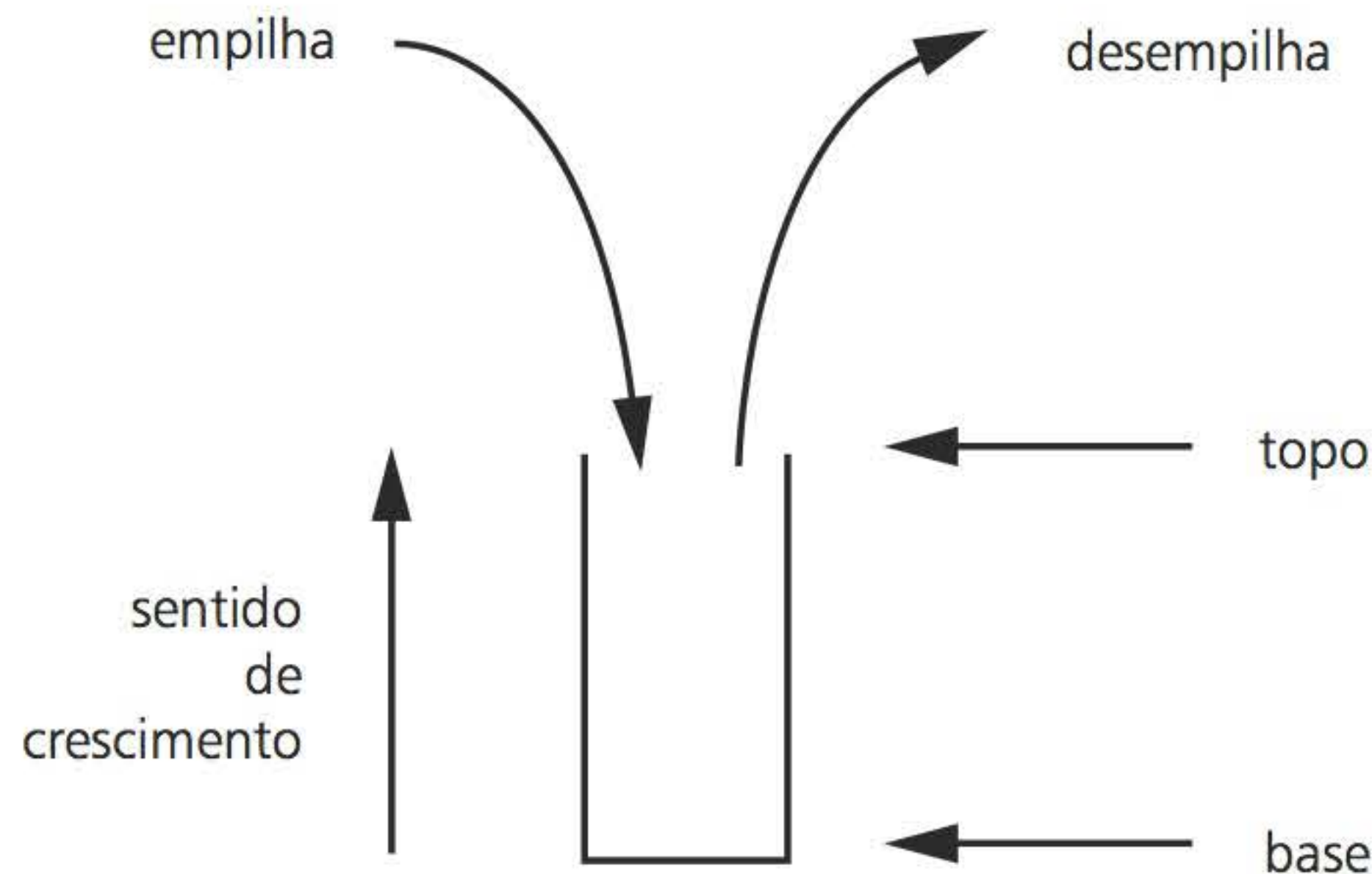


figura 4.3 Estrutura do tipo pilha.

De forma análoga ao exemplificado, outras estruturas de dados podem ser construídas, como vetores, matrizes, filas, listas e listas encadeadas.

4.3.4 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 4.19 – 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 4.4, trata o endereçamento indireto de **A** (e onde: cada circunferência rotulada representa um desvio incondicional para a correspondente instrução; entenda-se **A+k** como uma notação simplificada de **A+1** repetida **k** vezes). 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
```

Um teste com endereçamento indireto:

```
r:   se T então vá_para End_A senão vá_para End_B
```

□

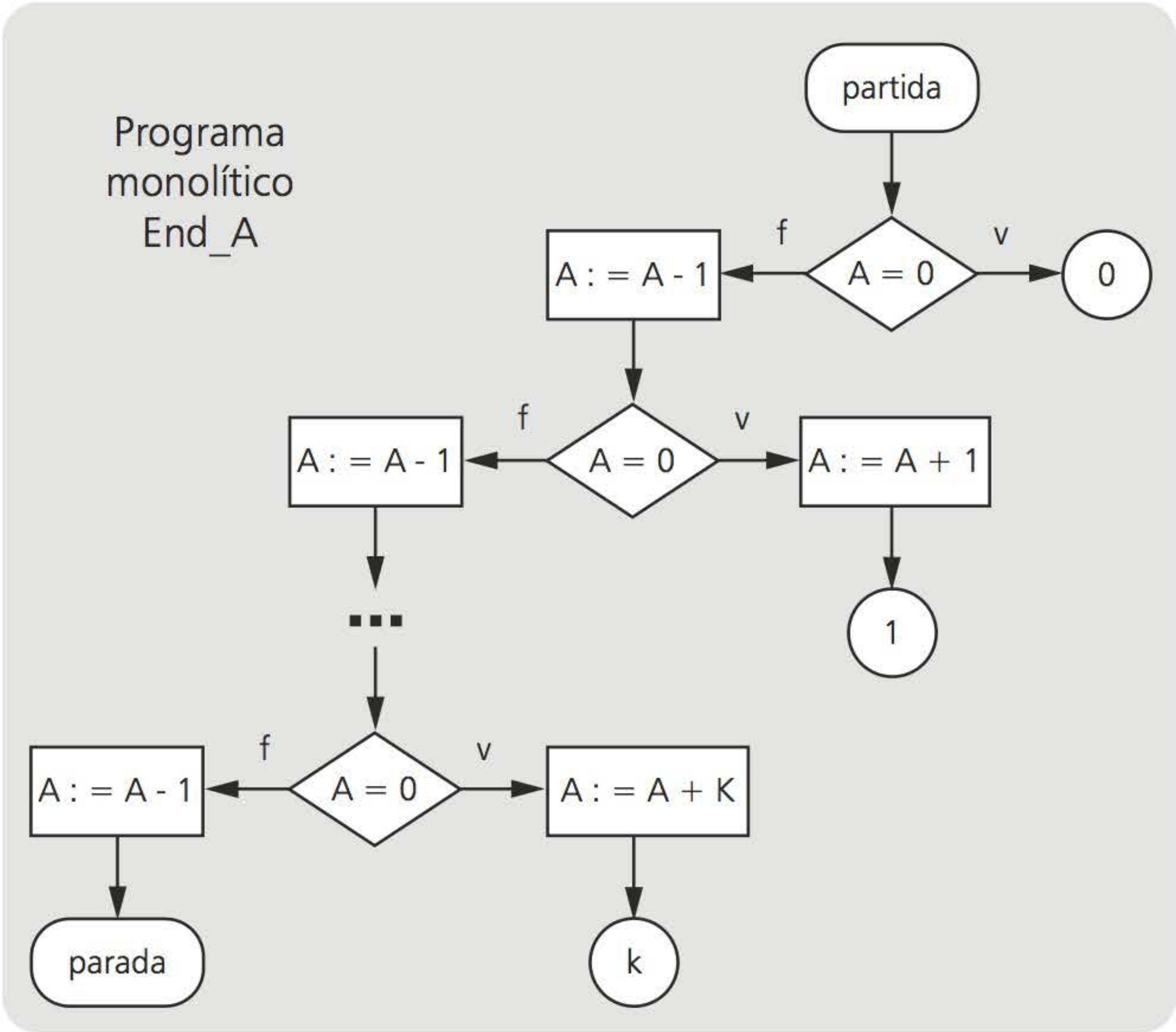


figura 4.4 Fluxograma para tratar endereçamento indireto.

observação 4.3 – Programa monolítico x 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. Tais demonstrações não serão estudadas neste livro. □

4.3.5

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 por meio de outra máquina universal, denominada máquina de Turing, estudada adiante, a qual se prova ser equivalente à máquina Norma.

4.4

→ conclusões

Neste capítulo, foram introduzidas as máquinas de registradores. É especialmente interessante observar como uma máquina tão simples e rudimentar pode ser tão poderosa. De fato, os recursos necessários para computar são bem poucos. Todos os demais recursos existentes

em máquinas reais como nos computadores modernos são apenas artifícios para facilitar a programação da computação.

Em particular, foi estudada a máquina Norma que possui três tipos de instruções/teste (**ad**, **sub** e **zero**) sobre registradores naturais. Evidências internas de que Norma é uma máquina universal foram apresentadas, evidências externas serão vistas nos próximos capítulos.

4.5

→ exercícios

exercício 4.1 Analise o programa monolítico abaixo e assinale a opção que contém a codificação correta (considere a seguinte ordem de operações **F=1**, **G=2**, **H=3** e dos testes **T₁=1**, **T₂=2**).

Programa monolítico	
1:	faça F vá_para 2
2:	se T₁ então vá_para 3 senão vá_para 1
3:	faça G vá_para 4
4:	se T₂ então vá_para 1 senão vá_para 5
5:	faça H vá_para 6
6:	se T₂ então vá_para 0 senão vá_para 5

- a
- $2^{3675} \cdot 3^{5250} \cdot 5^{135025} \cdot 7^{151260} \cdot 11^{496371875} \cdot 13^{30256}$
- b
- $2^{375} \cdot 3^{525} \cdot 5^{1350625} \cdot 7^{151630} \cdot 11^{4963311875} \cdot 13^{3025}$
- c
- $2^{3675} \cdot 3^{5250} \cdot 5^{13505625} \cdot 7^{1512630} \cdot 11^{49633171875} \cdot 13^{302526}$
- d
- $2^{375} \cdot 3^{5250} \cdot 5^{1355625} \cdot 7^{151260} \cdot 11^{49633171875} \cdot 13^{30256}$
- e
- $2^{3675} \cdot 3^{5250} \cdot 5^{13505625} \cdot 7^{152630} \cdot 11^{4963171875} \cdot 13^{3526}$

exercício 4.2 Codifique o seguinte programa monolítico e escolha a alternativa que representa o resultado de forma correta.

Programa monolítico	
1:	se T₁ então vá_para 2 senão vá_para 3
2:	faça F₁ vá_para 3
3:	faça F₂ vá_para 4
4:	se T₁ então vá_para 1 senão vá_para 0

- a
- $2^{51450} \cdot 3^{128625} \cdot 5^{13505625} \cdot 7^{30}$
- b
- $2^2 \cdot 3^{12825} \cdot 5^{13505625} \cdot 7^{10}$
- c
- $2^{450} \cdot 3^{108625} \cdot 4^{13505625} \cdot 7^{20}$
- d
- $2^{40532} \cdot 3^{128888} \cdot 4^{13505625} \cdot 7^{70}$
- e
- $2^{2345} \cdot 3^{1555555} \cdot 5^{13505625} \cdot 7^{30}$

exercício 4.3 Dado um programa monolítico abaixo, sabe-se que sua codificação é da forma $p = 2^x \cdot 3^y \cdot 5^z$.

Programa monolítico	
1:	se T_1 então vá_para 4 senão vá_para 3
2:	faça F_1 vá_para 1
3:	se T_1 então vá_para 2 senão vá_para 1

Marque a alternativa correta:

- ☐ **a** $x = 643125, y = 128625$ e $z = 26250$
- ☐ **b** $x = 1286250, y = 105$ e $z = 1050$
- ☐ **c** $x = 643125, y = 210$ e $z = 525$
- ☐ **d** $x = 36750, y = 128625$ e $z = 26250$
- ☐ **e** $x = 1286250, y = 128625$ e $z = 1050$

exercício 4.4 Considere o seguinte programa monolítico:

Programa monolítico	
1:	se T então vá_para 2 senão vá_para 3
2:	faça F vá_para 6
3:	se U então vá_para 5 senão vá_para 4
4:	faça G vá_para 0
5:	faça F vá_para 0
6:	se U então vá_para 4 senão vá_para 1

Assinale a quádrupla ordenada que *não* corresponde a nenhuma codificação das instruções rotuladas existentes neste programa:

- ☐ **a** (0, 1, 1, 0)
- ☐ **b** (0, 2, 0, 0)
- ☐ **c** (1, 1, 2, 3)
- ☐ **d** (1, 2, 4, 1)
- ☐ **e** (1, 2, 5, 4)

exercício 4.5 Considere o seguinte programa monolítico:

Programa monolítico	
1:	faça F vá_para 2
2:	se T_1 então vá_para 1 senão vá_para 3
3:	faça G vá_para 4
4:	se T_2 então vá_para 5 senão vá_para 1

Marque a correspondente codificação correta:

- ☐ a $2^{(2^03^15^27^2)} + 3^{(2^13^15^17^3)} + 5^{(2^03^25^47^4)} + 7^{(2^13^25^57^1)}$
- ☐ b $2^{(2^13^25^57^1)} \cdot 3^{(2^13^15^17^3)} \cdot 5^{(2^03^25^47^4)} \cdot 7^{(2^03^15^27^2)}$
- ☐ c $2^{(2^0+3^1+5^2+7^2)} \cdot 3^{(2^1+3^1+5^1+7^3)} \cdot 5^{(2^0+3^2+5^4+7^4)} \cdot 7^{(2^1+3^2+5^5+7^1)}$
- ☐ d $2^{(2^23^15^27^2)} \cdot 3^{(2^33^15^17^3)} \cdot 5^{(2^03^25^47^4)} \cdot 7^{(2^13^25^57^1)}$
- ☐ e $2^{(2^03^15^27^2)} \cdot 3^{(2^13^15^17^3)} \cdot 5^{(2^03^25^47^4)} \cdot 7^{(2^13^25^57^1)}$

exercício 4.6 Considere o seguinte programa monolítico:

Programa monolítico	
1:	faça F vá_para 2
2:	se T ₁ então vá_para 1 senão vá_para 3
3:	faça F ₂ vá_para 0

Marque a alternativa que representa o número natural associado pela função de codificação **codigo(P)**:

- ☐ a 340341750
- ☐ b $2^{7350} \cdot 3^{5145} \cdot 5^{18}$
- ☐ c $2^{3675} \cdot 3^{10290} \cdot 5^9$
- ☐ d 583200
- ☐ e 77145

exercício 4.7 Qual dos números abaixo representa a codificação de um programa monolítico?

- ☐ a $2^{3675} \cdot 3^{18375} \cdot 5^{4410}$
- ☐ b $2^{11025} \cdot 3^{1470} \cdot 5^{5292}$
- ☐ c $2^{9261} \cdot 3^{6174} \cdot 5^{9450}$
- ☐ d $2^{8505} \cdot 3^{13230} \cdot 5^{2835}$
- ☐ e 9447840

exercício 4.8 Qual dos números abaixo *não* representa a codificação de uma instrução de programa monolítico?

- ☐ a 210
- ☐ b 450
- ☐ c 315
- ☐ d 750
- ☐ e 525

exercício 4.9 Considere o seguinte programa monolítico codificado:

$$p = 2^{51450} \cdot 3^{105} \cdot 5^9$$

Marque a alternativa que representa a reescrita como um programa iterativo:

- a

enquanto T faça (F;G)
- b

até T faça (F);G
- c

até T faça (F;G)
- d

enquanto T faça (F);G
- e

enquanto T faça (F)

exercício 4.10 Codifique o seguinte programa iterativo e marque a opção correta:

- enquanto T₁ faça (F₁);
F₂;
(Se T₂
então até T₁ faça (F₃;F₁);✓
senão ✓)

a

$2^{(2^1 3^1 5^2 7^3)} \cdot 3^{(2^0 3^1 5^1 7^1)} \cdot 5^{(2^1 3^2 5^4 7^4)} \cdot 7^{(2^0 3^2 5^5 7^0)} \cdot 11^{(2^1 3^3 5^6 7^6)} \cdot 13^{(2^1 3^1 5^7 11^7)} \cdot 17^{(2^0 3^1 5^0 7^5)}$

b

$2^{(2^1 3^1 5^2 7^3)} \cdot 3^{(2^0 3^1 5^1 7^1)} \cdot 5^{(2^0 3^2 5^4 7^4)} \cdot 7^{(2^1 3^2 5^5 7^0)} \cdot 11^{(2^1 3^1 5^0 7^6)} \cdot 13^{(2^0 3^3 5^7 7^7)} \cdot 17^{(2^0 3^1 5^5 7^5)}$

c

$2^{(2^0 3^1 5^2 7^3)} \cdot 3^{(2^1 3^6 5^1 7^1)} \cdot 5^{(2^1 3^2 5^4 7^4)} \cdot 7^{(2^0 3^2 5^5 7^0)} \cdot 11^{(2^1 3^3 5^6 7^6)} \cdot 13^{(2^1 3^1 5^7 7^7)} \cdot 17^{(2^0 3^1 5^0 7^7)}$

d

$2^{(2^1 3^1 5^2 7^3)} \cdot 3^{(2^0 3^1 5^1 7^1)} \cdot 5^{(2^0 3^2 5^4 7^4)} \cdot 7^{(2^1 3^2 5^5 7^0)} \cdot 11^{(2^0 3^3 5^6 7^6)} \cdot 13^{(2^0 3^1 5^7 7^7)} \cdot 19^{(2^1 3^1 5^0 7^5)}$

e

$2^{(2^0 3^1 5^2 7^3)} \cdot 3^{(2^1 3^1 5^1 7^1)} \cdot 5^{(2^1 3^2 5^4 7^4)} \cdot 7^{(2^0 3^2 5^5 7^0)} \cdot 11^{(2^1 3^3 5^6 7^6)} \cdot 13^{(2^5 3^1 5^7 7^7)} \cdot 19^{(2^0 3^1 5^0 7^5)}$

exercício 4.11 Codifique o fluxograma ilustrado na figura 4.5 como um número natural, usando a codificação de programas introduzida no exemplo 4.2 – Codificação de programas monolíticos.

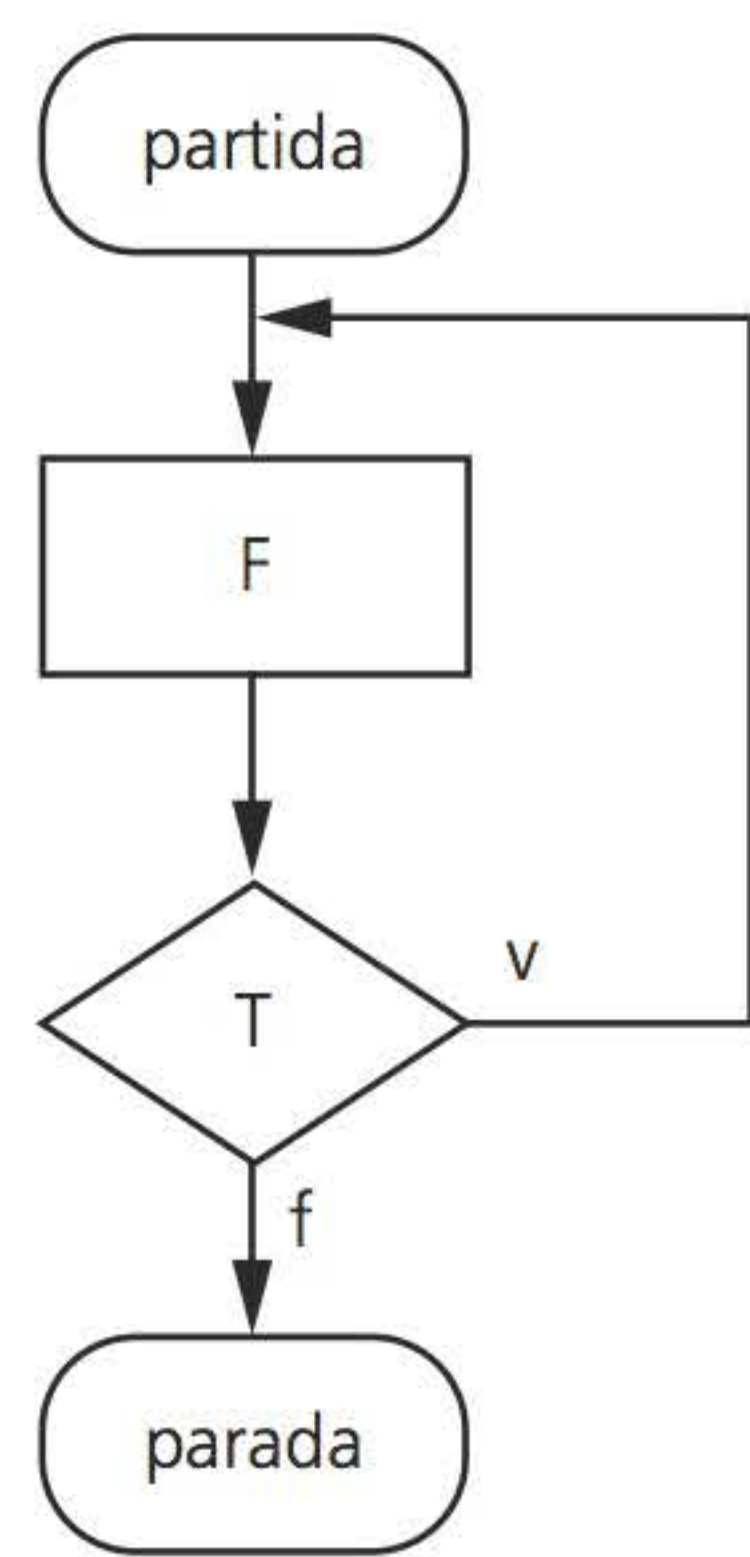


figura 4.5 Fluxograma.

exercício 4.12 Codifique o programa da figura 4.6 usando a codificação de programas monolíticos.

Programa monolítico M	
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 4.6 Programa monolítico.

exercício 4.13 Desenvolva um *software* que seja capaz de processar qualquer programa de máquina Norma. A entrada para o *software* contém:

- o programa no formato de instruções rotuladas simples;
- o valor de entrada a ser computado pelo programa;

A saída é a computação do programa para a dada entrada. Teste o *software* para o cálculo do fatorial de 3.

exercício 4.14 Mostre como a instrução abaixo pode ser construída como uma macro em Norma:

r_1 : se $A < 2$ então vá_para r_2 senão vá_para r_3

exercício 4.15 Desenvolva os programas, em Norma, que realizam as operações e testes abaixo:

- a** $A := B - C$
- b** $\text{div}(B, C)$: divisão inteira de **B** por **C**
- c** $\text{fat}(A)$: fatorial;
- d** $\text{pot}(x, y)$: potência;
- e** $\text{teste}(A > B)$
- f** $\text{teste}(A \geq B)$
- g** $\text{teste}(A \leq B)$
- h** $\text{mdc}(A, B)$: máximo divisor comum;
- i** $\text{mod}(A, B)$: resto da divisão inteira;
- j** $\text{primos}(A, B)$: todos os primos entre os valores de **A** e de **B**;
- k** $\text{teste_nperf}(A)$: verifica se é um número perfeito.

exercício 4.16 Desenvolva os programas, em Norma, que realizam as operações abaixo nos inteiros (utilize a representação sinal-magnitude introduzida no exemplo 4.15 - Inteiros):

- a** $A := B + C$
- b** $A := B \cdot C$
- c** $A := B - C$
- d** $A := \text{div}(B, C)$

exercício 4.17 Desenvolva os programas, em Norma, que realizam as operações abaixo nos números racionais (utilize a representação de racionais introduzida no exemplo 4.16 - Racionais):

- a** $A := B + C$
- b** $A := B \cdot C$
- c** $A := B - C$
- d** $A := \text{div}(B, C)$

exercício 4.18 Seja a *máquina Normaneg*, a qual é, em todos os aspectos, idêntica à 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 4.19 Seja $P = \{ I, 1 \}$ definido na máquina Norma e suponha o conteúdo dos registradores **A** e **B** igual a **0** e **5** respectivamente. Marque o programa que realiza $A := B$ na máquina Norma:

a

Programa monolítico

1: se **zero_B** então vá_para 5 senão vá_para 2
2: faça **ad_D** vá_para 3
3: faça **sub_B** vá_para 4
4: faça **sub_A** vá_para 1

b

Programa monolítico

1: se **zero_A** então vá_para 4 senão vá_para 2
2: faça **ad_A** vá_para 3
3: faça **ad_B** vá_para 1

c

Programa monolítico

1: se **zero_A** então vá_para 5 senão vá_para 2
2: faça **ad_A** vá_para 3
3: faça **sub_B** vá_para 1

d

Programa monolítico

1: se **zero_B** então vá_para 5 senão vá_para 2
2: faça **ad_A** vá_para 3
3: faça **sub_B** vá_para 1

e Nenhuma das alternativas anteriores está correta.

exercício 4.20 A máquina Norma é composta basicamente por um conjunto de registradores naturais e apenas três instruções sobre cada um deles (adição e subtração de uma unidade e verificação se o conteúdo do registrador é **zero**). Portanto, é correto afirmar que:

- a** Seus registradores podem assumir valores naturais tão grandes quanto necessários;
- b** Os registradores podem assumir qualquer valor;
- c** Seu modelo formal encontra-se baseado em uma estrutura chamada fita;
- d** Não é capaz de simular mecanismos de recursão;
- e** Pode ser definida por uma 8-upla da forma $(\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$.

exercício 4.21 Sobre a máquina Norma, analise as seguintes afirmações:

- I. É uma máquina extremamente simples, porém com poder computacional igual ao de qualquer computador atual;
- II. Em norma, sub-rotinas e mecanismos de recursão, como os definidos em programas recursivos, podem ser simulados por programas monolíticos, usando endereçamento indireto;
- III. É impossível programar operações matemáticas complexas (radiciação, seno e cosseno, por exemplo) em Norma, visto que apenas as instruções de somar/diminuir 1 unidade a um registrador não dão suporte para que isso seja feito.

Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas III está correta;
- ☐ **d** Apenas I e II estão corretas;
- ☐ **e** Apenas II e III estão corretas.

exercício 4.22 Sobre a máquina Norma, analise as seguintes afirmações:

- I. Possui um conjunto finito de registradores naturais como memória;
- II. Possui poder computacional, no mínimo, igual ao de qualquer computador moderno;
- III. Sub-rotinas e mecanismos de recursão podem ser simulados por programas monolíticos, usando endereçamento indireto.

Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas I e III estão corretas;
- ☐ **d** Apenas II e III estão corretas;
- ☐ **e** I, II e III estão corretas.

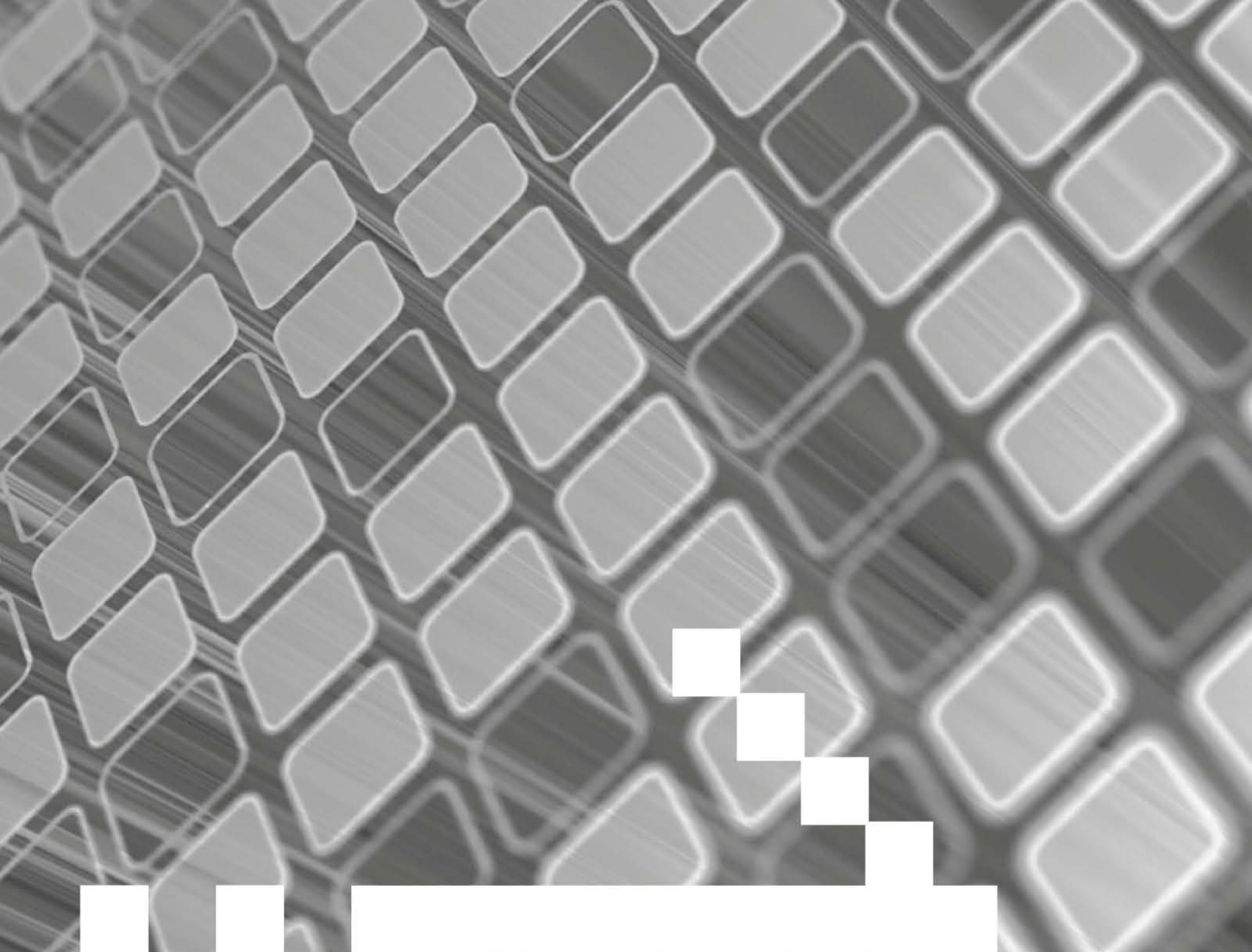
Termos-chaves

codificação de programas, p. 106

máquina universal, p. 105

máquina Norma, p. 105, p.109

Número de Gödel, p. 105



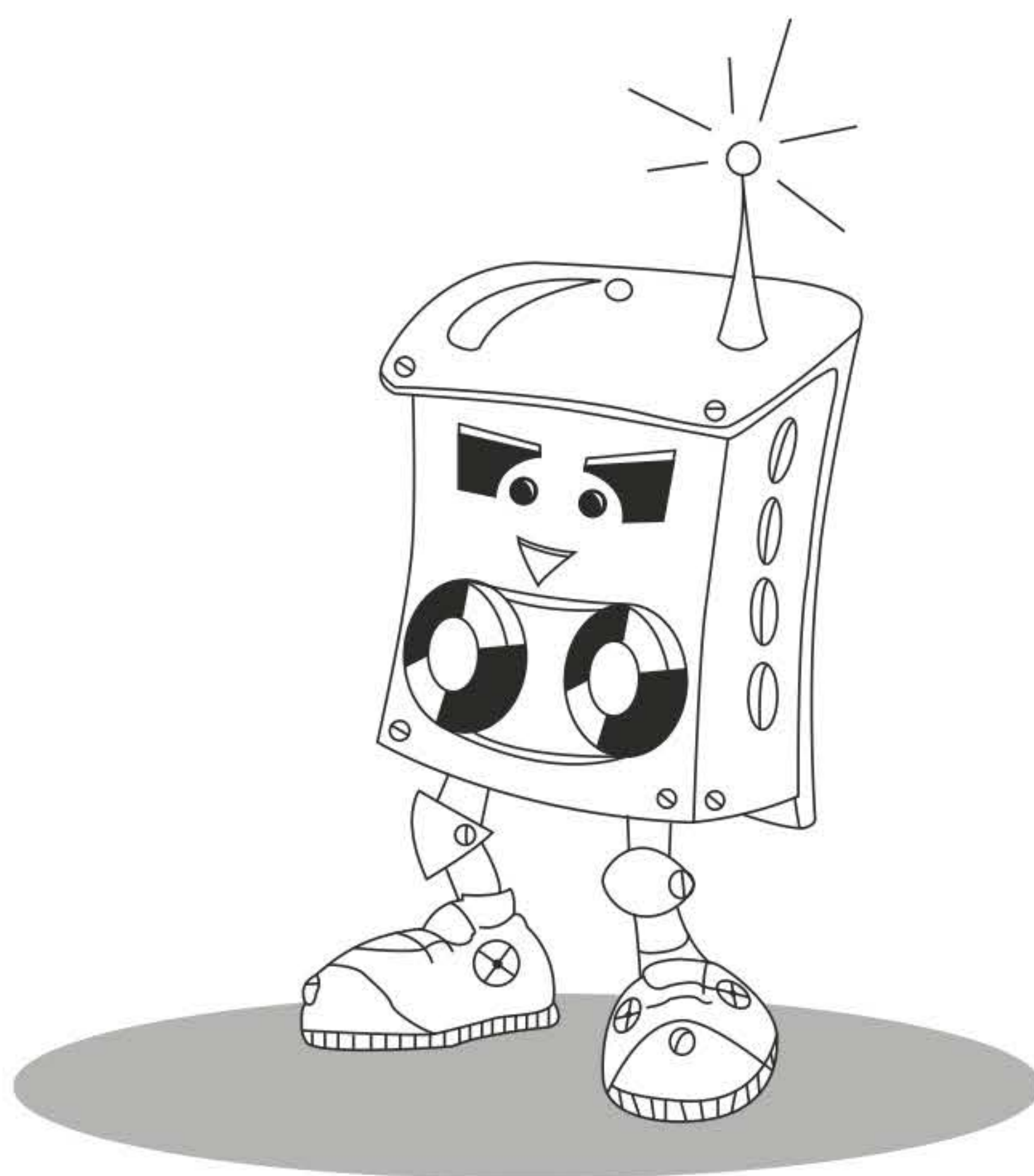
capítulo

5

máquina de Turing

■ ■ Provavelmente, o modelo mais utilizado como formalização de algoritmo é a máquina de Turing. Trata-se de um modelo muito simples (inspirado em uma pessoa que realiza cálculos) e similar a um computador atual, embora tenha sido proposto alguns anos antes do primeiro computador digital.

A máquina de Turing é uma máquina universal (algumas evidências de tal fato são apresentadas) e são estudadas duas abordagens: reconhecimento de linguagens e processamento de funções. Uma terceira abordagem (solucionabilidade de problemas) é estudada no capítulo subsequente.



Provavelmente, o modelo mais utilizado como formalização de *algoritmo* é a *máquina de Turing*, proposta em 1936, por Alan Turing. O trabalho de Turing é particularmente significativo por ter sido o primeiro a identificar programas escritos para uma “máquina computacional”, como noções intuitivas do efetivamente computável.

Basicamente, uma máquina de Turing é um mecanismo simples, que formaliza a ideia de uma pessoa que realiza cálculos, usando um instrumento de escrita e um apagador. O modelo formal é baseado em uma *fita* infinita (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.

No que se refere aos recursos “tanto quanto necessários”, entende-se que a máquina de Turing possui “tantas células de armazenamento de dados quanto necessário”.

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. Algumas são relativamente simples de serem enunciadas como, por exemplo, uma função que nomeia todas as funções.

5.1

→ 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 em uma fita ou 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 sequências 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 deve ser finito. Para representar informações mais complexas, é possível utilizar sequências 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.

5.2

→ 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.

Adota-se que a fita é finita à esquerda e infinita (tão grande quanto necessário) à direita e é dividida em células, cada uma armazenando 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". Isso é ilustrado na figura 5.1, onde β e $\odot$ representam "branco" e "marcador de início de fita", respectivamente.

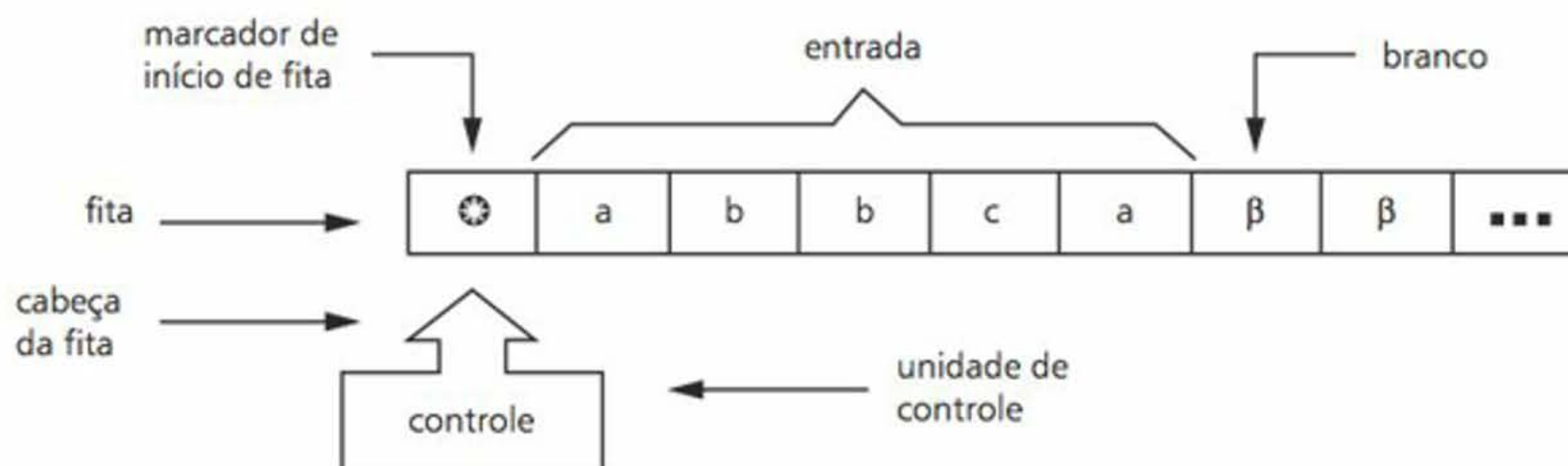


figura 5.1 Fita e unidade de controle de uma máquina de Turing.

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.

5.3

→ modelo formal

definição 5.1 – Máquina de Turing

Uma *máquina de Turing* é uma 8-upla:

$$M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$$

onde:

Σ alfabeto de símbolos de entrada;

Q conjunto de estados possíveis da máquina, o qual é finito;

Π programa ou função de transição:

$$\Pi: Q \times (\Sigma \cup V \cup \{\beta, \odot\}) \rightarrow Q \times (\Sigma \cup V \cup \{\beta, \odot\}) \times \{E, D\}$$

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;
- β símbolo especial *branco*;
- $\odot$ 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

- estado corrente
- símbolo lido da fita

para determinar

- 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:

$$\Pi(\text{estado corrente, símbolo lido}) = (\text{novo estado, símbolo gravado, sentido do movimento})$$

ou seja (suponha que $p, q \in Q$, $a_u, a_v \in (\Sigma \cup V \cup \{\beta, \odot\})$ 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 5.2. Neste caso, os estados iniciais e finais são representados como ilustrado na figura 5.3.

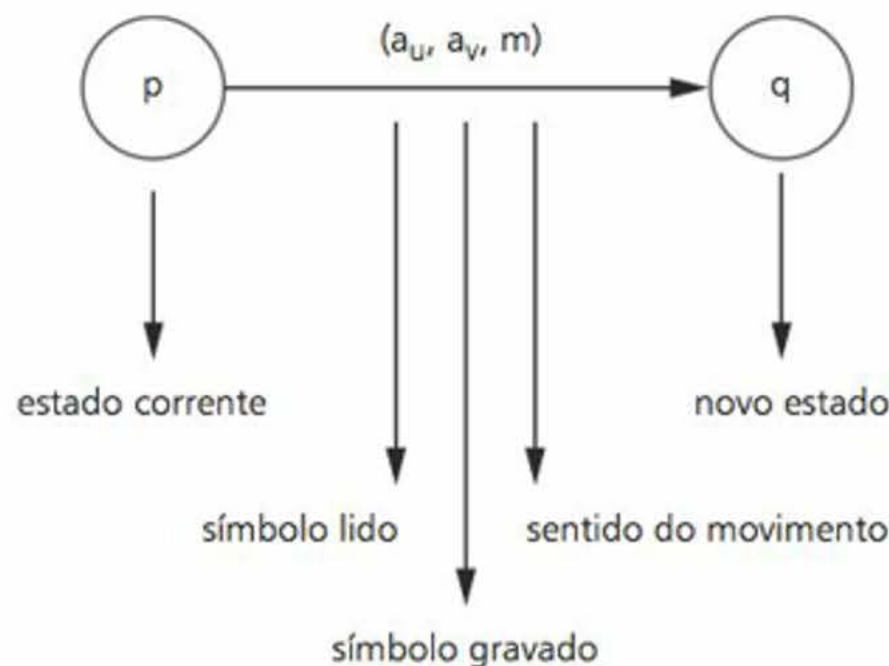


figura 5.2 Representação da função programa como um grafo.



figura 5.3 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 5.4 (ilustrada para o caso $\Pi(p, a_u) = (q, a_v, m)$).

Π	$\odot$	...	a_u	...	a_v	...	β
p			(q, a_v, m)				
q							
...							

figura 5.4 Representação da função programa como uma tabela.

O processamento de uma máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ 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 atingir uma condição de parada (se ocorrer), pois eventualmente pode ficar em *loop* infinito.

Ao parar, duas informações estão disponíveis e podem ser usadas como resultado ou saída do processamento: o estado da máquina e o conteúdo da fita. Em relação ao estado, a parada do processamento 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 para 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 para 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 para e a palavra de entrada é rejeitada.

observação 5.2 – Máquina de Turing $\times$ (programa e máquina)

É importante observar 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. $\square$

observação 5.3 – Definição alternativa de máquina de Turing

Existem diversas definições alternativas de máquina de Turing na bibliografia em geral. Note-se que as variações entre as diversas definições não alteram o poder computacional do formalismo. As variações mais comuns estão nas características da fita e no movimento da cabeça como, por exemplo:

- a** *Inexistência do marcador de início de fita.* É frequente 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** *Fita infinita nos dois lados.* Trata-se de uma definição alternativa tradicional de máquina de Turing, com infinitas células tanto para a esquerda quanto para a direita;
- c** *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;
- d** *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.

De certa forma, estas definições alternativas são algumas *evidências internas* de que se trata de uma máquina universal. □

5.4

→ máquinas de Turing como reconhecedores de linguagens

Uma das abordagens do estudo das máquinas de Turing, ou de 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 5.4 – Linguagem aceita por uma máquina de Turing

Seja $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ 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:

$$\text{ACEITA}(M) = \{ w \mid M \text{ ao processar } w \in \Sigma^* \text{ para em um estado } q_f \in F \}$$

- b** A *linguagem rejeitada* por M , denotada por **REJEITA**(M) é o conjunto de todas as palavras de Σ^* rejeitadas por M , ou seja:

$$\text{REJEITA}(M) = \{ w \mid M \text{ ao processar } w \in \Sigma^* \text{ para em um estado } q \notin F \}$$

- c** A linguagem para a 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:

$$\text{ACEITA}(M) \cup \text{REJEITA}(M) \cup \text{LOOP}(M) = \Sigma^*$$

$$\text{ACEITA}(M) \cap \text{REJEITA}(M) = \emptyset$$

$$\text{ACEITA}(M) \cap \text{LOOP}(M) = \emptyset$$

$$\text{REJEITA}(M) \cap \text{LOOP}(M) = \emptyset$$

e, portanto:

$$ACEITA(M) \cap REJEITA(M) \cap LOOP(M) = \emptyset$$

Consequentemente, o complemento de:

$$ACEITA(M) \text{ é } REJEITA(M) \cup LOOP(M)$$

$$REJEITA(M) \text{ é } ACEITA(M) \cup LOOP(M)$$

$$LOOP(M) \text{ é } ACEITA(M) \cup REJEITA(M)$$

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 5.5.

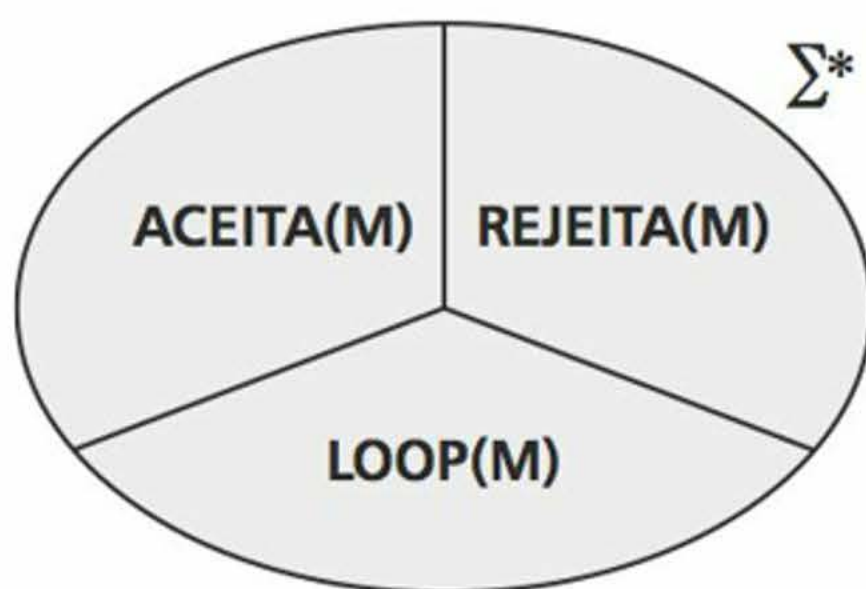


figura 5.5 Particionamento do conjunto de palavras em classes de equivalência induzido por uma máquina de Turing.

exemplo 5.1 – Máquina de Turing – duplo balanceamento

Considere a linguagem:

$$\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$$

A máquina de Turing:

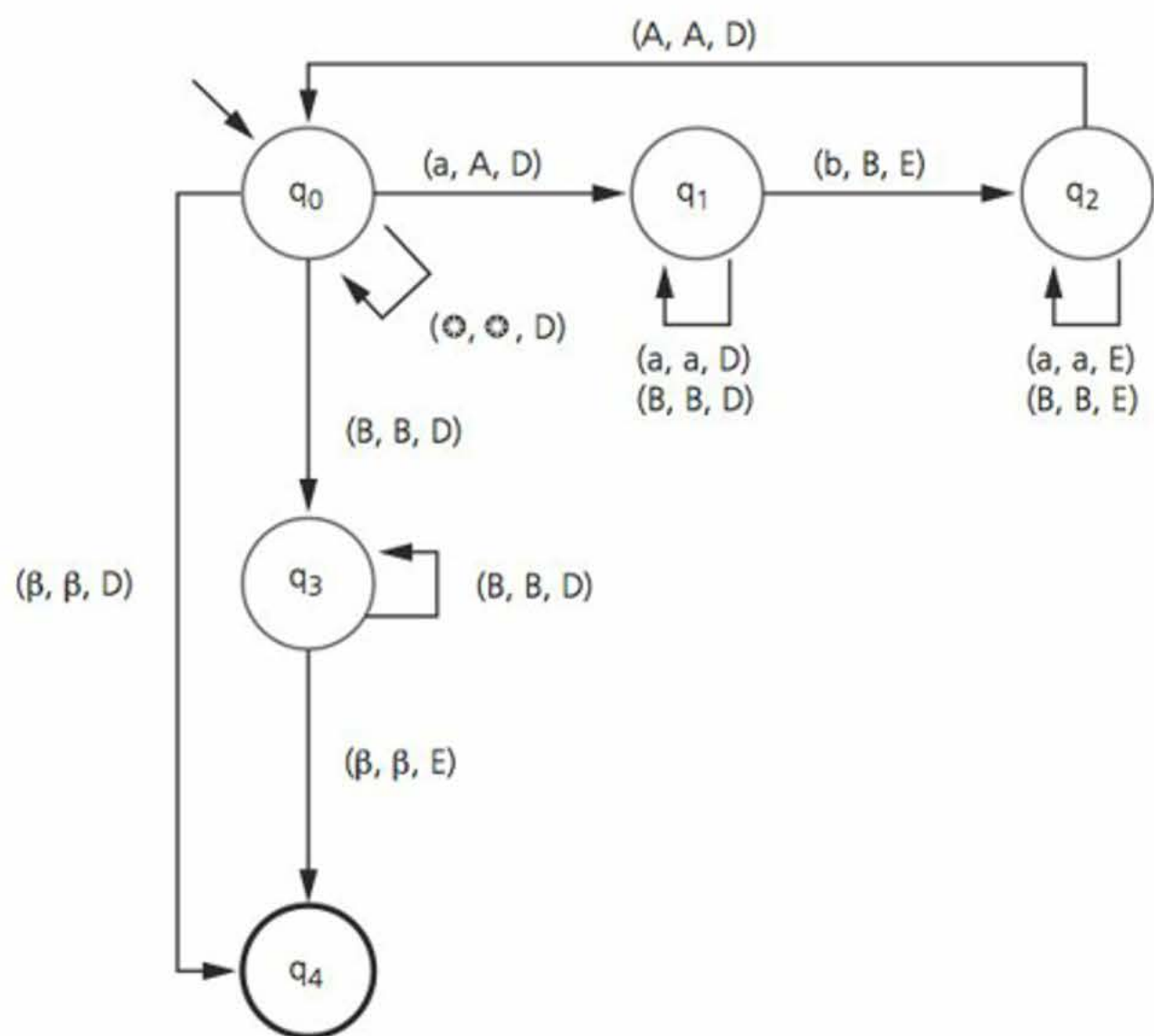
$$MT_Duplo_Bal = (\{ a, b \}, \{ q_0, q_1, q_2, q_3, q_4 \}, \Pi, q_0, \{ q_4 \}, \{ A, B \}, \beta, \odot)$$

ilustrada na figura 5.6, é tal que:

$$ACEITA(MT_Duplo_Bal) = \text{Duplo_Bal}$$

$$REJEITA(MT_Duplo_Bal) = \Sigma^* - \text{Duplo_Bal}$$

e, portanto, $LOOP(MT_Duplo_Bal) = \emptyset$



Π	$\odot$	a	b	A	B	β
q_0	$(q_0, \odot, D)$	(q_1, A, D)			(q_3, B, D)	(q_4, β, 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, β, E)
q_4						

figura 5.6 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 para a 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 5.7 ilustra a sequência do processamento da máquina de Turing **Duplo_Bal** para a entrada $w = aabb$.

Essa 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. $\square$

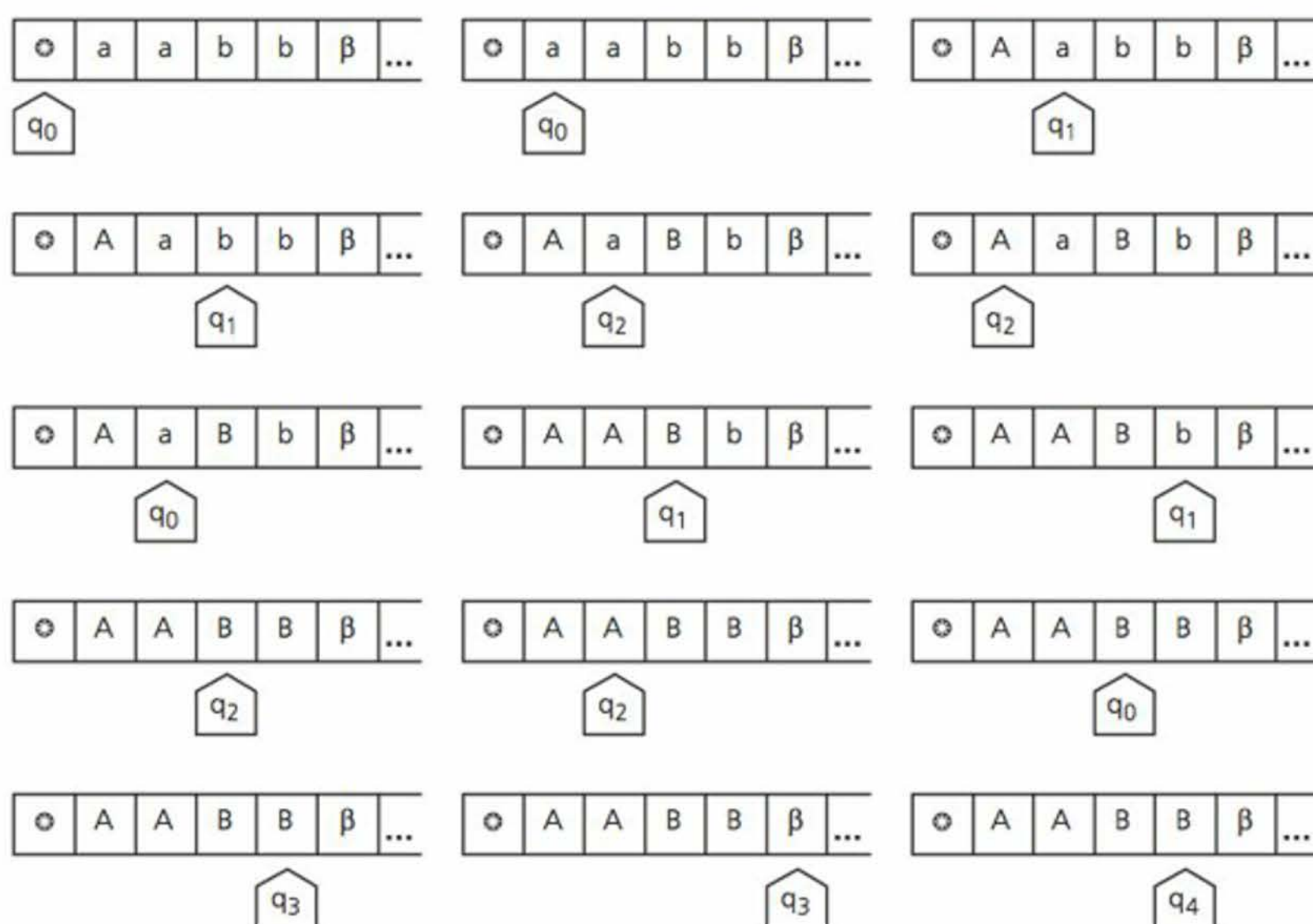


figura 5.7 Computação de uma máquina de Turing.

Uma linguagem aceita por uma máquina de Turing é dita linguagem 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 das *funções recursivas*, assunto estudado em capítulo específico.

definição 5.5 – Linguagem enumerável recursivamente

Uma linguagem aceita por uma máquina de Turing é dita *linguagem enumerável recursivamente* ou *linguagem recursivamente enumerável* $\square$

exemplo 5.2 – 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 5.1 (as demais são sugeridas como exercício):

- a** $\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$
- b** $\text{Triplo_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$
- c** $\text{Palavra_Palavra} = \{ ww \mid w \text{ é palavra sobre os símbolos } a \text{ e } b \}$

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. $\square$

Segundo a *hipótese de Church* a ser estudada adiante, a máquina de Turing é considerada o dispositivo mais genérico de computação. Portanto, a classe das linguagens enumeráveis recursivamente define a classe de todas as linguagens que podem ser reconhecidas mecanicamente.

Observe que a classe das linguagens enumeráveis recursivamente inclui algumas linguagens 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 pertence a L e que, ao ser processada por M , leva a máquina a entrar em *loop* infinito. Assim, pode-se afirmar que:

- se w pertence a L , M para e aceita a entrada;
- se w não pertence a L (w pertence ao complemento de L , ou seja, $\Sigma^* - L$) 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 para para qualquer entrada, aceitando ou rejeitando.

definição 5.6 – Linguagem recursiva

Uma linguagem L é dita *linguagem recursiva* se existe uma máquina de Turing M tal que:

$$\begin{aligned} \text{ACEITA}(M) &= L \\ \text{REJEITA}(M) &= \Sigma^* - L \\ \text{LOOP}(M) &= \emptyset \end{aligned}$$

 $\square$

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 para*, 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 para. Entretanto, tais reconhecedores podem ser muito ineficientes.

exemplo 5.3 – Linguagem recursiva

É fácil verificar que as seguintes linguagens são recursivas:

- a** $\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$
- b** $\text{Triplo_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$
- c** $\{ w \mid w \in \{ a, b \}^* \text{ e tem o dobro de símbolos } a \text{ que } b \}$ □

A seguir, são enunciadas algumas das principais propriedades das linguagens enumeráveis recursivamente e das linguagens recursivas. Tais propriedades podem ser resumidas como segue:

- a** O complemento de uma linguagem recursiva é uma linguagem recursiva. Consequentemente, existe um algoritmo que sempre para e que reconhece o complemento da linguagem;
- b** Uma linguagem L é recursiva se, e somente se, L e seu complemento são linguagens enumeráveis recursivamente;
- c** A classe das linguagens recursivas está contida propriamente na classe das linguagens enumeráveis recursivamente.

A primeira propriedade é demonstrada a seguir e as demais são detalhadas e demonstradas nos capítulos subsequentes.

teorema 5.7 – 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 para para qualquer entrada. Ou seja:

$$\begin{aligned} \text{ACEITA}(M) &= L \\ \text{REJEITA}(M) &= \Sigma^* - L \\ \text{LOOP}(M) &= \emptyset \end{aligned}$$

Seja M' uma máquina universal construída a partir de M , mas invertendo-se as condições de ACEITA por REJEITA e vice-versa. Portanto, M' aceita $\Sigma^* - L$ e sempre para para qualquer entrada. Ou seja:

$$\begin{aligned} \text{ACEITA}(M') &= \Sigma^* - L \\ \text{REJEITA}(M') &= L \\ \text{LOOP}(M') &= \emptyset \end{aligned}$$

Logo $\Sigma^* - L$ é uma linguagem recursiva. □

5.5

→ 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 podem 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 compará-la e diferenciá-la da definição de função computada introduzida no capítulo 2 – Programas, máquinas e computações.

definição 5.8 – 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 $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ que computa f , ou seja:

- a** Para $(w_1, w_2, \dots, w_n) \in (\Sigma^*)^n$, tem-se que a palavra de entrada para M é:

$$\odot w_1 w_2 \dots w_n$$

- b** Se f é definida para $(w_1, w_2, \dots, w_n)$, então o processamento de M para a entrada $\odot w_1 w_2 \dots w_n$ é tal que:

- para (aceitando ou rejeitando);
- o conteúdo da fita é (excetuando-se os símbolos brancos):

$$\odot w$$

- c** Se f é indefinida para $(w_1, w_2, \dots, w_n)$, então M , ao processar a entrada $\odot w_1 w_2 \dots w_n$, fica em *loop* infinito. □

observação 5.9 – Função Turing-computável x condição de parada

Na definição acima, é considerado como resultado do processamento de M somente o conteúdo gravado na fita. Portanto, em relação à função computável, um processamento que para em um estado não final é perfeitamente válido. Se a função computável não for definida para a entrada, é porque o processamento é infinito. □

observação 5.10 – 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 de delimitação, como exemplificado a seguir para a n -upla $(w_1, w_2, \dots, w_n)$:

$$\odot w_1 \# w_2 \# \dots \# w_n$$

□

exemplo 5.4 - Máquina de Turing: concatenação

Considere a função (total):

concatenação: $(\{ a, b \}^*)^2 \rightarrow \{ a, b \}^*$

tal que associa ao par (w_1, w_2) a palavra $w_1 w_2$. A máquina de Turing:

$\text{Conc} = (\{ a, b, \# \}, \{ q_0, q_1, q_2, q_3, q_4 \}, \Pi, q_0, \{ q_4 \}, \emptyset, \beta, \odot)$

é ilustrada na figura 5.8.

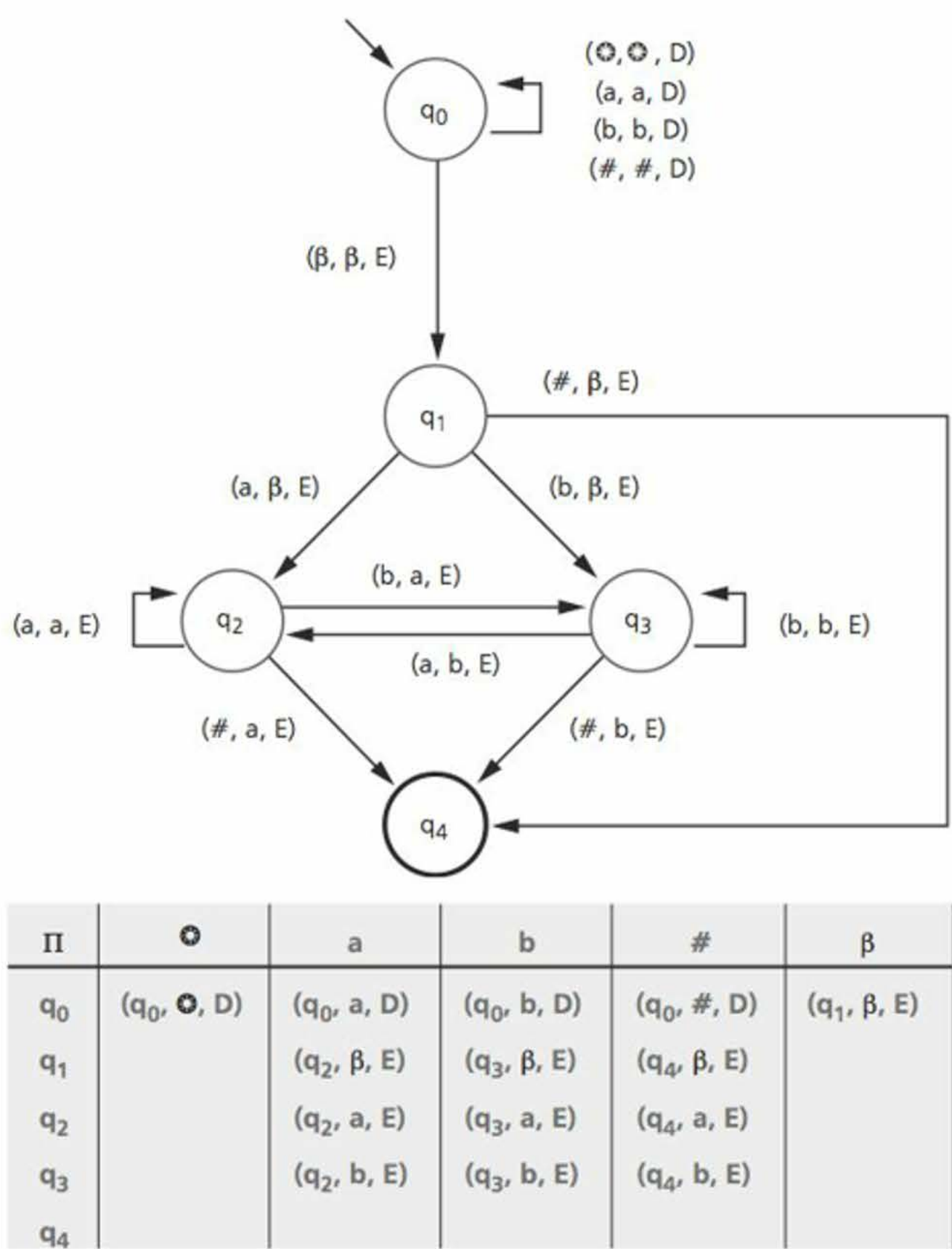


figura 5.8 Grafo e tabela de transições da máquina de Turing - concatenação.

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 para. 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 5.5 - 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:

$$\text{Quadr} = (\{ 1 \}, \{ q_0, q_1, q_2, \dots, q_{13} \}, \Pi, q_0, \{ q_{13} \}, \{ A, B, C \}, \beta, \odot)$$

ilustrada na figura 5.10 e onde Π é como na figura 5.9.

Π	$\odot$	1	A	B	C	β
q_0	$(q_0, \odot, D)$	(q_1, A, D)		(q_0, B, D)		(q_3, β, 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	$(q_{13}, \odot, D)$			(q_4, β, E)		
q_4			(q_5, A, D)	(q_4, B, E)		
q_5				(q_6, C, E)		(q_{12}, β, E)
q_6	$(q_7, \odot, D)$		$(q_6, 1, E)$		(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}	$(q_{13}, \odot, D)$		$(q_{12}, 1, E)$		$(q_{12}, 1, E)$	
q_{13}						

figura 5.9 Tabela de transições da máquina de Turing – função quadrado.

O algoritmo apresentado recebe como entrada a palavra:

$$\odot n_1$$

onde n_1 denota o valor n representado em unário sobre $\{1\}$. Assim, $(n_1)^2$ é simplesmente n_1 concatenado consigo mesmo, ou seja:

$$(n_1)^2 = n_1 n_1$$

Tal concatenação é obtida como segue:

- no ciclo em q_0 , q_1 e q_2 , é gerado $\odot n_A n_B$ (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 $n_A (n-1)_B$
- em q_5 até q_{11} , a subpalavra $(n-1)_B$ é usada para controlar concatenações sucessivas, resultando em:

$$\odot n_A (n-1)_C (n-1)_1 (n-1)_1 \dots (n-1)_1 \quad \text{onde } (n-1)_1 \text{ é repetida } n-1 \text{ vezes}$$

- por fim, em q_{12} , as subpalavras $n_A (n-1)_C$ são substituídas por $n_1 (n-1)_1$, resultando em:

$$\odot n_1 (n-1)_1 (n-1)_1 (n-1)_1 \dots (n-1)_1 \quad \text{onde } (n-1)_1 \text{ é repetida } n \text{ vezes}$$

- ou seja, o comprimento da palavra resultante é:

$$n + (n-1) \cdot n = n + (n^2 - n) = n^2$$

A figura 5.11 ilustra a sequência do processamento da máquina de Turing **Quadr** para a entrada $w = 111$ (ou seja, 3 em unário). Por simplicidade, a representação está na forma de sequência de palavras, sendo que cada palavra representa uma descrição instantânea da máquina na qual $(q_i)x$ indica o estado corrente q_i e o posicionamento da cabeça da fita, apontando para a célula que contém o símbolo x . □

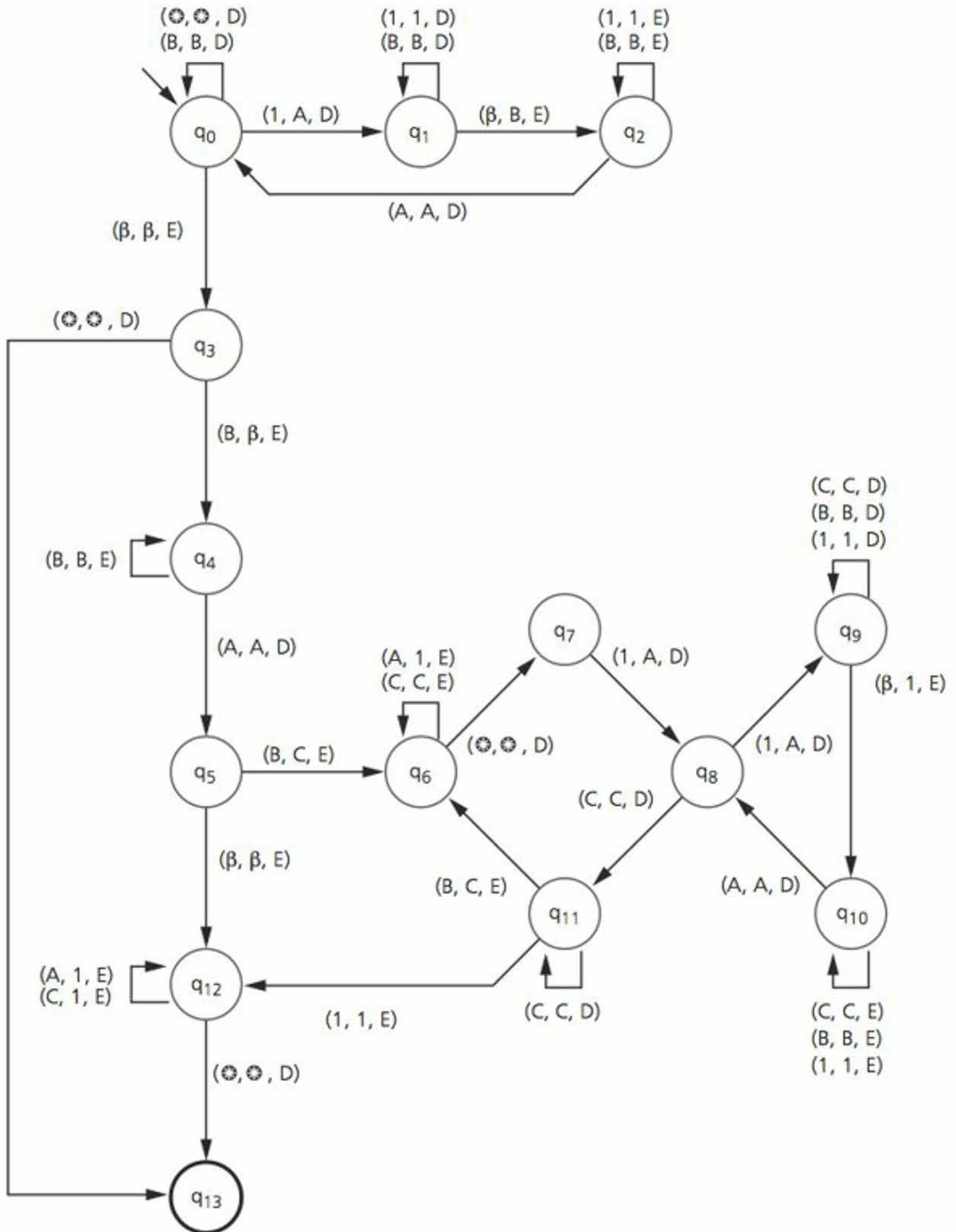


figura 5.10 Grafo da máquina de Turing – função quadrado.

$(q_0)\odot 111\beta$	$\odot(q_6)A11CB\beta$	$\odot AA1CC(q_9)11\beta$
$\odot(q_0)111\beta$	$(q_6)\odot 111CB\beta$	$\odot AA1CC1(q_9)1\beta$
$\odot A(q_1)11\beta$	$\odot(q_7)111CB\beta$	$\odot AA1CC11(q_9)\beta$
$\odot A1(q_1)1\beta$	$\odot A(q_8)11CB\beta$	$\odot AA1CC1(q_{10})11\beta$
$\odot A11(q_1)\beta$	$\odot AA(q_9)1CB\beta$	$\odot AA1CC(q_{10})111\beta$
$\odot A1(q_2)1B\beta$	$\odot AA1(q_9)CB\beta$	$\odot AA1C(q_{10})C111\beta$
$\odot A(q_2)11B\beta$	$\odot AA1C(q_9)B\beta$	$\odot AA1(q_{10})CC111\beta$
$\odot(q_2)A11B\beta$	$\odot AA1CB(q_9)\beta$	$\odot AA(q_{10})1CC111\beta$
$\odot A(q_0)11B\beta$	$\odot AA1C(q_{10})B1\beta$	$\odot A(q_{10})A1CC111\beta$
$\odot AA(q_1)1B\beta$	$\odot AA1(q_{10})CB1\beta$	$\odot AA(q_8)1CC111\beta$
$\odot AA1(q_1)B\beta$	$\odot AA(q_{10})1CB1\beta$	$\odot AAA(q_9)CC111\beta$
$\odot AA1B(q_1)\beta$	$\odot A(q_{10})A1CB1\beta$	$\odot AAAC(q_9)C111\beta$
$\odot AA1(q_2)BB\beta$	$\odot AA(q_8)1CB1\beta$	$\odot AAACC(q_9)111\beta$
$\odot AA(q_2)1BB\beta$	$\odot AAA(q_9)CB1\beta$	$\odot AAACC1(q_9)11\beta$
$\odot A(q_2)A1BB\beta$	$\odot AAAC(q_9)B1\beta$	$\odot AAACC11(q_9)1\beta$
$\odot AA(q_0)1BB\beta$	$\odot AAACB(q_9)1\beta$	$\odot AAACC111(q_9)\beta$
$\odot AAA(q_1)BB\beta$	$\odot AAACB1(q_9)\beta$	$\odot AAACC11(q_{10})11\beta$
$\odot AAAB(q_1)B\beta$	$\odot AAACB(q_{10})11\beta$	$\odot AAACC1(q_{10})111\beta$
$\odot AAABB(q_1)\beta$	$\odot AAAC(q_{10})B11\beta$	$\odot AAACC(q_{10})1111\beta$
$\odot AAAB(q_2)BB\beta$	$\odot AAA(q_{10})CB11\beta$	$\odot AAAC(q_{10})C1111\beta$
$\odot AAA(q_2)BBB\beta$	$\odot AA(q_{10})ACB11\beta$	$\odot AAA(q_{10})CC1111\beta$
$\odot AA(q_2)ABBB\beta$	$\odot AAA(q_8)CB11\beta$	$\odot AA(q_{10})ACC1111\beta$
$\odot AAA(q_0)BBB\beta$	$\odot AAAC(q_{11})B11\beta$	$\odot AAA(q_8)CC1111\beta$
$\odot AAAB(q_0)BB\beta$	$\odot AAA(q_6)CC11\beta$	$\odot AAAC(q_{11})C1111\beta$
$\odot AAABB(q_0)B\beta$	$\odot AA(q_6)ACC11\beta$	$\odot AAACC(q_{11})1111\beta$
$\odot AAABBB(q_0)\beta$	$\odot A(q_6)A1CC11\beta$	$\odot AAAC(q_{12})C1111\beta$
$\odot AAABB(q_3)B\beta$	$\odot(q_6)A11CC11\beta$	$\odot AAA(q_{12})C11111\beta$
$\odot AAAB(q_4)B\beta$	$(q_6)\odot 111CC11\beta$	$\odot AA(q_{12})A111111\beta$
$\odot AAA(q_4)BB\beta$	$\odot(q_7)111CC11\beta$	$\odot A(q_{12})A1111111\beta$
$\odot AA(q_4)ABB\beta$	$\odot A(q_8)11CC11\beta$	$\odot(q_{12})A11111111\beta$
$\odot AAA(q_5)BB\beta$	$\odot AA(q_9)1CC11\beta$	$(q_{12})\odot 111111111\beta$
$\odot AA(q_6)ACB\beta$	$\odot AA1(q_9)CC11\beta$	$\odot(q_{13})111111111\beta$
$\odot A(q_6)A1CB\beta$	$\odot AA1C(q_9)C11\beta$	

figura 5.11 Computação da máquina de Turing – função quadrado.

definição 5.11 – 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 $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ que computa f e sempre para para qualquer entrada. $\square$

A definição de uma função Turing-computável total garante que, para todos os valores de entrada, a função está definida, ou seja, existe uma máquina de Turing que a computa e que sempre para para qualquer entrada. Sugere-se como exercício verificar se os exemplos acima de funções Turing-computáveis são totais.

observação 5.12 – Máquina de Turing como reconhecedor x 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 de rejeição, gravar na fita uma informação sobre tal condição e, somente após, parar. Esta informação pode ser do tipo binário 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. Nesse 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.

Consequentemente, são conceitos relacionados:

- Função Turing-computável e linguagem enumerável recursivamente;
- Função Turing-computável total e linguagem recursiva.



5.6

→ conclusões

Neste capítulo, foi visto o formalismo máquina de Turing, um dos modelos mais utilizados como formalização de algoritmo. Trata-se de um modelo simples, poderoso e que formaliza a ideia de uma pessoa que realiza cálculos. É muito similar a um computador moderno, embora tenha sido proposto alguns anos antes do primeiro computador digital. Na realidade, não se trata de uma máquina no sentido usual dado a essa palavra, mas sim de um programa para uma máquina universal.

Duas das abordagens da máquina de Turing foram apresentadas: reconhecimento de linguagens e processamento de funções. Uma terceira abordagem será vista em um capítulo subsequente: solucionabilidade de problemas. Algumas evidências de que se trata de uma máquina universal foram apresentadas.

5.7

→ exercícios

exercício 5.1 Qual a importância do estudo da máquina de Turing na ciência da computação?

exercício 5.2 Desenvolva máquinas de Turing que aceitem as seguintes linguagens:

- a** $L_1 = \emptyset$
- b** $L_2 = \{ \epsilon \}$
- c** $L_3 = \{ w \mid w \text{ tem o mesmo número de símbolos } a \text{ e } b \}$
- d** $L_4 = \{ w \mid \text{o décimo símbolo da direita para a esquerda é } a \}$
- e** $L_5 = \{ waw \mid w \text{ é palavra de } \{ a, b \}^* \}$
- f** $L_6 = \{ ww \mid w \text{ é palavra de } \{ a, b \}^* \}$
- g** $L_7 = \{ ww^r \mid w \text{ é palavra de } \{ a, b \}^* \text{ onde } w^r \text{ denota a palavra reversa de } w \}$
- h** $L_8 = \{ www \mid w \text{ é palavra de } \{ a, b \}^* \}$
- i** $L_9 = \{ w \mid w = a^1 b^2 a^3 b^4 \dots a^{n-1} b^n \text{ e } n \text{ é número natural par} \}$
- j** $L_{10} = \{ w \mid w = a^n b^n \text{ ou } w = b^n a^n \}$
- k** $L_{11} = \{ w \mid w = a^i b^j c^k, \text{ onde ou } i = j \text{ ou } j = k \}$
- l** $\text{Triplo_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$
- m** $\text{Dobro} = \{ w \mid w \in \{ a, b \}^* \text{ e tem o dobro de símbolos } a \text{ que } b \}$

exercício 5.3 Desenvolva máquinas de Turing que realizam as operações e testes abaixo:

- a** $A - B$: subtração nos inteiros
- b** $\text{div}(A, B)$: divisão inteira de A por B
- c** $\text{fat}(A)$: fatorial;
- d** $\text{pot}(A, B)$: potência;
- e** $\text{teste}(A > B)$: nos inteiros
- f** $\text{teste}(A \geq B)$: nos inteiros
- g** $\text{teste}(A \leq B)$: nos inteiros
- h** $\text{mdc}(A, B)$: máximo divisor comum;
- i** $\text{teste_primo_entre_si}(A, B)$: verifica se são primos entre si, ou seja, se $\text{mdc}(A, B) = 1$;
- j** $\text{mod}(A, B)$: resto da divisão inteira;
- k** $\text{teste_primo}(A)$: verifica se A é primo;
- l** $\text{teste_nperf}(A)$: verifica se é um número perfeito.

exercício 5.4 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. Então:

Construa uma máquina de Turing sobre $\Sigma = \{ v, f, e, \text{ou} \}$ tal que:

$$\text{ACEITA}(M) = \{ EB \mid EB \text{ é } v \}$$

$$\begin{aligned} \text{REJEITA}(M) &= \{ EB \mid EB \text{ é f} \} \\ \text{LOOP}(M) &= \{ w \mid w \text{ não é EB} \} \end{aligned}$$

exercício 5.5 Desenvolva uma máquina de Turing que receba com entrada uma palavra sobre o alfabeto $\{a, b, c\}$ sem qualquer ordem e ordene os símbolos supondo que $a < b$ e $b < c$. Por exemplo, para a entrada **abccba**, o resultado é **aabbcc**.

exercício 5.6 Desenvolva uma máquina de Turing sobre o alfabeto $\{ (,) \}$ que verifique o correto uso de parênteses em expressões matemáticas. Por exemplo, são expressões válidas:

$()$
 $((()))$
 $((())())()$

exercício 5.7 Desenvolva máquinas de Turing que aceitem as seguintes linguagens:

Sugestão: procure usar máquinas de Turing já definidas em exercícios anteriores e implemente uma noção de macro ou de composição de máquinas de forma modular.

- a** $L_{12} = \{ (ww^r)^3 \mid w \text{ é palavra de } \{a, b\}^* \text{ onde } w^r \text{ denota a palavra reversa de } w \}$
- b** $L_{13} = \{ ww^r w^r w \mid w \text{ é palavra de } \{a, b\}^* \text{ onde } w^r \text{ denota a palavra reversa de } w \}$
- c** $L_{14} = \{ (a^n b^n)^m \mid n \text{ e } m \text{ são números naturais} \}$

exercício 5.8 Qual a diferença fundamental entre a classe das linguagens recursivas e a das linguagens enumeráveis recursivamente? Qual a importância de se distinguir essas duas classes?

exercício 5.9 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 5.10 Dê a máquina de Turing **MT**, sobre o alfabeto $\{a, b\}$, tal que:

$$\begin{aligned} \text{ACEITA}(\text{MT}) &= \{ w \mid w \text{ tem o mesmo número de símbolos } a \text{ e } b \} \\ \text{REJEITA}(\text{MT}) &= \{ w \mid w \text{ cuja diferença entre o número de símbolos } a \text{ e } b \text{ é } 1 \} \\ \text{LOOP}(\text{MT}) &= \{a, b\}^* - (\text{ACEITA}(\text{MT}) \cup \text{REJEITA}(\text{MT})) \end{aligned}$$

Por exemplo:

$$\begin{aligned} ab &\in \text{ACEITA}(\text{MT}) \\ aba &\in \text{REJEITA}(\text{MT}) \\ aaba &\in \text{LOOP}(\text{MT}). \end{aligned}$$

exercício 5.11 Considere a seguinte máquina de Turing cuja função programa Π é como na figura 5.12:

$$(\{a, b\}, \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \Pi, q_0, \{q_7\}, \{A, B\}, \beta, \odot)$$

Π	$\odot$	a	b	A	B	β
q_0	$(q_0, \odot, D)$	(q_1, A, D)	(q_0, b, D)	(q_0, A, D)	(q_0, B, D)	(q_4, β, E)
q_1		(q_1, a, D)	(q_1, b, D)		(q_2, B, E)	(q_2, β, E)
q_2	$(q_5, \odot, D)$	(q_2, a, E)	(q_3, B, E)	(q_2, A, E)	(q_2, B, E)	
q_3	$(q_0, \odot, D)$	(q_3, a, E)	(q_3, b, E)	(q_0, A, D)		
q_4	$(q_7, \odot, D)$		(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	$(q_6, \odot, D)$	(q_6, a, D)	(q_6, b, D)	(q_6, A, D)	(q_6, B, D)	(q_6, β, E)
q_7						

figura 5.12 Tabela de transições da máquina de Turing.

- a** Verifique qual a configuração final após a computação para as seguintes palavras:
 ab
 aba
 aaba
- b** Qual a linguagem aceita?
- c** A linguagem aceita é apenas enumerável recursivamente ou é também recursiva? Por quê?

exercício 5.12 Elabore uma máquina de Turing **MT_Palíndroma** (determinística ou não) que sempre para 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 $\in$ ACEITA(MT_Palíndroma)
 abab $\in$ REJEITA(MT_Palíndroma)

exercício 5.13 Elabore uma máquina de Turing **M** sobre o alfabeto $\{a, b\}$ tal que:

ACEITA(M) = $\{w \mid w \text{ inicia com } a \text{ e, após cada } a, \text{ existe pelo menos um } b\}$
 LOOP(M) = $\{w \mid w \notin \text{ACEITA(M)} \text{ e existe pelo menos um } b \text{ entre dois } a\}$
 REJEITA(M) = $\{a, b\}^* - (\text{ACEITA(M)} \cup \text{LOOP(M)})$

Por exemplo:

ab, abbab $\in$ ACEITA(M)
 b, baa, baab $\in$ REJEITA(M)
 aba, baba, abbaba $\in$ LOOP(M)

exercício 5.14 Verifique se as respectivas funções computáveis são totais:

- a** Função **concatenação** do exemplo 5.4 – Máquina de Turing: concatenação;
b Função **quadrado** do exemplo 5.5 – Máquina de Turing: função quadrado.

exercício 5.15 Sobre a máquina de Turing, analise as seguintes afirmações:

- I. Uma linguagem aceita por uma máquina de Turing pode ser dita linguagem recursiva;
- II. A classe das linguagens enumeráveis recursivamente está contida propriamente na classe das linguagens recursivas;
- III. O complemento de uma linguagem recursiva é uma linguagem recursiva.

Marque a alternativa correta:

- ☐ **a** Apenas I e II estão corretas;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas II e III estão corretas;
- ☐ **d** Apenas I e III estão corretas;
- ☐ **e** I, II e III estão corretas.

exercício 5.16 Considere a seguinte máquina de Turing:

$$M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$$

Marque a alternativa *errada*:

- ☐ **a** $ACEITA(M)$, é o conjunto de todas as palavras pertencentes a Σ^* aceitas por M ;
- ☐ **b** $REJEITA(M)$, é o conjunto de todas as palavras pertencentes a Σ^* rejeitadas por M ;
- ☐ **c** $ACEITA(M) \cap REJEITA(M) = \emptyset$
- ☐ **d** $ACEITA(M) \cup REJEITA(M) = \Sigma^*$
- ☐ **e** O complemento de $ACEITA(M)$ é $REJEITA(M) \cup LOOP(M)$

exercício 5.17 Considere a seguinte máquina de Turing cuja função programa Π é como na figura 5.13:

$$M = (\{a, b\}, \{q_0, q_1, q_2, q_3, q_4, q_f\}, \Pi, q_0, \{q_f\}, \emptyset, \beta, \odot)$$

Relacione a primeira coluna de acordo com a segunda, considerando o reconhecimento das palavras por M :

- | | |
|----------------------|-------------|
| (1) $\in ACEITA(M)$ | () aababa |
| (2) $\in REJEITA(M)$ | () abba |
| (3) $\in LOOP(M)$ | () bbab |
| | () aabbba |
| | () aaaabba |

Marque a alternativa que corresponde a correta correlação:

- ☐ **a** 1 – 1 – 3 – 2 – 1
- ☐ **b** 3 – 1 – 2 – 2 – 1
- ☐ **c** 2 – 3 – 1 – 1 – 2
- ☐ **d** 3 – 1 – 2 – 1 – 2
- ☐ **e** 1 – 2 – 1 – 3 – 3

Π	$\odot$	a	b	β
q_0	$(q_0, \odot, D)$	(q_0, a, D)	(q_1, b, D)	(q_4, β, E)
q_1		(q_0, a, E)	(q_2, b, D)	
q_2		(q_3, b, D)		
q_3				(q_f, β, E)
q_4		(q_2, a, D)	(q_3, a, E)	(q_4, β, E)
q_f				

figura 5.13 Tabela de transições da máquina de Turing.

exercício 5.18 Considere a seguinte máquina de Turing cuja função programa Π é como na figura 5.14:

$(\{a, b, \#\}, \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_f\}, \Pi, q_0, \{q_8\}, \{A, B\}, \beta, \odot)$

Marque a alternativa correta:

- a** $ACEITA(M) = \{w \# w^r \mid w \in \{a, b, \#\}^* \text{ e } w^r \text{ é a reversa de } w\}$
- b** $ACEITA(M) = \{w \# w \mid w \in \{a, b, \#\}^+\}$
- c** $ACEITA(M) = \{w \# w \mid w \in \{a, b, \#\}^*\}$
- d** $ACEITA(M) = \emptyset$
- e** Nenhuma das alternativas acima esta correta.

Π	$\odot$	a	b	#	A	B	β
q_0	$(q_0, \odot, D)$	(q_1, A, D)	(q_5, B, D)	$(q_7, \#, D)$			(q_f, β, E)
q_1		(q_1, a, D)	(q_1, b, D)	$(q_2, \#, D)$			
q_2		(q_3, A, E)			(q_2, A, D)	(q_2, B, D)	
q_3	$(q_8, \odot, D)$			$(q_4, \#, E)$	(q_3, A, E)	(q_3, B, E)	
q_4		(q_4, a, E)	(q_4, b, E)		(q_0, A, D)	(q_0, B, D)	
q_5		(q_5, a, D)	(q_5, b, D)	$(q_6, \#, D)$			
q_6			(q_3, B, E)		(q_6, A, D)	(q_6, B, D)	
q_7					(q_7, A, D)	(q_7, B, D)	
q_8							(q_f, β, E)
q_f							

figura 5.14 Tabela de transições da máquina de Turing.

exercício 5.19 Sobre a máquina de Turing, analise as seguintes afirmações:

- I. O termo função Turing-computável reflete a classe das linguagens recursivas;
- II. A partir da definição de função Turing-computável, um processamento que para em um estado não final não tem resultado definido;

- III. O conjunto formado pela união de todas as funções Turing-computável totais possui a propriedade de que o conjunto das palavras que deixam a máquina em *loop* é vazio.

Marque a alternativa correta:

- ☐ a Apenas I está correta;
- ☐ b Apenas II está correta;
- ☐ c Apenas III está correta;
- ☐ d Apenas I e II estão corretas;
- ☐ e Apenas II e III estão corretas.

exercício 5.20 Sobre a máquina de Turing, marque a alternativa errada:

- ☐ a Uma função parcial $f: (\Sigma^*)^n \rightarrow \Sigma^*$ é dita função Turing-computável, ou simplesmente função computável, se existe uma máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ que computa f ;
- ☐ b 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 $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$ que computa f e sempre para para qualquer entrada;
- ☐ c A função programa total sempre induz uma função Turing-computável total;
- ☐ d A definição de uma função Turing-computável total garante que a função está definida para todos os valores de entrada;
- ☐ e Toda máquina de Turing vista como um reconhecedor de linguagem também pode ser vista como um processador de função.

exercício 5.21 Como, sem perda de generalidade, pode-se supor que a função programa de uma máquina de Turing seja total?

exercício 5.22 Desenvolva um simulador universal de máquina de Turing. Recebe como entrada uma representação de uma máquina de Turing no formato de uma 8-upla ordenada e a palavra de entrada a ser processada. Para um processamento finito, gera como saída: o estado atingido (incluindo a condição aceita/rejeita); o conteúdo da fita; e o número de movimentos realizados pela cabeça de fita. Sugere-se a implementação da opção execução passo a passo da máquina simulada, para permitir uma análise da computação.

Termos-chaves

classe das linguagens enumeráveis
recursivamente, p. 141

classe das linguagens recursivas, p. 141

Função Turing-computável, p. 143

Função Turing-computável total, p. 148

hipótese de Church, p. 141

linguagem enumerável recursivamente,
p. 140, p. 141

linguagem recursiva, p. 141

máquina de Turing, p. 132, p. 134

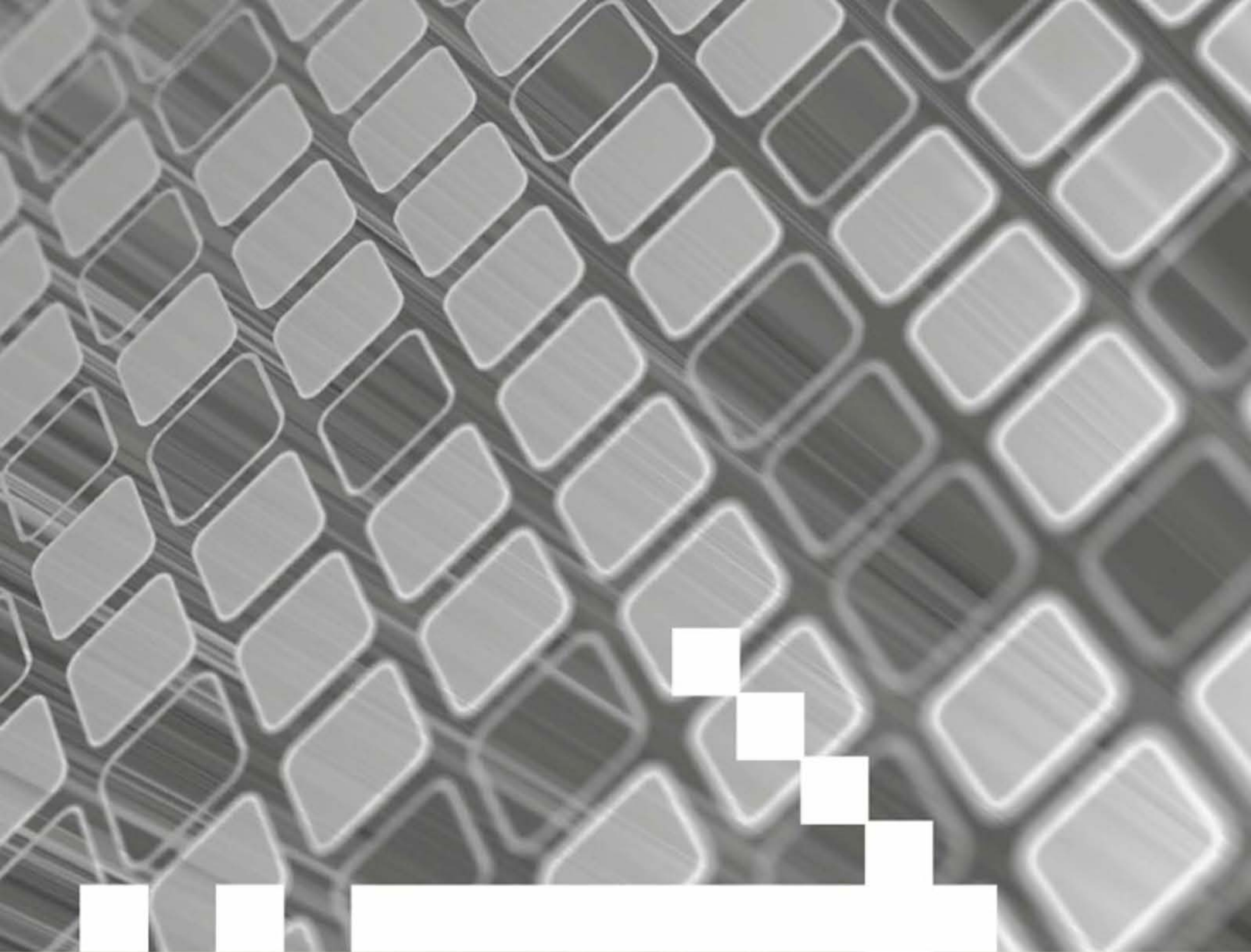
máquina universal, p. 137

processamento de funções, p. 132

propriedades das linguagens, p. 142

reconhecimento de linguagens, p. 132

solucionabilidade de problemas, p. 132

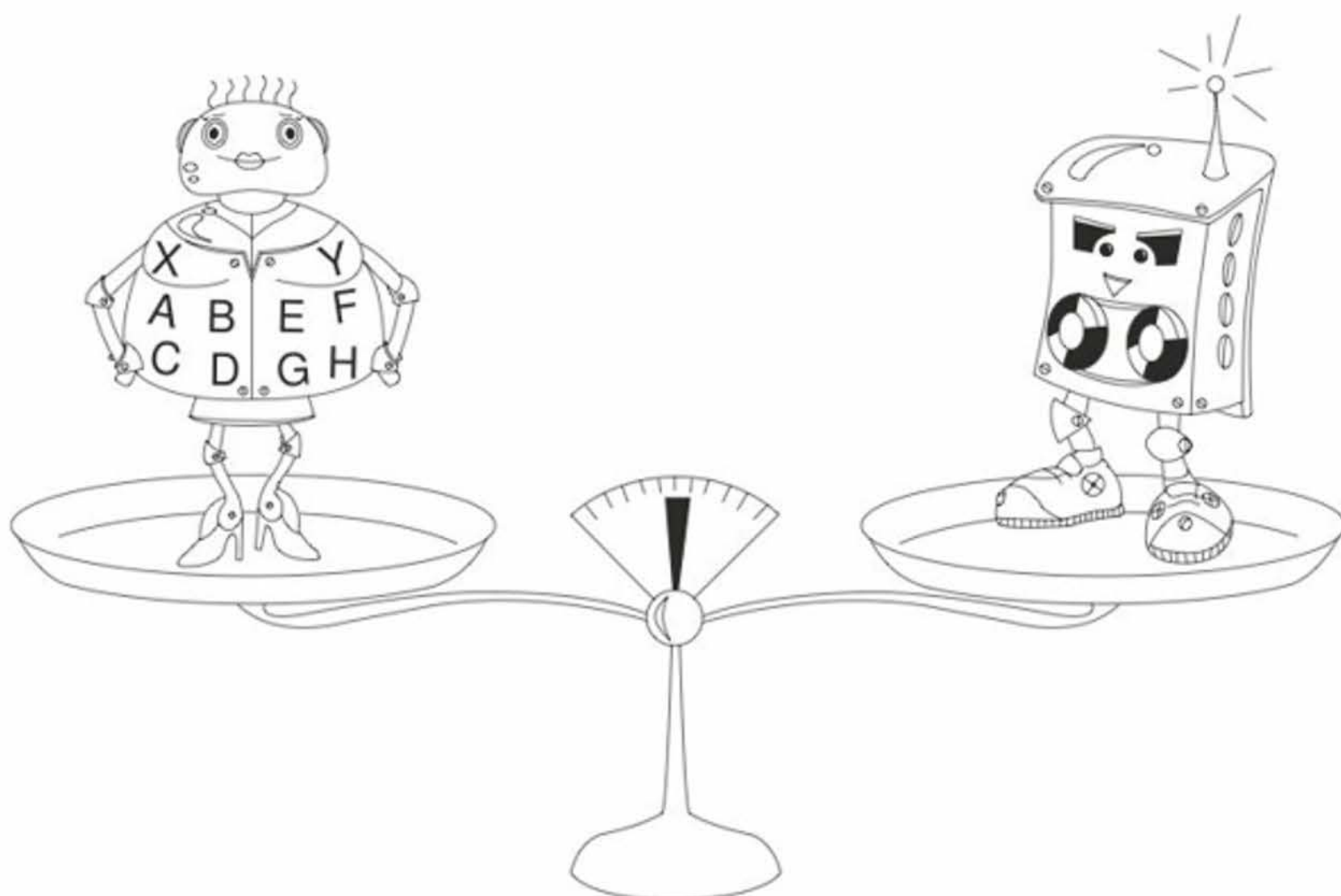


capítulo

6

máquinas universais e hipótese de Church

■ ■ A tese de Church supõem que o limite máximo do que é computável pode ser expresso como um algoritmo em uma máquina de Turing. Como a noção de algoritmo é intuitiva, a tese não é demonstrável, sendo aceita (suposta) como verdadeira. Este capítulo apresenta algumas evidências internas e externas que corroboram a tese de Church como a verificação de que as máquinas de Turing e Norma são equivalentes. E também a constatação de que as modificações apresentadas sobre a máquina de Turing não ampliam a classe das funções computadas.



Máquinas universais podem ser entendidas de duas formas:

- se é capaz de simular todas as outras máquinas;
- se toda função computável pode ser expressa como um programa nesta máquina.

Ambos os entendimentos não podem ser provados, pois:

- não há como definir o conjunto de todas as máquinas possíveis e, portanto, uma prova exaustiva de simulação não seria possível;
- o conceito de função computável não é matematicamente preciso.

Como é impossível formalizar a demonstração de que uma máquina é universal, buscam-se evidências para tal comprovação analogamente ao desenvolvido nos capítulos referentes às máquinas Norma e Turing. Nesse caso, foram buscadas evidências internas, pois, nos referidos capítulos, foram exploradas variações nas definições das máquinas que não aumentaram o poder computacional das mesmas, ou seja, o conjunto das funções computadas por estas máquinas permaneceu o mesmo.

Neste capítulo, é mostrado que o conjunto das funções computadas por Norma e Turing é o mesmo, por meio da demonstração de equivalência entre essas máquinas, o que constitui uma evidência externa.

Evidências como estas levaram Alonzo Church a apresentar, em 1936, a tese 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.

Apesar de a noção intuitiva de algoritmo impossibilitar a demonstração de que a máquina de Turing é, efetivamente, o mais genérico dispositivo de computação, todas as evidências internas e externas imaginadas foram sempre verificadas. Isso reforça a tese de Church, ou seja, os demais modelos de máquinas propostos, bem como qualquer extensão de suas capacidades, possuem, no máximo, a mesma capacidade computacional da máquina de Turing. Assim, a tese de Church foi assumida como uma hipótese para toda a *teoria da computação clássica*, razão pela qual também é conhecida como *hipótese de Church*.

Neste capítulo, também se verifica que algumas extensões da capacidade 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.

6.1

→ equivalência entre as máquinas de Turing e Norma

A seguir, é apresentada a prova de que a máquina de Turing é equivalente à máquina Norma (a prova completa é de R. Bird). Além de reforçar as evidências de que ambas são máquinas universais, também fica caracterizado que, de fato, a Norma pode definir e manipular cadeias de caracteres (conforme introduzido quando do estudo desta máquina).

Em resumo, a prova é como segue:

- a** $Turing \leq Norma$. A estrutura de fita da máquina de Turing é simulada na Norma usando uma estrutura de arranjo unidimensional;
- b** $Norma \leq Turing$. Conforme já introduzido, 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 pares da fita, e Y, as ímpares maiores que 1.

Lembre-se de 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 6.1 – Máquina de Turing $\leq$ máquina Norma

O formalismo máquina de Turing pode ser simulado pelo formalismo máquina Norma.

→ Prova:

Como já 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 $M = (\Sigma, Q, \Pi, q_0, F, V, B, \odot)$. 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 $\Sigma \cup V = \{a_1, a_2, \dots, a_n\}$, o símbolo a_i é codificado como o natural i , e os símbolos especiais B e $\odot$ como **zero** e $n+1$, respectivamente;

Estados. Para os estados de $Q = \{q_0, q_1, \dots, q_n\}$ em M , o programa P em norma possui correspondentes instruções rotuladas por 2^{u+1} , para $u \in \{0, 1, \dots, n\}$. Portanto, o rótulo inicial de P é $2 = 2^{0+1}$ (pois q_0 é o estado inicial de M), e, para qualquer $q_f \in F$, 2^{f+1} é o rótulo antecessor ao rótulo final de P , preparando os dados para a função de saída em norma;

Estado corrente. O estado corrente de M é simulado em norma, usando o registrador Q , o qual assume valores em $\{2^{u+1} \mid u \in \{0, 1, \dots, n\}\}$ (correspondendo aos estados $q_0, q_1, \dots, q_n$);

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 2^{u+1} é, na realidade, uma chamada de macro, a qual implementa a transição de M , contendo instruções rotuladas para o novo estado, gravação na fita e movimento da cabeça:

2^{u+1} :	faça $A := 3^Q \cdot 5^{X(C)}$ vá_para End_A	
...		
a :	faça $X(C) := s$ vá_para a_1	grava na fita
a_1 :	faça ad_C vá_para a_2	move a cabeça para a direita
a_2 :	faça $Q := 2^{v+1}$ vá_para End_Q	novo estado

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 2^{u+1} , é especificado um desvio incondicional para uma instrução rotulada pelo par $(2^{u+1}, r)$, usando uma codificação de primos, similar à introduzida no estudo da máquina Norma (lembre que o conteúdo de Q e $X(C)$ são 2^{u+1} e r , respectivamente). Por simplicidade, os detalhes de como implementar o cálculo da expressão $3^Q \cdot 5^{X(C)}$ 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 estudo da máquina Norma. O conteúdo do registrador **A** é denotado por a , rótulo inicial da macro que implementa a transição. O conteúdo do registrador **Q** é denotado por 2^{v+1} , rótulo que denota o novo estado q_v .

Adicionalmente, a cada estado final q_f se associa um rótulo 2^{f+1} , antecessor do rótulo final, que corresponde ao trecho de programa em P a seguir. Neste rótulo, o conteúdo de X ("fita") é atribuído a Y (pois, em norma, a função de saída retorna o valor do registrador Y), assim, os dados para a função de saída de Norma são preparados e o processamento é desviado para o rótulo final.

2^{f+1} : faça $Y := X$ vá_para fim

onde *fim* é um rótulo final.

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.

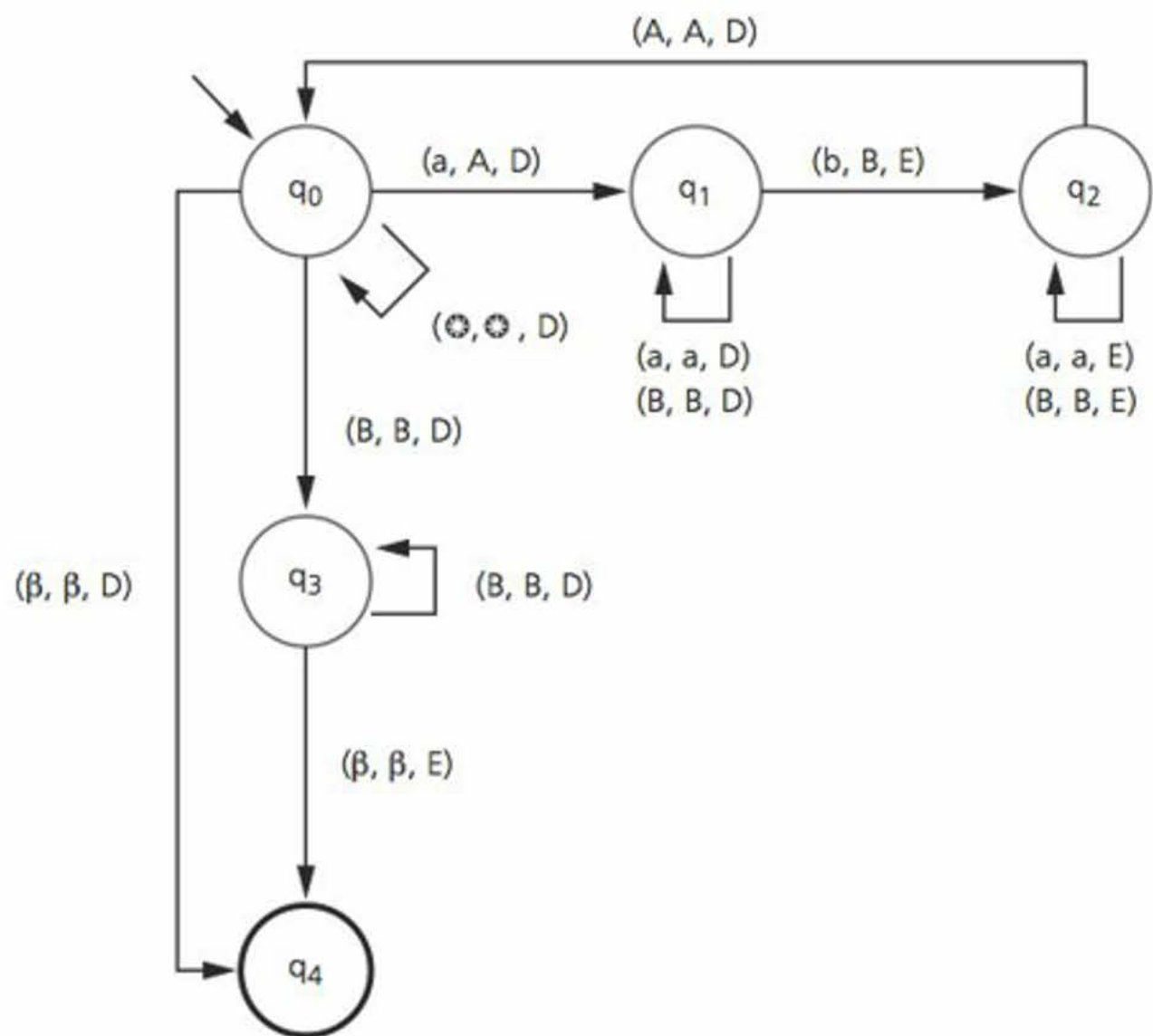
Sugere-se, como exercício, a simulação, por Norma, das condições de **ACEITA** e de **REJEITA** de parada da máquina de Turing, lembrando que a função programa é suposta total. Ao transformar a função programa de parcial em total, as transições originalmente indefinidas ($\Pi(q_u, a_r)$ indefinida) são definidas usando um estado terminação especialmente introduzido. A correspondente simulação de Norma faz com que, a partir desse estado de terminação, desvie para um rótulo sem instrução associada, encerrando a execução. $\square$

exemplo 6.1 – Máquina de Turing $\leq$ máquina Norma: duplo balanceamento

Considere a máquina de Turing:

$$MT_Duplo_Bal = (\{ a, b \}, \{ q_0, q_1, q_2, q_3, q_4 \}, \Pi, q_0, \{ q_4 \}, \{ A, B \}, \beta, \odot)$$

introduzida anteriormente e ilustrada na figura 6.1.



Π	$\odot$	a	b	A	B	β
q_0	$(q_0, \odot, D)$	(q_1, A, D)			(q_3, B, D)	(q_4, β, 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, β, E)
q_4						

figura 6.1 Grafo e tabela de transições da máquina de Turing – duplo balanceamento.

Para cada estado $q_u \in \{q_0, q_1, q_2, q_3, q_4\}$, tem-se uma instrução rotulada, onde o rótulo é calculado por 2^{u+1} . Os rótulos calculados são: 2, 4, 8, 16 e 32. A instruções rotuladas correspondentes aos estados, são:

2:	faça $A = 3^Q \cdot 5^{X(C)}$ vá_para End_A	estado q_0
4:	faça $A = 3^Q \cdot 5^{X(C)}$ vá_para End_A	estado q_1
8:	faça $A = 3^Q \cdot 5^{X(C)}$ vá_para End_A	estado q_2
16:	faça $A = 3^Q \cdot 5^{X(C)}$ vá_para End_A	estado q_3
32:	faça $Y = X$ vá_para fim	estado q_4

A expressão $3^Q \cdot 5^{X(C)}$, que determina o valor do registrador A , serve para distinguir as diferentes transições definidas em função do estado e do símbolo lido, garantindo rótulos diferentes. Observa-se, ainda, que o rótulo fim é um rótulo final. O registrador X é o registrador de entrada, o qual contém o conteúdo da fita de Turing. Ao final da computação transfere o conteúdo do registrador X para o registrador de saída de Y . Os símbolos da fita são codificados por números naturais, ou seja, os símbolos $\odot$, a , b , A , B e β serão codificados pelos números: 5, 1, 2, 3, 4 e 0, respectivamente. Estão sendo usados ainda os registradores Q (indica estado corrente, começando por 2) e C (posição da cabeça da fita).

A expressão $3^Q \cdot 5^{X(C)}$ indica as transições da função programa da máquina de Turing. Neste caso, existem 12 transições, às quais correspondem 12 conjuntos de três instruções rotuladas cada, que vão indicar: novo estado, novo símbolo e movimento na fita.

a Para o estado inicial q_0 :

$\Pi(q_0, \odot) = (q_0, \odot, D)$ e, portanto:

$$Q = 2^{u+1} = 2^{0+1} = 2$$

$$A = 3^Q \cdot 5^{X(C)} = 3^2 \cdot 5^5 = 28125$$

28125: faça $X(C) = 5$ vá_para 28126

28126: faça $C = C+1$ vá_para 28127

28127: faça $Q = 2$ vá_para End_Q

$$(2^{u+1} = 2^{0+1} = 2)$$

$\Pi(q_0, a) = (q_1, A, D)$ e, portanto:

$$Q = 2$$

$$A = 3^Q \cdot 5^{X(C)} = 3^2 \cdot 5^1 = 45$$

45: faça $X(C) = 3$ vá_para 46

46: faça $C = C+1$ vá_para 47

47: faça $Q = 4$ vá_para End_Q

$$(2^{u+1} = 2^{1+1} = 4)$$

$\Pi(q_0, B) = (q_3, B, D)$ e, portanto:

$$Q = 2$$

$$A = 3^Q \cdot 5^{X(C)} = 3^2 \cdot 5^4 = 5625$$

5625: faça $X(C) = 4$ vá_para 5626

5626: faça $C = C+1$ vá_para 5627

5627: faça $Q = 16$ vá_para End_Q

$$2^{u+1} = 2^{3+1} = 16)$$

$\Pi(q_0, \beta) = (q_4, \beta, D)$ e, portanto:

$$Q = 2$$

$$A = 3^Q \cdot 5^{X(C)} = 3^2 \cdot 5^0 = 9$$

9: faça $X(C) = 0$ vá_para 10

10: faça $C = C+1$ vá_para 11

11: faça $Q = 32$ vá_para End_Q

$$(2^{u+1} = 2^{4+1} = 32)$$

b Para o estado q_1 :

$\Pi(q_1, a) = (q_1, a, D)$ e, portanto:

$$Q = 2^{u+1} = 2^{1+1} = 4$$

$$A = 3^Q \cdot 5^{X(C)} = 3^4 \cdot 5^1 = 405$$

405: faça $X(C) = 1$ vá_para 406

406: faça $C = C+1$ vá_para 407

407: faça $Q = 4$ vá_para End_Q

$$(2^{u+1} = 2^{1+1} = 4)$$

$\Pi(q_1, b) = (q_2, B, E)$ e, portanto:

$$Q = 4$$

$$A = 3^Q \cdot 5^{X(C)} = 3^4 \cdot 5^2 = 2025$$

2025: faça $X(C) = 4$ vá_para 2026

2026: faça $C = C-1$ vá_para 2027

2027: faça $Q = 8$ vá_para End_Q

$$(2^{u+1} = 2^{2+1} = 8)$$

$\Pi(q_1, B) = (q_1, B, D)$ e, portanto:

$$Q = 4$$

$$A = 3^Q \cdot 5^{X(C)} = 3^4 \cdot 5^4 = 50625$$

50625: faça $X(C) = 4$ vá_para 50626

50626: faça $C = C+1$ vá_para 50627

50627: faça $Q = 4$ vá_para End_Q

$$(2^{u+1} = 2^{1+1} = 4)$$

c Para o estado q_2 :

$\Pi(q_2, a) = (q_2, a, E)$ e, portanto:

$$Q = 2^{u+1} = 2^{2+1} = 8$$

$$A = 3^Q \cdot 5^{X(C)} = 3^8 \cdot 5^1 = 32805$$

32805: faça $X(C) = 1$ vá_para 32806

32806: faça $C = C-1$ vá_para 32807

32807: faça $Q = 8$ vá_para End_Q

$$(2^{u+1} = 2^{2+1} = 8)$$

$\Pi(q_2, A) = (q_3, A, D)$ e, portanto:

$$Q = 8$$

$$A = 3^Q \cdot 5^{X(C)} = 3^8 \cdot 5^3 = 820125$$

820125: faça $X(C) = 3$ vá_para 820126

820126: faça $C = C+1$ vá_para 820127

820127: faça $Q = 16$ vá_para End_Q

$$(2^{u+1} = 2^{3+1} = 16)$$

$\Pi(q_2, B) = (q_2, B, E)$ e, portanto:

$$Q = 8$$

$$A = 3^Q \cdot 5^{X(C)} = 3^8 \cdot 5^4 = 4100625$$

4100625: faça $X(C) = 4$ vá_para 4100626

4100626: faça $C = C-1$ vá_para 4100627

4100627: faça $Q = 8$ vá_para End_Q

$$(2^{u+1} = 2^{2+1} = 8)$$

d Para o estado q_3 :

$\Pi(q_3, B) = (q_3, B, D)$ e, portanto:

$$Q = 2^{u+1} = 2^{3+1} = 16$$

$$A = 3^Q \cdot 5^{X(C)} = 3^{16} \cdot 5^4 = 26904200625$$

26904200625: faça $X(C) = 4$ vá_para 26904200626

26904200626: faça $C = C+1$ vá_para 26904200627

26904200627: faça $Q = 16$ vá_para End_Q

$$(2^{u+1} = 2^{3+1} = 16)$$

$\Pi(q_3, \beta) = (q_4, \beta, E)$ e, portanto:

$$Q = 16$$

$$A = 3^Q \cdot 5^{X(C)} = 3^{16} \cdot 5^0 = 43046721$$

43046721: faça $X(C) = 0$ vá_para 43046722

43046722: faça $C = C-1$ vá_para 43046723

43046723: faça $Q = 32$ vá_para End_Q

$$(2^{u+1} = 2^{4+1} = 32)$$

Isso resulta no programa em Norma da figura 6.2, que computa um reconhecedor do duplo-balanceamento:

Programa monolítico duplo balanceamento

```

2:      faça  $A = 3^Q \cdot 5^{X(C)}$  vá_para End_A
4:      faça  $A = 3^Q \cdot 5^{X(C)}$  vá_para End_A
8:      faça  $A = 3^Q \cdot 5^{X(C)}$  vá_para End_A
9:      faça  $X(C) = 0$  vá_para 10
10:     faça  $C = C+1$  vá_para 11
11:     faça  $Q = 32$  vá_para End_Q
16:     faça  $A = 3^Q \cdot 5^{X(C)}$  vá_para End_A
32:     faça  $Y = X$  vá_para fim
45:     faça  $X(C) = 3$  vá_para 46
46:     faça  $C = C+1$  vá_para 47
47:     faça  $Q = 4$  vá_para End_Q
405:    faça  $X(C) = 1$  vá_para 406
406:    faça  $C = C+1$  vá_para 407
407:    faça  $Q = 4$  vá_para End_Q
2025:   faça  $X(C) = 4$  vá_para 2026
2026:   faça  $C = C-1$  vá_para 2027
2027:   faça  $Q = 8$  vá_para End_Q
5625:   faça  $X(C) = 4$  vá_para 5626
5626:   faça  $C = C+1$  vá_para 5627
5627:   faça  $Q = 16$  vá_para End_Q
28125:   faça  $X(C) = 5$  vá_para 28126
28126:   faça  $C = C+1$  vá_para 28127
28127:   faça  $Q = 2$  vá_para End_Q
32805:   faça  $X(C) = 1$  vá_para 32806
32806:   faça  $C = C-1$  vá_para 32807
32807:   faça  $Q = 8$  vá_para End_Q
50625:   faça  $X(C) = 4$  vá_para 50626
50626:   faça  $C = C+1$  vá_para 50627
50627:   faça  $Q = 4$  vá_para End_Q
820125:  faça  $X(C) = 3$  vá_para 820126
820126:  faça  $C = C+1$  vá_para 820127
820127:  faça  $Q = 16$  vá_para End_Q
4100625: faça  $X(C) = 4$  vá_para 4100626
4100626: faça  $C = C-1$  vá_para 4100627
4100627: faça  $Q = 8$  vá_para End_Q
43046721: faça  $X(C) = 0$  vá_para 43046722
43046722: faça  $C = C-1$  vá_para 43046723
43046723: faça  $Q = 32$  vá_para End_Q
26904200625: faça  $X(C) = 4$  vá_para 26904200626
26904200626: faça  $C = C+1$  vá_para 26904200627
26904200627: faça  $Q = 16$  vá_para End_Q

```

figura 6.2 Programa monolítico – duplo balanceamento.

Para a palavra de entrada **aabb**, o valor inicial armazenado da fita da máquina de Turing é:

⊙aabbβββ...

correspondendo a seguinte representação no registrador **X** de Norma:

5	1	1	2	2	0	0
---	---	---	---	---	---	---

Também:

- o conteúdo do registrador **Q** = 2 (estado inicial q_0 , codificado como 2^{0+1});
- o conteúdo do registrador **C** = 1 (denota a cabeça da fita posicionada no início). □

teorema 6.2 – Máquina Norma $\leq$ máquina de Turing

O formalismo máquina Norma pode ser simulado pelo formalismo máquina de Turing.

→ Prova:

Já foi introduzido que, em Norma, sub-rotinas e mecanismos de recursão como os definidos nos programas recursivos podem ser simulados por programas monolíticos, usando endereçamento indireto. Assim, é suficiente considerar o caso de programas monolíticos. Então, suponha um programa monolítico **P** para Norma, mas com somente dois registradores **X** e **Y**. Então, a simulação de **P** para Norma por uma máquina de Turing $M = (\Sigma, Q, \Pi, q_0, F, V, \beta, \odot)$, 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 as **x** primeiras 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 iniciais (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 $K = X$, ilustrado na figura 6.3) ou ímpares (caso $K = Y$, ilustrado na figura 6.4), 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 ;

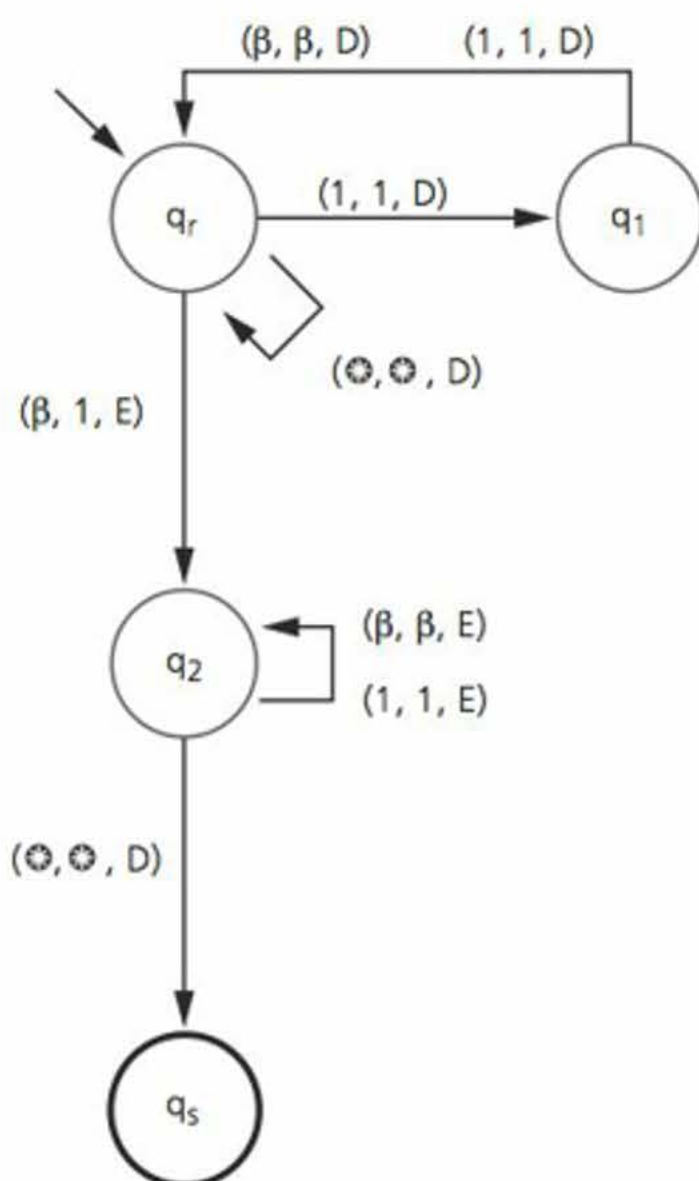


figura 6.3 Grafo da máquina de Turing: ad_X .

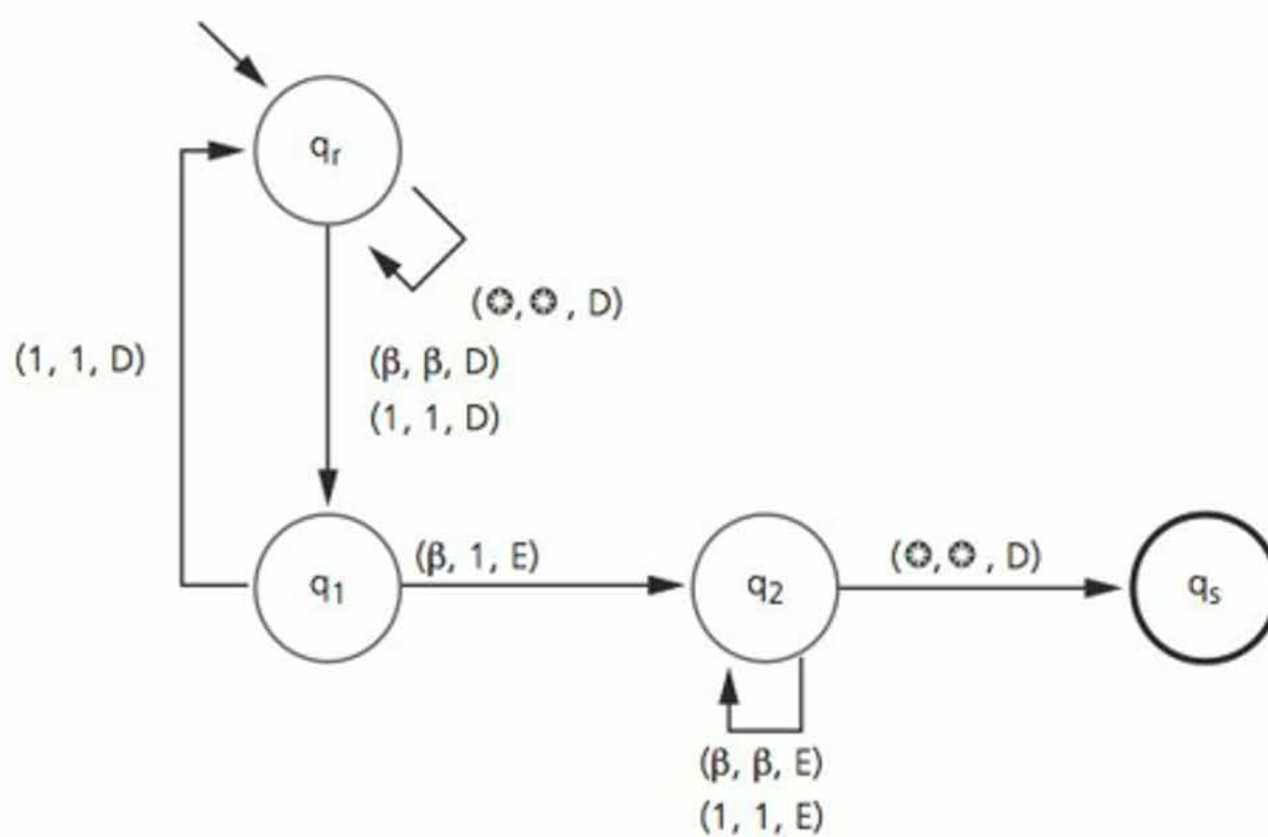


figura 6.4 Grafo da máquina de Turing: ad_Y .

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 $K = X$, ilustrado na figura 6.5) ou ímpares (caso $K = Y$, ilustrado na figura 6.6), 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 ;

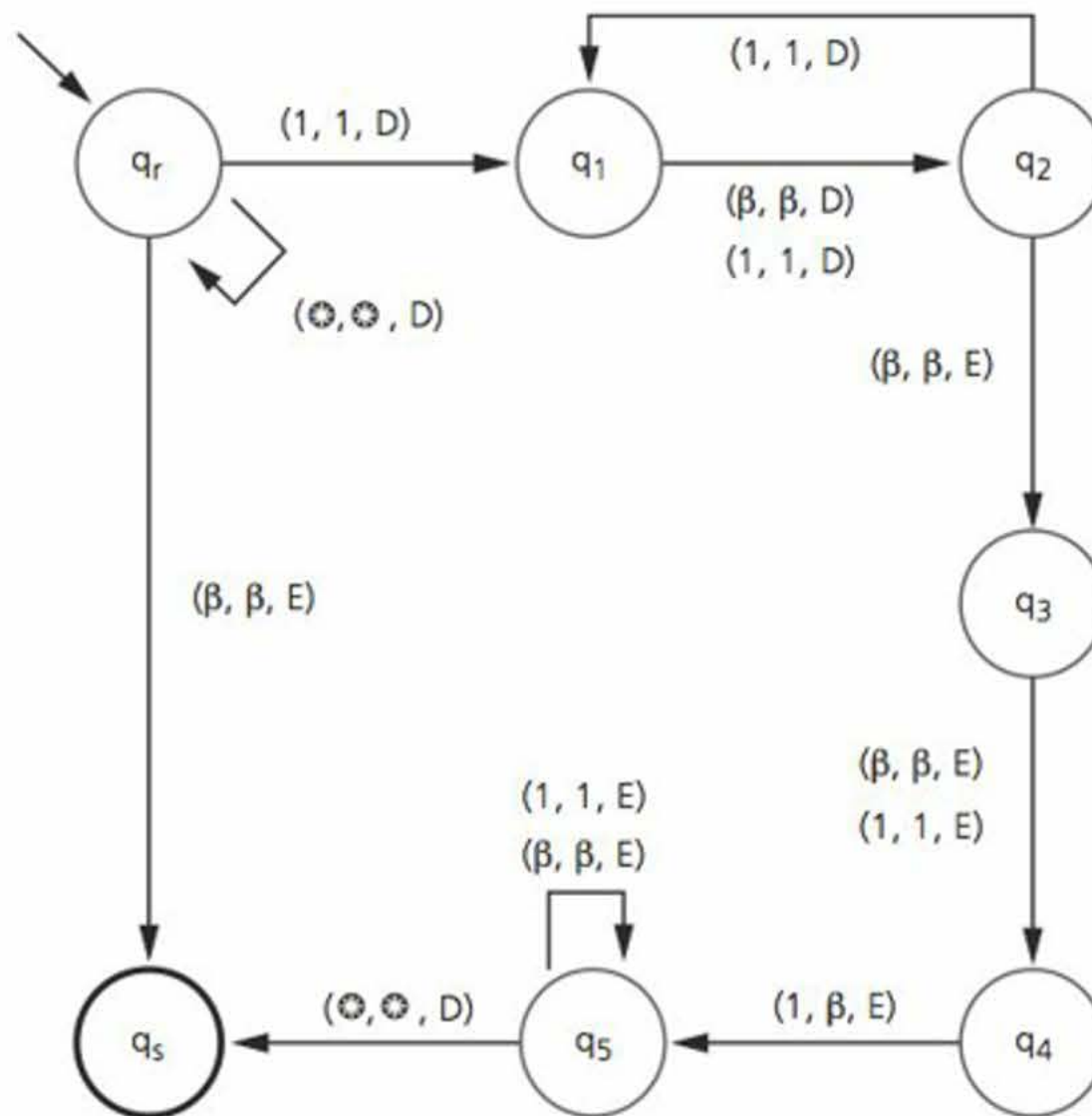


figura 6.5 Grafo da máquina de Turing: sub_X .

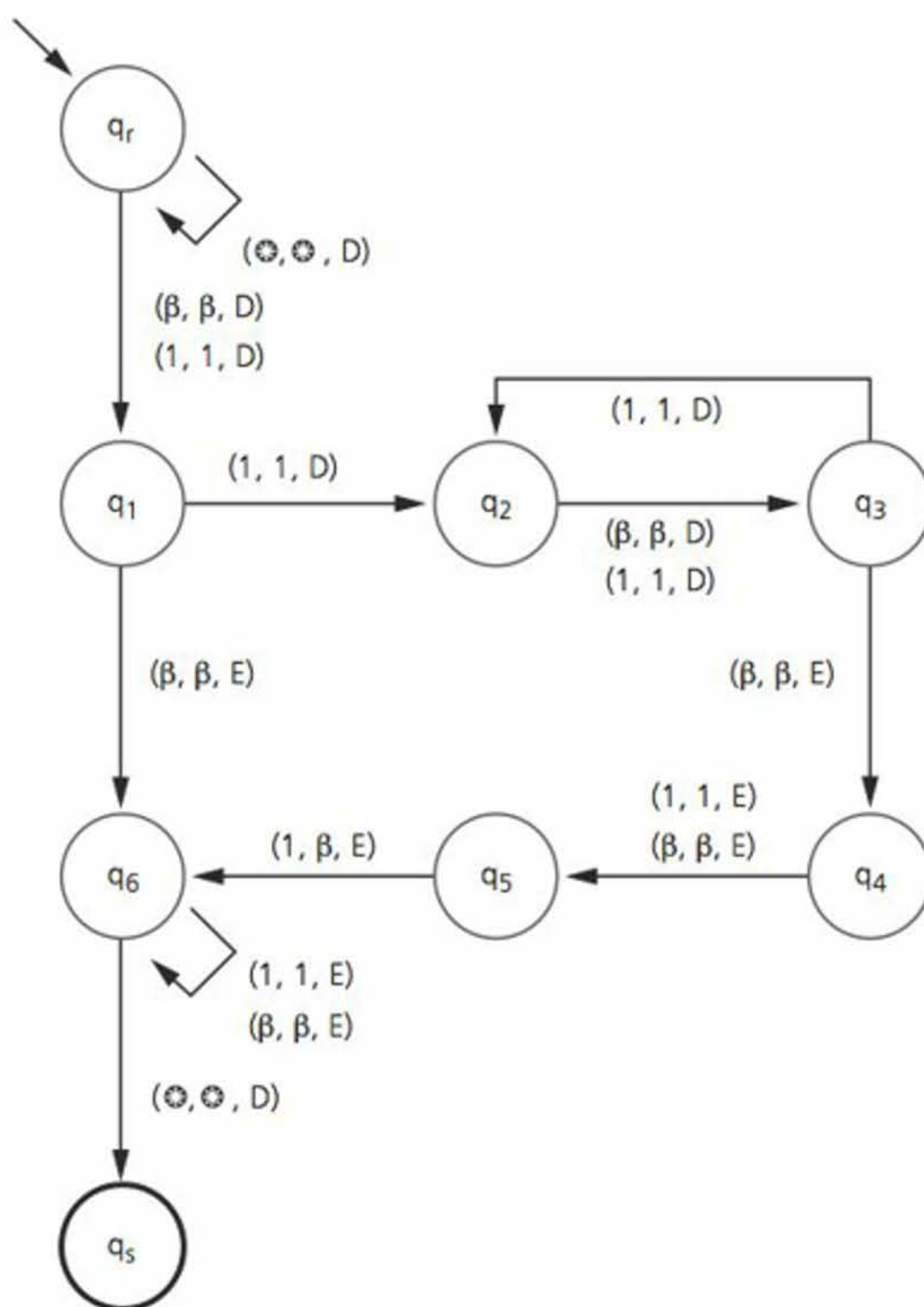


figura 6.6 Grafo da máquina de Turing: Sub_Y .

c Teste. Uma instrução rotulada de P da seguinte forma:

r : se zero_K vá_para v senão vá_para f

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 $K = X$, ilustrado na figura 6.7) ou segunda célula ímpar (caso $K = Y$ - lembre que a primeira célula ímpar é reservada para o marcador de início de fita $\odot$ - ilustrado na figura 6.8);

- c.2) Caso a célula pesquisada contenha o símbolo branco, reposiciona a cabeça no início da fita e assume o estado q_v ; caso contrário, reposiciona a cabeça no início da fita e assume o estado q_f .

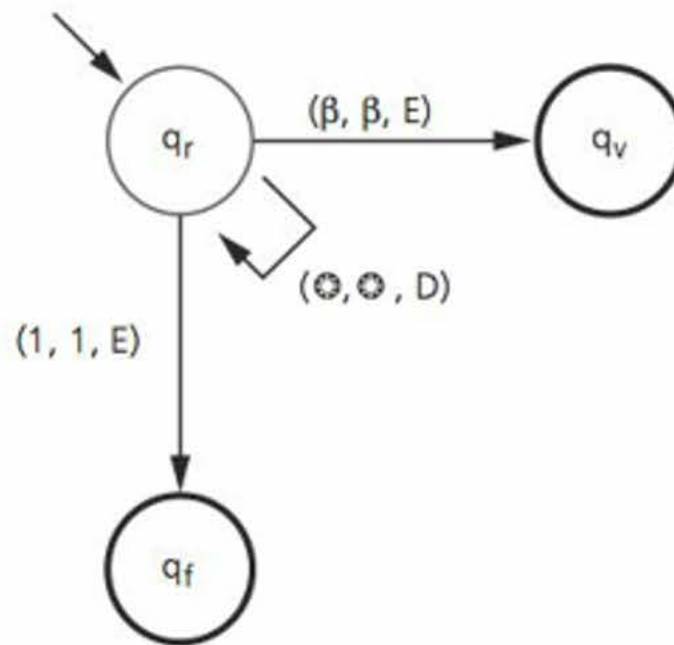


figura 6.7 Grafo da máquina de Turing: zero_X .

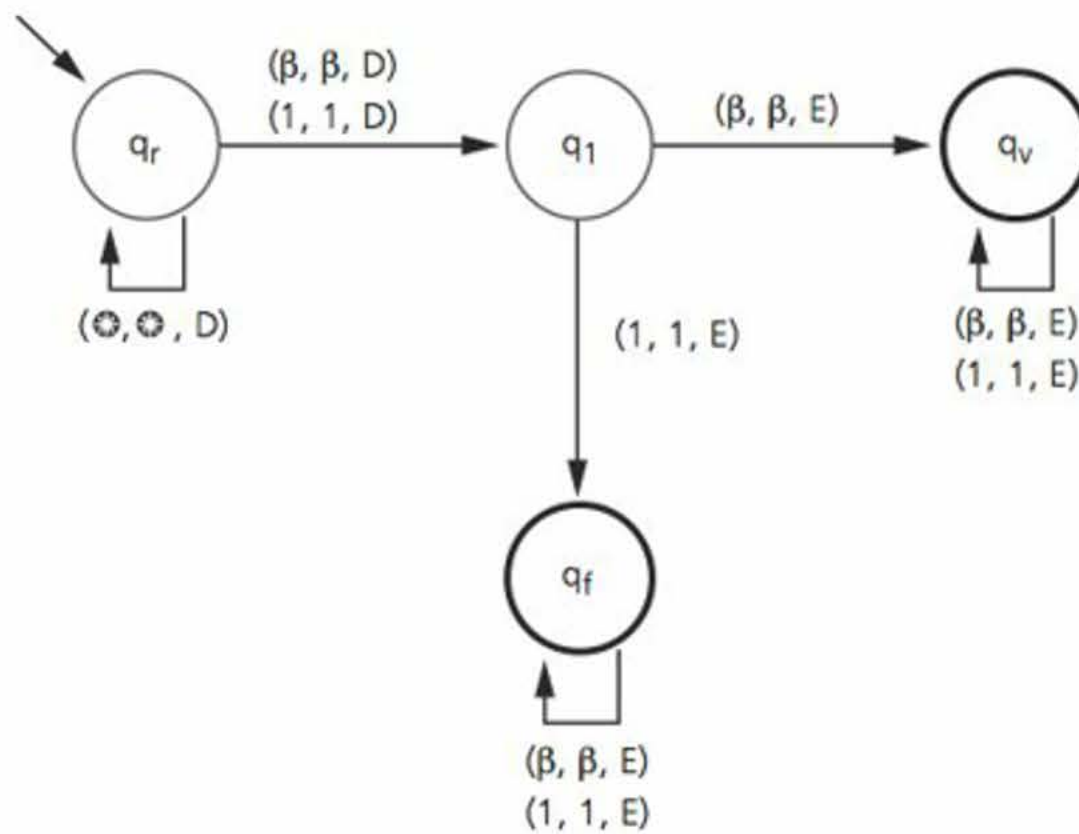


figura 6.8 Grafo da máquina de Turing: zero_Y .

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. $\square$

6.2

→ modificações sobre a máquina de Turing

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:

- Máquina de Turing não determinística;
- 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.

6.2.1 máquina de Turing não determinística

A *máquina de Turing não determinística* é tal que, para o mesmo valor de entrada, pode existir mais de um valor de saída. Assim, 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 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, nem no símbolo lido dos demais caminhos alternativos.

Para uma máquina de Turing M não determinística e para uma palavra w :

- w pertence a **ACEITA**(M), se existe pelo menos um caminho alternativo que aceita a palavra;
- w pertence a **REJEITA**(M), se todas as alternativas rejeitam a entrada;
- w pertence a **LOOP**(M), se nenhum caminho aceita a palavra e pelo menos um fica em *loop*.

6.2.2 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 6.9. Assim, as células pares representam a parte direita da fita, e as ímpares, a parte esquerda. O símbolo $\odot$ é usado para controlar a fronteira entre as partes esquerda e direita.

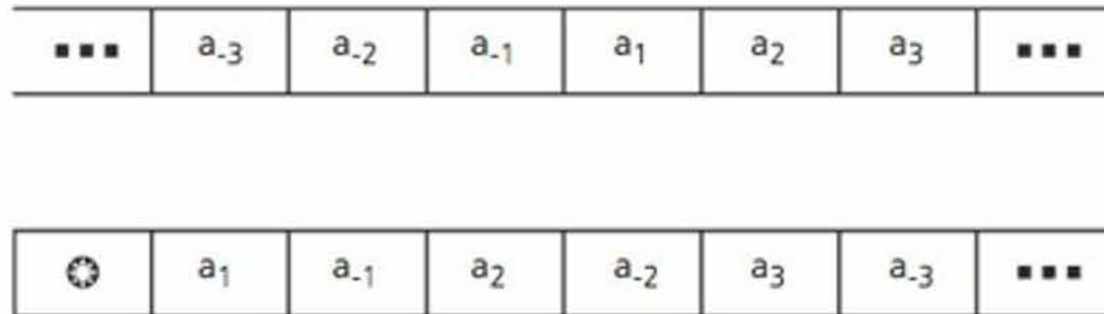


figura 6.9 Simulação de uma fita infinita dos dois lados.

6.2.3 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 6.10 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 três componentes para representar as células de cada uma das três fitas, e as demais três componentes reservadas para marcar a posição corrente das cabeças de cada fita (representadas na figura 6.10 pelo símbolo $\blacktriangle$).

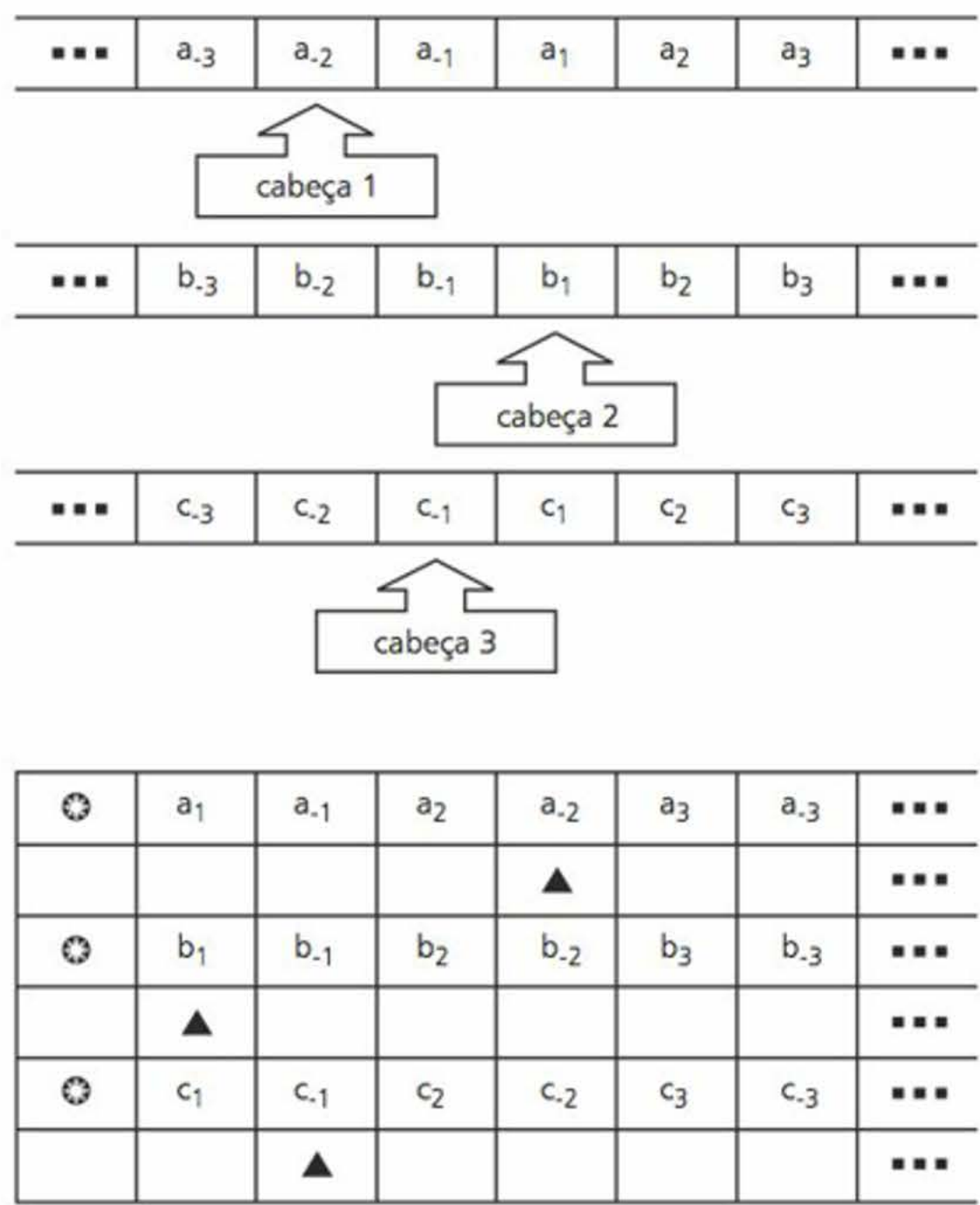


figura 6.10 Simulação de três fitas infinitas dos dois lados.

6.2.4 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 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.

6.3**→ 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* (introduzidos em capítulo específico), bem como trabalhos mais recentes, como máquina Norma e autômato com pilhas, resultaram em conceitos equivalentes ao de Turing. 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 *tese de Church* ou *tese 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 tese 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 tese de Church não é demonstrável. Entretanto, como é assumida como verdadeira (ou seja, é assumida como hipótese) para toda a ciência da computação, também é conhecida como *hipótese de Church* ou *hipótese de Turing-Church*.

6.4**→ conclusões**

Neste capítulo, foi apresentada a tese de Church, a qual supõe que o limite máximo do que é computável pode ser expresso como um algoritmo em uma máquina de Turing. Como a noção de algoritmo é intuitiva, a tese não é demonstrável, sendo aceita (suposta) como verda-

deira. Corroboraram para isso algumas evidências externas e internas, como a verificação de que as máquinas de Turing e Norma são equivalentes e a constatação de que as modificações sobre a máquina de Turing apresentadas nesse capítulo não ampliam a classe das funções computadas.

Nos próximos capítulos, serão vistos outros formalismos que também terão no máximo o mesmo poder computacional da máquina de Turing.

6.5

→ exercícios

exercício 6.1 Complete a prova do teorema 6.1 - Máquina de Turing $\leq$ máquina Norma de tal forma que a máquina Norma simule as condições **ACEITA** e **REJEITA** de parada da máquina de Turing.

exercício 6.2 No contexto do teorema 6.2 - Máquina Norma $\leq$ máquina de Turing, suponha que a máquina Norma tenha três registradores (X, Y e Z). Então:

- a** Como seriam representados esses registradores na fita da máquina de Turing?
- b** Dê a função programa, na forma de grafo, que implemente as seguintes operações: ad_X , sub_X , $zero_X$;
- c** Dê a função programa, na forma de grafo, que implemente as seguintes operações: ad_Y , sub_Y , $zero_Y$;
- d** Dê a função programa, na forma de grafo, que implemente as seguintes operações: ad_Z , sub_Z , $zero_Z$.

exercício 6.3 No contexto do teorema 6.2 - Máquina Norma $\leq$ máquina de Turing, suponha que a máquina Norma tenha cinco registradores. Discuta como seriam representados os registradores na fita e como seriam implementadas as operações da máquina Norma na máquina de Turing.

exercício 6.4 Considere a seguinte máquina de Turing (introduzida anteriormente), ilustrada na figura 6.11:

$$Conc = (\{ a, b, \# \}, \{ q_0, q_1, q_2, q_3, q_4 \}, \Pi, q_0, \{ q_4 \}, \emptyset, \beta, \odot)$$

No contexto do teorema 6.1 - Máquina de Turing $\leq$ máquina Norma, qual o correspondente programa (na forma de instruções rotuladas) na máquina Norma?

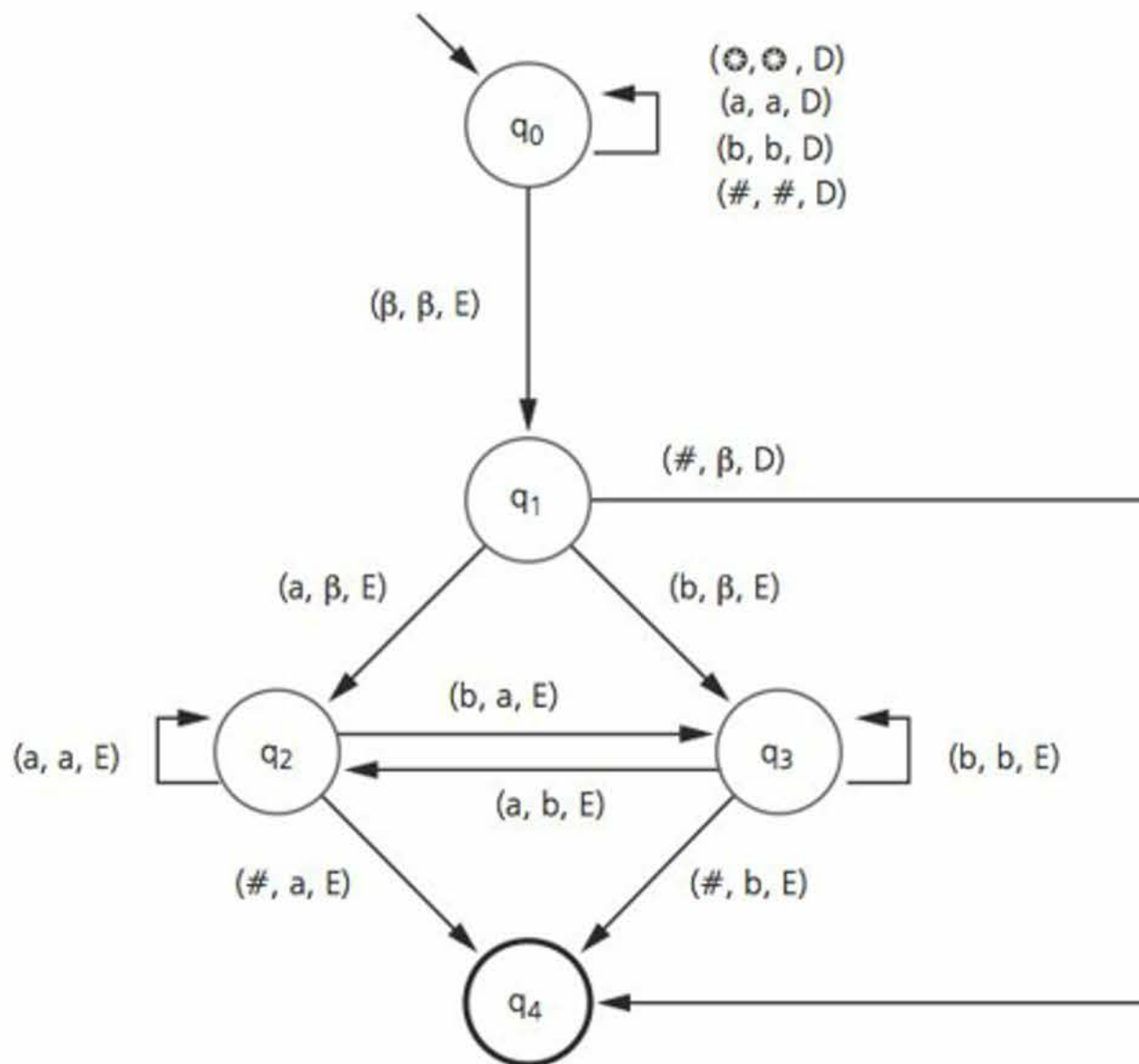


figura 6.11 Grafo da máquina de Turing - concatenação.

exercício 6.5 No contexto do teorema 6.1 - Máquina de Turing $\leq$ máquina Norma, marque a alternativa errada:

- a** A fita é simulada por um arranjo unidimensional em X , sendo que cada célula da fita corresponde a uma posição do arranjo;
- b** O símbolo de cada célula é codificado como um número natural: cada símbolo a_i do alfabeto $\Sigma = \{a_1, a_2, \dots, a_n\}$, é codificado como o natural i ; e os símbolos especiais de início de fita e branco são codificados como $n+1$ e zero, respectivamente;
- c** Para cada estado de um programa em Turing há uma correspondente instrução rotulada, sendo que ao estado inicial q_0 é associado ao rótulo 2^{0+1} de P e o estado final q_f é associado o rótulo final 2^{f+1} de P ;

- d** A cabeça da fita é simulada pelo registrador C , que contém a posição corrente do arranjo X e é inicializado com o valor 1;
- e** O estado corrente da máquina de Turing é simulado pelo registrador Q , o qual assume os valores em $\{2^1, 2^2, \dots, 2^{n+1}\}$, correspondendo aos estados $\{q_0, q_1, \dots, q_n\}$, respectivamente.

exercício 6.6 No contexto do teorema 6.2 - Máquina Norma $\leq$ máquina de Turing, marque a alternativa correta:

- a** Cada rótulo é representado por um estado na máquina de Turing;
- b** Se o registrador X contém o valor x , então seu conteúdo é armazenado nas x primeiras células pares da fita e, se o registrador Y contém o valor y , então seu conteúdo é armazenado nas y primeiras células ímpares da fita;
- c** Os alfabetos de entrada e auxiliar juntos precisam ter, no mínimo, dois símbolos para poder simular a máquina Norma;
- d** A função de transição precisa ser total;
- e** A instrução ad_k pode ser simulada procurando-se a primeira célula em branco e substituindo-a pelo símbolo 1.

exercício 6.7 No contexto do teorema 6.1 - Máquina de Turing $\leq$ máquina Norma e do teorema 6.2 - Máquina Norma $\leq$ máquina de Turing, marque a alternativa correta:

- a** Na máquina Norma não existe meio de prever todas as condições de ACEITA e de REJEITA da máquina de Turing;
- b** A máquina de Turing simula os estados da máquina Norma por meio de uma fila recursiva;
- c** Pode-se representar a fita da máquina de Turing na máquina Norma por um arranjo unidimensional no registrador X , onde cada célula da fita corresponde a uma posição do arranjo em X ;
- d** Não são necessárias funções de codificação e de decodificação para os conjuntos de entrada e de saída nos formalismos em questão.
- e** É impossível a simulação da máquina Norma por uma máquina de Turing com fita infinita para os dois lados, uma vez que parte da fita ficará vazia e se terá perda de informação.

exercício 6.8 A tese de Church afirma que (marque a alternativa correta):

- a** Qualquer programa pode ser representado na forma de fluxograma;
- b** Qualquer máquina abstrata é uma máquina universal;
- c** A codificação de conjuntos estruturados é o modo mais eficiente de representar uma máquina universal;
- d** Qualquer função computável pode ser processada por uma máquina de Turing;
- e** Todo programa iterativo pode ser representado por meio de um programa monolítico.

exercício 6.9 A tese de Church não pode ser demonstrada porque (marque a alternativa correta):

- ☐ **a** O conceito de função computável não é matematicamente preciso;
- ☐ **b** O não determinismo não aumenta o poder computacional da máquina de Turing;
- ☐ **c** Atualmente, não há poder computacional que execute o algoritmo proposto pela tese de Church;
- ☐ **d** Para qualquer sistema de computação, o tempo necessário para simular a tese de Church é maior do que o tempo de existência do sistema solar;
- ☐ **e** Uma função computável não tem equivalência na máquina de Turing.

exercício 6.10 Sobre a tese de Church, analise as seguintes afirmações:

- I. A capacidade de computação representada pela máquina de Turing é o limite máximo do que pode ser atingido por qualquer dispositivo de computação;
- II. Qualquer outra forma de expressar algoritmos terá, no máximo, a mesma capacidade computacional da máquina de Turing;
- III. Como a noção de algoritmo é intuitiva, a tese de Church não é demonstrável.

Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas I e III estão corretas;
- ☐ **d** Apenas II e III estão corretas;
- ☐ **e** I, II e III estão corretas.

Termos-chaves

classe das funções computadas, p. 157

equivalência entre máquinas, p. 159

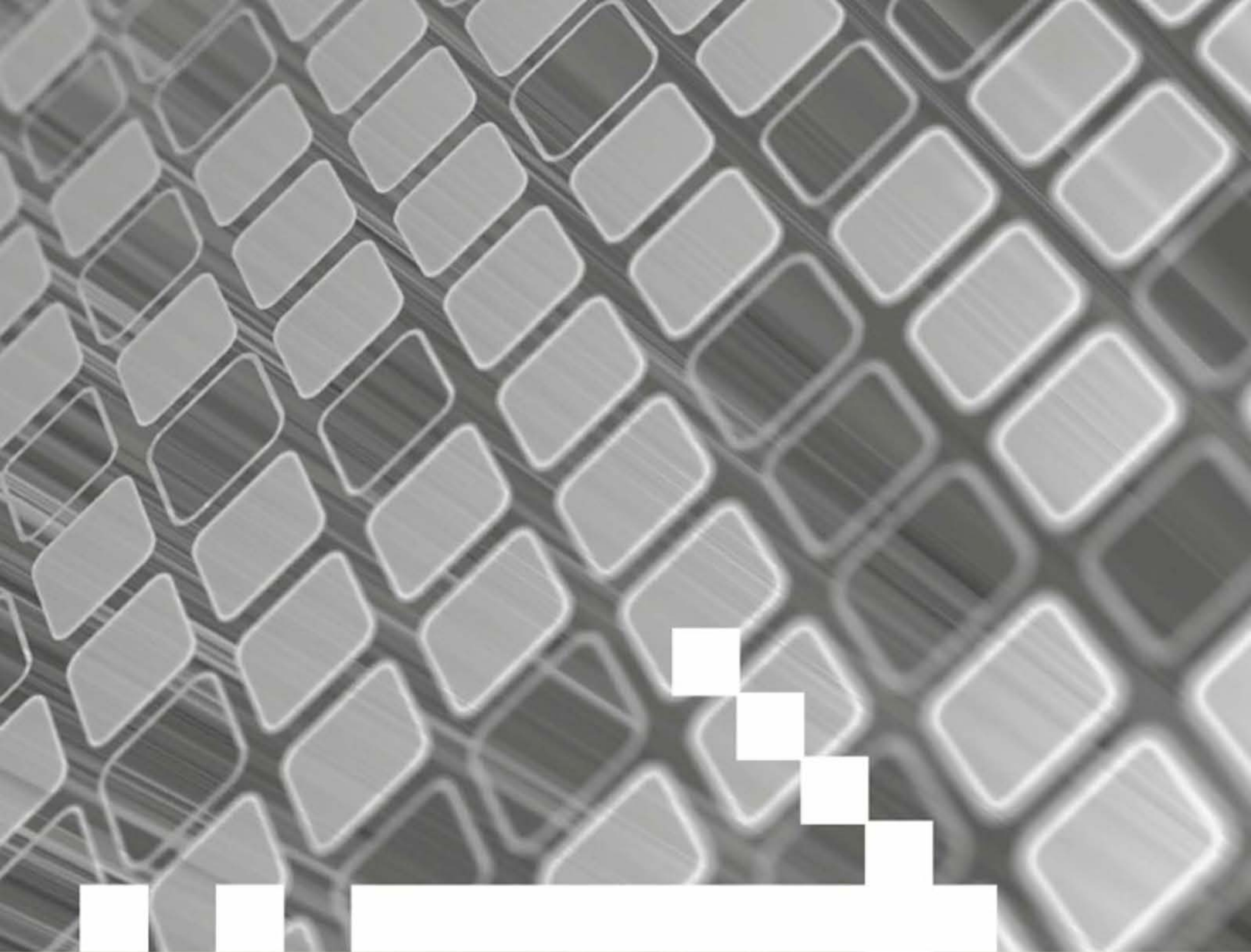
função computável, p. 158

hipótese de Church, p. 159, p. 175

máquina de Turing não determinística, p. 172

máquina universal, p. 158

tese de Church, p. 158, p. 175

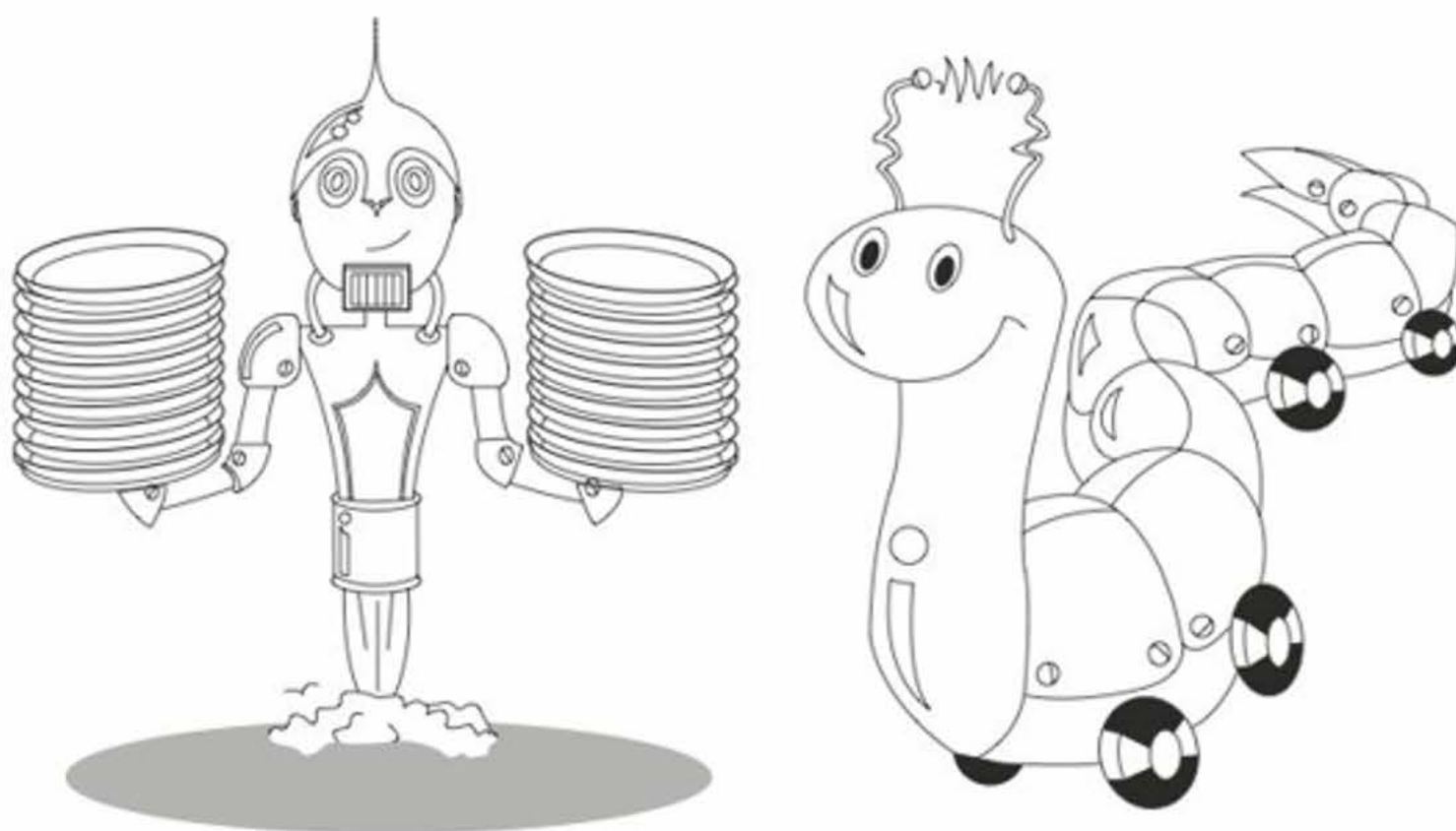




capítulo

outros modelos de máquinas universais

- ■ São apresentados outros modelos de máquinas universais equivalentes à máquina de Turing: máquina de Post (diagrama de fluxos e uma variável do tipo fila), máquina com pilhas (diagrama de fluxos e variáveis do tipo pilha) e autômato com pilhas (diagrama de estados e variáveis do tipo pilha). Também versões não determinísticas são estudadas. Uma interessante conclusão diz respeito à estrutura de dados das máquinas universais: uma fita (Turing), uma fila (Post) e duas pilhas (máquina e autômato com pilhas) são suficientes.



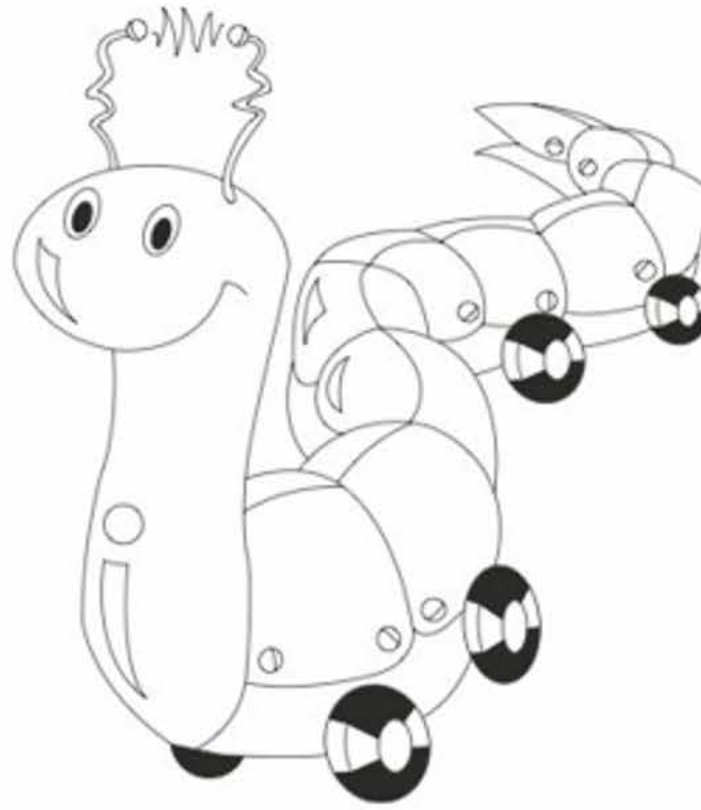
Neste capítulo, é verificado que os seguintes formalismos são equivalentes à máquina de Turing (note o tipo de estrutura de dados de cada modelo):

- a** *Máquina de Post*. Baseada na estrutura de dados do tipo *fila* (o primeiro dado armazenado é o primeiro a ser recuperado);
- b** *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. Trata-se de uma família de máquinas com poder computacional variado. Para alguns casos da família, o não determinismo aumenta o poder computacional;
- c** *Autômato com duas pilhas*. Analogamente, é baseada na estrutura de dados do tipo pilha, com exatamente duas pilhas, com programas expressos na forma de grafos, podendo ser determinístico ou não determinístico.

Um resultado importante que pode ser estabelecido a partir das equivalências destas máquinas universais, bem como da máquina Norma 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.

As equivalências estudadas neste capítulo são mais algumas evidências de que se tratam, de fato, de máquinas universais.

**7.1****→ máquina de Post**

Assim como Turing, Emil Leon Post propôs, também em 1936, um modelo de máquina universal denominado *máquina de Post*. A principal característica da máquina de Post é que ela 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 7.1. Sugere-se como exercício uma comparação das estruturas *fita* e *fila*.

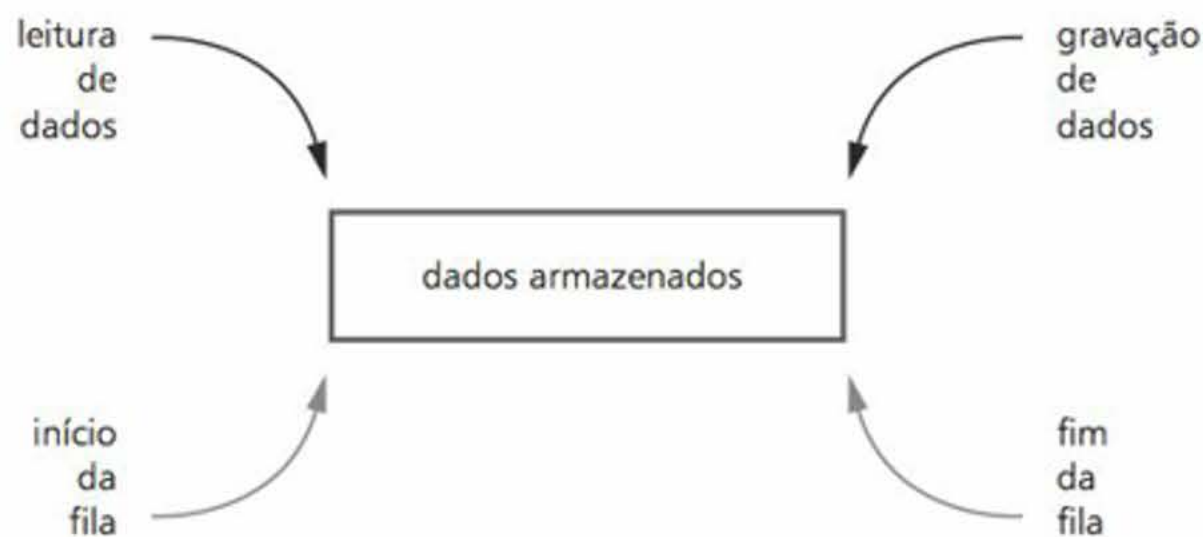


figura 7.1 Estrutura do tipo fila.

Uma máquina de Post consiste, basicamente, de duas partes:

- a** *Variável X*. Trata-se de uma variável do tipo fila que é utilizada como entrada, saída e memória de trabalho.
- b** *Programa*. É uma sequência finita de instruções, representada 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 7.1 – 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 7.2 (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 7.2 (direita);

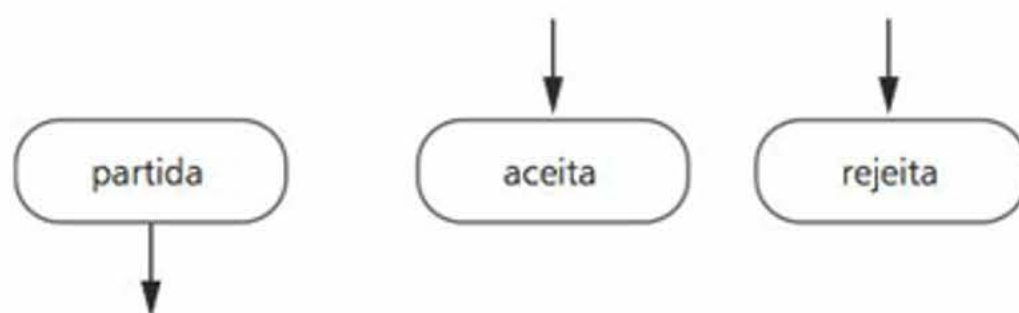


figura 7.2 Partida (esquerda) e parada (direita) em um diagrama de fluxos.

- 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 7.3, onde $X \leftarrow \text{ler}(X)$ denota uma *leitura destrutiva* ou seja, que lê o símbolo mais à esquerda da palavra, retirando da mesma o símbolo lido.

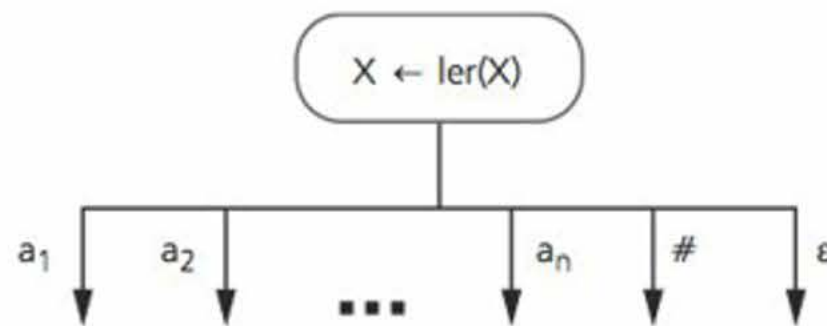


figura 7.3 Desvio em um diagrama de fluxos.

- 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 7.4, supondo que $s \in \Sigma \cup \{ \# \}$. □

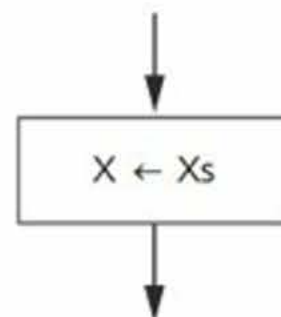


figura 7.4 Atribuição em um diagrama de fluxos.

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 aceita 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 7.1 – Máquina de Post – duplo balanceamento

Considere a seguinte linguagem introduzida anteriormente:

$$\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$$

A máquina de Post:

$$\text{Post_Duplo_Bal} = (\{ a, b \}, D, \#)$$

onde D é como ilustrado na figura 7.5, é tal que:

$$\text{ACEITA}(\text{Post_Duplo_Bal}) = \text{Duplo_Bal}$$

$$\text{REJEITA}(\text{Post_Duplo_Bal}) = \Sigma^* - \text{Duplo_Bal}$$

e, portanto, $\text{LOOP}(\text{Post_Duplo_Bal}) = \emptyset$

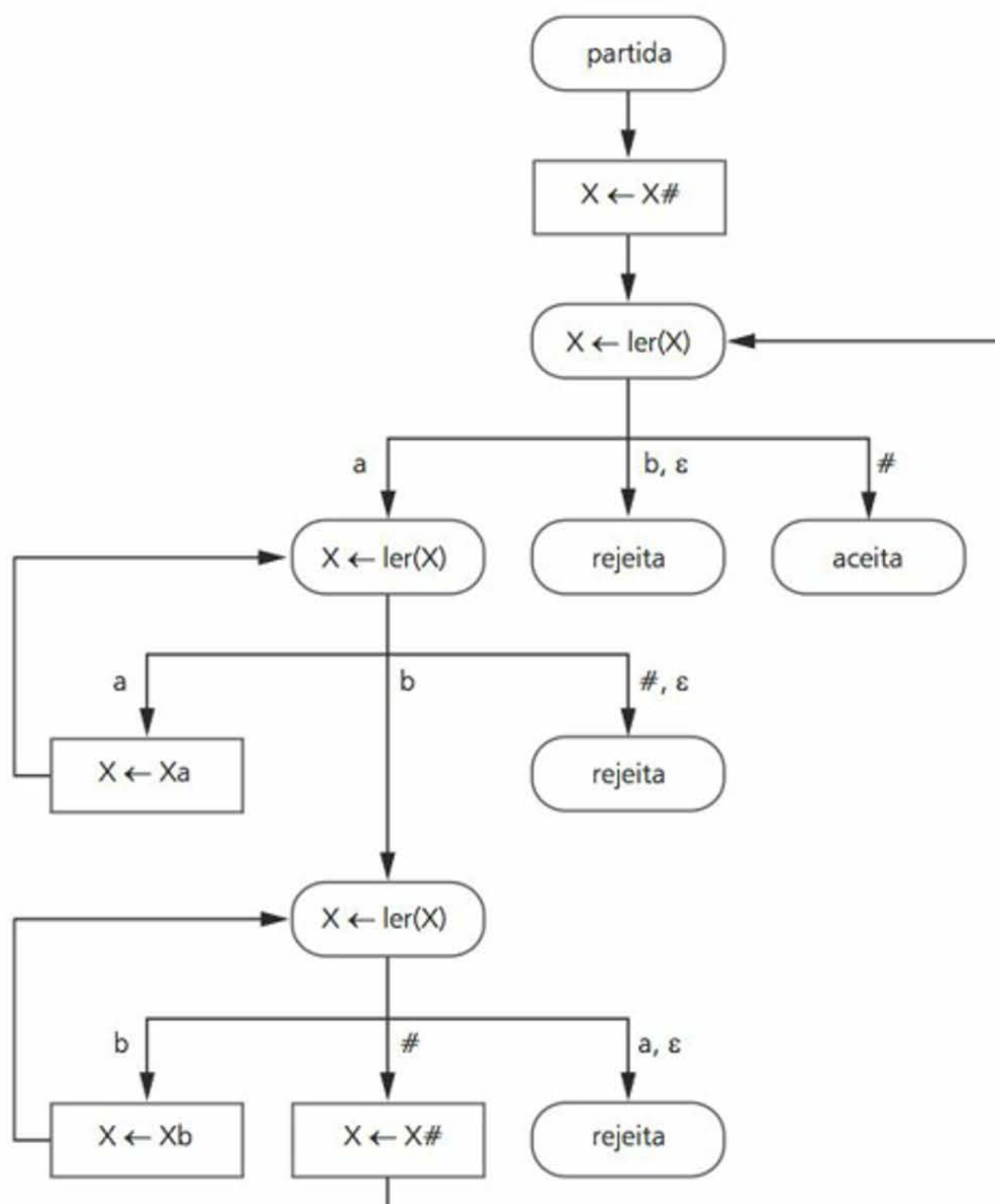


figura 7.5 Diagrama de fluxo da máquina de Post – duplo balanceamento.

O algoritmo lê e remove o primeiro símbolo a . Após, realiza uma varredura circular em busca do correspondente b . Essa varredura é realizada por meio de sucessivas leituras (e remoções), armazenando-se 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). Esse ciclo é repetido até restar a palavra vazia ou ocorrer alguma condição de rejeição. Compare este algoritmo com o algoritmo correspondente para a máquina de Turing. $\square$

exemplo 7.2 – Máquina de Post – duplo balanceamento em qualquer ordem

Considere a seguinte linguagem introduzida anteriormente:

$$\text{Duplo_Bal_2} = \{ w \mid w = a^n b^n \text{ ou } w = b^n a^n, n \geq 0 \}$$

A máquina de Post:

$$\text{Post_Duplo_Bal_2} = (\{ a, b \}, D, \#)$$

onde D é como ilustrado na figura 7.6, é tal que:

$$\text{ACEITA}(\text{Post_Duplo_Bal_2}) = \text{Duplo_Bal_2}$$

$$\text{REJEITA}(\text{Post_Duplo_Bal_2}) = \Sigma^* - \text{Duplo_Bal_2}$$

e, portanto, $\text{LOOP}(\text{Post_Duplo_Bal_2}) = \emptyset$

Este programa pode ser facilmente adaptado para reconhecer a linguagem “mesmo número de símbolos a ’s e de b ’s”, o que se sugere como exercício. $\square$

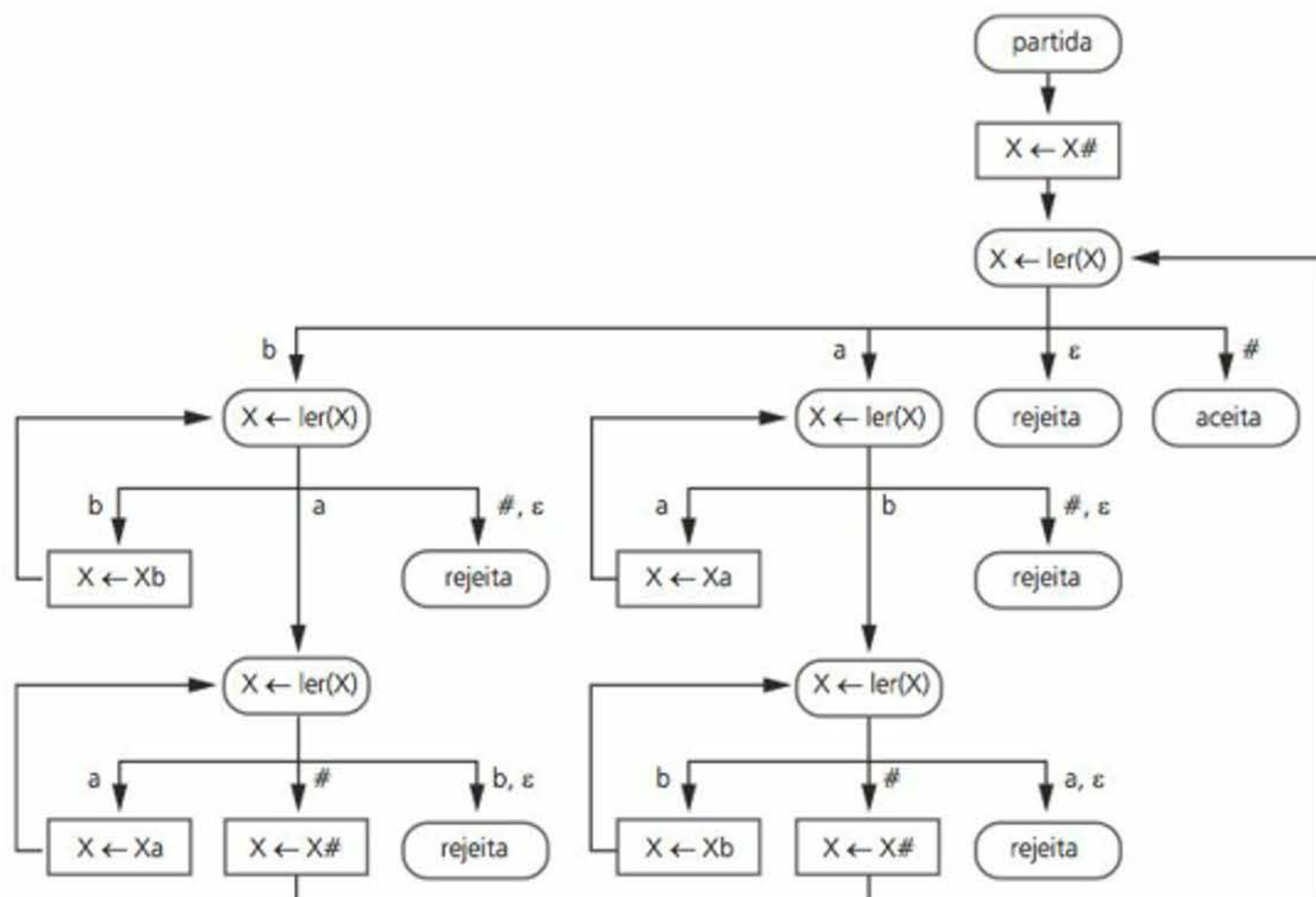


figura 7.6 Diagrama de fluxo da máquina de Post – $a^n b^n$ ou $b^n a^n$.

A seguir, é enunciado o teorema que prova que a máquina de Post é equivalente à máquina de Turing, reforçando a evidência de que são máquinas universais. A prova é omitida e, resumidamente, é como segue:

a *Máquina de Turing $\leq$ máquina de Post.*

- A estrutura de fita da máquina de Turing é simulada pela máquina de Post, usando a variável X , onde a posição corrente da cabeça corresponde à primeira posição em X . Por exemplo, a fita ilustrada na figura 7.7 é simulada pela variável X com o seguinte conteúdo:

$$X = a_3 a_4 \dots a_n \# a_1 a_2$$

- o movimento para a esquerda da cabeça da fita exige a execução de tantas instruções **desvio** e instruções **atribuição** quantos forem os símbolos da palavra corrente;
- o movimento para a direita da cabeça da fita exige a execução de uma instrução **desvio** (leitura com teste), consumindo um símbolo variável X (no início da fila) e a correspondente **atribuição** (no fim da fila);
- a simulação dos estados é como segue:
 - estado inicial é simulado pela instrução **partida**, seguido pela execução de um **desvio**;
 - estados finais são simulados pela instrução **aceita**;
 - demaís estados são simulados por instruções **desvio** (leitura com teste);
- as condições de rejeição (função programa indefinida ou movimento inválido) são simuladas pela instrução **rejeita**;

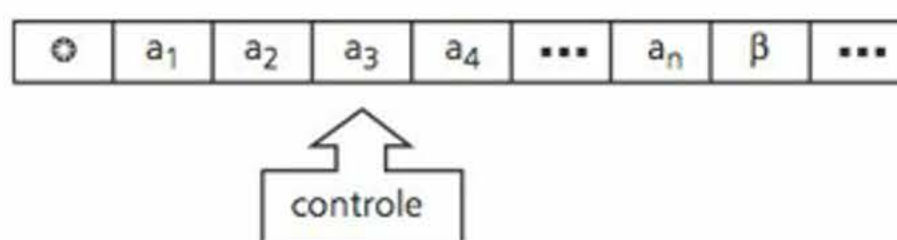


figura 7.7 Fita e posição da cabeça da fita.

b *Máquina de Post $\leq$ máquina de Turing.*

- A variável X da máquina de Post é simulada na máquina de Turing usando a estrutura de fita, onde a primeira posição da fila corresponde à posição corrente da cabeça da fita. Por exemplo, a fita e a cabeça da fita, como ilustrado na figura 7.8, corresponde a:

$$X = a_1 a_2 a_3 \dots a_m \# a_{m+1} \dots a_n$$

- o **desvio** realiza a leitura com remoção do símbolo mais à esquerda da variável X , o que pode ser simulado pela substituição do conteúdo da correspondente célula de fita pelo símbolo branco, resultando em símbolos brancos entre o marcador de início de fita e primeiro símbolo não branco (no caso de inconveniência desses brancos, pode-se realizar um deslocamento do conteúdo para a esquerda);
- a **atribuição** de um símbolo s (à direita do conteúdo da variável X), pode ser simulada movendo a cabeça para a direita até o primeiro símbolo branco, substituindo-o pelo símbolo s e, retornando para a posição correspondente ao primeiro símbolo da fila;

- a instrução **partida** pode ser simulada usando o estado inicial;
- uma instrução **aceita** pode ser simulada usando um estado final;
- uma instrução **rejeita** pode ser simulada usando uma condição excepcional de parada (como um movimento inválido).

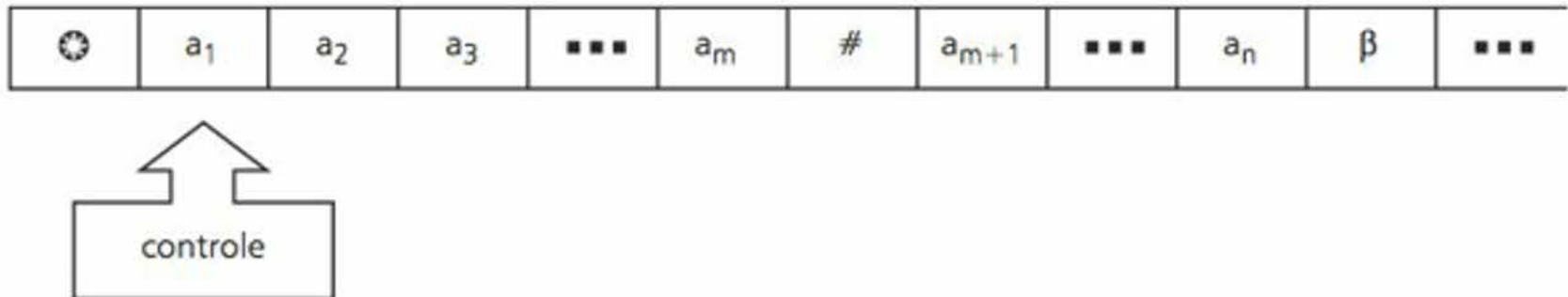
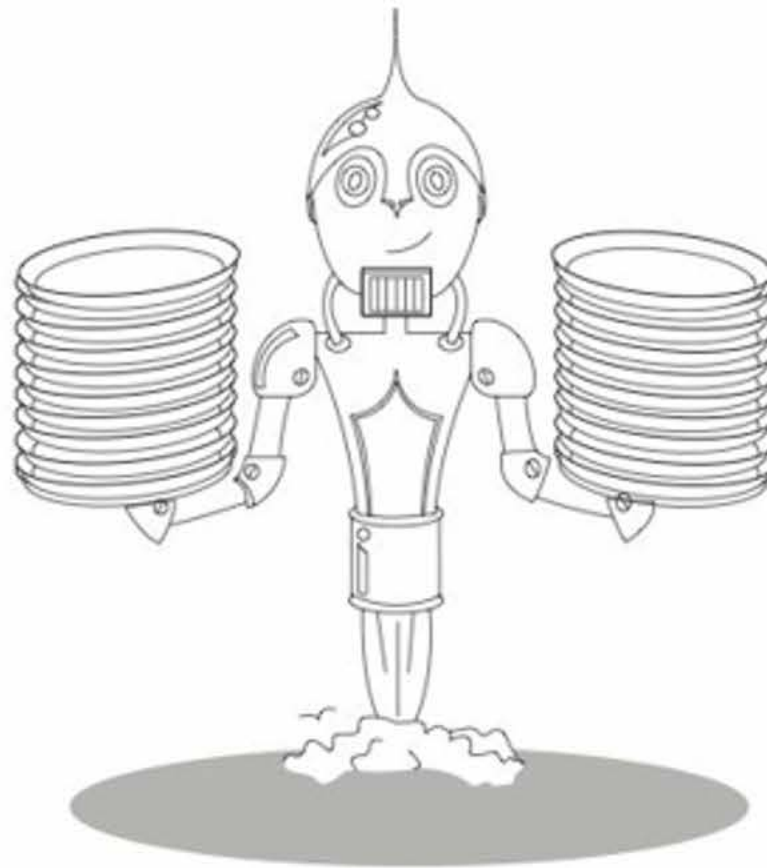


figura 7.8 Simulação da fila pela fita.

teorema 7.2 – Equivalência entre máquina de Turing e máquina de Post

O formalismo máquina de Turing pode ser simulado pelo formalismo máquina de Post e vice-versa. Logo, são equivalentes. □



7.2

→ **máquina com pilhas**

A *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 7.20, 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 pilhas*, possui um poder computacional relativamente restrito, pois não tem memória auxiliar para armazenar informações de trabalho. Por exemplo, não existe máquina finita 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*, 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 das máquinas finitas, ainda possui uma capacidade computacional restrita. Por exemplo, não existe máquina com uma pilha capaz de reconhecer um triplo balanceamento como em $\{ a^n b^n c^n \mid n \geq 0 \}$ (procure intuir por que não existe tal máquina). Entretanto, também é uma classe de máquinas de fundamental importância no estudo das *linguagens formais*, como em compiladores de linguagens tipo Pascal;
- c** *Máquina com duas pilhas.* Conforme será comentado adiante, o formalismo máquina com duas pilhas possui o mesmo poder computacional que os formalismos máquina de Turing e máquina de Post;
- d** *Máquina com mais de duas pilhas.* A classe das máquinas com mais de duas pilhas possui o mesmo poder computacional que a classe das 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.

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 sequência 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 com-

primento 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 observar 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 a leitura (e remoção) são sempre na mesma extremidade.

definição 7.3 – 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 7.9 (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 7.9 (direita);

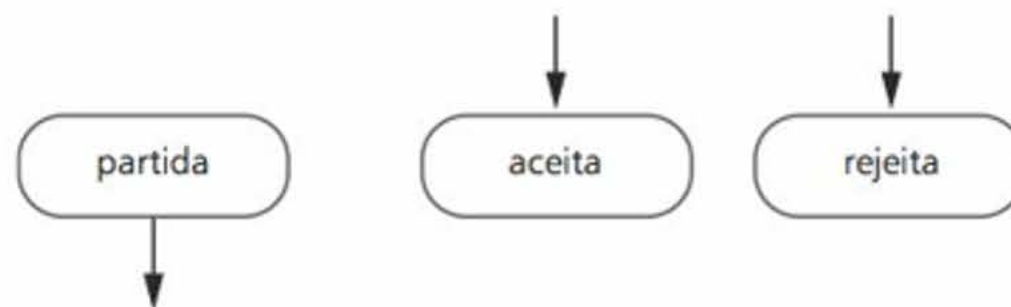


figura 7.9 Partida (esquerda) e parada (direita) em um diagrama de fluxos.

- 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

7.10, onde $X \leftarrow \text{ler}(X)$ ou $Y_i \leftarrow \text{ler}(Y_i)$ denotam 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.

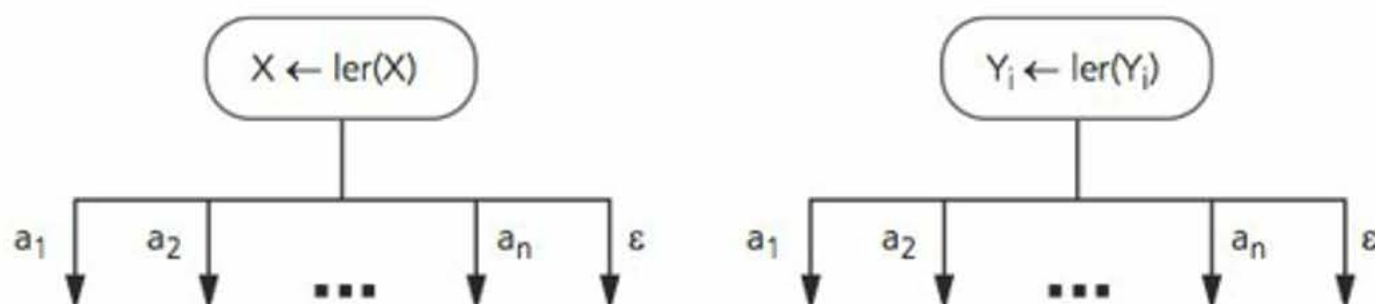


figura 7.10 Desvio (esquerda) e desempilha (direita) em um diagrama de fluxos.

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 7.11, supondo que $s \in \Sigma$. $\square$

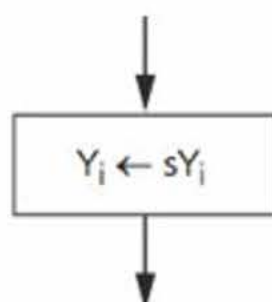


figura 7.11 Empilha em um diagrama de fluxos.

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 aceita 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 7.3 – Máquina com pilhas – duplo balanceamento

Considere a seguinte linguagem introduzida anteriormente:

$$\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$$

A máquina com pilhas:

$$\text{Pilhas_Duplo_Bal} = (\{ a, b \}, D)$$

onde D é como ilustrado na figura 7.12, é tal que:

$$ACEITA(Pilhas_Duplo_Bal) = Duplo_Bal$$

$$REJEITA(Pilhas_Duplo_Bal) = \Sigma^* - Duplo_Bal$$

e, portanto, $LOOP(Pilhas_Duplo_Bal) = \emptyset$

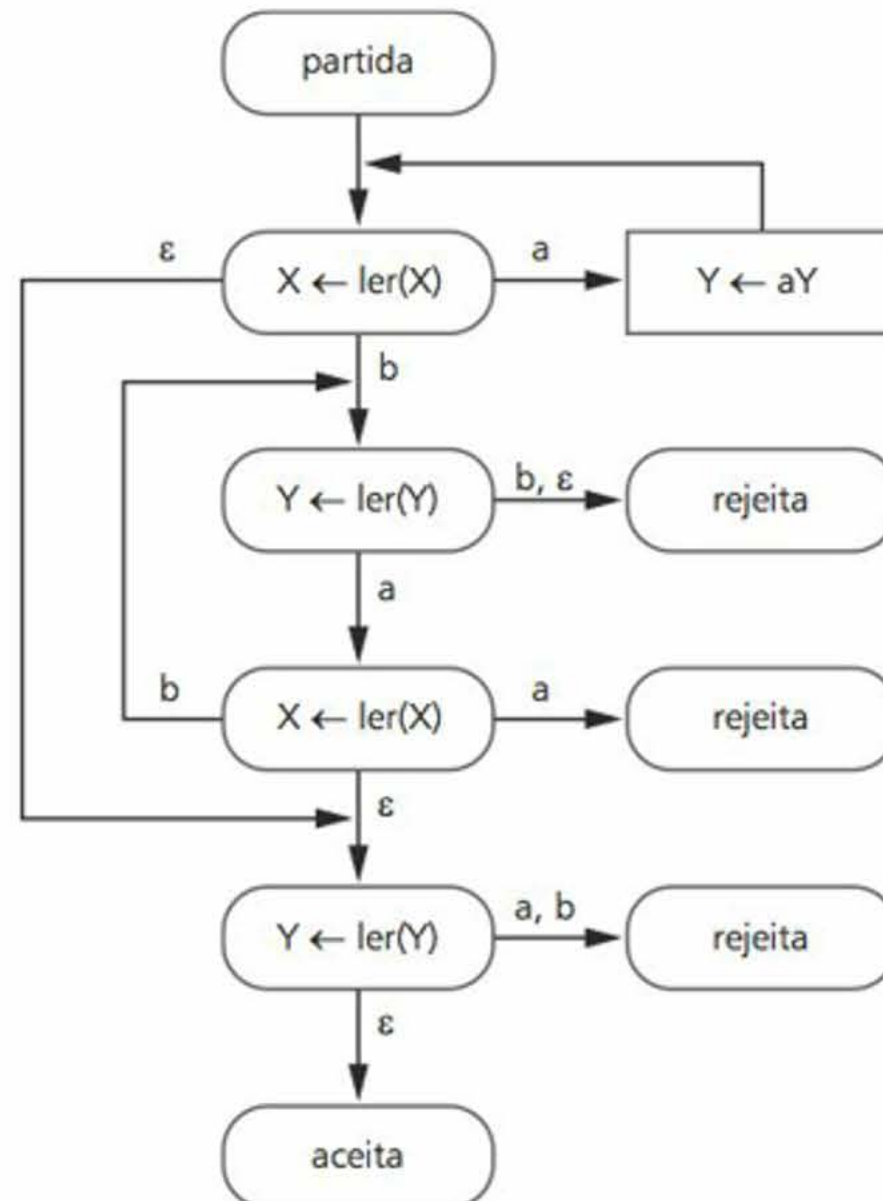


figura 7.12 Diagrama de fluxo da máquina com pilhas – duplo balanceamento.

O algoritmo lê o prefixo de símbolos a e empilha na única pilha utilizada Y . Em seguida, para cada símbolo b em X deve existir um correspondente a em Y . Compare este algoritmo com o apresentado no exemplo 7.1 - Máquina de Post – duplo balanceamento. $\square$

exemplo 7.4 – Máquina com pilhas – triplo balanceamento

Considere a seguinte linguagem introduzida anteriormente:

$$Triplo_Bal = \{ a^n b^n c^n \mid n \geq 0 \}$$

A máquina com pilhas:

$$Pilhas_Triplo_Bal = (\{ a, b, c \}, D)$$

onde D é como ilustrado na figura 7.13, é tal que:

$$ACEITA(Pilhas_Triplo_Bal) = Triplo_Bal$$

$$REJEITA(Pilhas_Triplo_Bal) = \Sigma^* - Triplo_Bal$$

e, portanto, $LOOP(Pilhas_Triplo_Bal) = \emptyset$

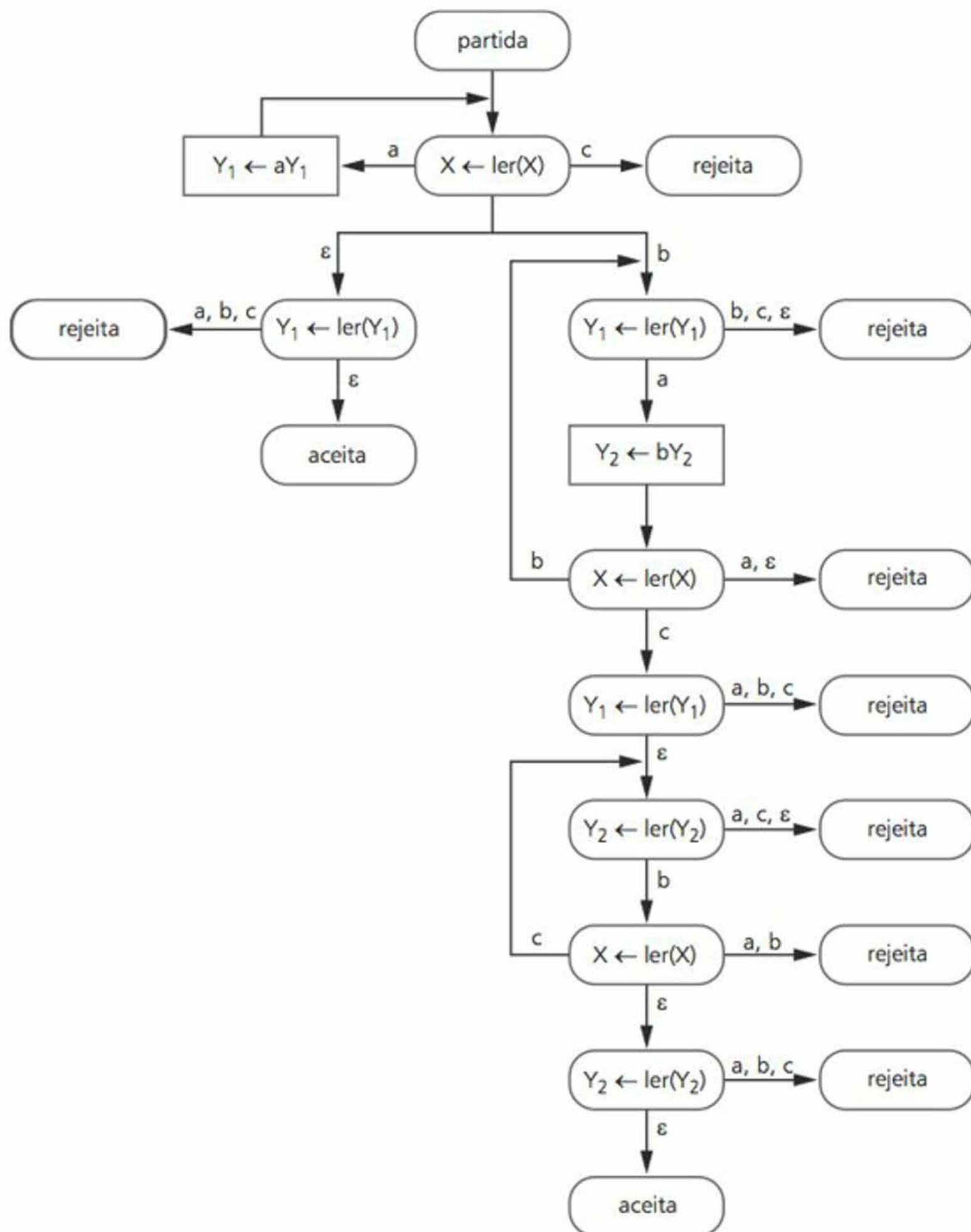


figura 7.13 Diagrama de fluxo da máquina com pilhas – triplo balanceamento.

Note-se que foram usadas duas pilhas. O algoritmo lê o prefixo de símbolos a e empilha em Y_1 . Em seguida, 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 . $\square$

exemplo 7.5 – Máquina finita – prefixo

Considere a seguinte linguagem:

$$\text{Prefixo_aaa} = \{ w \mid w \in \{ a, b \}^* \text{ e } w \text{ possui a subpalavra } aaa \text{ como prefixo} \}$$

Por exemplo, **aaabab**, **aaaaaaab** e **aaa** são palavras de **Prefixo_aaa**. A máquina finita:

$$\text{Finita_Prefixo_aaa} = (\{ a, b \}, D)$$

onde D é como ilustrado na figura 7.14, é tal que:

$$\text{ACEITA}(\text{Finita_Prefixo_aaa}) = \text{Prefixo_aaa}$$

$$\text{REJEITA}(\text{Finita_Prefixo_aaa}) = \Sigma^* - \text{Prefixo_aaa}$$

e, portanto, $\text{LOOP}(\text{Finita_Prefixo_aaa}) = \emptyset$

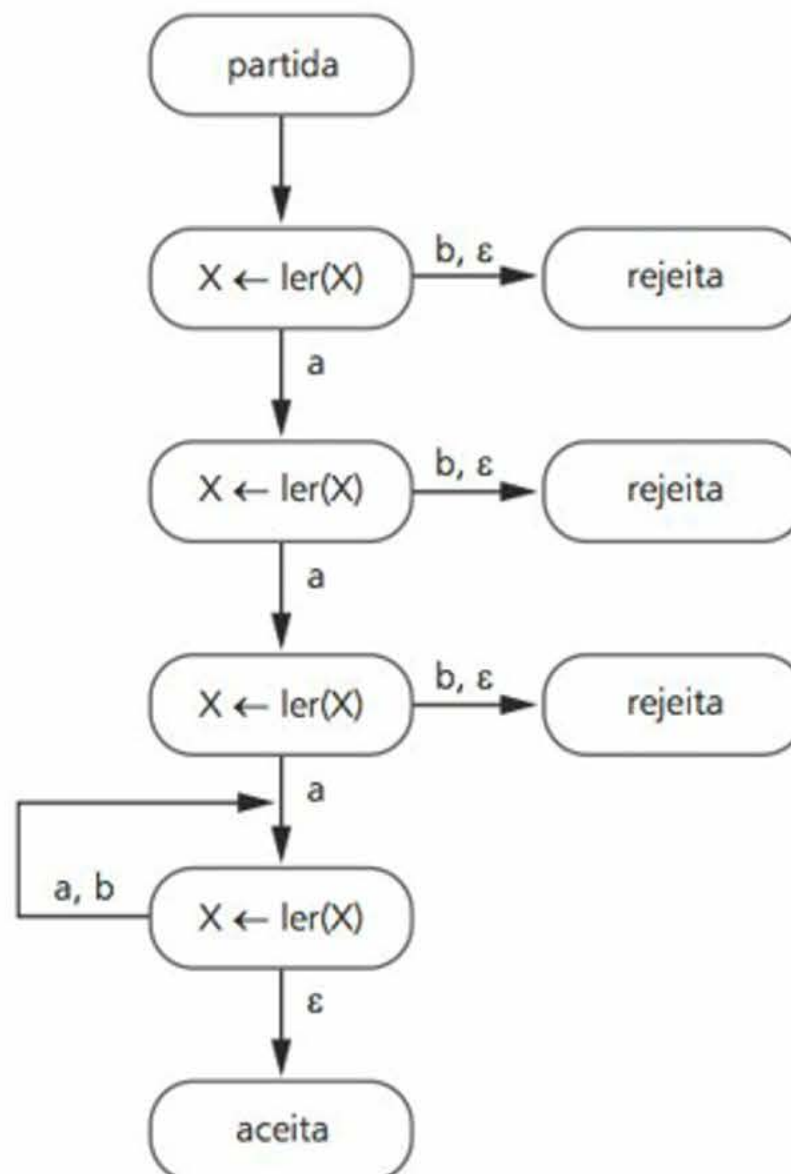


figura 7.14 Diagrama de fluxo da máquina finita – prefixo.

Note-se que *não* foram usadas pilhas. O algoritmo lê o prefixo **aaa** executando três leituras. Em seguida, qualquer sufixo é aceito. $\square$

7.3

→ autômato com pilhas

O *autômato com pilhas* ou *autômato com duas pilhas* é uma máquina universal similar à máquina com duas pilhas. Uma importante diferença é que o programa é especificado usando as noções de estados e de transições (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 o autômato com estados é, em geral, mais indicado para estudos teórico-formais, pois facilita provas e estudos de facilidades especiais, por exemplo, o 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 e autômato. 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, a leitura e gravação das pilhas e define o estado da máquina.

As pilhas são dividida em células, armazenando, em cada uma, um símbolo do alfabeto auxiliar (pode ser igual ao alfabeto de entrada). Lembre-se de 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, seu tamanho corrente é igual ao tamanho da palavra armazenada e seu valor inicial é vazio.

A unidade de controle possui um número finito e predefinido de estados. Possui também uma cabeça de fita e uma cabeça para cada pilha, como segue:

- a** *Cabeça da fita*. Unidade de leitura que acessa uma célula da fita de cada vez e movimentta-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 que se 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 7.4 – Autômato com pilhas

Um *autômato com pilhas* ou *autômato com duas 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:*

$$\Pi: Q \times (\Sigma \cup \{\epsilon, ?\}) \times (V \cup \{\epsilon, ?\}) \times (V \cup \{\epsilon, ?\}) \rightarrow Q \times (V \cup \{\epsilon\}) \times (V \cup \{\epsilon\})$$

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 programa 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;
- 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, $\Pi(p, ?, a, \epsilon) = (q, \epsilon, b)$ indica que:

se:

- no estado p
- a entrada foi completamente lida (na fita)
- o topo da pilha 1 contém o símbolo a
- não lê da pilha 2

então:

- assume o estado q
- não grava na pilha 1
- grava o símbolo b no topo da pilha 2

De forma análoga à máquina de Turing, a função programa de um autômato com pilhas pode ser representada como um grafo direto, como na figura 7.15.

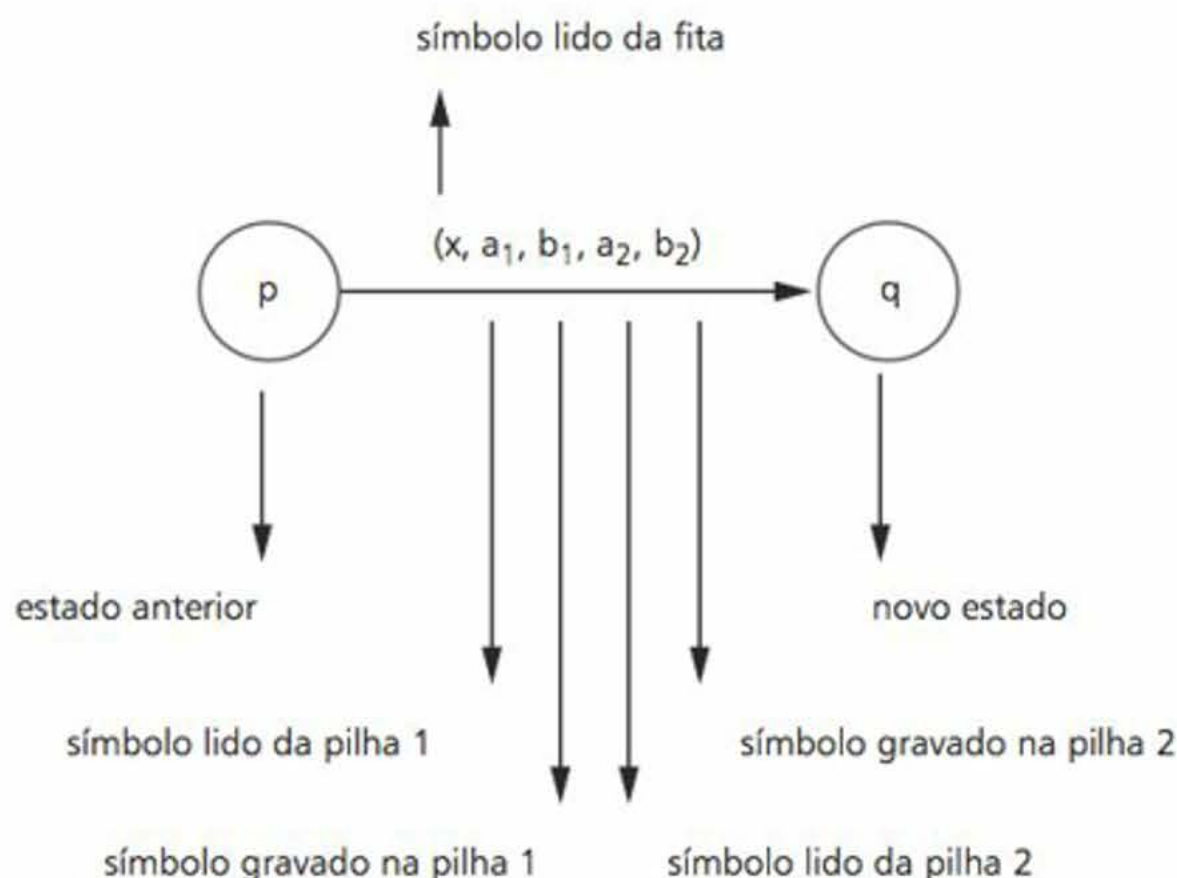


figura 7.15 Representação da função programa como um grafo.

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 para e a palavra de entrada é aceita;
- b** *Função indefinida.* A função programa é indefinida para o argumento: o autômato para e a palavra de entrada é rejeitada.

exemplo 7.6 – Autômato com pilhas – duplo balanceamento

Considere a seguinte linguagem introduzida anteriormente:

$$\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$$

O autômato com pilhas

$$A2P_Duplo_Bal = (\{ a, b \}, \{ q_0, q_1, q_f \}, \Pi, q_0, \{ q_f \}, \Sigma \cup \{ B \})$$

onde Π é como abaixo, é tal que $\text{ACEITA}(\text{A2P_Duplo_Bal}) = \text{Duplo_Bal}$:

$$\Pi(q_0, a, \varepsilon, \varepsilon) = (q_0, B, \varepsilon)$$

$$\Pi(q_0, b, B, \varepsilon) = (q_1, \varepsilon, \varepsilon)$$

$$\Pi(q_0, ?, ?, \varepsilon) = (q_f, \varepsilon, \varepsilon)$$

$$\Pi(q_1, b, B, \varepsilon) = (q_1, \varepsilon, \varepsilon)$$

$$\Pi(q_1, ?, ?, \varepsilon) = (q_f, \varepsilon, \varepsilon)$$

O autômato com pilhas pode ser representado pelo grafo ilustrado na figura 7.16. 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, a pilha 1 estiver vazia. Observe que o conjunto $\text{LOOP}(\text{A2P_Duplo_Bal})$ é vazio. Compare com o exemplo 7.3 - Máquina com pilhas – duplo balanceamento. $\square$

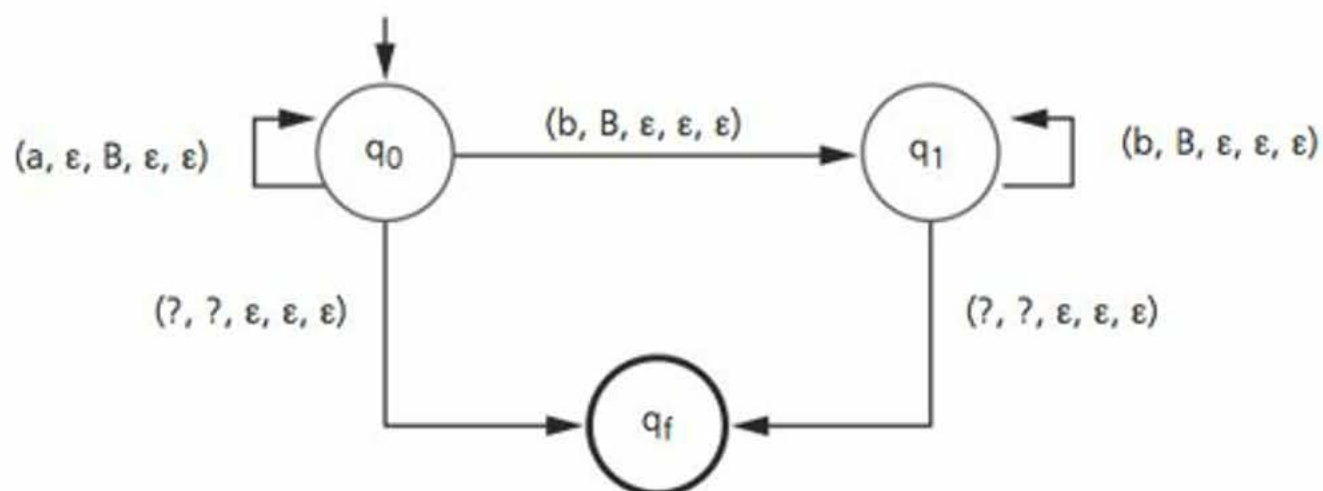


figura 7.16 Grafo do autômato com pilhas - duplo balanceamento.

Note-se que o algoritmo do 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 7.7 – Autômato com pilhas – triplo balanceamento

Considere a seguinte linguagem introduzida anteriormente:

$$\text{Triplo_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$$

O autômato com pilhas

$$\text{A2P_Triplo_Bal} = (\{ a, b, c \}, \{ q_0, q_1, q_2, q_f \}, \Pi, q_0, \{ q_f \}, \Sigma \cup \{ B, C \})$$

ilustrado na figura 7.17, é tal que $\text{ACEITA}(\text{A2P_Triplo_Bal}) = \text{Triplo_Bal}$.

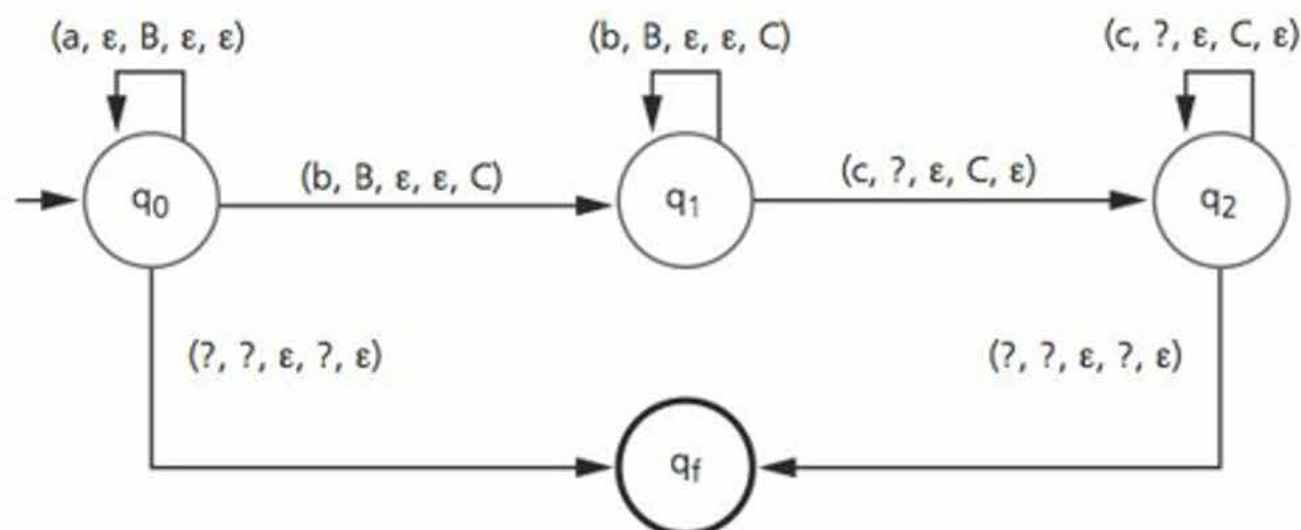


figura 7.17 Grafo do autômato com pilhas - triplo balanceamento.

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. Observe que este autômato sempre para. Compare com o exemplo 7.4 - Máquina com pilhas – triplo balanceamento. $\square$

A seguir, é enunciado o teorema que prova que o formalismo autômato com duas pilhas é equivalente ao formalismo máquina de Turing, reforçando a evidência de que ambas são máquinas universais (bem como máquina de Post e máquina Norma). Resumidamente, a prova é como abaixo (a prova formal é omitida). Note-se que a mesma ideia básica pode ser usada para provar que o formalismo máquina com pilhas é equivalente ao formalismo máquina 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 (ver figura 7.18): a pilha 1 simula o conteúdo da fita à esquerda da cabeça da fita, e a pilha 2, o conteúdo à direita, bem como da célula apontada pela cabeça;
- 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.



figura 7.18 Simulação da fita por duas pilhas.

teorema 7.5 – Equivalência entre máquina de Turing e 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. $\square$

7.4

→ máquinas não determinísticas

O não determinismo é uma importante generalização dos modelos de máquinas. A mesma ideia aplicada na máquina de Turing não determinística é válida para a variável X da máquina de Post ou para as pilhas do autômato com pilhas.

Como já introduzido, a facilidade do *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, nem no símbolo lido dos demais caminhos alternativos.

exemplo 7.8 – Autômato com pilhas não determinístico – palavra e reversa

Considere a seguinte linguagem:

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	abaabaabaaba
abbbba	bbbbbbbbbaabbbbbbbbb

Esse tipo de linguagem pode ser considerado 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:

APN_Palavra_Reversa = $(\{ a, b \}, \{ q_0, q_1, q_f \}, \Pi, q_0, \{ q_f \}, \{ a, b \})$

ilustrado na figura 7.19, é tal que:

ACEITA(APN_Palavra_Reversa) = Palavra_Reversa

É não determinístico devido às duas alternativas de movimentos a partir de q_0 , para os mesmos 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 foi identificada toda a primeira metade);

- ocorre um movimento não determinística 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;
- observe que a pilha 2 não foi usada nem para leitura e nem para gravação (usou-se o símbolo ϵ nas respectivas posições das transições). Portanto, esta solução poderia ser desenvolvida usando apenas uma pilha (nesse caso, o não determinismo é obrigatório?). □

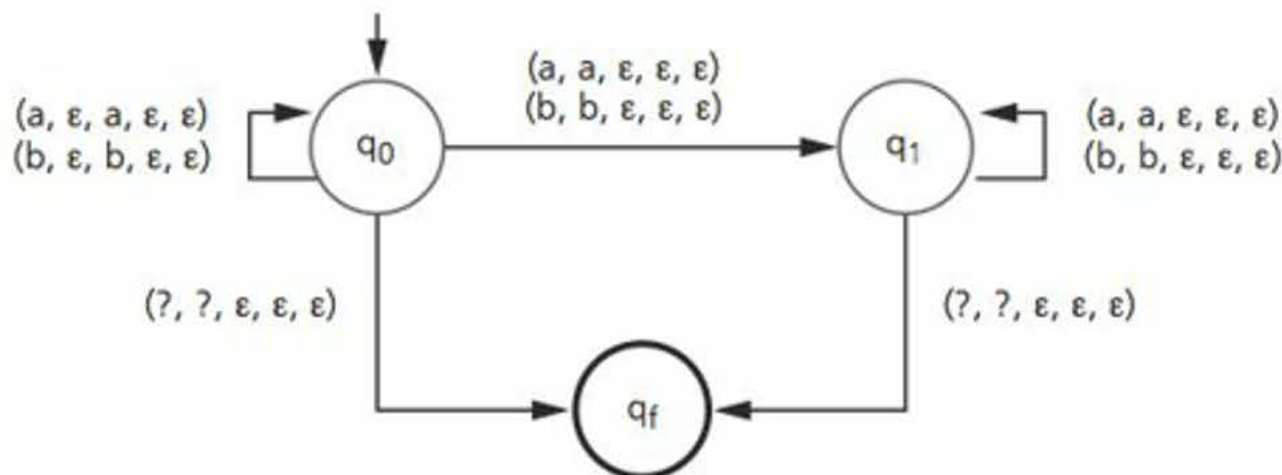


figura 7.19 Grafo do autômato com pilhas não determinístico - palavra e reversa.

O seguinte teorema não será demonstrado.

teorema 7.6 – Equivalência entre máquina de Turing e máquinas não determinísticas

O formalismo máquina de Turing pode ser simulado pelos seguintes formalismos e vice-versa.

- a** Máquina de Post não determinística;
- b** Máquina com pilhas 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.

7.5

→ hierarquia de classes de máquinas

Todos os formalismos de máquinas introduzidos para representar algoritmos, bem como suas diversas modificações, são equivalentes à máquina de Turing, ou seja, são máquinas universais. Lembre-se de que, no caso da máquina com pilhas, são necessárias pelo menos duas pilhas para ele ser considerado uma máquina universal. Assim, os diversos modelos, bem como suas modificações, podem ser usados indistintamente, dependendo do caso, para facilitar construções ou provas.

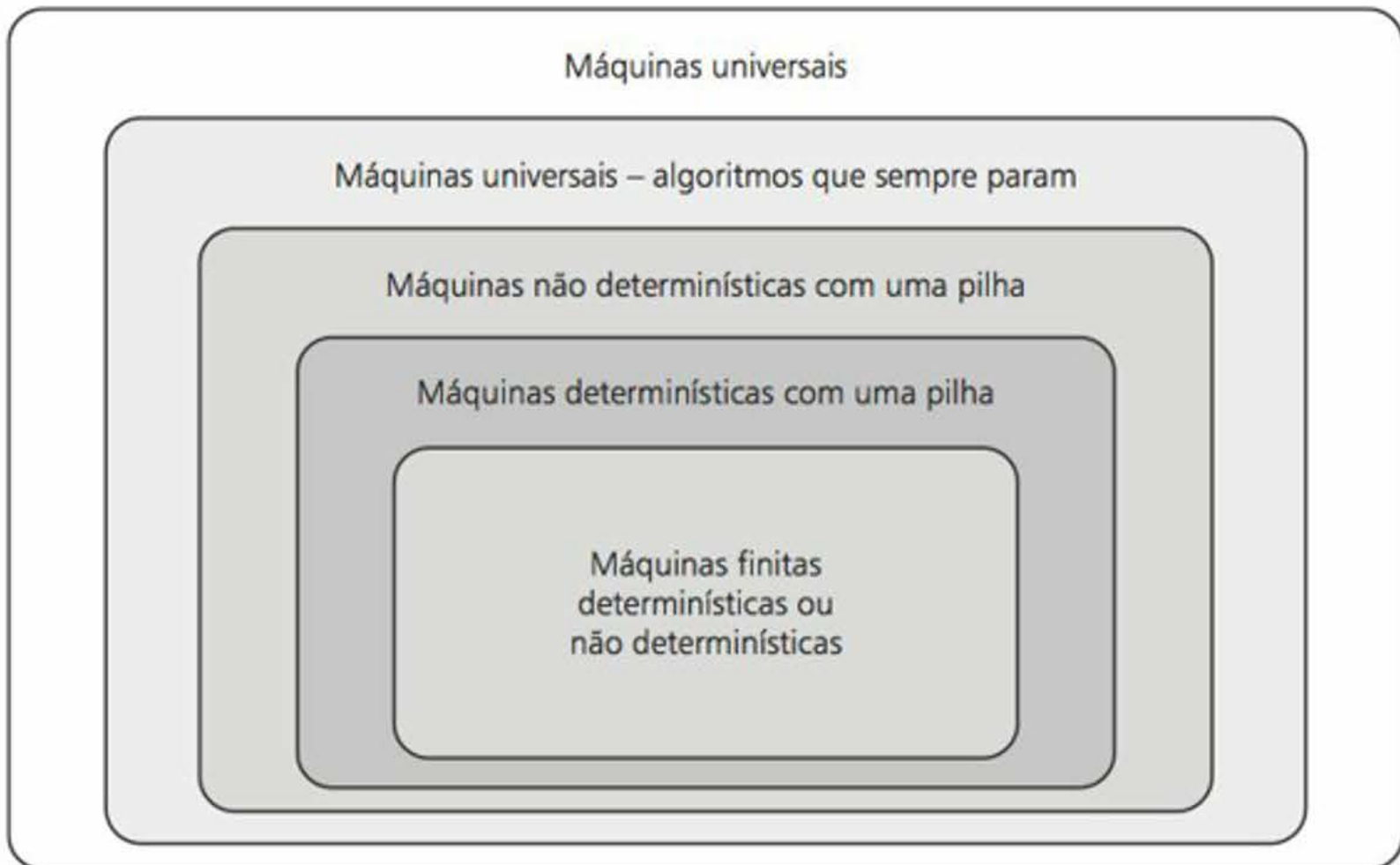


figura 7.20 Hierarquia de classes de máquinas.

A figura 7.20 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 7.21, a qual é como segue:

- a** *Linguagens regulares.* Correspondem à classe das máquinas finitas (sem pilhas). 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 das máquinas ou autômato 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 7.8 - Autômato com pilhas não determinístico – palavra e reversa):

$$\text{Palavra_Reversa} = \{ ww^r \mid w \text{ pertence a } \{ a, b \}^* \}$$

O tempo de reconhecimento de uma entrada é diretamente proporcional ao dobro do tamanho da entrada;

- c** *Linguagens livres do contexto.* Correspondem à classe das máquinas ou autômato com uma pilha não determinísticos. Constituem uma classe de fundamental importância, pois incluem linguagens de programação como Algol, Pascal e C. Entretanto, algumas linguagens muito simples não pertencem a esta classe de linguagens como:

$$\text{Triplo_Bal} = \{ a^n b^n c^n \mid n \geq 0 \}$$

$$\text{Palavra_Palavra} = \{ ww \mid w \text{ é palavra sobre os símbolos } a \text{ e } b \}$$

Os melhores algoritmos 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 para 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, correspondem à classe de todas as linguagens que podem ser reconhecidas mecanicamente.

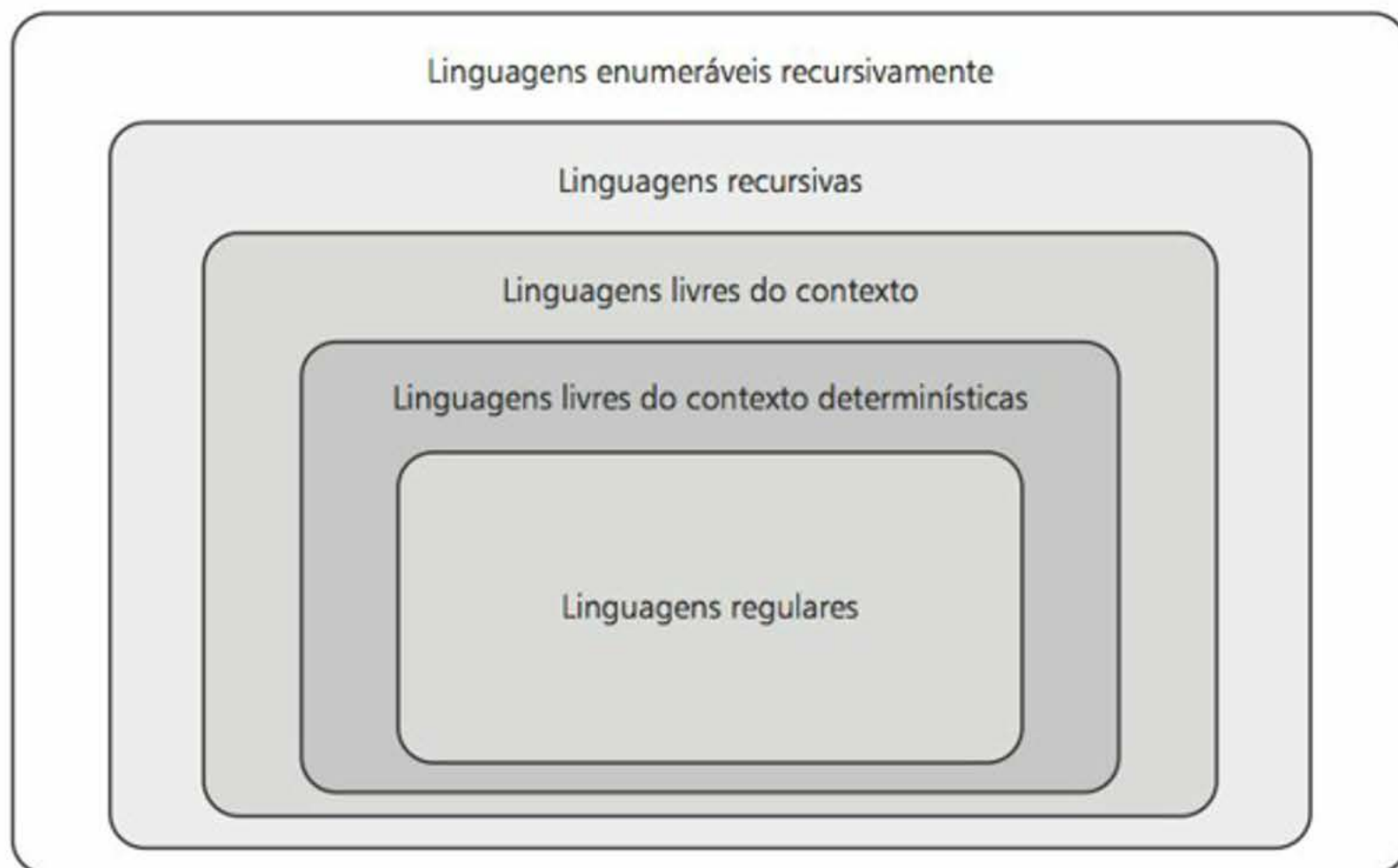


figura 7.21 Hierarquia de classes de linguagens.

7.6

→ conclusões

Neste capítulo, foram estudados os formalismos:

- Máquina de Post (baseado em uma variável do tipo fila);
- Máquinas com pilhas, constituídas por uma família de máquinas com capacidades ou poder computacional distintos. Tem-se a máquina finita (sem pilha alguma de memória auxiliar), a máquina com uma ou com duas (ou mais) pilhas. Duas pilhas são suficientes para atingir o poder computacional equivalente da máquina de Turing;
- Autômato com pilhas, análogos às máquinas com pilhas, diferenciado-se na forma de expressar a função programa, substituindo diagramas de fluxos por grafos diretos, representando estados e transições;
- Máquinas não determinísticas cujos efeitos não aumentam o poder computacional de suas correspondentes versões determinísticas, excetuando-se as máquinas ou autômato com uma pilha.

No próximo capítulo, serão vistos outros formalismos equivalentes, mas do tipo denotacional, baseados no conceito de recursão.

7.7

→ exercícios

exercício 7.1 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. Em particular, compare as estruturas de dados de cada modelo.

exercício 7.2 Desenvolva máquinas de Post, determinísticas ou não, que aceitem as seguintes linguagens:

- a** $L_1 = \emptyset$
- b** $L_2 = \{ \varepsilon \}$
- c** $L_3 = \{ w \mid w \text{ tem o mesmo número de símbolos } a \text{ e } b \}$
- d** $L_4 = \{ w \mid \text{o décimo símbolo da direita para a esquerda é } a \}$
- e** $L_5 = \{ waw \mid w \text{ é palavra de } \{ a, b \}^* \}$
- f** $L_6 = \{ ww \mid w \text{ é palavra de } \{ a, b \}^* \}$
- g** $L_7 = \{ ww^r \mid w \text{ é palavra de } \{ a, b \}^* \text{ onde } w^r \text{ denota a palavra reversa de } w \}$
- h** $L_8 = \{ www \mid w \text{ é palavra de } \{ a, b \}^* \}$
- i** $L_9 = \{ w \mid w = a^1 b^2 a^3 b^4 \dots a^{n-1} b^n \text{ e } n \text{ é número natural par} \}$
- j** $L_{10} = \{ w \mid w = a^n b^n \text{ ou } w = b^n a^n \}$
- k** $L_{11} = \{ w \mid w = a^i b^j c^k, \text{ onde ou } i = j \text{ ou } j = k \}$

exercício 7.3 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 . Construa uma máquina de Post sobre $\Sigma = \{v, f, e, ou\}$ tal que:

$$\begin{aligned} \text{ACEITA}(M) &= \{EB \mid EB \text{ é } v\} \\ \text{REJEITA}(M) &= \{EB \mid EB \text{ é } f\} \\ \text{LOOP}(M) &= \{w \mid w \text{ não é EB}\} \end{aligned}$$

exercício 7.4 Dê a máquina de Post MP, sobre o alfabeto $\{a, b\}$, tal que:

$$\begin{aligned} \text{ACEITA}(MP) &= \{w \mid w \text{ tem o mesmo número de símbolos } a \text{ e } b\} \\ \text{REJEITA}(MP) &= \{w \mid w \text{ cuja diferença entre o número de símbolos } a \text{ e } b \text{ é } 1\} \\ \text{LOOP}(MP) &= \{a, b\}^* - (\text{ACEITA}(MP) \cup \text{REJEITA}(MP)) \end{aligned}$$

Por exemplo:

$$\begin{aligned} ab &\in \text{ACEITA}(MP) \\ aba &\in \text{REJEITA}(MP) \\ aaba &\in \text{LOOP}(MP). \end{aligned}$$

exercício 7.5 Elabore uma máquina de Post P sobre o alfabeto $\{a, b\}$, tal que:

$$\begin{aligned} \text{ACEITA}(P) &= \{a^m b^n \mid m > n \geq 0\} \\ \text{REJEITA}(P) &= \{a, b\}^* - \text{ACEITA}(P) \end{aligned}$$

exercício 7.6 Para a demonstração do teorema 7.2 – Equivalência entre máquina de Turing e máquina de Post, na simulação da máquina de Turing pela máquina de Post, é necessário que a máquina de Post 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 $a_1 a_2 \dots a_{n-1} a_n$ passa a ser $a_n a_1 a_2 \dots a_{n-1}$. Desenvolva uma máquina que processe essa rotação.

exercício 7.7 Desenvolva autômato com duas pilhas ou máquinas com pilhas, determinísticas ou não, que aceitem as linguagens do exercício 7.2.

exercício 7.8 Prove que uma máquina com n pilhas ($n > 2$) é equivalente a uma máquina com duas pilhas.

exercício 7.9 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 7.10 Sobre a máquina de Post, marque a alternativa *errada*:

- a** A leitura numa máquina de Post é destrutiva, pois o símbolo lido é retirado da variável X
- b** A variável X não possui tamanho finito;
- c** A variável X é do tipo fila, ou seja, o último símbolo gravado é o primeiro a ser lido;

- ☐ **d** O símbolo # é o único símbolo auxiliar;
- ☐ **e** A máquina de Post pode ficar em *loop* infinito.

exercício 7.11 Sobre a máquina de Post, analise as seguintes afirmações:

- I. A concatenação do símbolo # no fim da pilha tem de ser feita na primeira instrução do diagrama de fluxo;
- II. Um diagrama de fluxos tem sempre $n+2$ ramificações após uma atribuição, onde n é o numero de símbolos do alfabeto de entrada;
- III. A pilha da máquina de Post é infinita.

Marque a alternativa correta:

- ☐ **a** Apenas I e II estão corretas;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas III está correta;
- ☐ **d** Apenas I e III estão corretas;
- ☐ **e** Nenhuma afirmação está correta.

exercício 7.12 Sobre a máquina de Post, marque a alternativa correta:

- ☐ **a** Para quando o programa atinge uma instrução de parada (**aceita** ou **rejeita**) ou por indefinição;
- ☐ **b** Um programa em Post sempre contém exatamente uma instrução de partida e duas de parada (**aceita** e **rejeita**);
- ☐ **c** No início da execução de um programa, a fita sempre contém a palavra de entrada concatenada com o símbolo de final de fila #.
- ☐ **d** Após a leitura de um símbolo, o programa pode retirá-lo da fila;
- ☐ **e** Um programa em Post é uma sequência finita de instruções de quatro tipos possíveis (**partida**, **parada**, **desvio** e **atribuição**).

exercício 7.13 Sobre a máquina de Post, analise as seguintes afirmações:

- I. O programa para esta classe de máquinas é total;
- II. A variável X é do tipo fila e é utilizada como memória de trabalho, entrada e saída;
- III. As instruções do tipo desvio são definidas como funções totais.

Marque a alternativa correta:

- ☐ **a** Apenas II está correta;
- ☐ **b** Apenas I e II estão corretas;
- ☐ **c** Apenas II e III estão corretas.
- ☐ **d** I, II e III estão corretas;
- ☐ **e** Nenhuma afirmação está correta.

exercício 7.14 Sobre a simulação do formalismo máquina de Turing pelo do formalismo máquina de Post, analise as seguintes afirmações:

- I. O estado inicial é simulado pela instrução **partida**, seguido pela execução de um **desvio**, e os estados finais, simulados pela instrução **aceita**, sendo os demais estados simulados por instruções de **desvio**;
- II. A variável X é simulada pela fita e a posição da cabeça da fita representa a posição mais à esquerda da fila;
- III. Para simular o movimento para a esquerda da cabeça da fita é necessário executar tantos **desvios** quantos forem os símbolos da palavra corrente;
- IV. Uma indefinição na função programa da máquina de Turing é simulada em Post pela instrução **rejeita**.

Marque a alternativa correta:

- ☐ **a** Apenas I, II e III estão corretas;
- ☐ **b** Apenas II e III estão corretas;
- ☐ **c** Apenas I e IV estão corretas;
- ☐ **d** Apenas I, III e IV estão corretas;
- ☐ **e** Nenhuma das alternativas anteriores está correta.

exercício 7.15 Sobre a família das máquinas com pilhas, analise as seguintes afirmações:

- I. Uma máquina finita, que corresponde a uma máquina sem pilhas, possui um poder computacional relativamente restrito, pois não tem memória auxiliar de trabalho;
- II. A classe das máquinas sem pilhas é de fundamental importância no estudo das linguagens formais como, por exemplo, no desenvolvimento de reconhecedores de linguagens de programação tipo Pascal;
- III. A classe das máquinas com uma pilha possui poder computacional maior que o da classe das máquinas sem pilhas, sendo capaz, inclusive, de reconhecer um triplo balanceamento.

Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas I e III estão corretas;
- ☐ **d** Apenas II e III estão corretas;
- ☐ **e** I, II e III estão corretas.

exercício 7.16 Sobre a família das máquinas com pilhas, marque a alternativa correta:

- ☐ **a** Uma máquina com pilhas não possui alfabeto auxiliar, mas pode usar símbolos do alfabeto de entrada como símbolos auxiliares;
- ☐ **b** Uma máquina finita (sem pilhas) determinística não pode reconhecer a linguagem duplo balanceamento, mas uma não determinística pode;

- ☐ **c** Uma máquina com uma pilha não é uma máquina universal, a menos que seja não determinística;
- ☐ **d** Assim como na máquina de Post, a máquina com pilhas possui uma fita somente para leitura;
- ☐ **e** Nenhuma das anteriores.

exercício 7.17 No formalismo máquina com pilhas, existe maneira de aumentar o poder computacional, realizando alguma das alterações abaixo? Marque a alternativa correta:

- ☐ **a** Múltiplas pilhas;
- ☐ **b** Não determinismo;
- ☐ **c** Múltiplas cabeças;
- ☐ **d** Maior número de instruções básicas;
- ☐ **e** Nenhuma das anteriores.

exercício 7.18 Sobre máquina com Pilhas, marque a alternativa errada (caso todas estejam corretas, marque a última alternativa):

- ☐ **a** $X \leftarrow \text{Ler}(X)$ denota uma leitura, onde se lê o símbolo mais à esquerda de X ou do topo de Y , não retirando da estrutura o símbolo lido;
- ☐ **b** Uma máquina com pilhas é uma dupla $M = (\Sigma, D)$ onde Σ é o alfabeto de símbolos de entrada e D é o diagrama de fluxos;
- ☐ **c** Uma máquina com pilhas reconhece o triplo balanceamento;
- ☐ **d** Uma máquina com pilhas possui a memória de entrada separada das memórias de trabalho e de saída;
- ☐ **e** Nenhuma das anteriores.

exercício 7.19 Sobre autômato com duas pilhas, marque a alternativa errada:

- ☐ **a** Um autômato com duas pilhas é composto basicamente por: fita, pilhas, unidade de controle e função de transição;
- ☐ **b** A situação inicial de cada pilha é vazia;
- ☐ **c** Uma importante diferença entre um autômato com duas pilhas e uma máquina com duas pilhas é que, no autômato, o programa é especificado usando a noção de estados e transições e, na máquina, é especificado como um diagrama de fluxos.
- ☐ **d** O que reflete o estado corrente do autômato é a unidade de controle.
- ☐ **e** A função programa ou função de transição é uma função total.

exercício 7.20 Sobre autômato com duas pilhas $M = (\Sigma, Q, \Pi, q_0, F, V)$, marque a alternativa errada:

- ☐ **a** Σ indica o alfabeto de entrada;
- ☐ **b** Q indica o conjunto de estados possíveis;
- ☐ **c** Π indica a função de transição;
- ☐ **d** F indica que a palavra é falsa;
- ☐ **e** V indica o alfabeto auxiliar.

exercício 7.21 Sobre a família dos autômato com pilhas, analise as seguintes afirmações:

- I. Para cada autômato não determinístico, sempre há um autômato determinístico com duas pilhas equivalente.
- II. Pode-se simular um autômato finito por uma máquina de Turing e vice-versa.
- III. As estruturas de dados do tipo pilha podem ser usadas livremente para leitura, gravação e entrada de dados.
- IV. Pode ocorrer de um autômato, para uma dada palavra de entrada, nunca atingir uma condição de parada.

Marque a alternativa correta:

- ☐ **a** Apenas I e II estão corretas;
- ☐ **b** Apenas I, II e III estão corretas;
- ☐ **c** I é correta e III é errada;
- ☐ **d** III e IV são erradas;
- ☐ **e** I, II, III e IV estão corretas.

exercício 7.22 Relacionando a hierarquia das classes de máquinas com a hierarquia das classes de linguagens, numere a primeira coluna de acordo com a seguida:

() Linguagens recursivas	1. Máquinas determinísticas com uma pilha
() Linguagens regulares	2. Máquinas universais
() Linguagens livres do contexto	3. Máquinas determinísticas ou não determinísticas sem pilha
() Linguagens enumeráveis recursivamente	4. Máquinas universais – algoritmos que sempre param
() Linguagens livres do contexto determinísticas	5. Máquinas não determinísticas com uma pilha

A numeração correta, de cima para baixo, é:

- ☐ **a** 2 – 5 – 4 – 1 – 3
- ☐ **b** 4 – 5 – 1 – 2 – 3
- ☐ **c** 4 – 5 – 2 – 3 – 1
- ☐ **d** 4 – 3 – 5 – 2 – 1
- ☐ **e** 2 – 4 – 5 – 1 – 3

exercício 7.23 Sobre não determinismo, analise as seguintes afirmações:

- I. Uma máquina de Turing não determinística possui um poder computacional superior ao de uma máquina de Post determinística;

- II. Uma máquina de Turing não determinística possui poder computacional superior ao de uma máquina finita, sendo assim capaz de simulá-la;
- III. Está demonstrado que nenhuma outra forma de expressar algoritmos terá capacidade superior à da máquina de Turing.

Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas III está correta;
- ☐ **d** Apenas II e III estão corretas;
- ☐ **e** I, II e III estão corretas.

exercício 7.24 Sobre não determinismo, analise as seguintes afirmações:

- I. Somente máquinas com uma pilha não determinística têm a capacidade para reconhecer a linguagem:

$$\text{Palavra_Reversa} = \{ ww^r \mid w \text{ pertence a } \{ a, b \}^* \}$$

- II. O não determinismo pode modificar o poder computacional das máquinas universais estudadas;
- III. Uma palavra w pertencente a uma linguagem L é entrada em uma máquina M não determinística. Sabendo que, em um caminho alternativo, a palavra foi aceita, em outro, foi rejeitada, e em todos os demais, ficou em *loop*, então pode-se afirmar que w pertence a **LOOP(M)**.

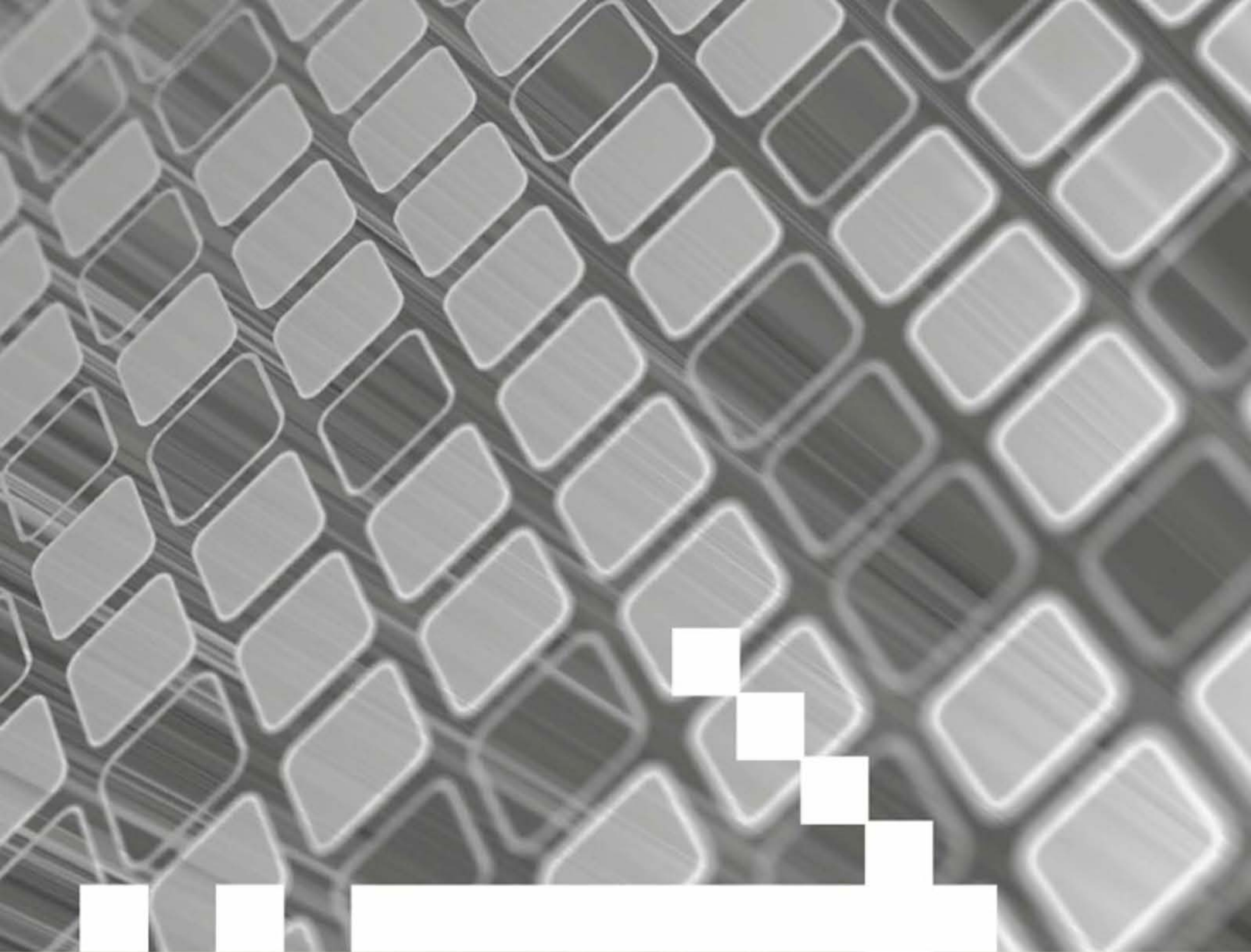
Marque a alternativa correta:

- ☐ **a** Apenas I está correta;
- ☐ **b** Apenas II está correta;
- ☐ **c** Apenas I e II estão corretas;
- ☐ **d** I, II e III estão corretas;
- ☐ **e** I, II e III estão erradas.

Termos-chaves

autômato com duas pilhas, p. 182, p. 197
autômato com pilhas, p. 196, p. 197
autômato com uma pilha não determinístico, p. 204
hierarquia de classes de linguagens, p. 203
hierarquia de classes de máquinas, p. 203
máquina com duas pilhas, p. 190

máquina com pilhas, p. 189, p. 191
máquina com uma pilha, p. 190
máquina de Post, p. 182, p. 184
máquina finita, p. 190
máquina universal, p. 182
não determinismo, p. 201



capítulo

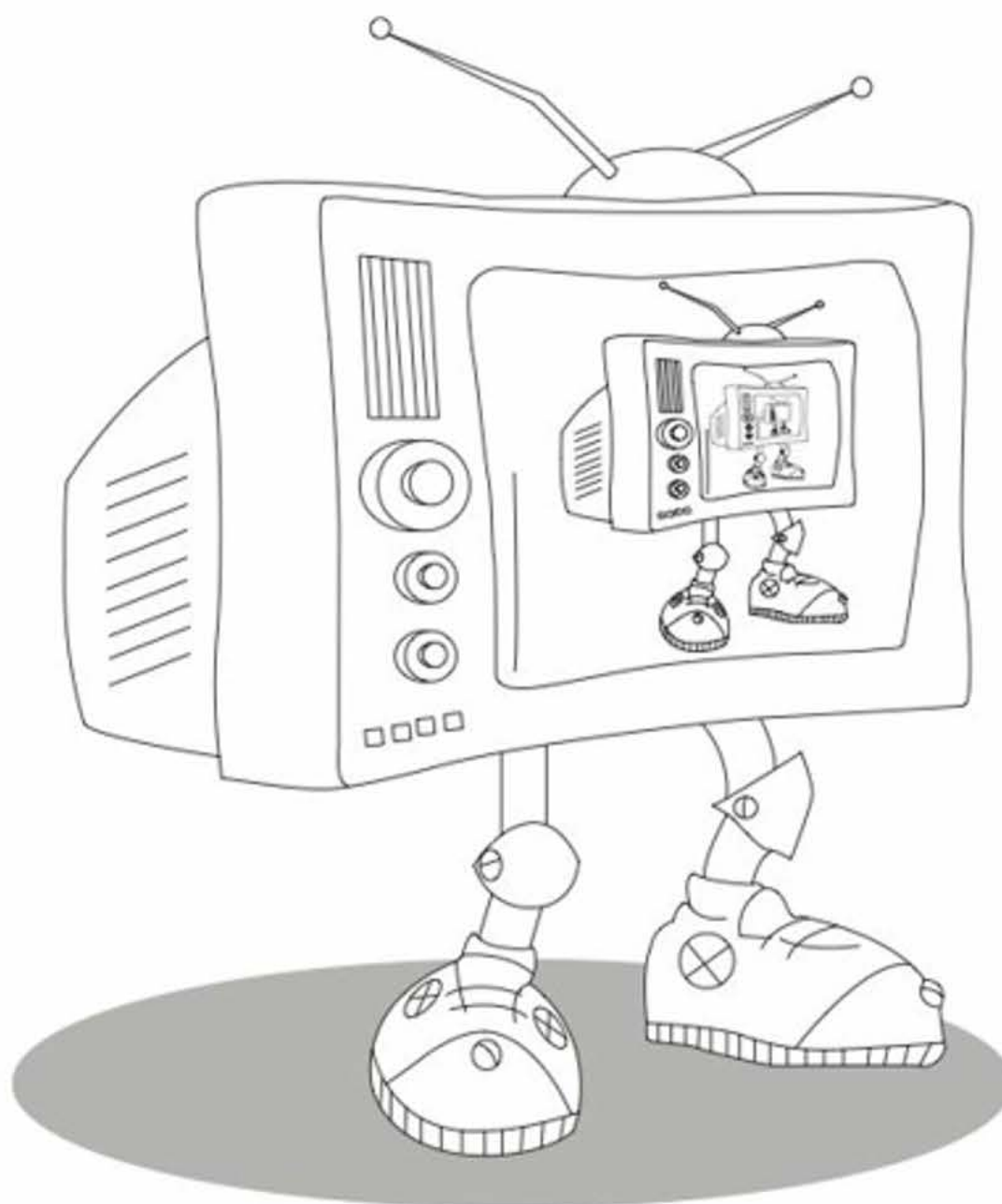


funções recursivas

■ ■ São estudados dois formalismos denotacionais (funcionais) do tipo recursivo (função definida em termos de si mesma), os quais são equivalentes ao formalismo máquina de Turing, reforçando a hipótese de Church. São eles:

funções recursivas parciais ou funções recursivas de Kleene, definidas sobre funções naturais zero, sucessor e projeção, bem como as operações substituição composicional, recursão e minimização, cálculo Lambda, formalismo definido sobre uma linguagem e um conjunto de axiomas e regras que permitem inferir quando dois termos denotam a mesma função.

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



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-se o que era verdadeiro antes da ocorrência;
- c** *Denotacional ou funcional*. Em geral, trata-se de uma função construída a partir de funções elementares de forma composicional, no sentido em que o algoritmo denotado pela função pode ser determinado em termos de suas funções componentes.

Até o momento, os formalismos estudados são do tipo operacional, como a máquina Norma, a máquina de Turing, a máquina de Post, etc.

Formalismos axiomáticos não fazem parte do objetivo desta publicação. Como curiosidade, possivelmente o formalismo axiomático mais usual para o estudo de computabilidade é a gramática (similar às gramáticas das linguagens naturais, por exemplo, português). 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, na qual as seguintes equivalências podem ser estabelecidas:

- Autômato finito $\Leftrightarrow$ gramáticas regulares;
- Autômato com uma pilha não determinístico $\Leftrightarrow$ gramáticas livres do contexto;
- Máquinas universais $\Leftrightarrow$ gramáticas irrestritas.

Neste capítulo, são estudados dois formalismos denotacionais (funcionais) do tipo *recursivo* (permite a definição de uma função em termos de si mesma) denominados:

- Funções recursivas parciais, ou funções recursivas de Kleene, introduzidas por S. C. Kleene (1936), as quais, como o próprio nome indica, são funções parciais definidas recursivamente;
- Cálculo Lambda (λ -Cálculo), introduzido por A. Church (1936), formalismo para definição de função, aplicação de função e recursão.

Assim como Turing, Kleene e Church tinham como objetivo formalizar a noção intuitiva de função computável. Nos dias atuais, isto significaria dizer que eles formalizaram o que é possível computar em um computador. Quando foi provado que as classes das funções Turing-computáveis, das funções recursivas parciais e das funções Lambda-computáveis são iguais, a tese de Church cresceu significativamente em termos de credibilidade, fortalecendo-a como hipótese para toda a teoria da computação clássica.

8.1

→ funções recursivas de Kleene

As funções recursivas parciais, ou funções recursivas de Kleene, são funções construídas sobre três funções naturais básicas:

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

juntamente com as seguintes operações:

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

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

8.1.1 substituição composicional

A substituição composicional definida a seguir generaliza o conceito usual de composição de funções.

definição 8.1 – Substituição composicional

Sejam as funções parciais:

$$g: \mathbb{N}^k \rightarrow \mathbb{N} \quad \text{e} \quad f_1, f_2, \dots, f_k: \mathbb{N}^n \rightarrow \mathbb{N}$$

A função parcial:

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

é a *substituição composicional de funções*, ou simplesmente *substituição de funções*, definida a partir de $g, f_1, f_2, \dots, f_k$ como segue:

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

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

$f_i(x_1, x_2, \dots, x_n)$ é definida para todo $i \in \{1, 2, \dots, k\}$

$g(f_1(x_1, x_2, \dots, x_n), f_2(x_1, x_2, \dots, x_n), \dots, f_k(x_1, x_2, \dots, x_n))$ é definida. □

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

exemplo 8.1 – Substituição de funções – funções constantes

Suponha as seguintes funções:

$f_{\text{zero}}: \mathbb{N} \rightarrow \mathbb{N}$ tal que, $\forall x \in \mathbb{N}, f_{\text{zero}}(x) = 0$

função constante zero

$\text{sucessor}: \mathbb{N} \rightarrow \mathbb{N}$ tal que, $\forall x \in \mathbb{N}, \text{sucessor}(x) = x + 1$

função sucessor

$\text{adição}: \mathbb{N}^2 \rightarrow \mathbb{N}$ tal que, $\forall x, y \in \mathbb{N}, \text{adição}(x, y) = x + y$

função adição

As seguintes funções são definidas, usando substituição de funções:

$f_{\text{um}} = \text{sucessor}(f_{\text{zero}}): \mathbb{N} \rightarrow \mathbb{N}$

função constante um

$f_{\text{dois}} = \text{sucessor}(f_{\text{um}}): \mathbb{N} \rightarrow \mathbb{N}$

função constante dois

$f_{\text{três}} = \text{adição}(f_{\text{um}}, f_{\text{dois}}): \mathbb{N} \rightarrow \mathbb{N}$

função constante três

Sugere-se, como exercício, verificar se, usando as mesmas funções f_{zero} , sucessor e adição , existem outras formas de definir equivalentemente as funções f_{um} , f_{dois} e $f_{\text{três}}$. □

exemplo 8.2 – Substituição de funções – média de uma turma de alunos

Para visualizar melhor a ideia da substituição em um contexto aplicado, suponha uma turma com k alunos na qual foram realizadas n provas, sendo a nota final de cada aluno $i \in \{1, 2, \dots, k\}$ dada pela função $\text{nota}_i(p_1, p_2, \dots, p_n)$. Assim, o cálculo da média de toda a turma, considerando as provas, é dado pela função turma , a qual é a **média** das notas dos alunos, ou seja:

$$\text{turma}(p_1, p_2, \dots, p_n) = \text{média}(\text{nota}_1(p_1, p_2, \dots, p_n), \text{nota}_2(p_1, p_2, \dots, p_n), \dots, \text{nota}_k(p_1, p_2, \dots, p_n)) \quad \square$$

8.1.2 recursão primitiva

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 8.2 – Recursão primitiva

Sejam $f: \mathbb{N}^n \rightarrow \mathbb{N}$ e $g: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ funções parciais. A função parcial:

$$h: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$$

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

$$\begin{aligned} h(x_1, x_2, \dots, x_n, 0) &= f(x_1, x_2, \dots, x_n) \\ h(x_1, x_2, \dots, x_n, y + 1) &= g(x_1, x_2, \dots, x_n, y, h(x_1, x_2, \dots, x_n, y)) \end{aligned}$$

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

$$\begin{aligned} f(x_1, x_2, \dots, x_n) &\text{ é definida; } \\ g(x_1, x_2, \dots, x_n, i, h(x_1, x_2, \dots, x_n, i)) &\text{ é definida para todo } i \in \{1, 2, \dots, y\} \end{aligned} \quad \square$$

exemplo 8.3 – Recursão primitiva – adição

Suponha as seguintes funções:

$$\begin{array}{ll} \text{id}: \mathbb{N} \rightarrow \mathbb{N} & \text{função identidade} \\ \text{sucessor}: \mathbb{N} \rightarrow \mathbb{N} & \text{função sucessor} \\ \text{proj}_{3_3}: \mathbb{N}^3 \rightarrow \mathbb{N} & \text{função projeção da 3ª componente de uma tripla} \end{array}$$

A função adição nos naturais:

$$\text{adição}: \mathbb{N}^2 \rightarrow \mathbb{N} \text{ tal que, } \forall x, y \in \mathbb{N}, \text{adição}(x, y) = x + y$$

é definida, usando recursão primitiva, como segue:

$$\begin{aligned} \text{adição}(x, 0) &= \text{id}(x) \\ \text{adição}(x, y + 1) &= \text{proj}_{3_3}(x, y, \text{sucessor}(\text{adição}(x, y))) \end{aligned}$$

Por exemplo, $\text{adição}(3, 2)$ é como abaixo. Note-se que, nas instâncias de $\text{proj}_{3_3}(x, y, \text{sucessor}(\text{adição}(x, y)))$, somente a componente $\text{sucessor}(\text{adição}(x, y))$ é

importante na lógica apresentada. A função proj_{3_3} , bem como as componentes x e y , estão presentes somente para satisfazer a definição de recursão primitiva. De fato, funções do tipo projeção são fundamentais em recursão primitiva, como será visto adiante.

```

adição(3, 2) =
proj33(3, 1, sucessor(adição(3, 1))) =
proj33(3, 1, sucessor(proj33(3, 0, sucessor(adição(3, 0))))) =
proj33(3, 1, sucessor(proj33(3, 0, sucessor(id( 3 ))))) =
proj33(3, 1, sucessor(proj33(3, 0, sucessor( 3 )))) =
proj33(3, 1, sucessor(proj33(3, 0, 4))) =
proj33(3, 1, sucessor( 4 )) =
proj33(3, 1, 5 ) =
5

```

□

8.1.3 minimização

O conceito de minimização que segue não é intuitivo na noção de funções recursivas parciais. Entretanto, é fundamental para garantir que a classe de funções recursivas parciais definida adiante possa conter qualquer função computável.

definição 8.3 – Minimização

Seja $f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ uma função parcial. A função parcial:

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

é dita definida por *minimização* de f e é tal que, $\forall (x_1, x_2, \dots, x_n, y) \in \mathbb{N}^{n+1}$:

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

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

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

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

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):

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

observação 8.4 – Valores constantes como funções

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

A técnica usada para representar valores constantes como funções se baseia no fato de que o número de funções que existem a partir de um conjunto unitário 1 (com um único elemento) para outro conjunto A , é o cardinal de A . Portanto, existe uma função para denotar cada elemento de A . Assim, por exemplo, para os conjuntos $1 = \{ * \}$ e **Booleano** = $\{ V, F \}$, existem as seguintes funções:

$\text{const}_V: 1 \rightarrow \text{Booleano}$ tal que $\text{const}_V(*) = V$	denota a constante V
$\text{const}_F: 1 \rightarrow \text{Booleano}$ tal que $\text{const}_F(*) = F$	denota a constante F

Seguindo o princípio denotacional, para a função **negação**: **Booleano** $\rightarrow$ **Booleano, tem-se que:**

$\text{const}_F = \text{negação} \circ \text{const}_V$
 $\text{const}_V = \text{negação} \circ \text{const}_F$

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

$\text{const}_F: \rightarrow \text{Booleano}$ denota a constante F

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

exemplo 8.4 – Minimização – função número zero

Suponha a função constante $f_{\text{zero}}: \mathbb{N} \rightarrow \mathbb{N}$ (sendo que $f_{\text{zero}}(x) = 0$). Seja $\text{const}_{\text{zero}}$ o natural 0 visto como função (não confundir com a função constante f_{zero}):

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

A função número $\text{const}_{\text{zero}}$ pode ser definida, usando minimização, como segue:

$\text{const}_{\text{zero}} = \min\{ y \mid f_{\text{zero}}(y) = 0 \}$

De fato, o menor natural y tal que $f_{\text{zero}}(y) = 0$ é 0 . $\square$

exemplo 8.5 – Minimização, recursão primitiva – antecessor

Suponha a função número $\text{const}_{\text{zero}}$ (exemplo 8.4 - Minimização – função número zero) bem como a seguinte função de projeção:

$\text{proj}_{2_1}: \mathbb{N}^2 \rightarrow \mathbb{N}$ projeção da 1ª componente de uma dupla: $\text{proj}_{2_1}(x, y) = x$

A seguinte função antecessor nos naturais:

antecessor: $\mathbb{N} \rightarrow \mathbb{N}$ antecessor

pode ser definida, usando recursão primitiva (supondo que antecessor de 0 é 0), como segue:

$$\begin{aligned}\text{antecessor}(0) &= \text{const}_{\text{zero}} \\ \text{antecessor}(y + 1) &= \text{proj}_{2,1}(y, \text{antecessor}(y))\end{aligned}$$

Por exemplo, $\text{antecessor}(2)$ é como segue:

$$\begin{aligned}\text{antecessor}(2) &= & (1) \\ \text{proj}_{2,1}(1, \text{antecessor}(1)) &= \\ \text{proj}_{2,1}(1, \text{proj}_{2,1}(0, \text{antecessor}(0))) &= \\ \text{proj}_{2,1}(1, \text{proj}_{2,1}(0, \text{const}_{\text{zero}})) &= \\ \text{proj}_{2,1}(1, \text{proj}_{2,1}(0, 0)) &= \\ \text{proj}_{2,1}(1, 0) &= \\ 1\end{aligned}$$

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

exemplo 8.6 – Minimização, recursão primitiva – subtração

Suponha a função número $\text{const}_{\text{zero}}$ (exemplo 8.4 - Minimização – função número zero), a função antecessor (exemplo 8.5 - Minimização, recursão primitiva – antecessor), bem como as seguintes funções:

$$\begin{aligned}\text{id}: \mathbb{N} &\rightarrow \mathbb{N} & \text{identidade: } \text{id}(x) &= x \\ \text{proj}_{3,3}: \mathbb{N}^3 &\rightarrow \mathbb{N} & \text{projeção da 3ª componente de uma tripla: } \text{proj}_{3,3}(x, y, z) &= z\end{aligned}$$

A seguinte função subtração nos naturais:

$$\text{sub}: \mathbb{N}^2 \rightarrow \mathbb{N} \quad \text{subtração}$$

pode ser definida usando recursão primitiva:

$$\begin{aligned}\text{sub}(x, 0) &= \text{id}(x) \\ \text{sub}(x, y + 1) &= \text{antecessor} \circ \text{proj}_{3,3}(x, y, \text{sub}(x, y))\end{aligned}$$

Por exemplo, $\text{sub}(3, 2)$ é como segue:

$$\begin{aligned}\text{sub}(3, 2) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, \text{sub}(3, 1)) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, \text{antecessor} \circ \text{proj}_{3,3}(3, 0, \text{sub}(3, 0))) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, \text{antecessor} \circ \text{proj}_{3,3}(3, 0, \text{id}(3))) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, \text{antecessor} \circ \text{proj}_{3,3}(3, 0, 3)) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, \text{antecessor}(3)) &= \\ \text{antecessor} \circ \text{proj}_{3,3}(3, 1, 2) &= \\ \text{antecessor}(2) &= \\ 1\end{aligned}$$

Novamente, foi necessário usar uma função de projeção. Sugere-se como exercício determinar o valor de $\text{sub}(2, 3)$. Neste caso, o uso da função número $\text{const}_{\text{zero}}$ se faz necessário. $\square$

8.1.4 função recursiva parcial e total

Funções recursivas parciais são definidas a partir de três funções básicas sobre o conjunto dos números naturais, como segue:

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

É interessante observar que, somente com essas três funções, bem como com as construções de substituição composicional, recursão primitiva e minimização, é possível definir qualquer função intuitivamente computável.

definição 8.5 – Função recursiva parcial

Uma *função recursiva parcial*, ou *função recursiva de Kleene*, é indutivamente definida como segue:

- a** *Funções básicas.* As seguintes funções são recursivas parciais:

$$\text{const}_{\text{zero}}: 1 \rightarrow \mathbb{N}$$

$$\text{sucessor}: \mathbb{N} \rightarrow \mathbb{N}$$

$$\text{proj}_{n_i}: \mathbb{N}^n \rightarrow \mathbb{N}$$

$$\text{const}_{\text{zero}}(*) = 0$$

$$\text{sucessor}(x) = x + 1$$

$$\text{proj}_{n_i}(x_1, x_2, \dots, x_n) = x_i$$

- b** *Substituição composicional de funções.* Se $g: \mathbb{N}^k \rightarrow \mathbb{N}$ e $f_1, f_2, \dots, f_k: \mathbb{N}^n \rightarrow \mathbb{N}$ são funções recursivas parciais, então a seguinte função é recursiva parcial:

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

e é definida pela *substituição composicional de funções* a partir de $g, f_1, f_2, \dots, f_k$ como segue:

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

- c** *Recursão primitiva.* Se $f: \mathbb{N}^n \rightarrow \mathbb{N}$ e $g: \mathbb{N}^{n+2} \rightarrow \mathbb{N}$ são funções recursivas parciais, então a seguinte função é recursiva parcial:

$$h: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$$

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

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

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

- d** *Minimização.* Se $f: \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ é função recursiva parcial, então a seguinte função é recursiva parcial:

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

e é definida por *minimização* de f , como segue:

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

exemplo 8.7 – Funções recursivas parciais

- a** *Função identidade.* A função identidade:

$$\text{id}: \mathbb{N} \rightarrow \mathbb{N} \quad \text{id}(x) = x$$

é recursiva parcial, pois é uma função básica de projeção, ou seja:

$$\text{id} = \text{proj}_{1,1} \quad \text{projeta a primeira (e única) componente de uma 1-upla}$$

- b** As seguintes funções, exemplificadas anteriormente, são recursivas parciais:

adição: $\mathbb{N}^2 \rightarrow \mathbb{N}$	adição
sub: $\mathbb{N}^2 \rightarrow \mathbb{N}$	subtração
f_{zero}: $\mathbb{N} \rightarrow \mathbb{N}$	constante zero
f_{um}: $\mathbb{N} \rightarrow \mathbb{N}$	constante um
f_{dois}: $\mathbb{N} \rightarrow \mathbb{N}$	constante dois
$f_{\text{três}}$: $\mathbb{N} \rightarrow \mathbb{N}$	constante três
$\text{const}_{\text{zero}}$: $\{*\} \rightarrow \mathbb{N}$	número zero
antecessor: $\mathbb{N} \rightarrow \mathbb{N}$	antecessor

□

Uma função recursiva total, como o próprio nome indica, nada mais é que uma função recursiva parcial que é total.

definição 8.6 – 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 8.1 – Substituição composicional
- definição 8.2 – Recursão primitiva
- definição 8.3 – Minimização

verificando as condições para que a função resultante dessas definições seja total. Sugere-se, como exercício, verificar quais das funções do exemplo 8.7 – Funções recursivas parciais são totais.

O seguinte teorema (a demonstração é omitida) é mais um reforço para a hipótese de Church.

teorema 8.7 – Funções recursivas x 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. □

Consequentemente, a seguinte relação entre classes também pode ser estabelecida:

- Funções recursivas parciais e linguagens enumeráveis recursivamente;
- Funções recursivas totais e linguagens recursivas.

8.2

→ cálculo Lambda

A importância do cálculo Lambda transcende o estudo da computabilidade, tendo importantes aplicações no estudo da lógica e das linguagens de programação. Em particular, foi inspiração de muitas *linguagens de programação funcionais* como, por exemplo, *Lisp* e *Haskell*. No que segue, inicialmente é feita uma apresentação geral do cálculo Lambda (não tipado) o qual é usado como motivação para as construções formais.

8.2.1 aspectos gerais do cálculo Lambda

Considere a função $f: \mathbb{N} \rightarrow \mathbb{N}$ tal que:

$$f(x) = x + 1$$

É importante reparar que o que efetivamente está sendo definido não é a função f propriamente dita, mas sim $f(x)$, ou seja, o resultado da aplicação de f ao argumento x (o qual é suposto assumir valores nos números naturais). Claramente $f(x) = x + 1$ não é uma função, mas uma equação (pois $x + 1 - f(x) = 0$).

Uma forma de definir funções com mais rigor é usando a *linguagem Lambda*. 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* (elementos que definem ou representam funções na *linguagem Lambda*):

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

que pode ser interpretado como segue:

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

- b** *Aplicação Lambda*. Determina o valor da função aplicada a um dado argumento. Por exemplo, a função f acima aplicada ao argumento 3 é denotada pelo seguinte termo Lambda:

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

que pode ser interpretado como segue:

aplicação da função $\lambda x.(x + 1)$ ao valor 3

Assim, a notação Lambda permite representar e diferenciar a regra de associação (função) da aplicação da mesma a um argumento. Também, a notação deixa claro qual variável é o dado de entrada. Como primeiro exemplo, foi considerada a função sucessor $\lambda x.(x + 1)$, a qual faz a associação da regra $x \mapsto x + 1$.

No caso de funções de mais de uma variável, como a adição de números, pode parecer necessário definir uma abstração Lambda para cada variável. Entretanto, uma função de duas variáveis pode ser vista como uma função de uma variável quando se fixa um dos parâmetros. Por exemplo, para a função de adição, fixando o segundo argumento no valor 1, obtém-se a função sucessor $\lambda x.(x + 1)$. O mesmo raciocínio permite construir as funções correspondentes às regras $x \mapsto x + 2$, $x \mapsto x + 3$, ... Generalizando este raciocínio, a adição pode ser representada pelo seguinte termo Lambda:

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

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

Em resumo, o cálculo Lambda é a linguagem Lambda juntamente com um conjunto de axiomas e regras que permitem inferir quando dois termos Lambda denotam a mesma função.

8.2.2 linguagem Lambda

Uma *linguagem Lambda* é o conjunto dos termos Lambdas sobre um conjunto de variáveis.

definição 8.8 – Termo Lambda

Seja V um conjunto infinito e enumerável de variáveis. Então um *termo Lambda*, λ -*termo*, *expressão Lambda* ou *palavra Lambda* sobre V é indutivamente definido como segue:

- i) *Variável*. $x \in V$ é um λ -termo;
- ii) *Abstração Lambda* ou λ -*abstração*. Se M é um λ -termo e $x \in V$, então também é um λ -termo:

$$(\lambda x.M)$$

- iii) *Aplicação Lambda* ou λ -*aplicação*. Se M e N são λ -termos, então também são λ -termos:

$$(M N)$$

□

Intuitivamente, na definição de λ -termo, tem-se que:

- abstração Lambda representa a operação de atribuir caráter funcional a um objeto. Assim, $(\lambda x.M)$ representa uma função que possui um parâmetro de entrada;
- aplicação Lambda $(M N)$ representa a operação de aplicação da função M a um objeto (uma entrada) N .

Em relação ao uso de parênteses, as seguintes simplificações de notação são usuais:

- a** *Parênteses externos.* Parênteses externos podem ser eliminados. Por exemplo, para os λ -termos M e N :

$M N$ é uma notação simplificada de $(M N)$

- b** *Associatividade à esquerda.* Parênteses podem ser eliminados, respeitando a propriedade associativa à esquerda, ou seja, para os λ -termos M , N e P , tem-se que:

$M N P$ é uma notação simplificada de $(M N) P$

- c** *Escopo de uma variável.* Parênteses podem ser eliminados, respeitando o escopo de uma variável em uma abstração, ou seja, para os λ -termos M , N e P e para a variável x , tem-se que:

$\lambda x.M N P$ é uma notação simplificada de $\lambda x.(M N P)$

Observe que:

- no cálculo Lambda dito puro, como definido neste livro, não existem constantes. As constantes usadas são introduzidas por razões puramente didáticas;
- os λ -termos são anônimos, ou seja, não são explicitamente nomeados.

exemplo 8.8 – Termos Lambda

Sejam x , y e z variáveis. As seguintes expressões são termos Lambda (na coluna da direita, são apresentados os casos aplicados, conforme a definição 8.8 – Termo Lambda):

x	i
$(x y)$	i, iii
$\lambda x.x$	i, ii
$\lambda x.y$	i, ii
$(\lambda x.x \lambda x.x)$	i, ii, iii
$(\lambda z.z (x y))$	i, ii, iii
	□

definição 8.9 – Linguagem Lambda

Seja V um conjunto infinito e enumerável de variáveis. Então uma *linguagem Lambda* ou λ -linguagem sobre V é o conjunto de todos os λ -termos sobre esse conjunto. □

8.2.3 variável livre e substituição

Detalhando a noção de aplicação, considere, por exemplo, o termo $\lambda x.x$, o qual representa uma função com entrada x resultando também em x . Portanto, é uma regra que determina a função identidade. Nesse caso, parece razoável que a aplicação de $\lambda x.x$ a algum objeto M tenha como resultado o próprio M . Isto é:

$$(\lambda x.x) M \text{ resulta em } M$$

o que é o mesmo que dizer que x foi substituído por M em x .

Considere outro caso: o que significa o objeto x (uma variável)? Naturalmente, não possui funcionalidade aparente. Pode-se construir $x y$, o que poderia significar a aplicação de x ao objeto (à entrada) y . Nesse caso, x deveria ser uma função, o que só pode acontecer para alguma interpretação de x , ou seja, lançando mão de um artifício semântico. Assim, não parece ser razoável entender que $x y$ resulta em outro termo, como fazendo “contas” como em uma expressão aritmética ou como feito na introdução acima.

Uma vez que o resultado de uma aplicação é calculado por meio de um processo de substituição, é introduzida uma notação para representar esse processo. Antes, é necessário caracterizar melhor a ocorrência de uma variável em um termo Lambda. Observe que as definições que seguem cobrem cada possibilidade de λ -termo na definição 8.8 – Termo Lambda.

definição 8.10 – Variável livre

Sejam x e y variáveis tal que $x \neq y$ e N, P λ -termos. A variável x é dita *variável livre* em um λ -termo, nos seguintes casos:

- i) *Variável*. x é livre em y . Em particular, x é livre em x
- ii) *Abstração Lambda*. x é livre em $\lambda y.N$
- iii) *Aplicação Lambda*. Se x é livre em N e em P , então x é livre em $N P$ □

definição 8.11 – Variável ligada

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

Portanto, uma variável é ligada se ocorre no escopo de um λ , ou seja, é “local” ao termo, enquanto a variável livre é “global” ao termo. Observando atentamente a definição, pode-se concluir que uma variável pode ocorrer ligada em uma parte do termo e livre em outra, como exemplificado a seguir.

exemplo 8.9 – Variável ligada, variável livre

a Para o termo $\lambda x.(x y)$:

x é variável ligada;
 y é variável livre;

b Para o termo $\lambda x.\lambda y.(x y)$:

x e y são variáveis ligadas;

c Para o termo $x \lambda x. \lambda y. (x y)$:

a primeira ocorrência da variável x é livre;
a segunda ocorrência da variável x é ligada;

d Para o termo $(\lambda x. ax) x$:

no subtermo ax , as variáveis a e x são livres
no subtermo $\lambda x. ax$, a variável a é livre e a variável x é ligada
no termo $(\lambda x. ax) x$, a variável x é ambos, ligada (primeira ocorrência) e livre (segunda ocorrência) $\square$

É interessante observar que a escolha de uma variável ligada em uma abstração Lambda não é importante, o que induz uma relação de *equivalência alfabética*. Por exemplo, os termos Lambda:

$\lambda x. x$ e $\lambda y. y$ são equivalentes alfabeticamente

pois representam a mesma função identidade. Entretanto, os termos Lambda:

x e y não são equivalentes alfabeticamente

pois as variáveis não estão ligadas. É usual identificar ("igualar") os termos equivalentes alfabeticamente.

definição 8.12 – Substituição

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

$$M [x \leftarrow X]$$

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

- i) *Variável.*
 - i.1) $x [x \leftarrow X] = X$
 - i.2) $y [x \leftarrow X] = y$
- ii) *Aplicação Lambda.* Suponha M_1, M_2 λ -termos. Então:

$$(M_1 M_2) [x \leftarrow X] = (M_1 [x \leftarrow X]) (M_2 [x \leftarrow X])$$
- iii) *Abstração Lambda.*
 - iii.1) $\lambda x. M [x \leftarrow X] = \lambda x. M$
 - iii.2) $(\lambda y. M) [x \leftarrow X] = \lambda y. (M [x \leftarrow X])$ se y não ocorre livre no subtermo X ou x não ocorre livre no subtermo M
 - iii.3) $(\lambda y. M) [x \leftarrow X] = \lambda z. (M [y \leftarrow z]) [x \leftarrow X]$ se y ocorre livre no subtermo X e x ocorre livre no subtermo M ; nesse caso, z não ocorre livre nos subtermos M e X $\square$

O item iii.3) na definição acima evita a realização de substituições que mudem a característica livre ou ligada de uma variável no termo. Por exemplo, se o item iii.2) for aplicado no termo $(\lambda u.v) [v \leftarrow u]$ resulta em $\lambda u.u$ fazendo com que uma substituição supostamente livre resulte em ligada. Por esta razão, a variável z é introduzida, resultando em $\lambda z.u$, ou seja:

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

exemplo 8.10 – Substituição

Para facilitar o entendimento, para cada termo são destacados os subtermos envolvidos na substituição realizada e, na coluna da direita, é explicitado o caso (item) usado, conforme a definição 8.12 - Substituição:

a	$x [x \leftarrow z] = z$	i.1
b	$y [x \leftarrow z] = y$	i.2
c	$(x y) [x \leftarrow z] =$ $(x [x \leftarrow z]) (y [x \leftarrow z]) =$ $z (y [x \leftarrow z]) =$ $z y$	ii i.1 i.2
d	$\lambda x.(x y) [x \leftarrow z] = \lambda x.(x y)$	iii.1
e	$(\lambda u.v) [v \leftarrow u] =$ $\lambda z. ((v [u \leftarrow z]) [v \leftarrow u]) =$ $\lambda z. (v [v \leftarrow u]) =$ $\lambda z.u$	iii.3 (u ocorre livre em u e v ocorre livre em v) i.2 i.1
f	$((\lambda p.p (p q)) (\lambda r.(p r))) [q \leftarrow p a] =$ $(\lambda p.p (p q)) [q \leftarrow p a] (\lambda r.(p r)) [q \leftarrow p a] =$ $(\lambda p.p (p q)) [q \leftarrow p a] \lambda r. ((p r) [q \leftarrow p a]) =$ $(\lambda p.p (p q)) [q \leftarrow p a] \lambda r. (p [q \leftarrow p a] r [q \leftarrow p a]) =$ $(\lambda p.p (p q)) [q \leftarrow p a] \lambda r. (p r [q \leftarrow p a]) =$ $(\lambda p.p (p q)) [q \leftarrow p a] \lambda r.(p r) =$ $\lambda z. ((p (p q) [p \leftarrow z]) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (p [p \leftarrow z] ((p q) [p \leftarrow z])) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (z ((p q) [p \leftarrow z])) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (z (p [p \leftarrow z] q [p \leftarrow z]) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (z (z q [p \leftarrow z]) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (z (z q) [q \leftarrow p a]) \lambda r.(p r) =$ $\lambda z. (z (z [q \leftarrow p a] q [q \leftarrow p a])) \lambda r.(p r) =$ $\lambda z. (z (z q [q \leftarrow p a])) \lambda r.(p r) =$ $\lambda z. (z (z p a)) \lambda r.(p r)$	ii iii.2 (r não ocorre livre em $p a$ e q não ocorre livre em $p r$) ii i.2 i.2 iii.3 (p ocorre livre em $p a$ e q ocorre livre em $p q$) ii i.1 ii i.1 i.2 ii i.2 i.1 □

8.2.4 cálculo Lambda

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

definição 8.13 – Cálculo Lambda

O *cálculo Lambda*, ou simplesmente λ -*cálculo*, consiste na linguagem Lambda munida dos axiomas e as regras de inferência abaixo. Sejam M , N e K λ -termos:

- i) Principal axioma:

$$(\lambda x.M) N = M [x \leftarrow N]$$

- ii) Axiomas e regras lógicas:

- ii.1) Igualdade

$$M = M$$

(reflexiva)

$$\text{se } M = N, \text{ então } N = M$$

(simétrica)

$$\text{se } M = N \text{ e } N = K, \text{ então } M = K$$

(transitiva)

- ii.2) Regras de compatibilidade

$$\text{se } M = N, \text{ então } K N = K M$$

(distributiva à esquerda)

$$\text{se } M = N, \text{ então } N K = M K$$

(distributiva à direita)

$$\text{se } M = N, \text{ então } \lambda x.N = \lambda x.M$$

- iii) Os termos M e N são *iguais no cálculo Lambda*, ou seja, $M = N$ se, e somente se, existe uma dedução de $M = N$ no λ -cálculo. $\square$

Portanto, o principal axioma formaliza que o resultado de uma aplicação é calculado via substituição.

É importante notar que, a partir da notação Lambda, origina-se uma linguagem e um cálculo. O cálculo objetiva verificar a igualdade de termos da linguagem. A noção de cálculo é explorada pelo conceito de *redução*, o qual pode ser interpretado como um “passo computacional” na busca de um “valor” representativo para um λ -termo qualquer. Portanto, a redução pode ser vista como uma operacionalização do λ -cálculo. As reduções são:

- *Redução alfa*, renomeação de variáveis ligadas;
- *Redução beta*, aplicação de uma função a um argumento, via substituição;
- *Redução iterada*, sucessiva aplicação de qualquer das reduções acima.

definição 8.14 – Redução beta

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

$$(\lambda x.M)N \triangleright M[x \leftarrow N]$$

β -redução

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

definição 8.15 – Redução alfa

Uma *redução alfa*, ou simplesmente α -*redução*, é a seguinte transformação entre λ -termos:

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

onde y não ocorre livre no subtermo M . $\square$

A α -*redução* formaliza a relação de *equivalência alfabética*, igualmente denotada pelo símbolo $\triangleright$. Assim, tanto o termo “redução” como o símbolo $\triangleright$ são usados para indicar α -redução ou β -redução.

definição 8.16 – Redução iterada

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

Com a finalidade de não sobrecarregar a notação, é usual adotar simplesmente o símbolo $\triangleright$ para denotar o processo de redução tanto em um passo quanto iterada.

exemplo 8.11 – Redução

Para cada item que segue, sugere-se como exercício detalhar as substituições conforme a definição 8.12 – Substituição:

- a** $(\lambda x.2 * x + 1) 3 \triangleright$
 $(2 * x + 1) [x \leftarrow 3] = 2 * 3 + 1 = 7$
- b** $(\lambda x.\lambda y.x + y) 5 \triangleright$
 $(\lambda y.x + y) [x \leftarrow 5] = \lambda y.5 + y$
- c** $((\lambda x.\lambda y.x + y) 5) 7 \triangleright$
 $((\lambda y.x + y) [x \leftarrow 5]) 7 = (\lambda y.5 + y) 7 \triangleright$
 $(5 + y)[y \leftarrow 7] = 5 + 7 = 12$
- d** $(\lambda x.\lambda y.x - y) (\lambda z.z / 2) \triangleright$
 $(\lambda y.x - y) [x \leftarrow \lambda z.z / 2] = \lambda y.(\lambda z.z / 2) - y$
- e** $(\lambda x.x x) (\lambda y.y) \triangleright$
 $x x [x \leftarrow \lambda y.y] = (\lambda y.y) (\lambda y.y) \triangleright$
 $y [y \leftarrow \lambda y.y] = \lambda y.y$
- f** $((\lambda x.\lambda y.x y) (\lambda y.z)) x \triangleright$
 $(\lambda y.x y [x \leftarrow \lambda y.z]) x = (\lambda y.(\lambda y.z) y) x \triangleright$
 $(\lambda y.z) y [y \leftarrow x] = (\lambda y.z) x \triangleright$
 $z [y \leftarrow x] = z$
- g** $(\lambda x.x x) (\lambda x.x x) \triangleright$
 $x x [x \leftarrow \lambda x.x x] = (\lambda x.x x) (\lambda x.x x)$
- h** $((\lambda x.\lambda y.\lambda z.(x y) z) (\lambda x.\lambda y.(x + y)) 3) 7 \triangleright$
 $((\lambda y.\lambda z.(x y) z) [x \leftarrow \lambda x.\lambda y.(x + y)] 3) 7 = ((\lambda y.\lambda z.(\lambda x.\lambda y.(x + y) y) z) 3) 7 \triangleright$
 $((\lambda z.(\lambda x.\lambda y.(x + y) y) z) [y \leftarrow 3]) 7 = ((\lambda z.(\lambda x.\lambda y.(x + y) 3) z) 7) \triangleright$
 $(\lambda x.\lambda y.(x + y) 3) z [z \leftarrow 7] = (\lambda x.\lambda y.(x + y) 3) 7 \triangleright$
 $(\lambda y.(x + y) [x \leftarrow 3]) 7 = (\lambda y.(3 + y)) 7 \triangleright$
 $(3 + y) [y \leftarrow 7] = 3 + 7 = 10$

Observe que, nos primeiros quatro itens, bem como no último, foi feito o uso “informal” de operações matemáticas e de constantes que não pertencem propriamente ao λ -cálculo. $\square$

8.2.5 tipos de dados básicos

A seguir, é exemplificado como o cálculo Lambda pode ser usado para modelar tipos de dados. Como os termos no cálculo Lambda são anônimos, no estudo que segue, para facilitar o entendimento e simplificar a notação, o símbolo “:=” é introduzido para nomear λ -termos. Por exemplo:

identidade := $\lambda x.x$

Tipo booleano

Os valores possíveis para os elementos do tipo de dado *booleano* ou *lógico* são **verdadeiro** e **falso**. Ambos valores são definidos como funções (λ -termos) e são ilustrados a seguir (modelagem conhecida como *booleanos de Church*).

exemplo 8.12 – Valores lógicos

Os valores lógicos são definidos como segue:

verdadeiro := $\lambda x.\lambda y.x$ e **falso** := $\lambda x.\lambda y.y$

Portanto:

- **verdadeiro** seleciona o primeiro entre dois termos da entrada, ou seja, a projeção da primeira componente;
- **falso** seleciona o segundo entre dois termos da entrada, ou seja, a projeção da segunda componente. $\square$

Assim, as operações booleanas são definidas usando λ -termos que têm como entrada um dos valores lógicos como exemplificados acima.

exemplo 8.13 – Negação

A operação de negação lógica $\neg$ pode ser definida invertendo-se as opções de escolha:

$\neg$:= $\lambda p.p \text{ falso verdadeiro}$

Por exemplo, o cálculo da negação lógica $\neg$ para o argumento **falso** é como segue:

$(\lambda p.p \text{ falso verdadeiro}) \text{ falso} \triangleright$
falso falso verdadeiro $\triangleright$
verdadeiro

falso: projeção da segunda componente $\square$

exemplo 8.14 – Conjunção lógica

A operação de conjunção lógica $\wedge$ definida da seguinte forma:

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

pode ser representada pelo λ -termo:

$$\wedge := \lambda p. \lambda q. p \ q \text{ falso}$$

então, se p for substituído pelo valor lógico:

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

Por exemplo, o cálculo da conjunção lógica $\wedge$ para os argumentos **falso** e **verdadeiro** é como segue:

$(\lambda p. \lambda q. p \ q \text{ falso}) \text{ falso} \text{ verdadeiro} \triangleright$
 $(\lambda q. \text{falso} \ q \text{ falso}) \text{ verdadeiro} \triangleright$
 $\text{falso} \text{ verdadeiro} \text{ falso} \triangleright$ falso: projeção da segunda componente
falso

Analogamente, o cálculo da conjunção lógica $\wedge$ para os argumentos **verdadeiro** e **verdadeiro** é como segue:

$(\lambda p. \lambda q. p \ q \text{ falso}) \text{ verdadeiro} \text{ verdadeiro} \triangleright$
 $(\lambda q. \text{verdadeiro} \ q \text{ falso}) \text{ verdadeiro} \triangleright$
 $\text{verdadeiro} \text{ verdadeiro} \text{ falso} \triangleright$ verdadeiro: projeção da primeira componente
verdadeiro □

exemplo 8.15 – Se-então-senão

A especificação de:

$$\text{se } p \text{ então } x \text{ senão } y$$

pode ser realizada simplesmente projetando a componente correspondente ao **então/senão**, dependendo do valor lógico da condição. Assim, se o cálculo da condição for:

- **verdadeiro**, então projeta a primeira componente;
- **falso**, então projeta a segunda componente.

O correspondente termo é como segue:

$$\text{se-então-senão} := \lambda p. \lambda x. \lambda y. p \ x \ y$$
 □

Tipo natural

Para modelar adequadamente o tipo de dado *natural*, deve-se antes estabelecer qual o uso que se objetiva. Possivelmente, o uso mais intuitivo é o de contagem, o qual é ilustrado a seguir (modelagem conhecida como *numerais de Church*).

exemplo 8.16 – Números naturais

Um número natural n representa quantas vezes uma função f é aplicada a uma entrada x :

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

Assim, tem-se que:

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

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

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

...

□

Observe que natural 0 e o valor lógico **falso** são representados por λ -termos equivalentes alfabeticamente.

exemplo 8.17 – Sucessor

A operação que recebe um número natural como argumento e retorna seu sucessor pode ser definida como:

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

Essa definição de sucessor prevê a utilização de equivalência alfabéticas para a representação dos números naturais (por exemplo, o natural n pode ser representado por $\lambda g. \lambda y. g^n(y)$). Com isso, se n é definido por $\lambda g. \lambda y. g^n(y)$, no cálculo do **sucessor** acarretará no termo final $\lambda f. \lambda x. f(f^n(x))$, que é equivalente à representação $\lambda f. \lambda x. f^{n+1}(x)$, definida para $n + 1$.

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

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

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

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

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

□

exemplo 8.18 – Adição

A operação de adição pode ser definida por meio da aplicação repetida da função **sucessor**. Assim:

$$\text{adição} := \lambda n. \lambda m. n \text{ sucessor } m$$

Nesse termo, na adição de x com y o argumento x é associado com n que, por atender à definição de natural vista anteriormente, receberá como primeiro argumento a função (**sucessor**) que será aplicada repetida vezes (x vezes) ao segundo argumento (y que é associado com m). Por exemplo, o cálculo da **adição** para os argumentos $2 := \lambda f. \lambda x. f^2(x)$ e $3 := \lambda f. \lambda x. f^3(x)$, é como segue:

$$(\lambda n. \lambda m. n \text{ sucessor } m) 2 3 \triangleright$$

$$(\lambda m. 2 \text{ sucessor } m) 3 \triangleright$$

$$2 \text{ sucessor } 3 = \lambda f. \lambda x. f^2(x) \text{ sucessor } 3 \triangleright$$

$$\lambda x. \text{sucessor}^2(x) 3 \triangleright$$

$$\text{sucessor}^2(3) = \text{sucessor}(\text{sucessor}(3)) = \text{sucessor}(4) = 5$$

□

8.2.6 recursão e ponto fixo

No cálculo Lambda, uma função não pode referenciar a si mesma diretamente. Entretanto, pode usar a si mesma como parâmetro, permitindo então, uma recursão. De certa forma, o operador de ponto fixo implementa esta ideia, permitindo definir recursão no cálculo Lambda.

definição 8.17 – Ponto fixo

Seja $f: A \rightarrow A$ uma função. Então $x \in A$ é um *ponto fixo* de f se:

$$f(x) = x \quad \square$$

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

exemplo 8.19 – Ponto fixo

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

$\text{id}: \mathbb{N} \rightarrow \mathbb{N}$	qualquer $n \in \mathbb{N}$
$\text{sucessor}: \mathbb{N} \rightarrow \mathbb{N}$	não tem
$n^2: \mathbb{N} \rightarrow \mathbb{N}$	0 e 1
$n^2 - 2n: \mathbb{N} \rightarrow \mathbb{N}$	0 e 3

$\square$

definição 8.18 – Operador de ponto fixo

Um *operador de ponto fixo* M é um λ -termo tal que, para qualquer λ -termo F , é fato que:

$$M F = F (M F) \quad \square$$

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

exemplo 8.20 – Operador de ponto fixo

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

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

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

$$\begin{aligned} M f &= \\ & (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) f \triangleright \\ & (\lambda x. f (x x)) (\lambda x. f (x x)) [f \leftarrow f] = (\lambda x. f (x x)) (\lambda x. f (x x)) \triangleright \\ & (f (x x)) [x \leftarrow \lambda x. f (x x)] = f (\lambda x. f (x x) \lambda x. f (x x)) = \\ & f (M f) \end{aligned} \quad \square$$

exemplo 8.21 – Fatorial

Considere o seguinte λ -termo:

$$F := \lambda r. \lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot (r (n - 1)))$$

Então, o fatorial pode ser definido como (suponha M como definido em exemplo 8.20 – Operador de ponto fixo):

$$\text{Fat} := M F$$

Por exemplo, o cálculo de Fat para o argumento 3 é como segue:

$$\begin{aligned} M F 3 &= \\ F (M F) 3 &= \\ \lambda r. \lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot (r (n - 1))) (M F) 3 &= \\ (\lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot ((M F) (n - 1)))) 3 &= \\ \text{se } 3 = 0 \text{ então } 1 \text{ senão } 3 \cdot ((M F) (3 - 1)) &= \\ 3 \cdot ((F (M F)) 2) &= \\ 3 \cdot (\lambda r. \lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot (r (n - 1))) (M F) 2) &= \\ 3 \cdot ((\lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot ((M F) (n - 1)))) 2) &= \\ 3 \cdot (\text{se } 2 = 0 \text{ então } 1 \text{ senão } 2 \cdot ((M F) (2 - 1))) &= \\ 3 \cdot (2 \cdot (F (M F) 1)) &= \\ 3 \cdot (2 \cdot (\lambda r. \lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot (r (n - 1))) (M F) 1)) &= \\ 3 \cdot (2 \cdot ((\lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot ((M F) (n - 1)))) 1)) &= \\ 3 \cdot (2 \cdot (\text{se } 1 = 0 \text{ então } 1 \text{ senão } 1 \cdot ((M F) (1 - 1)))) &= \\ 3 \cdot (2 \cdot (1 \cdot ((F (M F)) 0))) &= \\ 3 \cdot (2 \cdot (1 \cdot (\lambda r. \lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot (r (n - 1))) (M F) 0))) &= \\ 3 \cdot (2 \cdot (1 \cdot ((\lambda n. (\text{se } n = 0 \text{ então } 1 \text{ senão } n \cdot ((M F) (n - 1)))) 0))) &= \\ 3 \cdot (2 \cdot (1 \cdot (\text{se } 0 = 0 \text{ então } 1 \text{ senão } 0 \cdot ((M F) (0 - 1)))))) &= \\ 3 \cdot (2 \cdot (1 \cdot 1)) &= \\ 3 \cdot (2 \cdot 1) &= \\ 3 \cdot 2 &= \\ 6 \end{aligned}$$

Observe que em $MF3$ a recursão é dada pelo surgimento de “novas cópias” de MF . □

8.2.7 cálculo Lambda e computabilidade

O λ -cálculo é equivalente, computacionalmente, a qualquer linguagem de programação (e, portanto, é equivalente, computacionalmente, à máquina de Turing). Se um λ -termo for visto como um programa, e o processo de redução como sua execução, pode-se associar a cada termo da forma $\lambda x. M$ um programa que computa uma função (parcial ou total?) com entrada x e corpo de comandos M (normalmente dependente de x). O resultado de tal programa seria o termo obtido após um número finito de reduções e tal que não possa ser mais redu-

zido. Entretanto, para continuar com tal entendimento, é necessário observar os seguintes pontos:

- a** O processo de reduções, em alguns casos, não para (veja exemplo abaixo);
- b** O processo de reduções não é determinístico, pois as reduções não têm ordem preferencial de aplicação. Veja o seguinte exemplo que, conforme a sequência escolhida, produz “resultados” diferentes. Seja:

$$(\lambda x.z) ((\lambda y.y) y) (\lambda y.y y))$$

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

$$\begin{aligned} (\lambda x.z) ((\lambda y.y) y) (\lambda y.y y)) &\triangleright \\ (\lambda x.z) ((\lambda y.y) y) (\lambda y.y y)) &\triangleright \\ (\lambda x.z) ((\lambda y.y) y) (\lambda y.y y)) &\dots \end{aligned}$$

ou, aplicando-se a redução mais à esquerda, tem-se que:

$$(\lambda x.z) ((\lambda y.y) y) (\lambda y.y y)) \triangleright z$$

Observe que, em qualquer momento que for aplicada uma redução mais à esquerda na primeira sequência de reduções, obtém-se z , que obviamente não pode ser mais reduzido. Isto é, o exemplo acima tem a interessante propriedade de que qualquer sequência de reduções ou não para, ou tem como resultado o mesmo que as outras sequências.

Assim, é preciso ter que, se um λ -termo possui resultado definido a partir de alguma sequência de reduções, então esse resultado é único, isto é, qualquer outra sequência que produza resultado definido produz o mesmo resultado. De fato, esse resultado vale, mas não será demonstrado.

definição 8.19 – Igualdade induzida por redução beta

Sejam M e N dois λ -termos. A igualdade induzida por β -redução, denotada por:

$$M =_{\beta} N$$

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

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

O seguinte teorema estabelece que o formalismo cálculo Lambda é equivalente ao formalismo máquina de Turing, bem como aos demais formalismos equivalentes estudados. O teorema não será demonstrado.

teorema 8.20 – Cálculo Lambda e função computável

Uma função $f: \mathbb{N} \rightarrow \mathbb{N}$ é computável se, e somente se, existe um termo Lambda F tal que, para qualquer $x, y \in \mathbb{N}$:

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

8.3

$\rightarrow$ funções recursivas e ciência da computação

8.3.1 importância das funções recursivas

O estudo das funções recursivas e da recursão em geral é de fundamental importância na ciência da computação. Não apenas são formalismos tão poderosos como as máquinas universais, como também fornecem uma abordagem (denotacional) completamente diferente da operacional. Em comparação com as máquinas universais, possuem um poder de expressão significativo, no sentido em que uma simples função pode representar um algoritmo consideravelmente complexo.

Em muitas instituições de ensino, linguagens baseadas em funções recursivas são usadas como uma primeira linguagem de programação. 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 e em pouco tempo (em alguns casos, em minutos).

Além disso, funções recursivas e, em particular, o cálculo Lambda possuem importantes aplicações em outros contextos da ciência da computação, por exemplo, lógica.

De qualquer forma, quase a totalidade das linguagens de programação modernas como Pascal ou C possuem recursão como um construtor básico de programas. A arquitetura da maioria dos atuais computadores possui ainda facilidades para implementar recursão. Consequentemente, o processamento de recursão possui, em geral, bons níveis de eficiência.

8.3.2 linguagem de programação funcional

No que segue, é exemplificada a aplicação das funções recursivas e, em particular, do cálculo Lambda no contexto da programação funcional, usando como exemplo uma breve introdução da linguagem *Haskell*. Na leitura, sugere-se atenção especial na identificação da semelhança de Haskell com o cálculo Lambda.

A *programação funcional* é um estilo de programação baseada em funções e composição de funções (constituindo um programa) em vez de comandos. Assim, um programa é uma expressão funcional a qual é avaliada, e não uma sequência de comandos a serem executados por um computador (como em Pascal, por exemplo).

Uma *linguagem de programação funcional* é uma linguagem que suporta e encoraja esse estilo de programação. Assim, resumidamente, uma linguagem de programação funcional pode ser considerada como operações tipadas (ou seja, com tipos associados) e construtores que permitem definir novos tipos e operações mais complexos. Inclusive, uma linguagem de programação funcional considerada *pura* não possui variáveis e nem atribuições. Portanto, uma linguagem de programação funcional é composta basicamente por:

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

Haskell é um exemplo de linguagem de programação funcional pura na qual todas as funções devem seguir o conceito matemático de função, isto é, os mesmos valores do domínio (parâmetros atuais) aplicados à função resultam nos mesmos elementos do contradomínio (produz as mesmas saídas). Ou seja, o resultado da aplicação de uma função é independente de qualquer estado implícito do sistema. Por exemplo, considere o seguinte exemplo de função constante em Haskell:

```
x = 1
```

Trata-se de uma função constante a qual sempre retorna o valor 1. Considere também a seguinte função:

```
f x = x + 1
```


Trata-se de uma função f que, para o parâmetro x , retorna o valor $x + 1$. Ou seja, são exemplos de funções em Haskell que satisfazem o conceito de função matemática.

Considere o seguinte exemplo (menos trivial) de função em Haskell para calcular as raízes de uma equação polinomial de segundo grau $ax^2 + bx + c = 0$

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

Trata-se de uma função que, para a , b e c , retorna o par ordenado de valores correspondendo às raízes da equação. A palavra chave `let` é usada para declarar `delta`, a qual fica acessível apenas dentro do escopo da função `baskara`. Por exemplo, aplicando os valores 1, -5 e 6, a expressão seria reduzida (no sentido da redução do cálculo Lambda) pelo computador de forma similar à seguinte sequência:

```
baskara 1 -5 6
```

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

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

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

```
( 3, 2 )
```


8.4

→ conclusões

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

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

A importância do cálculo Lambda transcende o estudo da computabilidade, tendo importantes aplicações no estudo da lógica e das linguagens de programação. No contexto das linguagens de programação funcionais, uma breve introdução da linguagem Haskell foi apresentada.

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

8.5

→ exercícios

exercício 8.1 Usando as funções recursivas parciais f_{zero} , **sucessor** e **adição**, verifique se existem outras formas de definir equivalentemente as funções f_{um} , f_{dois} e $f_{\text{três}}$ apresentadas no exemplo 8.1 – Substituição de funções – funções constantes.

exercício 8.2 Compare e diferencie claramente as funções f_{zero} (exemplo 8.1 – Substituição de funções – funções constantes) e a função constante $\text{const}_{\text{zero}}$ (exemplo 8.4 – Minimização – função número zero).

exercício 8.3 Determine o valor de **sub**(2, 3) para a função recursiva do exemplo 8.6 – Minimização, recursão primitiva – subtração.

exercício 8.4 Desenvolva funções recursivas totais sobre $\mathbb{N}$ para as seguintes operações:

- a** Multiplicação;
- b** Quadrado (n^2);

- c** Fatorial ($n!$).

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

exercício 8.5 As funções recursivas de Kleene são definidas sobre $\mathbb{N}$. Como poderia ser uma função recursiva para tratar palavras?

exercício 8.6 Usando a solução do exercício acima, demonstre que as funções que seguem são recursivas (Kleene). Em cada caso, verifique se é total ou parcial (suponha $\Sigma = \{a, b\}$):

- a** esquerda: $\Sigma^* \rightarrow \Sigma^*$ tal que retorna o primeiro símbolo de x ;
- b** direita: $\Sigma^* \rightarrow \Sigma^*$ tal que retorna o último símbolo de x ;
- c** comprimento: $\Sigma^* \rightarrow \{a\}^*$ tal que retorna o número de símbolos que compõem x , em unário;
- d** ordem_lexicográfica: $\{a\}^* \rightarrow \Sigma^*$ tal que retorna a x -ésima palavra (valor em unário) na ordem lexicográfica;
- e** compara: $(\Sigma^*)^2 \rightarrow \Sigma^*$ tal que retorna:
 - ε , se $x = y$
 - a , se $x > y$
 - b , se $x < y$

exercício 8.7 *Função de Ackermann.* A função de Ackermann é um importante exemplo no estudo das funções recursivas e das funções *recursivas primitivas* (funções recursivas, mas totais e sem minimização). Historicamente, foi considerada a possibilidade de que as funções recursivas primitivas definissem a classe das funções totais computáveis. Entretanto, a função de Ackermann (prova-se) é um exemplo de função recursiva (total) a qual não é primitiva.

A função de Ackermann $\text{ack}: \mathbb{N}^2 \rightarrow \mathbb{N}$ é tal que:

$$\begin{aligned} \text{ack}(0, y) &= y + 1 \\ \text{ack}(1, 0) &= 2 \\ \text{ack}(x, 0) &= x + 2, \text{ para } x \geq 2 \\ \text{ack}(x + 1, y + 1) &= \text{ack}(\text{ack}(x, y + 1), y) \end{aligned}$$

- a** A definição acima satisfaz a definição de função recursiva?
- b** Calcule, passo a passo:
 - $\text{ack}(0, 0) = 1$
 - $\text{ack}(2, 1) = 5$
 - $\text{ack}(2, 2) = 7$
- c** Defina, formalmente, a função recursiva de um argumento $\text{ack}(x, 2)$

exercício 8.8 Em relação ao exemplo 8.11 – Redução, detalhe as substituições, conforme a definição 8.12 – Substituição.

exercício 8.9 Usando os exemplos referentes aos tipos booleano (exemplo 8.12 – Tipo booleano) e natural (exemplo 8.13 – Tipo natural), desenvolva λ -termo para:

- a** Operação ou lógico;
- b** Teste se é zero;
- c** Teste se é menor ou igual;
- d** Operação predecessor, na qual:
 $\text{pred}(0) = 0$
 $\text{pred}(n) = n - 1$, para $n > 0$
- e** Teste se é número par;
- f** Operação multiplicação;
- g** Operação exponencial.

exercício 8.10 Em relação à composição de funções:

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

exercício 8.11 Desenvolva um λ -termo correspondente à seguinte função:

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

exercício 8.12 Considere os seguintes λ -termo:

$\text{teste}_{\text{zero}}(n) := \lambda n. n(\lambda x. \text{falso}) \text{ verdadeiro}$

$\text{predecessor}(n) := \lambda n. \lambda f. \lambda x. (n(\lambda z. \lambda w. w(z f)))(\lambda w. x) (\lambda u. u)$

- a** Compare os termos $\text{teste}_{\text{zero}}$ e predecessor definidos acima com os seus termos desenvolvidos nos correspondentes exercícios anteriores;
- b** Usando os exemplos referentes aos tipos booleano (exemplo 8.12 – Tipo booleano), natural (exemplo 8.13 – Tipo natural) e operador de ponto fixo (exemplo 8.15 – Operador de ponto fixo), bem como os termos definidos no enunciado deste exercício, desenvolva um λ -termo correspondente à seguinte função:

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

exercício 8.13 Considere o seguinte λ -termo:

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

- a** Compare o termo $\text{teste}_{\text{par}}$ definido acima com o seu termo desenvolvido no correspondente exercício anterior;
- b** Usando os exemplos referentes aos tipos booleano (exemplo 8.12 – Tipo booleano), natural (exemplo 8.13 – Tipo natural) e operador de ponto fixo (exemplo 8.15 – Operador de ponto fixo), bem como os termos definidos no enunciado deste exercício e dos anteriores, desenvolva um λ -termo correspondente à seguinte função:

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

Termos-chaves

abstração lambda, p. 224

aplicação lambda, p. 224

cálculo Lambda, p. 215, p. 229

classe das funções recursivas parciais, p. 218

classes das funções Turing-computáveis, p. 215

função recursiva parcial, p. 215, p. 221

função recursiva total, p. 216, p. 222

função recursiva de Kleene, p. 215

funções Lambda-computáveis, p. 215

funções Turing-computáveis totais, p. 216

gramática irrestrita, p. 215

gramática livre do contexto, p. 215

gramática regular, p. 215

hierarquia de Chomsky, p. 215

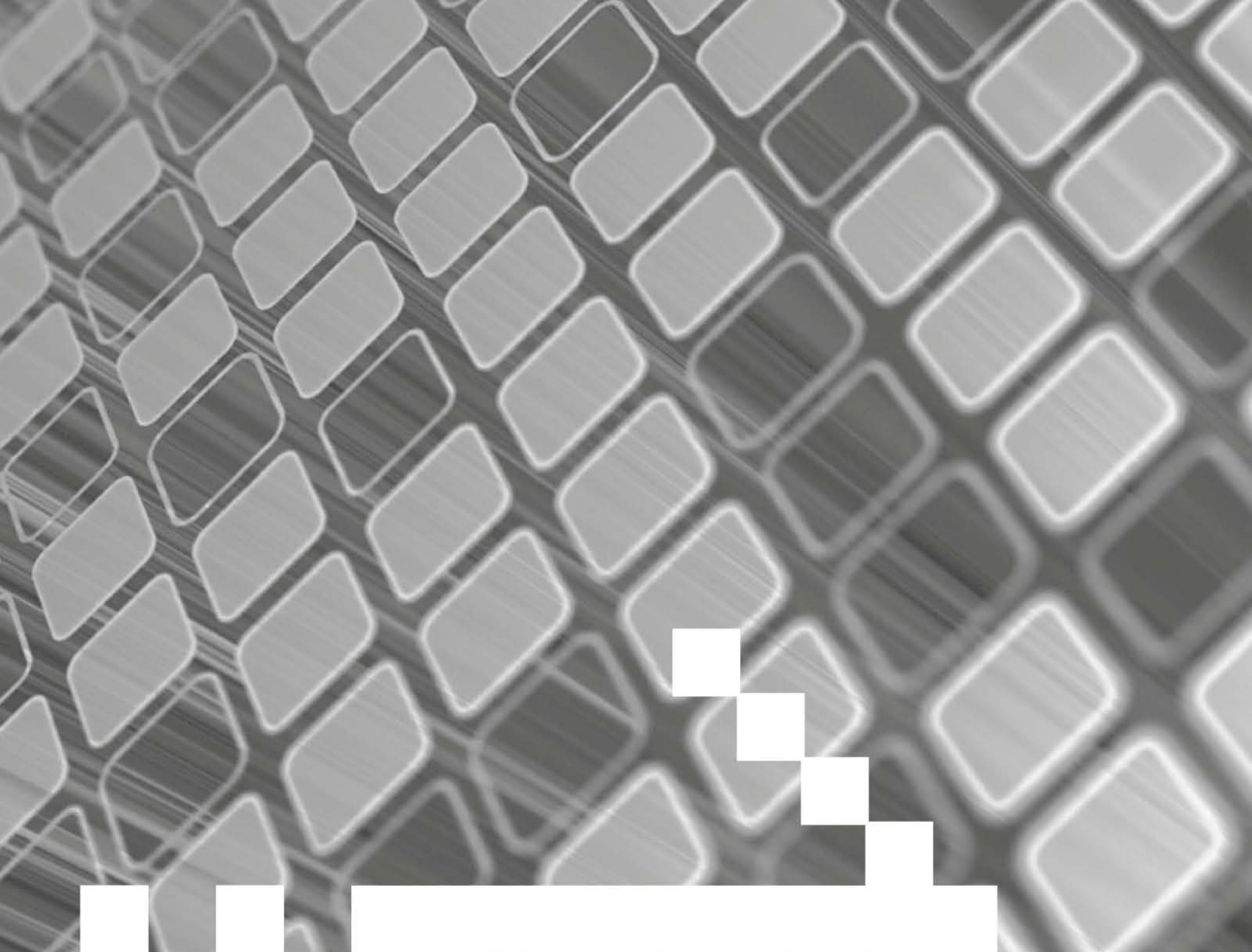
linguagem Lambda, p. 224, p. 225

teoria da computação clássica, p. 215

termo lambda, p. 224

variável ligada, p. 226

variável livre, p. 226



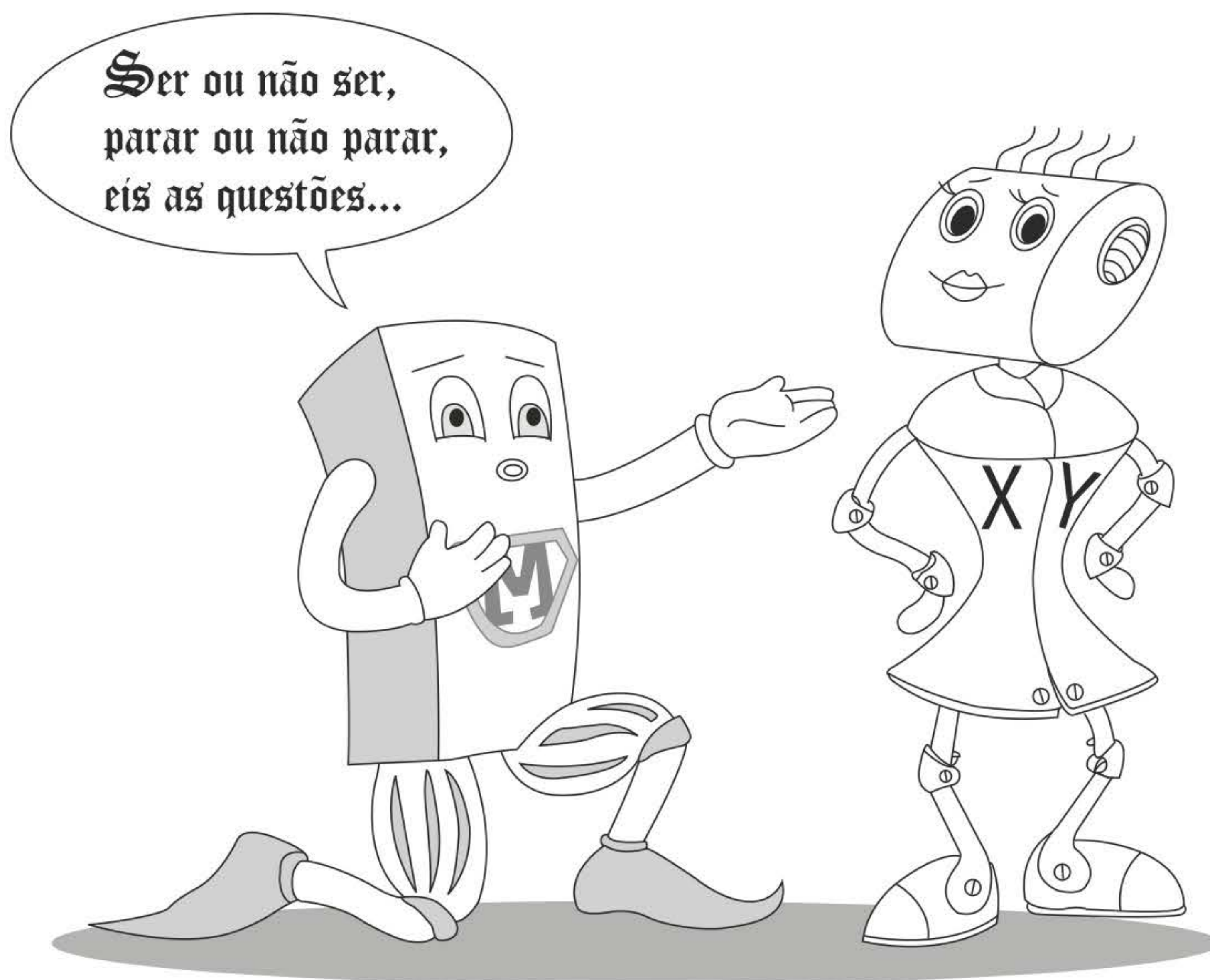
capítulo

9

computabilidade

■ ■ O estudo da computabilidade objetiva determinar a solucionabilidade de problemas (existência de algoritmos). Portanto, investiga os limites do que pode ser implementado em um computador evitando a pesquisa de soluções inexistentes.

A abordagem é centrada nos problemas de decisão (do tipo sim/não) e usa frequentemente o princípio da redução: investigação da solucionabilidade de um problema a partir de outro. Infelizmente, muitos dos problemas importantes são não solucionáveis como o detector universal de *loops*. De fato, informalmente falando, existem muito mais problemas não solucionáveis do que solucionáveis.



O objetivo do estudo da *computabilidade* é determinar a *solucionabilidade* de problemas, ou seja, investigar a existência ou não de algoritmos que solucionem determinada classe de problemas. Mais ainda, investigar os limites da computabilidade e, conseqüentemente, os limites do que pode efetivamente ser implementado em um computador. Por conseqüência, o estudo da solucionabilidade objetiva evitar a pesquisa de soluções inexistentes de problemas.

Um exemplo de problema que não tinha solução, mas que foi estudado por muitos anos (cerca de 70 anos) foi um dos 23 problemas propostos por David Hilbert, em Paris, no ano de 1900, durante o II Congresso Internacional de Matemática. Ele apresentou uma lista de 23 problemas ainda não solucionados, que seriam o desafio para o novo século. Entre eles, consta como o décimo problema o seguinte enunciado:

Dada uma equação diofantina $P(a_0, \dots, a_n) = 0$, onde P é um polinômio com coeficientes inteiros, quer-se saber se existe um procedimento efetivo que seja capaz de determinar, em um tempo finito, se suas raízes são inteiras.

Sabe-se que as equações diofantinas são equações algébricas com coeficientes inteiros.

Buscava-se estabelecer fundamentos rigorosos para a aritmética, visando representar todas as leis científicas em equações matemáticas. O princípio era estabelecer a consistência formal da

aritmética e, em seguida estender às demais áreas do conhecimento. Procurou-se definir uma linguagem puramente sintática (sem significado) denominada de sistema formal. Recaiu-se então no problema da existência ou não de um procedimento mecânico efetivo que decidisse a veracidade ou falsidade de qualquer enunciado aritmético.

Em 1928, David Hilbert propôs o *problema de decisão* (também conhecido como *Entscheidungsproblem*). O qual questionava a existência de um procedimento mecânico (baseado no trabalho de Gottfried Leibniz, que buscava mecanismo mecânico de manipulação de fórmulas) capaz de decidir se, dado um enunciado (proposição) da lógica de primeira ordem, ele seria válido ou não, em um tempo finito.

Esta ideia de considerar a matemática um sistema formal empolgou os matemáticos no século XIX. Pretendia-se obter uma teoria aritmética como um sistema formal consistente e completo. Isso infelizmente não foi possível, como foi afirmado por Kurt Gödel (em 1931) em seu *teorema da não completude* (*incompleteness theorem*):

Todas as formulações axiomáticas consistentes da teoria dos números incluem proposições indecidíveis, ou seja, que não podem ser provadas como verdadeiras ou como falsas. Portanto, se um sistema formal é consistente, ele não pode ser completo, e a consistência dos axiomas não pode ser provada usando o próprio sistema formal.

A demonstração usou uma codificação de primos (*número de Gödel*), de forma que as formulações axiomáticas eram codificadas em números naturais.

A resolução do problema de decisão começou a apresentar resultados em 1936, quando Alonzo Church provou que não existia nenhum algoritmo definido por meio de funções recursivas que decidisse se, para duas expressões dadas em cálculo Lambda, elas seriam ou não equivalentes. Alan Turing também contribuiu na resolução do problema de decisão, transformando-o em um *problema da parada* em sua máquina.

Na resolução do problema de decisão, uma boa parte do sucesso se deu pelo uso da técnica da *redução* (*princípio da redução*), na qual partes de um problema são transformadas em outro, o que permite usar resultados já conhecidos para se obter a resolução de partes do problema atual e, assim, resolvê-lo.

A solução definitiva do problema de decisão foi apresentada em 1970, pelo matemático Yuri Matiyasevich, o qual provou que tal procedimento efetivo (algoritmo) não existe. A prova é complexa, envolvendo teoria dos números e resultados anteriores sobre este problema. Basicamente, se tem:

Se R é uma relação que pode ser codificada em um número natural, então R é uma equação diofantina se, e somente se, R é enumerável.

O resultado disso é que o conjunto das equações diofantinas com raízes inteiras não é solucionável (decidível).

Neste capítulo, 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. Portanto, um problema de decisão é uma questão sobre um sistema formal com uma resposta do tipo **sim** ou **não**. Por exemplo:

Dados dois números naturais, eles são primos entre si?

A resposta para tal problema pode ser **sim** ou **não**, dependendo dos números tomados. Um problema de decisão também pode ser formalizado como o problema de verificar se uma determinada palavra pertence ou não a uma linguagem formal. O conjunto contém exatamente as palavras para as quais a resposta à pergunta é **sim**.

Assim, os problemas de decisão são problemas que objetivam determinar se um dado elemento de algum universo pertence ou não a um determinado conjunto **A** (ou, equivalentemente, se satisfaz uma determinada propriedade). Se existir um algoritmo que receba como entrada um elemento qualquer **x** e retorne como saída **sim**, caso **x** pertença ao conjunto **A**, ou **não**, caso contrário, então se diz que o problema de decisão para o conjunto **A** é solucionável. Do contrário, se não existir um algoritmo capaz de fazer essa avaliação, então se diz que o problema de decisão para o conjunto **A** é não solucionável. Tradicionalmente, é comum definir-se o problema de decisão em termos do conjunto de entradas para as quais o problema retorna **sim**. Nesse sentido, um problema de decisão é equivalente a uma linguagem formal.

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** *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*;
- b** *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), como Pascal, e verificar se são equivalentes, ou seja, se, de fato, reconhecem a mesma linguagem.

Alguns problemas não solucionáveis são parcialmente solucionáveis, ou seja, existe um algoritmo capaz de responder **sim**, embora, eventualmente, possa ficar processando indefinidamente 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):

- a classe dos problemas computáveis é contável;
- a classe dos problemas não computáveis é não contável.

Assim, informalmente, pode-se afirmar que o cardinal da classe dos problemas não computáveis é “muito maior” que o da 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 9.1):

- a** Sejam **A** e **B** dois problemas de decisão. Suponha que é possível modificar (“reduzir”) o **Problema A** de tal forma que ele se porte 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).

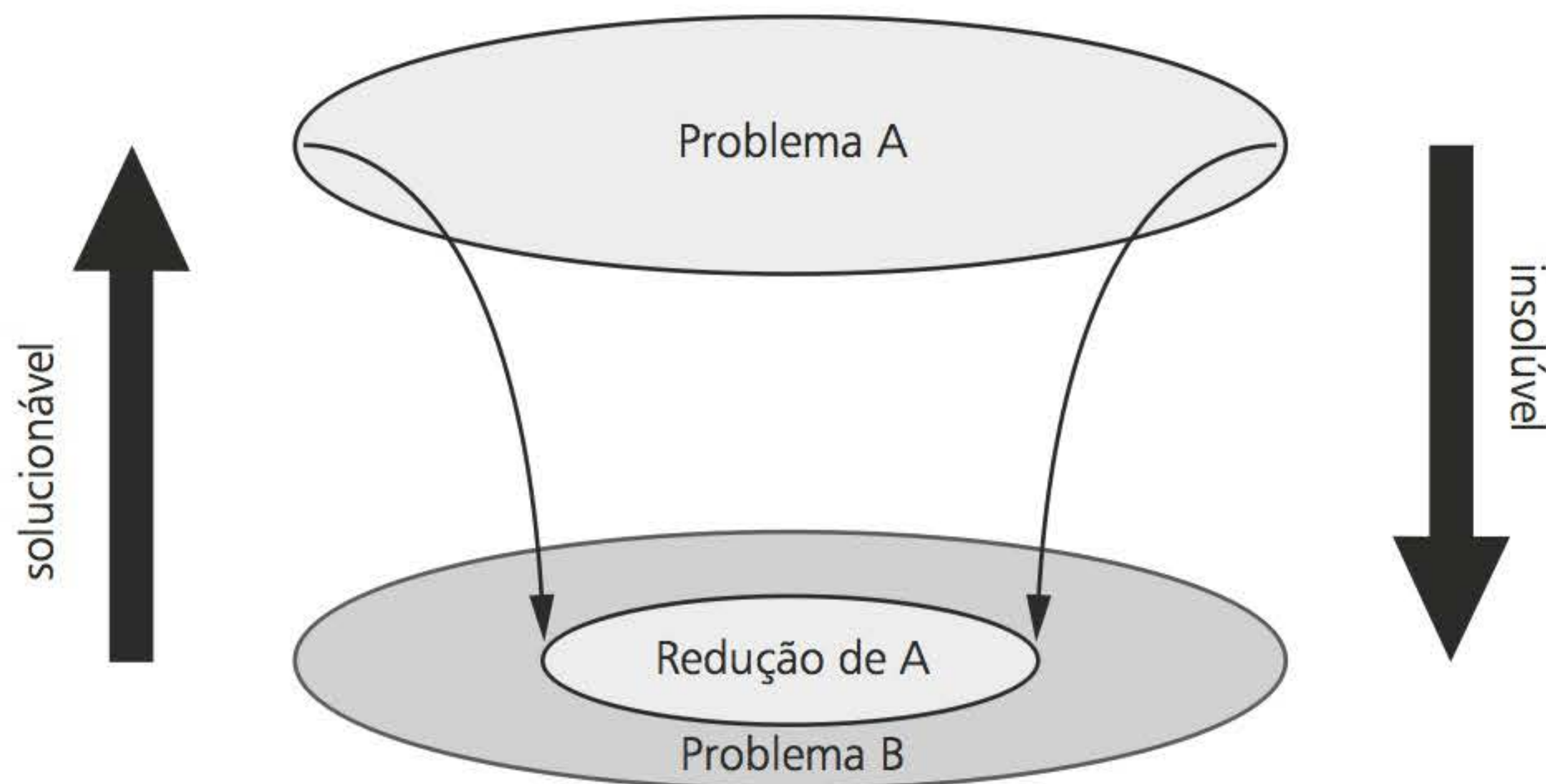


figura 9.1 Princípio da redução.

Como ilustração do princípio da redução, considere o caso de um aluno de um curso de ciência da computação. Para se formar, ele necessita ser aprovado em todas as disciplinas que compõem a grade curricular do curso, entre elas, a disciplina de teoria da computação. Considere então os problemas:

Problema A: “O aluno foi aprovado em teoria da computação?”

Problema B: “O aluno vai se formar?”

Se determinado aluno **x**, não é aprovado em teoria da computação, ou seja, a resposta do **Problema A** é **não**, pode-se determinar a solução do **Problema B**, para tal aluno, pois para se formar, precisa ter sido aprovado em teoria da computação. Logo, o aluno **x** não se formará (pelos menos até conseguir a aprovação nessa disciplina).

Já um outro aluno **y** tem sua formatura confirmada, ou seja, sabe-se que a resposta do **Problema B** é **sim**, logo, o aluno **y** também foi aprovado em teoria da computação (e nas demais disciplinas da grade curricular) e, portanto, a resposta do **Problema A** também é **sim**.

9.1

→ 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 9.2, induzidas pelas definições que seguem.

definição 9.1 – Problema solucionável

Um problema é dito *solucionável*, *totalmente solucionável* ou *decidível* se existe um algoritmo (máquina universal) que solucione o problema tal que sempre para para qualquer entrada, com uma resposta afirmativa (**ACEITA**) ou negativa (**REJEITA**). □

definição 9.2 – Problema não solucionável

Um problema é dito *não solucionável* ou *não decidível* se *não* existe um algoritmo (máquina universal) que solucione o problema tal que sempre para para qualquer entrada. □

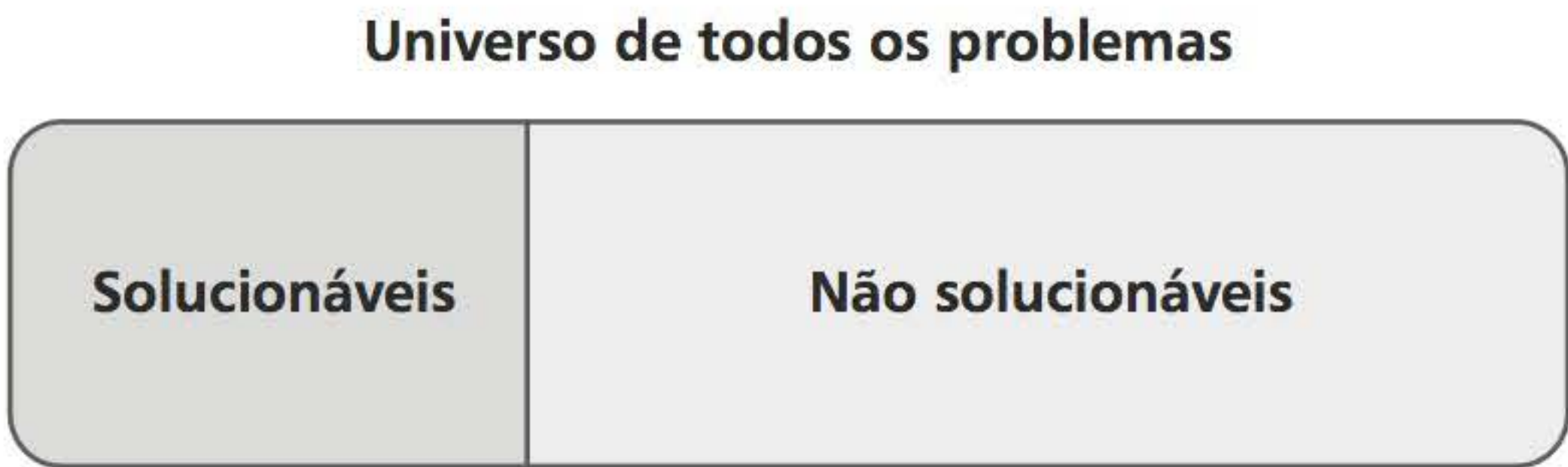


figura 9.2 Particionamento do conjunto de todos os problemas em classes.

Outra maneira de se particionar o conjunto de todos os problemas consiste nas duas classes ilustradas na figura 9.3, induzidas pela definição que segue.

definição 9.3 – 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 9.4 – 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**). □

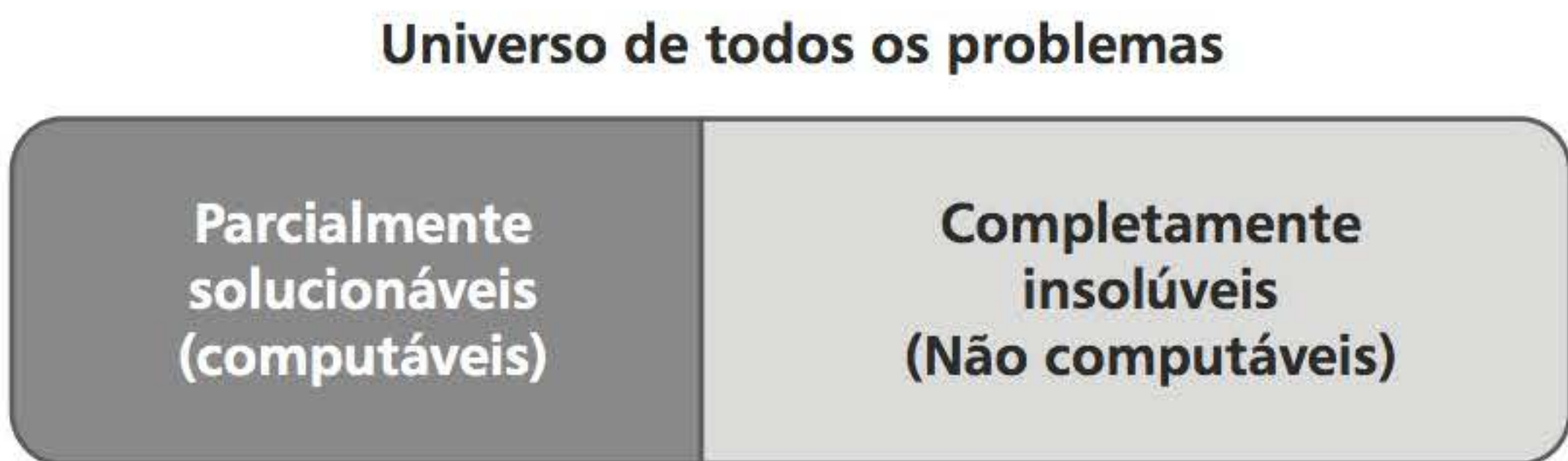


figura 9.3 Particionamento do conjunto de todos os problemas em classes.

É importante observar que alguns problemas não solucionáveis são parcialmente solucionáveis. Em relação ao relacionamento das classes de problemas, as seguintes conclusões podem ser estabelecidas (veja a figura 9.4):

- a união das classes dos problemas solucionáveis e dos não solucionáveis é o universo de todos os problemas;
- a união das classes dos problemas parcialmente solucionáveis e dos completamente insolúveis é o universo de todos os problemas;
- a classe dos problemas parcialmente solucionáveis contém propriamente a classe dos problemas solucionáveis e parte da classe dos problemas 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 processando indefinidamente.

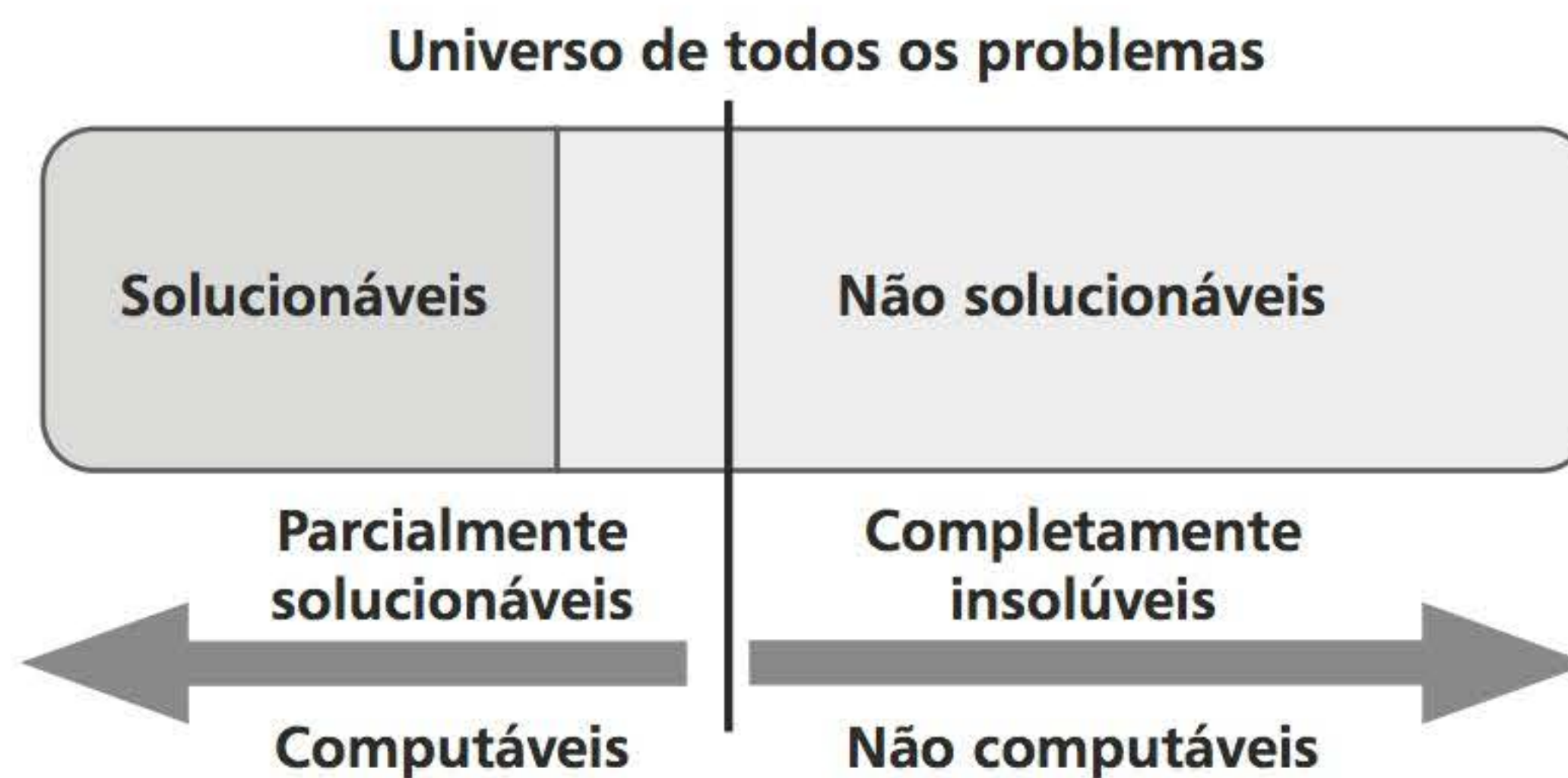


figura 9.4 Relação entre as classes de problemas.

9.2

→ problemas de decisão

Neste capítulo, são tratados alguns problemas de decisão ou do tipo **sim/não**. A ideia 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 ideia: dado um programa **P** para uma máquina universal **M**, decidir se a função computada $\langle P, M \rangle$ é total (ou seja, se a correspondente computação é finita). Lembre-se de 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 para 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, por exemplo, demonstrar limitações da capacidade de se expressar soluções por meio 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 9.5 – Solucionabilidade de problemas $\times$ problema de reconhecimento de linguagens

A investigação da solucionabilidade de problemas em máquinas universais pode ser vista como um problema de reconhecimento de linguagens. 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. Consequentemente, tem-se que:

a classe dos problemas solucionáveis é equivalente à classe das linguagens recursivas

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). $\square$

9.3

→ codificação de programas

Alguns problemas de decisão são definidos sobre programas ou máquinas. Assim, é necessário estabelecer uma forma de codificar programas ou máquinas.

No capítulo 4 – máquinas de registradores – Norma, foi mostrado como codificar um programa como um número natural. Como consequência, problemas de decisão sobre programas podem ser traduzidos em problemas de decisão sobre o conjunto dos naturais. Uma desvantagem da função de codificação introduzida, denominada de **código**, é que não é bijetora (mas é injetora). De fato, nem todo número 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 9.1 – Codificação bijetora de programas

Seja P o conjunto de todos os programas do tipo que está sendo considerado (monolítico, iterativo ou recursivo). Considere a sequência de inteiros $p_0, p_1, p_2, \dots$ formada pelos códigos dos programas de P (ordenados pela relação “menor” sobre os naturais). Então:

$$\text{código}_{\text{bij}}: P \rightarrow \mathbb{N}$$

onde, para qualquer $P \in P$, se o código de P é p_n , então (veja a figura 9.5):

$$\text{código_bij}(P) = n$$

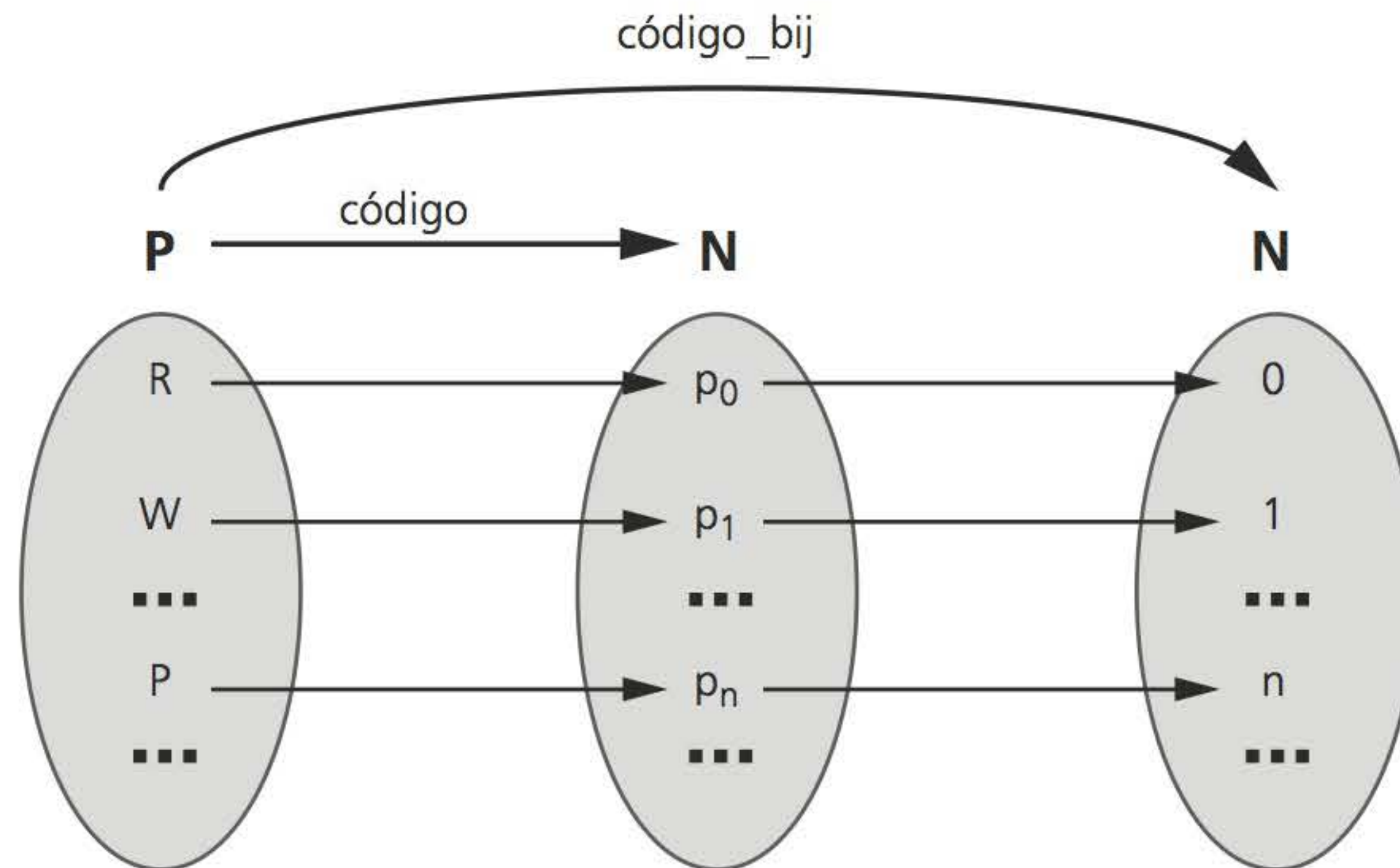


figura 9.5 Função bijetora de codificação de programas.

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 $p = \text{código}(P)$
- 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 $\text{código_bij}(P) = n + 1$

A inversa da função **código_bij** (que, obviamente, também é bijetora):

$$\text{código_bij}^{-1}: N \rightarrow P$$

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 . □

9.4

→ problema da autoaplicação

O problema da autoaplicação 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 9.6 – Problema da autoaplicação

O problema da autoaplicação é como segue (versão para a máquina Norma):

Dado um programa monolítico arbitrário P para a máquina Norma, decidir se a função computada $\langle P, \text{Norma} \rangle$ é definida para p , onde p é a codificação de P $\square$

Em outras palavras, o problema da autoaplicação 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 autoaplicação pode ser redefinido como uma linguagem, como segue:

$$L_{AA} = \{ p \mid \langle P, \text{Norma} \rangle(p) \text{ é definida,} \\ P \text{ é programa monolítico para Norma e } p = \text{código_bij}(P) \}$$

Assim, a questão da solucionabilidade do problema da autoaplicação 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:

$$\begin{aligned} n \in \text{ACEITA}(P) & \text{ se, e somente se, } \langle P, \text{Norma} \rangle(n) = 0 \\ n \in \text{REJEITA}(P) & \text{ se, e somente se, } \langle P, \text{Norma} \rangle(n) = 1 \\ n \in \text{LOOP}(P) & \text{ se, e somente se, } \langle P, \text{Norma} \rangle(n) \text{ é indefinida} \end{aligned}$$

teorema 9.7 – Problema da autoaplicação é parcialmente solucionável

A linguagem L_{AA} que traduz o problema da autoaplicação 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:

$$\begin{aligned} \text{ACEITA}(Q) &= L_{AA} \\ \text{REJEITA}(Q) \cup \text{LOOP}(Q) &= N - L_{AA} \end{aligned}$$

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 $p = \text{código_bij}(P)$ e simula P para a entrada p (como este programa pode ser definido?). Adicionalmente:

- $p \in \text{ACEITA}(Q)$ se, e somente se, $\langle P, \text{Norma} \rangle(p)$ é definido, ou seja, a computação de P em Norma é finita;
- $p \in \text{LOOP}(Q)$ se, e somente se, a computação de P em Norma é infinita;
- $\text{REJEITA}(Q) = \emptyset$ (consequência dos dois casos acima).

Portanto:

$$\begin{aligned} \text{ACEITA}(Q) &= L_{AA} \\ \text{REJEITA}(Q) \cup \text{LOOP}(Q) &= N - L_{AA} \end{aligned}$$

Logo, L_{AA} é enumerável recursivamente, e, portanto, o problema da autoaplicação é parcialmente solucionável. $\square$

teorema 9.8 – Problema da autoaplicação é não solucionável

A linguagem L_{AA} que traduz o problema da autoaplicação *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 9.6, tal que:

$ACEITA(Q) = L_{AA}$
 $REJEITA(Q) = N - L_{AA}$
 $LOOP(Q) = \emptyset$ (consequência dos dois casos acima)

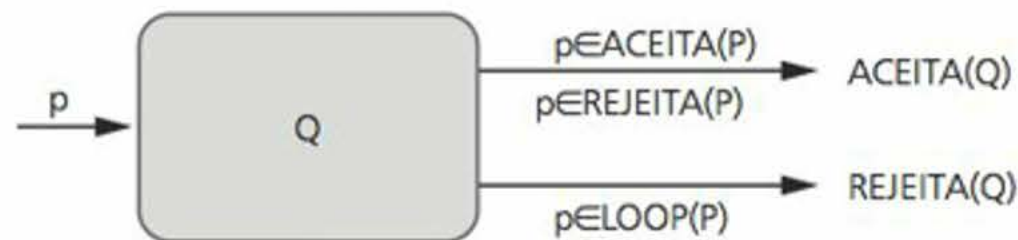


figura 9.6 Programa para a máquina Norma.

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 9.7, com a seguinte função:

- antes de terminar a computação, testa o valor da saída de Q ;

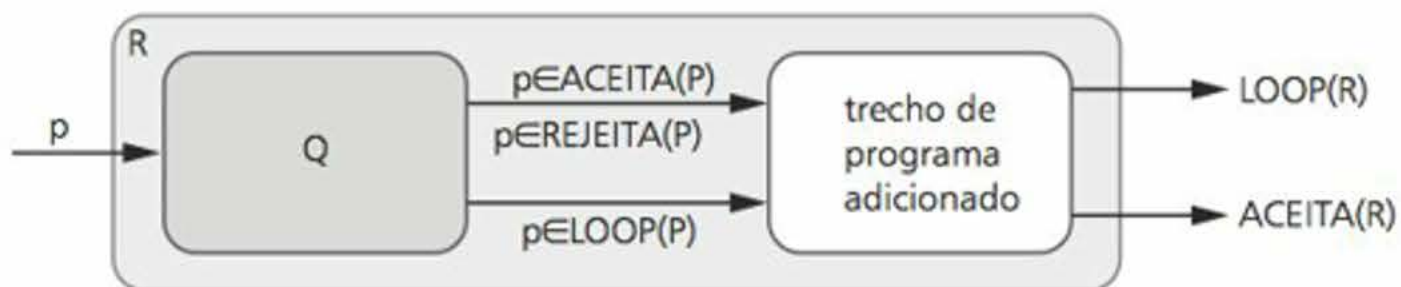


figura 9.7 Programa modificado para a máquina Norma.

- se Q aceita ou rejeita, então R fica em *loop* infinito;
- se Q fica em *loop* infinito, R aceita.

Assim, para a aplicação de R como entrada de R , tem-se que (suponha que $r = \text{código}(R)$):

- a** R fica em *loop* infinito quando Q , ao simular R , aceita ou rejeita. Ou seja, R fica em *loop* infinito quando R para;
- b** R para quando Q , ao simular R , fica em *loop* infinito. Ou seja, R para quando R fica em *loop* infinito.

Assim, fica caracterizada a contradição.

Logo, L_{AA} não é recursiva, e, portanto, o problema da autoaplicação é não solucionável. $\square$

9.5

→ máquina de redução

A *máquina de redução* implementa o princípio da redução e 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).

Neste capítulo, são supostas as seguintes generalizações:

- Função **código_bij** para qualquer máquina universal;
- Problema da autoaplicação para qualquer máquina universal.

Assim, sempre que possível, as definições e os resultados que seguem são independentes de formalismos.

definição 9.9 – 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 9.10 – 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 uma máquina de Turing de redução que sempre para e que reduz L_A a L_B , como ilustrado na figura 9.8.

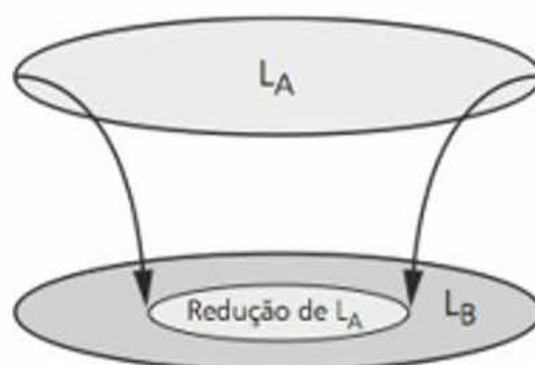


figura 9.8 Redução.

- a** Suponha que L_B é uma linguagem recursiva. Então existe M_B , máquina universal, que aceita L_B e sempre para para qualquer entrada. Seja a máquina universal M definida como na figura 9.9.

As seguintes conclusões podem ser estabelecidas:

- M sempre para 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 para para qualquer entrada. Logo, L_A é uma linguagem recursiva;

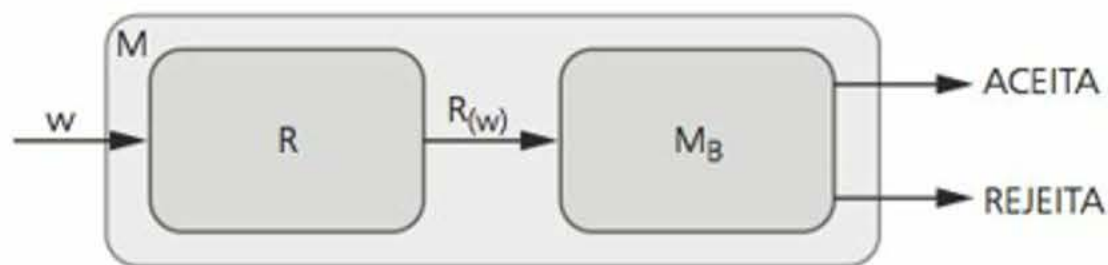


figura 9.9 Máquina universal.

- 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 9.10.

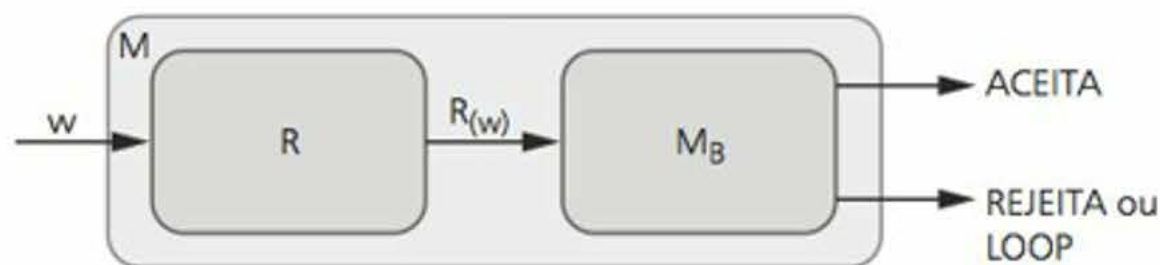


figura 9.10 Máquina universal.

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* infinito para a entrada w , pois M_B rejeita ou fica em *loop* infinito para a entrada $R(w)$.

Portanto, M aceita L_A , mas pode ficar em *loop* infinito 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 de que (suponha que p e q são proposições):

$$(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$$

□

9.6

→ problema da parada

Um dos mais importantes problemas não solucionáveis é conhecido como o *problema da parada*.

definição 9.11 – 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 verifica se M para, 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:

$$L_p = \{ (m, w) \mid m = \text{código_bij}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M) \}$$

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 9.12 – Problema da parada é não solucionável

A seguinte linguagem que traduz o problema da parada não é recursiva:

$$L_p = \{ (m, w) \mid m = \text{código_bij}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M) \}$$

→ **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:

$$\text{ACEITA}(M_p) = L_p$$

$$\text{REJEITA}(M_p) = \Sigma^* - L_p$$

$$\text{LOOP}(M_p) = \emptyset \text{ (consequência dos dois casos acima)}$$

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 9.11.

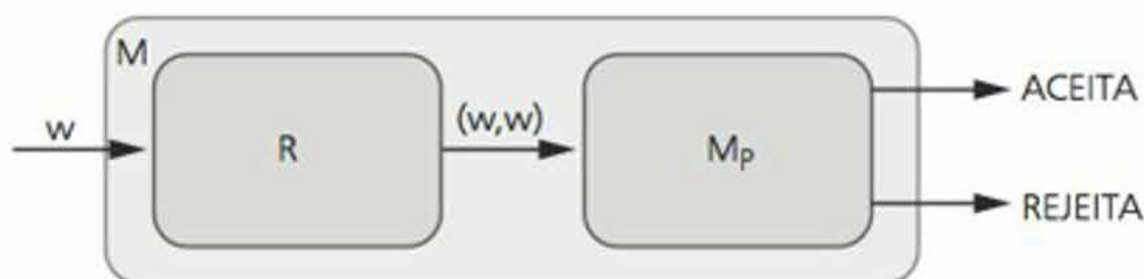


figura 9.11 Máquina construída usando o princípio da redução.

Claramente, o problema da autoaplicação foi reduzido ao problema da parada (figura 9.12), 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 9.10 – Redução – investigação da solucionabilidade, o problema da autoaplicação é solucionável, o que é um absurdo. Logo, é absurdo supor que o problema da parada é solucionável, e, portanto, é não solucionável. □

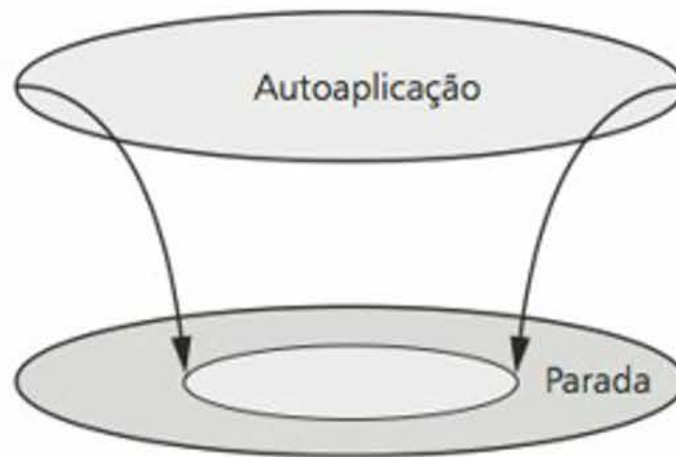


figura 9.12 Redução.

Analogamente ao problema da autoaplicação, 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 autoaplicação é parcialmente solucionável;
- usando o princípio da redução.

teorema 9.13 – Problema da parada é parcialmente solucionável

A seguinte linguagem que traduz o problema da parada é enumerável recursivamente:

$$L_p = \{ (m, w) \mid m = \text{código_bij}(M) \text{ e } w \in \text{ACEITA}(M) \cup \text{REJEITA}(M) \} \quad \square$$

9.7

→ 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, alguns destes problemas.

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 9.14 – 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 verifica se M para, aceitando ou rejeitando, ao processar a entrada vazia? □

A prova do seguinte teorema é sugerida como exercício e é genericamente mostrada na figura 9.13.

teorema 9.15 – Problema da parada da palavra vazia é não solucionável

A seguinte linguagem que traduz o problema da parada da palavra vazia não é recursiva:

$$L_\varepsilon = \{ m \mid m = \text{código_bij}(M) \text{ e } \varepsilon \in \text{ACEITA}(M) \cup \text{REJEITA}(M) \} \quad \square$$

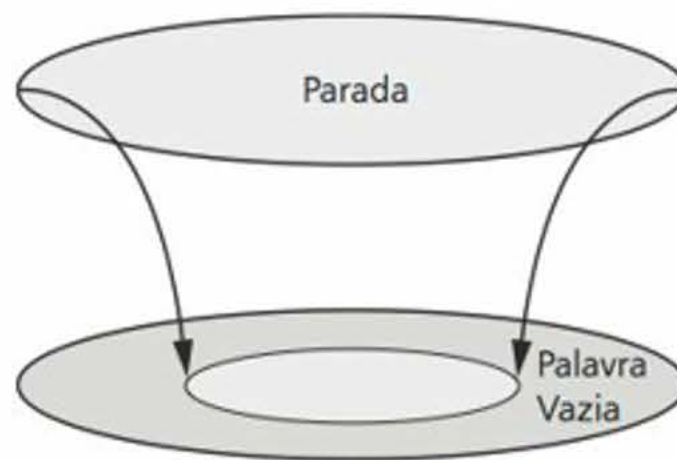


figura 9.13 Redução.

O *problema da totalidade* é uma variação do problema da parada, generalizado para toda entrada.

definição 9.16 – Problema da totalidade

O *problema da totalidade* é como segue:

Dada uma máquina universal M qualquer, existe um algoritmo que verifica se M para, aceitando ou rejeitando, ao processar qualquer entrada? □

A prova do seguinte teorema é sugerida como exercício e é genericamente mostrada na figura 9.14.

teorema 9.17 – Problema da totalidade é não solucionável

A seguinte linguagem que traduz o problema da totalidade não é recursiva:

$$L_T = \{ m \mid m = \text{código_bij}(M) \text{ e } \text{LOOP}(M) = \emptyset \} \quad \square$$

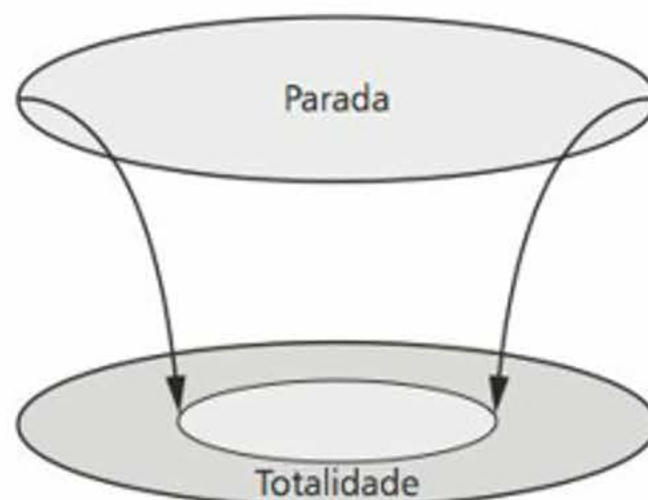


figura 9.14 Redução.

definição 9.18 – Problema da equivalência

O *problema da equivalência* é um problema de decisão (do tipo **sim/não**) que verifica a equivalência de duas máquinas universais. □

A prova do seguinte teorema é sugerida como exercício e é genericamente mostrada na figura 9.18.

teorema 9.19 – Problema da equivalência é não solucionável

A seguinte linguagem que traduz o problema da equivalência não é recursiva:

$$L_E = \{ (m, p) \mid m = \text{código_bij}(M), p = \text{código_bij}(P), \\ \text{ACEITA}(M) = \text{ACEITA}(P) \text{ e } \text{REJEITA}(M) = \text{REJEITA}(P) \}$$

□

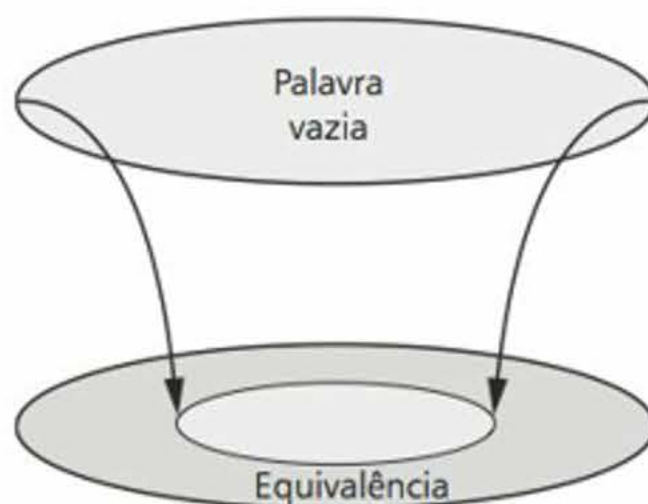


figura 9.15 Redução.

9.8

→ **problema da correspondência de Post**

O *problema da correspondência de 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 a equivalência de reconhecedores sintáticos de linguagens. O problema da correspondência de Post é definido sobre um *sistema de Post*.

definição 9.20 – 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 $i \in \{1, 2, \dots, n\}$:

$$S = \{ (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n) \}$$

Uma solução para um sistema de Post é uma sequência não vazia de números naturais com valores em $\{1, 2, \dots, n\}$:

$$i_1, i_2, \dots, i_k$$

tal que:

$$x_{i_1} x_{i_2} \dots x_{i_k} = y_{i_1} y_{i_2} \dots y_{i_k}$$

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 9.2 – 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 9.3 – 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 9.4 – 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 9.21 – Problema da correspondência de Post é não solucionável

A seguinte linguagem que traduz o problema da correspondência de Post *não* é recursiva:

$$L_{CP} = \{s \mid s = \text{código_bij}(S) \text{ e } S \text{ é um sistema de Post com pelo menos uma solução}\}$$

⇒ **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 de Post baseado na sequência de comandos executados por M para a entrada w , de tal forma que o sistema tenha solução se e somente se, M para 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 ideia 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 $w = a_1 a_2 \dots a_n$
- o valor inicial de X é w
- as componentes elementares do diagrama de fluxos de M são enumeradas sobre $\{ 1, 2, \dots, m \}$,

A relação entre os componentes elementares de M e os pares do correspondente sistema de Post é como segue (veja a figura 9.16):

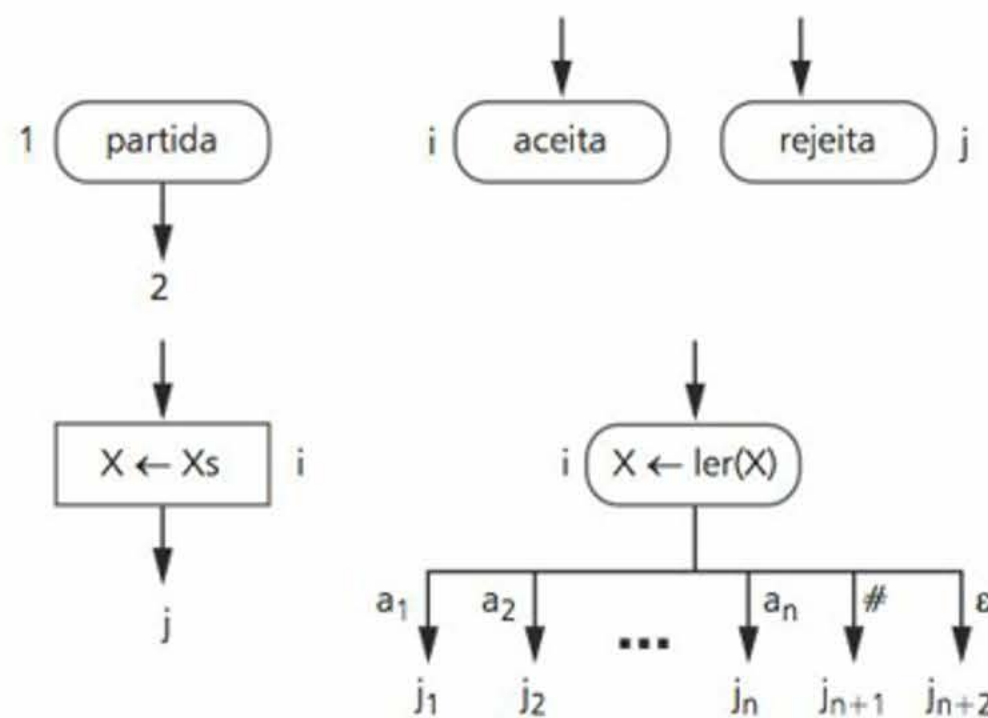


figura 9.16 Enumeração das instruções de um diagrama de fluxos de uma máquina de Post.

a *Partida.* Determina o seguinte par:

$$(1, 1 \ a_1 \ a_2 \dots a_n \ 2)$$

b *Desvio ou teste.* Determina os seguintes pares:

$$(i \ a_1, j_1), (i \ a_2, j_2), \dots, (i \ a_n, j_n), (i \ \#, j_{n+1}), (i \ \epsilon, j_{n+2})$$

c *Atribuição.* Determina o seguinte par:

$$(i, s \ j)$$

d *Parada.* Determina o seguinte par:

$$(i, \epsilon)$$

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 de Post somente tem solução se a máquina parar para a entrada w . Os demais detalhes são explicados por meio do exemplo que segue.

Portanto, é possível reduzir o problema da parada ao problema da correspondência de Post (figura 9.17).

Logo, o problema da correspondência de Post é não solucionável. □

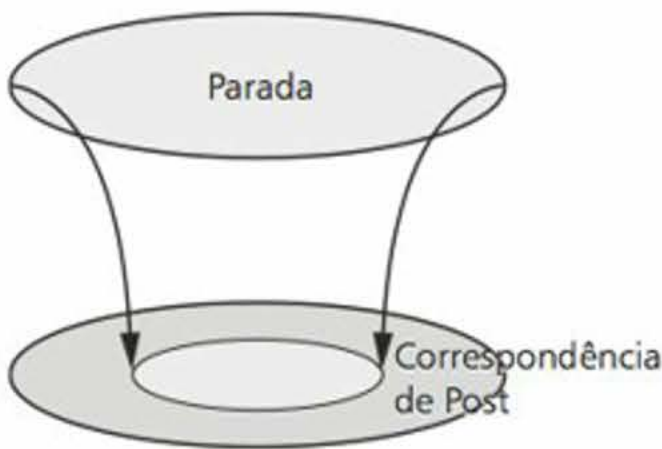


figura 9.17 Redução.

exemplo 9.5 – Máquina de Post → sistema de Post

Considere a máquina de Post **Post-Duplo_Bal** na figura 9.18 (introduzida anteriormente), a qual reconhece a linguagem $\text{Duplo_Bal} = \{ a^n b^n \mid n \geq 0 \}$, cujas instruções já estão enumeradas. Suponha a entrada $w = ab$. O sistema de Post correspondente, conforme o teorema 9.21, 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: (#, #)

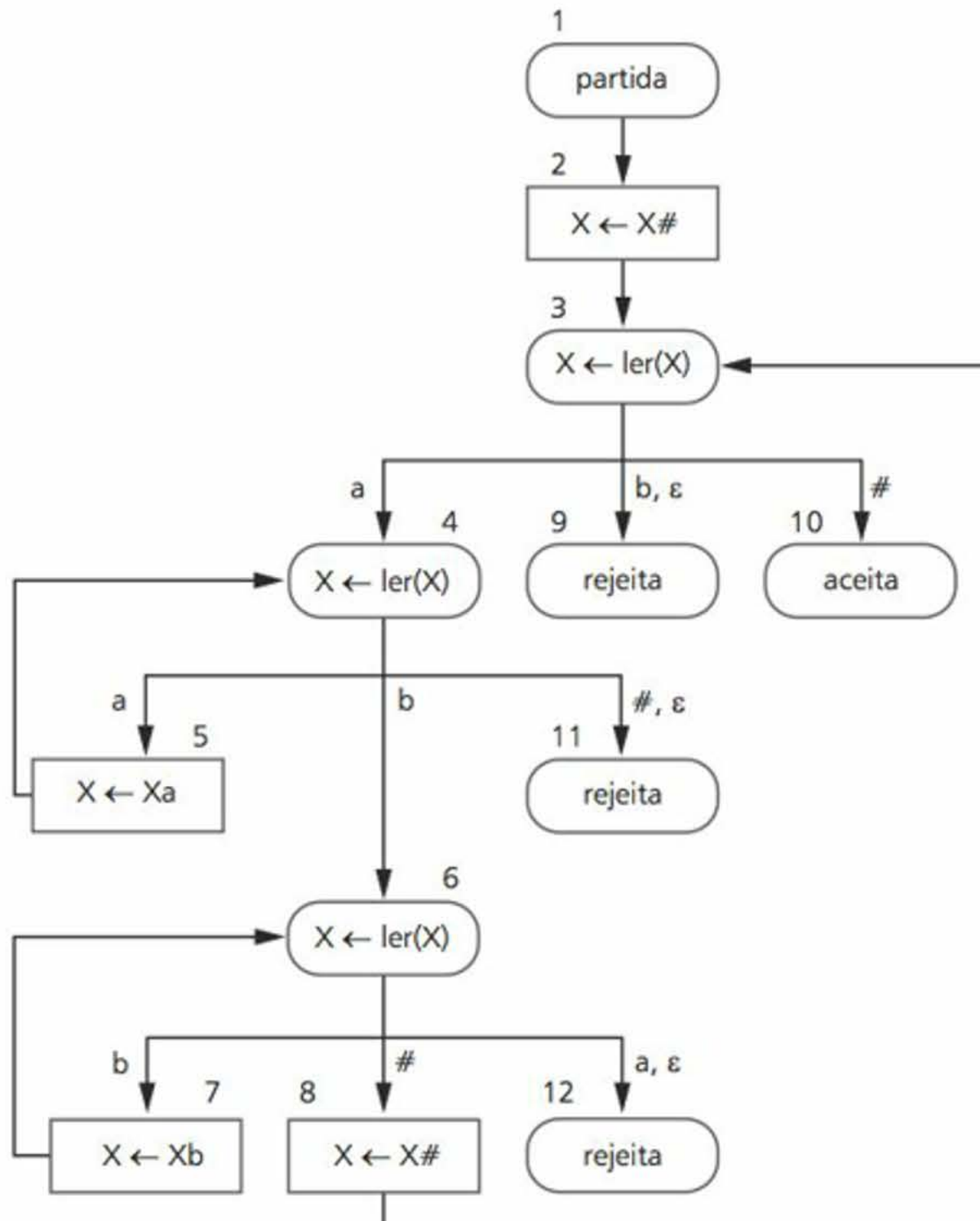


figura 9.18 Máquina de Post com as instruções enumeradas.

Uma solução para este sistema, a qual corresponde ao processamento da máquina de Post **Post-Duplo_Bal** para a entrada $w = ab$, é a sequência 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 =
= 1ab2 a b #3 a b # 4 b # 6 # 8 #3 # 10 ε

Repare que a informação contida entre dois números de instruções representa a evolução do valor da variável X : se o comando à esquerda é uma atribuição, corresponde ao valor da variável após a atribuição; se o comando à esquerda é um desvio, corresponde ao valor da variável antes do teste. □

9.9

→ propriedades da solucionabilidade

Relembrando, algumas das principais propriedades das linguagens enumeráveis recursivamente e das linguagens recursivas são:

- a** O complemento de uma linguagem recursiva é uma linguagem recursiva. Consequentemente, existe um algoritmo que sempre para 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.

A primeira propriedade já foi demonstrada no capítulo 5 – Máquina de Turing. Na segunda demonstração da propriedade, é usado o conceito de não determinismo. Para a demonstração da terceira propriedade, tem-se de apresentar uma linguagem que seja enumerável recursivamente e que não seja recursiva (não existe máquina que a reconheça e sempre pare).

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 parcialmente solucionáveis;
- a classe dos problemas solucionáveis está contida propriamente na classe dos problemas parcialmente solucionáveis.

As demonstrações que seguem são em termos de propriedades de linguagens. Relembrando, o seguinte teorema já foi demonstrado.

teorema 9.22 – O 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. $\square$

teorema 9.23 – 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 capítulo 5 – Máquina de Turing, $\Sigma^* - L$ 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:

$$\text{ACEITA}(M_1) = L$$

$$\text{ACEITA}(M_2) = \Sigma^* - L$$

Seja M uma máquina universal não determinística definida conforme o esquema ilustrado na figura 9.19 (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 para. Logo, L é recursiva. $\square$

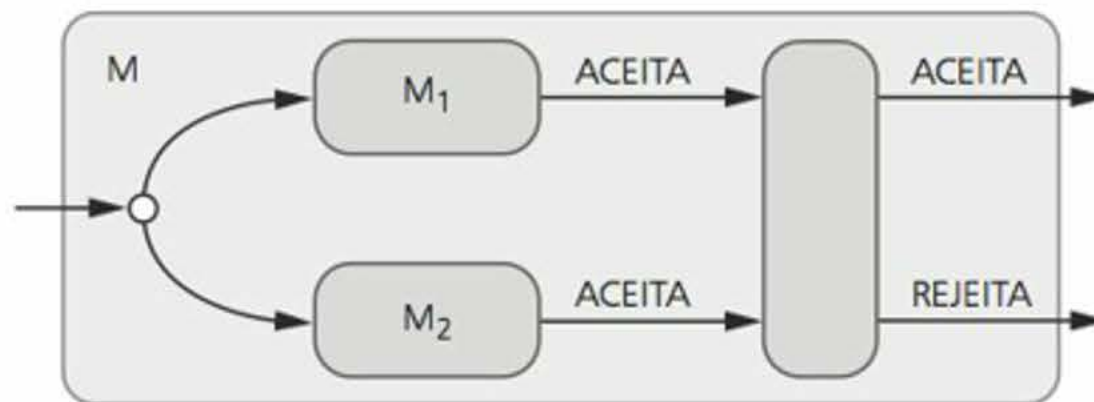


figura 9.19 Máquina universal não determinística.

teorema 9.24 – Linguagens recursivas $\subset$ enumeráveis recursivamente

A classe das linguagens recursivas está contida propriamente na classe das linguagens enumeráveis recursivamente.

➔ **Prova:**

Para mostrar que a inclusão é própria, basta mostrar que existe pelos menos uma linguagem enumerável recursivamente que não é recursiva. Lembre-se que:

- o problema da parada é parcialmente solucionável, ou seja, é enumerável recursivamente (teorema 9.13 – Problema da parada é parcialmente solucionável)
- o problema da parada é não solucionável, ou seja, não é recursivo (teorema 9.12 – Problema da parada é não solucionável).

Portanto, o problema da parada é não recursivo.

Logo, a classe das linguagens recursivas está contida propriamente na classe das linguagens enumeráveis recursivamente. $\square$

Importantes resultados podem ser obtidos a partir dessas propriedades, por exemplo, sobre o problema da parada (veja a figura 9.20 – Relação entre as classes de problemas, onde L_p é a linguagem associada ao problema da parada):

- o problema da parada é parcialmente solucionável;
- o problema da parada é não solucionável;
- portanto, o seu complemento, que é o *problema da não parada* (**Loop**), é não solucionável e completamente insolúvel, ou seja, não computável.



figura 9.20 Relação entre as classes de problemas.

Uma consequência imediata desse resultado é a inexistência de um algoritmo genérico para identificar *loops* infinitos em sistemas.

Da mesma forma, prova-se que o problema da não correspondência de Post (complemento do problema da correspondência de Post) é completamente insolúvel. De fato, demonstrou-se que o problema da correspondência de Post pode ser reduzido ao problema da parada, o qual é parcialmente solucionável e, também, não é solucionável. Em termos de linguagens, é enumerável recursivamente, mas não é recursivo. Assim, se ele fosse parcialmente solucionável, estaria satisfeita a hipótese da segunda propriedade e implicaria que ambos seriam solucionáveis, caindo em contradição. Logo, o problema da não correspondência de Post é não computável.

Sugere-se como exercício a aplicação do mesmo raciocínio acima para a demonstração de que os complementos dos seguintes problemas são completamente insolúveis:

- Definição 9.14 - Problema da parada da palavra vazia;
- Definição 9.16 - Problema da totalidade;
- Definição 9.18 - Problema da equivalência.

observação 9.25 – Cardinalidade das classes de problemas

Considerando que:

- os problemas computáveis são caracterizados pela existência de um algoritmo implementado na forma de um programa para uma máquina universal;
- é possível estabelecer uma função bijetora entre o conjunto de todos os programas e o conjunto dos números naturais (exemplo 9.1 - Codificação bijetora de programas);

conclui-se que o cardinal da classe dos problemas computáveis é igual ao cardinal dos números naturais, ou seja, é contável. Assim:

o cardinal da classe dos problemas computáveis é contável.

É fácil definir um conjunto não contável, como o conjunto de todas as funções nos inteiros (que caracterizam o universo “discreto”) e o conjunto de todas as funções nos reais (que caracterizam o universo “contínuo”). Assim:

existem classes de problemas que não são contáveis.

Portanto, o cardinal da classe dos problemas não computáveis é não contável. Assim, informalmente, pode-se afirmar que:

Existem muito mais problemas não computáveis do que computáveis. □

9.10

→ conclusões

O estudo da computabilidade objetiva determinar a solucionabilidade de problemas (existência de algoritmos). Portanto, investiga os limites do que pode ser implementado em um computador, evitando a pesquisa de soluções inexistentes.

A abordagem é centrada nos problemas de decisão (do tipo **sim/não**). A investigação da solucionabilidade é suportada por:

- Princípio da redução, o qual consiste na investigação da solucionabilidade de um problema a partir de outro, cuja classe de solucionabilidade é conhecida;
- Propriedades da solucionabilidade, as quais podem ser aplicadas para inferir a solucionabilidade ou não, conhecidas determinadas condições.

e permite, entre outros resultados, identificar classes de problemas:

- Solucionáveis: existe um algoritmo capaz de responder **sim** ou **não** e que sempre para;
- Parcialmente solucionáveis (computáveis): existe um algoritmo capaz de responder **sim**, embora, eventualmente, possa ficar processando indefinidamente para uma resposta que deveria ser **não**;
- E os respectivos complementos não solucionáveis e completamente insolúveis (não computáveis).

Um importante resultado deste estudo é que, informalmente, a classe dos problemas não computáveis é “muito maior” que o da classe dos problemas computáveis, ou seja, existem muito mais problemas não computáveis do que computáveis.

9.11

→ exercícios

exercício 9.1 Elabore uma linha de tempo sobre o desenvolvimento do conceito de computabilidade (use como base o correspondente exercício do capítulo 1 – Instrução).

exercício 9.2 Defina a relação entre as seguintes classes de problemas:

- a** Solucionáveis;
- b** Parcialmente solucionáveis (computáveis);
- c** Não solucionáveis;
- d** Completamente insolúveis (não computáveis).

exercício 9.3 Descreva uma sistemática genérica para traduzir problemas em linguagens.

exercício 9.4 Qual a importância do problema da autoaplicação no estudo da solucionabilidade de problemas?

exercício 9.5 Quais as ideias básicas do princípio da redução?

exercício 9.6 Desenvolva um algoritmo de **decodificação** (função inversa da **codificação**).

exercício 9.7 Esboce um programa que recebe como entrada $p = \text{código_bij}(P)$ e simula P para a entrada p .

exercício 9.8 Demonstre o teorema 9.15 - Problema da parada da palavra vazia é não solucionável. É parcialmente solucionável? E o seu complemento, como se classifica?

exercício 9.9 Demonstre o teorema 9.17 - Problema da totalidade é não solucionável. É parcialmente solucionável? E o seu complemento, como se classifica?

exercício 9.10 Demonstre o teorema 9.19 - Problema da equivalência é não solucionável. É parcialmente solucionável? E o seu complemento, como se classifica?

exercício 9.11 Demonstre que o *problema da parada da palavra constante* é não solucionável.

exercício 9.12 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 9.13 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 9.14 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 9.15 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 9.16 Seja **M** uma máquina universal não determinística definida conforme o esquema ilustrado na figura 9.19. Como seria a definição de **M**? A resposta pode ser específica para qualquer dos formalismo estudados.

exercício 9.17 Sobre o problema da correspondência de Post:

- a** Encontre a menor solução para os seguintes sistemas de Post:
 $S_1 = \{ (b, bbb), (babbb, ba), (ba, a) \}$
 $S_2 = \{ (\epsilon, a), (a, b), (b, aa), (aa, ab), (ab, ba), (ba, bb), (bb, \epsilon) \}$
- 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 sequência de 66 índices.

- c** Mostre que o seguinte sistema de Post não tem solução:
 $S_4 = \{ (ba, bab), (abb, bb), (bab, abb) \}$

exercício 9.18 Suponha **S** um sistema de Post com solução. Desenvolva um algoritmo que determine a menor sequência de índices que seja solução de **S**.

exercício 9.19 Marque a alternativa correta:

- a** Um problema não solucionável é aquele que não pode ser computável;
- b** Um problema é computável se existe um programa, em uma máquina de Turing, que para qualquer entrada. A computação sempre é finita, pois sempre para;
- c** Um problema parcialmente solucionável é aquele que tem partes que não se consegue resolver;
- d** Um problema é dito parcialmente solucionável se existe um programa em uma máquina de Turing que solucione o problema tal que para quando a resposta esperada é negativa (rejeitada);
- e** Um problema é computável quando ele é parcialmente solucionável.

exercício 9.20 Considere o seguinte:

- “O problema **A** é solucionável”;
- “O problema **B** é parcialmente solucionável. O complemento de **B** é **A**. O problema **C** pode ser reduzido ao problema **B**”.

O que pode dizer sobre **C**? Marque a alternativa correta:

- ☐ **a** É solucionável;
- ☐ **b** É não solucionável;
- ☐ **c** É parcialmente solucionável, mas não solucionável;
- ☐ **d** Seu complemento é não solucionável;
- ☐ **e** Não faz parte do complemento de **A**.

exercício 9.21 Numere a segunda coluna de acordo com a primeira:

I. Classe dos problemas solucionáveis	() Cardinal não contável
II. Classe dos problemas parcialmente solucionáveis	() Linguagens recursivas
III. Classe dos problemas completamente insolúveis	() Linguagens enumeráveis recursivamente

A numeração correta, de cima para baixo, é:

- ☐ **a** I, II, III
- ☐ **b** III, II, I
- ☐ **c** III, I, II
- ☐ **d** II, I, III
- ☐ **e** II, III, I

exercício 9.22 Analise as seguintes afirmações e marque cada uma com **V** (verdadeira) ou **F** (falsa):

- () A união das classes dos problemas solucionáveis com o complemento da classe dos problemas completamente insolúveis é o universo de todos os problemas;
- () A união das classes dos problemas solucionáveis, dos problemas computáveis, dos problemas parcialmente solucionáveis com a dos problemas não computáveis é o universo de todos os problemas;
- () Todo problema solucionável é parcialmente solucionável;
- () Não existem problemas não solucionáveis que possuam solução parcial.

A marcação correta, de cima para baixo, é:

- a** V, V, V, F
- b** F, V, V, F
- c** V, F, F, V
- d** F, V, F, F
- e** F, F, V, V

exercício 9.23 Analise as seguintes afirmações:

- I. A classe dos problemas solucionáveis é equivalente à classe das linguagens recursivas;
- II. Existe um algoritmo genérico que sempre para capaz de comparar quaisquer dois reconhecedores de linguagens livres do contexto, como Pascal, e verificar se são equivalentes, ou seja, se de fato, reconhecem a mesma linguagem;
- III. Dados um programa e uma entrada quaisquer, não existe algoritmo genérico capaz de verificar se o programa vai parar ou não para a entrada.

Marque a alternativa correta:

- a** Apenas a I está correta;
- b** Apenas a II está correta;
- c** Apenas a III está correta
- d** Apenas I e III estão corretas
- e** I, II e III estão corretas.

Termos-chaves

classe dos problemas computáveis, p. 248

classe dos problemas não computáveis,
p. 248

classe dos problemas parcialmente
solucionáveis, p. 248

máquina de redução, p. 256

problema completamente insolúvel, p. 250

problema da autoaplicação, p. 253, p. 254

problema da correspondência de Post, p.
261, p. 262

problema da equivalência, p. 261

problema da não parada, p. 267

problema da parada, p. 247, p. 258

problema da parada da palavra vazia, p. 259

problema da totalidade, p. 260

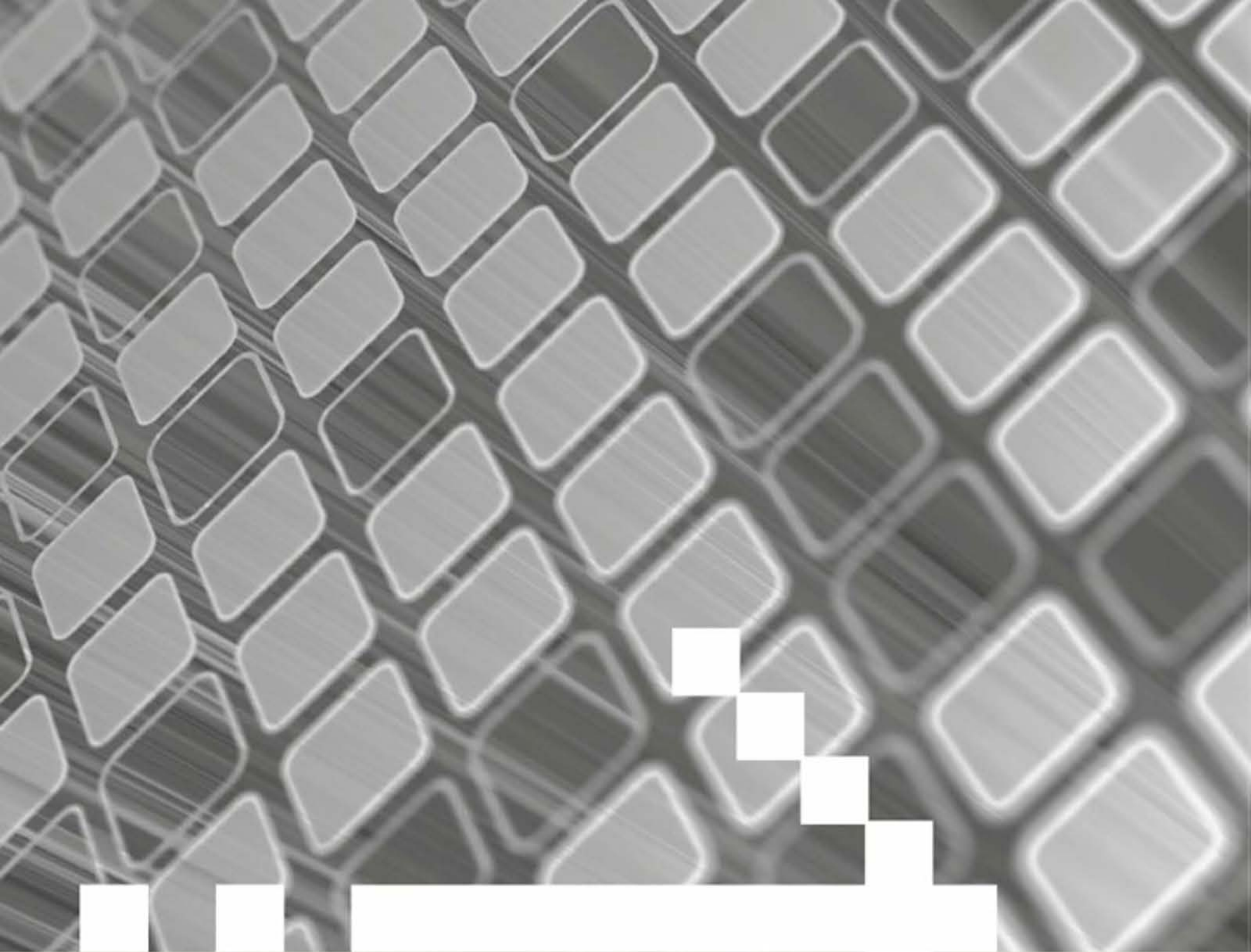
problema de decisão, p. 247, p. 251

problema não solucionável, p. 250

problema parcialmente solucionável,
p. 248, p. 250

problema solucionável, p. 250

sistema de Post, p. 261

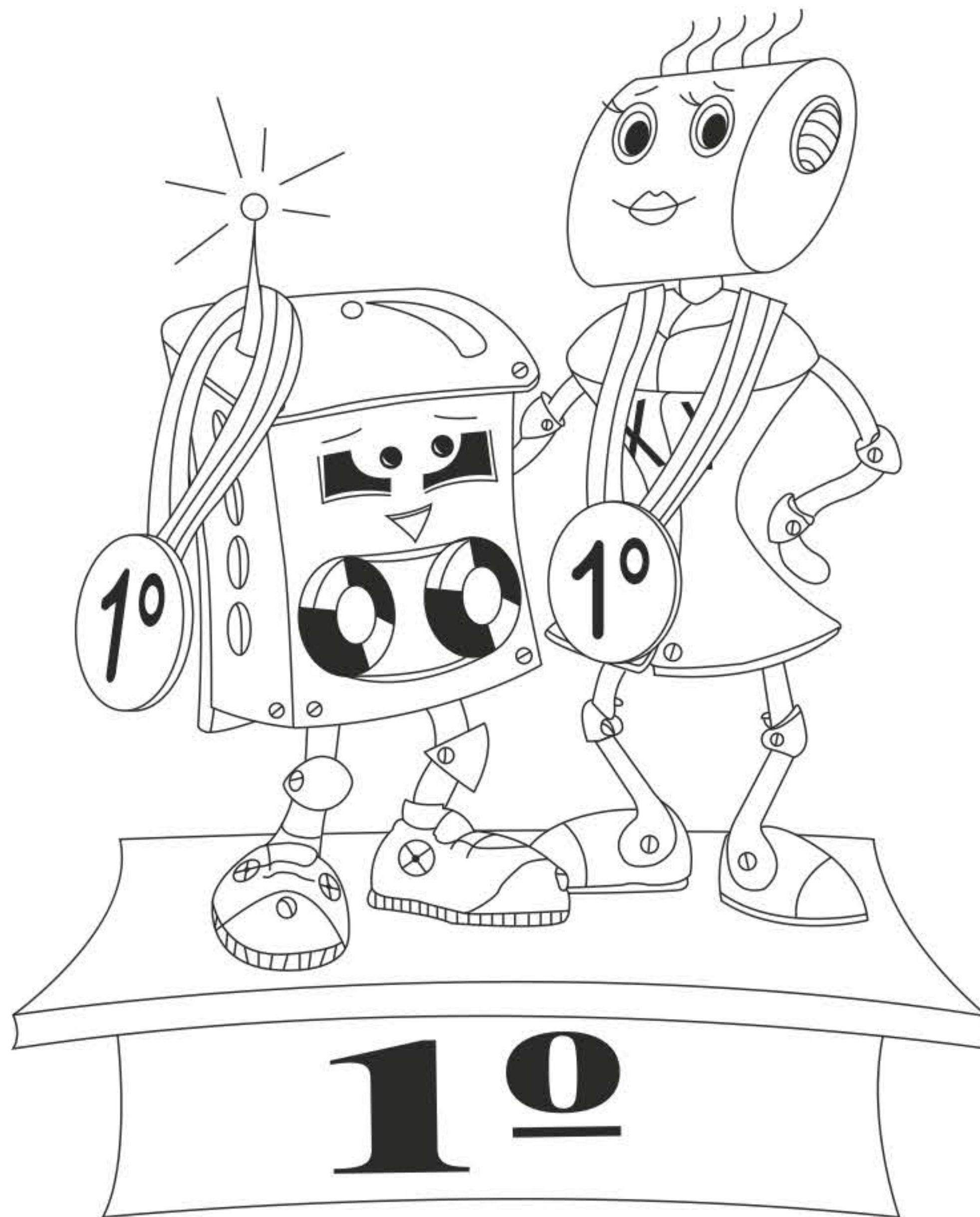


capítulo

10

conclusões

- ■ Apresenta um resumo dos conceitos fundamentais desenvolvidos ao longo do livro, visando a uma reflexão abrangente sobre abstração, formalização e investigação de solucionabilidade e dos limites do que pode ser efetivamente resolvido em um computador, enfim, do que é computável.



A abordagem adotada neste livro desenvolve os principais aspectos de teoria da computação, combinando abordagens históricas com abordagens próximas dos sistemas de 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 e futura.

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 modelos. 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 equivalentes 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 solucionáveis (computáveis) e problemas completamente insolúveis (não computáveis).

10.1**→ 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, podem-se fazer comparações entre programas e entre máquinas e definir-se 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 nessa máquina. Se esses programas são equivalentes em qualquer máquina, então eles são fortemente equivalentes.

Utiliza-se o conceito de equivalência de programas para eliminar instruções desnecessárias e para otimizar o programa. Dessa 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 desta que faça a mesma coisa. Se duas máquinas simulam-se mutuamente, é porque elas são equivalentes. Nesse 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.

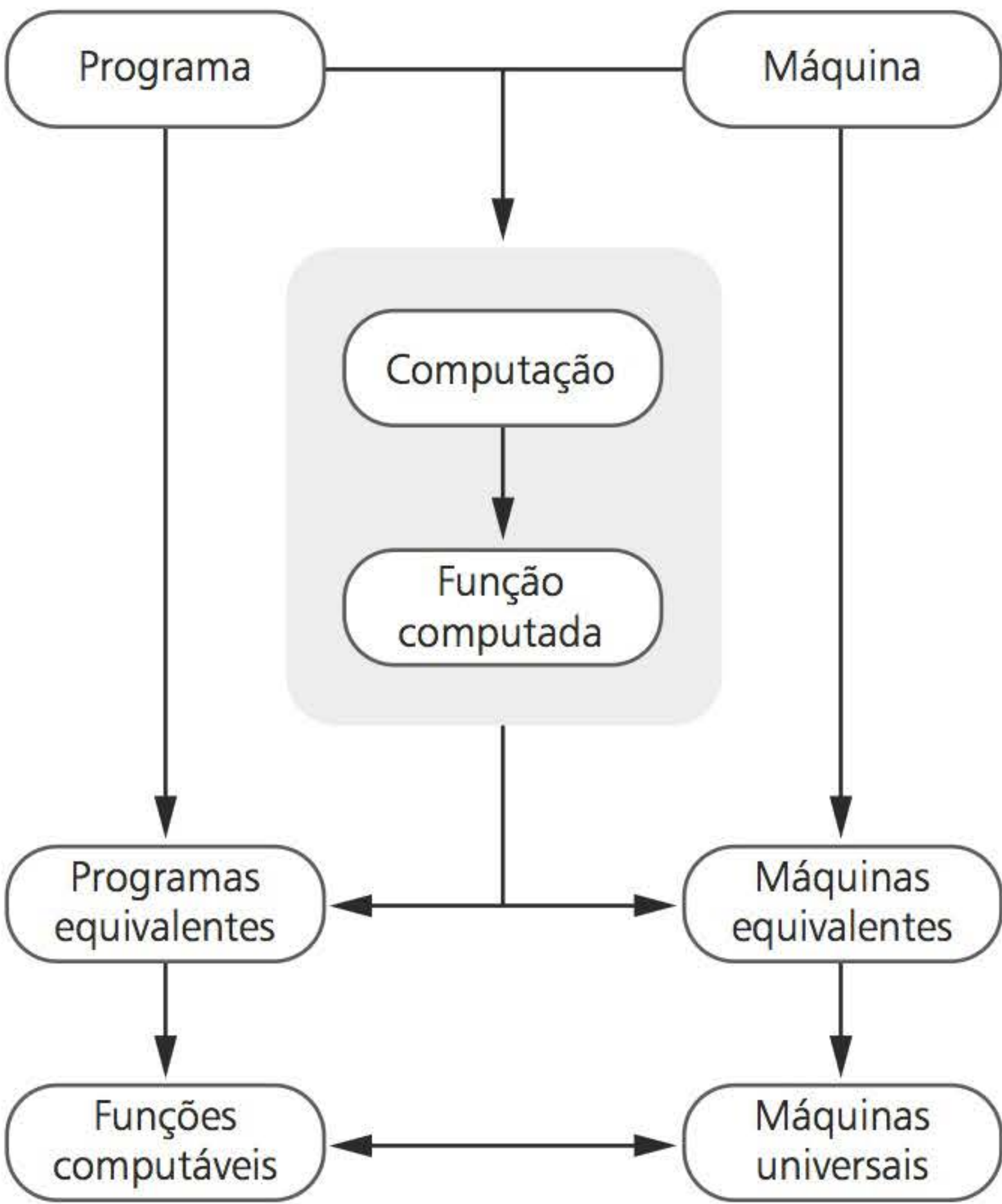


figura 10.1 Conceitos básicos desenvolvidos.

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 computável puder ser executada nela. E, segundo a tese de Church, diz-se que uma função computável é aquela que pode ser processada numa máquina de Turing ou equivalente. Assim, não se conseguem demonstrar tais afirmações, devido ao fato de a noção de algoritmo ser algo intuitivo. Entretanto, diversas evidências (muitas delas exploradas ao longo deste livro) fortalecem tal tese como uma hipótese da teoria da computação clássica.

Os problemas computáveis foram caracterizados como problemas parcialmente solucionáveis (não confundir com a ideia de que apenas parte do problema pode ser resolvido), ou seja, problemas que podem ser implementados na máquina de Turing (ou modelo equivalente) e que sempre reconhecem a linguagem associada ao problema (computação finita). Entretanto, podem existir palavras não pertencentes à linguagem que produzem uma computação infinita (como um *loop* infinito). Portanto, para qualquer palavra gerada do alfabeto da linguagem, nem sempre é decidível se a palavra pertence ou não à linguagem. Isso corresponde à linguagem enumerável recursivamente.

Nos problemas solucionáveis, sempre se pode decidir se uma palavra pertence ou não a linguagem associada, pois existe uma máquina de Turing que reconhece a linguagem (para acei-

tando a palavra) e que rejeita todas as palavras que não pertencem à linguagem (e, portanto, em tal máquina, o conjunto das palavras que a deixam em *loop* é vazio). Logo, são sinônimos de linguagens recursivas.

Uma vez estabelecida a relação dos problemas com as linguagens associadas, foram verificadas três importantes propriedades de linguagens para problemas. Destaca-se que o complemento de um problema solucionável é solucionável e que um problema é solucionável se, e somente se, ele e seu problema complementar são parcialmente solucionáveis. Estas duas propriedades são frequentemente usadas para estabelecer a solucionabilidade de problemas, juntamente com as definições das classes de solucionabilidade e com o princípio de redução (investigação da solucionabilidade de um problema a partir de outro, cuja classe de solucionabilidade é conhecida).

Da mesma forma, a demonstração de que um problema é completamente insolúvel ou não computável frequentemente implica o uso destas propriedades e a redução de problemas.

Como visto, o problema do detector universal de *Loops* (ou o problema da não parada - complemento do problema da parada) é não computável. Da mesma forma, prova-se que o problema da não correspondência de Post (complemento do Problema da Correspondência de Post) é não computável. Demonstrou-se que o problema da correspondência de Post pode ser reduzido ao problema da parada, o qual é parcialmente solucionável e, também, é não solucionável. Em termos de linguagens, é enumerável recursivamente, mas não é recursivo. Assim, se ele fosse parcialmente solucionável, estaria satisfeita a hipótese da segunda propriedade e implicaria que ambos seriam solucionáveis, caindo em contradição. Logo, o problema da não correspondência de Post é não computável.

Uma importante conclusão é que o cardinal da classe dos problemas computáveis é contável, enquanto o da classe dos problemas não computáveis é não contável. Ou seja, informalmente, pode-se afirmar que existem muito mais problemas não computáveis do que computáveis.

10.2

→ contribuições da teoria da computação

A 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, 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 por meio de suas características e propriedades possibilita que se identifiquem classes, as quais podem agrupar esses objetos. Os objetos herdam as pro-

priedades 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). Portanto, são temas abordados em análise e desenvolvimento de algoritmos e em complexidade de algoritmos;
- o não determinismo é um conceito introduzido que tem grande impacto na formação dos bacharéis em ciência da computação, em sistemas de informação e em engenharia da computação, pois é o primeiro tipo de composição não sequencial 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, 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-se 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;

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 independentemente de estruturas. Ou seja, permite tratar propriedades independentemente de implementação, o que constitui uma vantagem óbvia para a ciência da computação. Um exemplo típico dessa capacitação, abordado neste livro, é o uso do princípio da redução de problemas.

O aluno que acompanhou de forma satisfatória o conteúdo tratado possui formação teórica adequada para entender os princípios que constituem a ciência da computação e, portanto, de absorver as tecnologias atuais, bem como as evoluções e mudanças tecnológicas que porventura venham a ocorrer.

Termos-chaves

complexidade computacional, p. 280

complexidade estrutural, p. 280

engenharia da computação, p. 280

orientação a objetos, p. 280

otimização de algoritmos, p. 280

sistemas de informação, p. 280

leituras sugeridas

- AHO, A.; ULLMAN, J. *Foundations of computer science*. New York: W. H. Freeman, 1994. (Principles of Computer Science Series).
- ARBIB, M.; KFOURI, A.; MOLL, R. *A basis for theoretical computer science*. Berlin: Springer Verlag, 1981. (Monographs in Computer Science / The AKM Series in Theoretical Computer Science).
- BRAINERD, W. S.; LANDWEBER, L. H. *Theory of computation*. New York: John Wiley & Sons, 1974.
- BROOKSHEAR, J. G. *Teoría de la computación: lenguajes formales, autómatas y complejidad*. Wilmington: Addison Wesley Longman, 2000.
- CLARK, K.; COWELL, D. *Programs, machines and computation: an introduction to theory of computing*. London: McGraw-Hill, 1976.
- COHEN, D. I. A. *Introduction to computer theory*. New York: John Wiley & Sons, 1997.
- DEDEKIND, R. *Was sind und was sollen die Zahlen?* Braunschweig: Vieweg, 1888.
- EVEY, R. J. *The theory and applications of pushdown store machines: mathematical linguistics in machine translation*. Cambridge: Harvard Computation Laboratory, 1963. Report NSF-10.
- HAREL, D. *Algorithmics: the spirit of computing*. 3rd ed. Reading: Addison Wesley, 2004.
- HAREL, D. *Computers Ltd.: what they really can't do*. Oxford: Oxford University Press, 2003.
- HOPCROFT, J.; ULLMAN, J. *Introduction to automata theory, languages, and computation*. 3rd ed. Reading: Addison Wesley, 2006.
- JENSEN, K.; WIRTH, N. *Pascal user manual and report*. Berlin: Springer-Verlag, 1974.
- KNUTH, D. *Art of computer programming*. 3rd ed. Reading: Addison-Wesley, 1997. v. 1.
- LEWIS, H.; PAPADIMITRIOU, C. H. *Elements of the theory of computation*. 2nd ed. Upper Saddle River: Prentice-Hall, 1997.
- LUCCHESI, C. et al. *Aspectos teóricos da computação*. Rio de Janeiro: IMPA; UFRJ, 1979. (Projeto Euclides).
- MANNA, Z. *Mathematical theory of computation*. New York: Dover Publications, 2003.
- MENEZES, P. B. *Linguagens formais e autómatos*. 6. ed. Porto Alegre: Bookman, 2010. (Série Livros Didáticos do Instituto de Informática da UFRGS, n. 3).
- MENEZES, P. B. *Matemática discreta para computação e informática*. 3. ed. Porto Alegre: Bookman, 2010. (Série Livros Didáticos do Instituto de Informática da UFRGS, n. 19).
- MENEZES, P. B.; HAEUSLER, E. H. *Teoria das categorias para Ciência da Computação*. 2. ed. Porto Alegre: Bookman, 2008. (Série Livros Didáticos do Instituto de Informática da UFRGS, n. 12).

- MENEZES, P. B.; SERNADAS, A.; COSTA, J. Nonsequential automata semantics for a concurrent object-based language. *Electronic Notes in Theoretical Computer Science*, v. 14, p. 245-273, 1998.
- MENEZES, P. B.; TOSCANI, L. V.; LÓPEZ, J. G. *Aprendendo matemática discreta com exercícios*. Porto Alegre: Bookman, 2009. (Série Livros Didáticos do Instituto de Informática da UFRGS, n. 16).
- MINSKY, M. L. *Computation: finite and infinite machines*. Englewood Cliffs: Prentice Hall, 1967.
- OETTINGER, A. G. Automatic syntactic analysis and the pushdown store. In: SYMPOSIA IN APPLIED MATHEMATICS, 1961, Providence. *Proceedings...* Providence: American Mathematical Society, 1961. v. 12.
- PLUM, T. *Learning to program in C*. 2nd ed. Kamuela: Plum Hall, 1989.
- POST, E. Finite combinatory process: formulation I. *The Journal of Symbolic Logic*, v. 1, n. 3, p. 103-105, 1936.
- SAGASTUME, M.; BAUM, G. *Problemas, lenguajes y algoritmos*. Campinas: EBAI; Unicamp, 1986.
- SCHÜTZEMBERGER, M. P. On context-free languages and pushdown automata. *Information and Control*, v. 6, p. 246-264, 1967.
- SERNADAS, C. *Introdução à teoria da computação*. Lisboa: Editorial Presença, 1993.
- SHOENFIELD, J. R. *Recursion theory*. Berlin: Springer-Verlag, 1993. (Lecture Notes in Logic, v. 1). Reimpresso por A K Peters/CRC Press em 2001.
- SIPSER, M. *Introduction to the theory of computation*. 2nd ed. Boston: Course Technology, 2005.
- STOY, J. F. *Denotational semantics: the scott-strachey approach to programming language theory*. Cambridge: MIT Press, 1981. (Lecture Notes in Logic).
- TAYLOR, R. G. *Models of computation and formal languages*. New York: Oxford University Press, 1997.
- TOSCANI, L. V.; VELOSO, P. A. S. *Complexidade de algoritmos*. 2. ed. Porto Alegre: Bookman, 2008. (Série Livros Didáticos do Instituto de Informática da UFRGS, n. 13).
- WEGNER, P. *Programming language, information structure and machine organization*. New York: McGraw-Hill, 1968.

Citadas

- BIRD, R. *Programs and machines: an introduction to the theory of computation*. London: John Wiley & Sons, 1976. (Wiley Series in Computing).
- CHURCH, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, v. 58, n. 2, 1936.
- CLARKE, A. C. *3001: a odisséia final*. Bonsucesso: Nova Fronteira, 1997.
- FERREIRA, A. B. H. *Novo dicionário da língua portuguesa*. Bonsucesso: Nova Fronteira, 1984.
- GÖDEL, K. On formally undecidable propositions in principia mathematica and related systems. In: DAVIS, M. *The undecidable*. New York: Raven Press, 1965. p. 4-38.
- HILBERT, D. Mathematical problems: lecture delivered before the International Congress of Mathematicians at Paris in 1900. In: BROWDER, F. E. *Mathematical developments arising from Hilbert problems: proceedings of symposia in pure mathematics*. Providence: American Mathematical Society, 1976. p. 1-34, v. 28.
- KLEENE, S. C. Representation of events in nerve nets and finite automata. In: SHANNON, C. E.; MCCARTHY, J. (Ed.). *Automata studies*. Princeton: Princeton University Press, 1956.
- TURING, A. M. On computable numbers, with an application to the entscheidungsproblem. In: LONDON MATHEMATICAL SOCIETY, 1936, London. *Proceedings...* London: LMS, 1936.

A

Abstração lambda, 223
 Aceita, 138, 141, 142, 184, 185, 191, 192
 Ackermann (função), 241
 Adição (programa), 167
 Alfabeto, 24, 26
 auxiliar, 135, 197
 de símbolo de entrada, 134, 184, 191, 197
 Alfa (redução), 229, 230
 Algol (linguagem), 41, 141
 Algoritmo, 23, 85, 104, 132, 175
 de equivalência forte de programa monolítico, 92
 de Markov, 24
 de simplificação de ciclo infinito, 90
 para traduzir um fluxograma, 85
 Aplicação lambda, 224
 Arranjo unidimensional, 119
 Assembly ver Linguagem de montagem
 Associativa (concatenação), 28
 Associatividade à esquerda, 225
 Até (composição), 41, 42, 62
 Atribuição, 184, 185
 Autômato
 com duas pilhas, 182, 196, 197, 200, 201
 com pilha, 196-201
 com pilha não determinístico, 202, 215
 sem pilha, 215
 Auxiliar (símbolo), 184
 Axiomático (formalismo), 214

B

Base
 de indução, 83
 pilha, 119
 Bird, Richard, 25, 80, 105, 108
 Bit, 26
 Booleano
 de Church, 231
 tipo de dado, 231
 Branco (símbolo), 135

C

C (linguagem), 41, 204, 237
 Cabeça
 da fita, 134, 137, 160, 196
 da pilha, 196
 Cadeia
 de caractere, 25, 110, 121
 de conjunto, 89
 de símbolo, 26
 finita de conjunto, 89
 vazia, 26
 Cálculo lambda, 24, 29, 215, 223-237
 aspectos gerais, 223, 224
 e computabilidade, 235-237
 igualdade induzida por redução beta, 236
 Caractere, 25, 26
 Cardinalidade das classes de problemas, 268
 Church
 Alonzo, 24, 29, 158, 215, 247
 booleano de, 231
 hipótese de, 175
 numerais de, 232
 tese de, 158, 175
 Ciência da computação, 22
 Classe de solucionabilidade de problemas, 250, 251
 Codificação, 161, 171
 de conjuntos estruturados, 105-108
 de programas, 252, 253
 função de, 66
 Combinação, 175
 Compilador, 142
 Completamente insolúvel (problema), 250
 Composição
 até, 41, 42, 62
 concorrente, 36
 condicional, 41
 expressão de sub-rotina, 44
 programa iterativo, 40, 41, 62

enquanto (programa iterativo), 41, 62
 escolha, 35
 não determinística, 35
 sequencial, 35, 41, 61
 expressão de sub-rotina, 44
 programa iterativo, 41
 sucessiva de funções, 58
 Comprimento (palavra), 26
 Computabilidade, 245-273
 Computação, 22, 34, 45, 48-57
 da máquina de Turing, 148
 do programa monolítico, 49
 do programa recursivo, 51
 e função computada, 48-57
 finita, 49, 52
 infinita, 49, 52
 Computável, 250
 Concatenação
 associativa, 28
 elemento neutro, 28
 palavra, 28
 sucessiva, 29
 Concorrência
 falsa, 36
 modelo para, 36
 Concorrente (composição), 36
 Condição de parada, 143
 Condicional (composição), 41
 Configuração inicial, 49, 51
 Conjunto
 contável, 104
 de estado, 134, 197
 final, 134, 197
 de interpretação
 de operação, 109
 de teste, 109
 de todas as palavras, 27
Contractum, 229

D

Dado estruturado, 110, 117-120
 indexação direta, 117
 indexação indireta, 118
 Decidível (problema), 250
 Decodificação, 161, 171
 Decodificação (função de), 66
 Definição da máquina norma, 108-110
 Denotacional (formalismo), 214
 Descrição instantânea (máquina), 49, 51
 Desempilha, 120, 190, 191
 Desvio, 184, 185, 190, 191
 Detector universal de *loops*, 248
 Diagrama de fluxo, 184, 191

E

Elemento
 absorvente, 30
 inverso, 30
 neutro (concatenação), 28
 Empilha, 119, 190-192
 Endereçamento indireto e recursão, 110, 120, 121
 Enquanto (composição), 41, 62
Entscheidungsproblem (Hilbert), 23, 30, 247
 Equivalência
 alfabética, 227, 230
 de compilador, 248
 de máquina, 66
 de programa, 65
 e máquina, 57-66
 em máquina de traço, 82, 83
 em uma máquina, 65
 entre máquina de Turing, 201, 202
 entre a máquina de Turing e norma, 159-171
 forte, 88, 89
 de programa, 58-65, 82, 83
 monolítico, 88-95
 Escolha (composição), 35
 Escopo de uma variável, 225
 Estado, 160
 corrente, 160
 final, 136, 198
 final de rejeição, 137
 inicial, 134, 197
 Estrutura de controle, 35
 Estruturação
 iterativa, 35
 monolítica, 35
 recursiva, 35
 Etiqueta, 39
 antecessor, 84
 final, 84
 inicial, 84
 sucessora, 84
 Evidência
 externa, 105
 interna, 105
 Expressão, 44
 de sub-rotina, 44
 inicial, 44
 lambda, 224
 que define, 44

F

Falso (tipo de dado booleano), 231
 Fila, 182, 183
 Fita, 133, 160, 196
 de início, 135

infinita, 132, 137, 159
 marcador de início, 135, 136
 Fluxograma, 37, 38, 40, 85, 88
 operação, 37, 38, 85
 parada, 184, 190, 191
 partida, 184, 190, 191
 rotulação de nós, 85
 teste, 185, 191
 Formalismo
 axiomático, 214
 denotacional, 214
 funcional, 214
 operacional, 214
 Função
 básica, 221
 computada, 35, 54-57
 computável, 23, 143, 237
 total, 148
 concatenação, 153
 de Ackermann, 241
 de codificação, 66
 de decodificação, 66
 de entrada, 46, 109
 de saída, 46, 109
 de transição, 133, 134, 196, 197
 identidade, 53
 indefinida, 136, 198
 induzida por um traço, 81
 programa, 133, 134, 160, 197
 quadrado, 148, 153
 recursiva, 24, 140, 175, 213-243
 de Kleene, 215-223
 e ciência da computação, 237-239
 parcial, 215, 221
 e total, 221-223
 minimização, 218-221
 primitiva, 241
 total, 222
 Turing-computável, 143, 149, 223
 total, 148
 Funcional
 formalismo, 214
 linguagem de programação, 238
 programação, 238

G

Gödel
 incompleteness theorem, 23, 247
 Kurt, 23, 247
 número, 106, 247
 teorema da não completude, 23, 247
 Gramática
 irrestrita, 215

livre do contexto, 215
 regular, 215

H

Haskell (linguagem), 223, 238
 Hierarquia
 de Chomsky, 215
 de classe de linguagem, 203
 de classe de máquina, 202-203
 Hilbert,
 David, 23, 29, 247
 Entscheidungsproblem, 23, 30, 247
 problema de decisão, 23, 29, 30, 247
 Hipótese
 de Church, 24, 29, 141, 157-179, 222
 de indução, 83
 de Turing-Church., 175

I

Identificação de ciclo infinito, 89, 93
 Identificador
 de operação, 36
 de sub-rotina, 43
 de teste, 36
 Igualdade
 de função parcial, 58
 induzida por redução beta (cálculo lambda), 236
incompleteness theorem (Gödel), 23, 247
 Indexação
 direta, 117
 indireta, 118
 Instrução, 35
 etiquetada, 39
 composta, 84
 rotulada, 38, 39
 composta, 84-88, 93
 Intercalação, 36
 Interpretação
 de operação, 46
 de teste, 46
 Investigação da solucionabilidade, 256
 Iterativo (programa), 34, 61, 63

J

Java (linguagem), 41

K

Kleene
 função recursiva, 215, 221
 S. C., 24, 215

L**Lambda**

- abstração, 223
- aplicação, 224
- cálculo, 24, 29, 215, 223, 229, 237
- linguagem, 223-225
- termo, 224, 225

Leibniz, Gottfried, 247**Leitura destrutiva, 185, 192****Limite de uma cadeia finita de conjunto, 89****Linguagem, 25, 28**

- aceita, 137
- algol, 41, 141
- bloco-estruturada, 140
- C, 41, 204, 237
- com parêntese balanceado, 140
- de montagem, 40
- de programação funcional, 223, 238-239
 - pura, 238
- enumerável recursivamente, 141, 204, 266, 267
- formal, 23, 28, 36, 190
- Haskell, 223, 238
- Java, 41
- lambda, 223, 224, 225
- Lisp, 223
- livre do contexto, 204
 - determinística, 203
- Pascal, 41
- pura, 238
- recursiva, 141, 142, 204, 266, 267
- recursivamente enumerável, 141
- regular, 203
- rejeitada, 137
- snobol*, 24
- universal, 270

Lisp (linguagem), 223**Lógico (tipo de dado), 231****Loop, 138, 141, 142, 149, 187, 192****M****Magnitude, 115****Máquina, 34, 45-48**

- com duas pilhas, 190
- com mais de duas pilhas, 190
- com pilha não determinística, 202
- com pilha, 182, 189-195
- com uma pilha, 190
- configuração inicial, 49, 51
- de Post, 175, 183-189
 - não determinística, 202
- de redução, 256, 257
- de registrador, 24, 104-129
 - poderosa, 105
 - simples, 105

de traço, 80-83**de Turing, 24, 131-155, 160, 172, 188, 189, 200**

- com fita infinita à esquerda e à direita, 172, 173
- com fita multidimensional, 172
- com múltipla cabeça, 172, 175
- com múltiplas fitas, 172-174
- como processador de função, 143-149
- como reconhecedor de linguagem, 137-142
- multidimensional, 174
- não determinística, 172

descrição instantânea, 49, 51**equivalente, 35****finita, 190****interpretação**

- de operação, 46
- de teste, 46

não determinística, 201, 202**norma, 24, 105, 108-110, 160, 167**

- como máquina universal, 110-121

NormaNeg, 127**sem pilhas, 190****universal, 105, 110, 157-179, 215****valor**

- de entrada, 46
- de memória, 46
- de saída, 46

Marcador de início (fita), 135, 136**Matiyasevich, Yuri, 247****Minimização (função recursiva parcial), 218-221****Modelo**

- formal, 134-137
- para concorrência, 36

Modificação sobre a máquina de Turing, 172-175**Monolítico**

- iterativo, 64
- programa, 34, 37-40, 61, 62, 63

Movimento inválido, 136**Múltiplas**

- fitas, 159
- unidades de controle, 159

N**Não**

- computável (problema), 250
- decidível (problema), 250
- determinismo, 36, 159, 201
- determinista (composição), 35
- solucionável (problema), 250, 255, 258, 260-262

Natural (tipo de dado), 232**Nó, 85****Noção**

- como máquina, 133, 134
- intuitiva, 133

Norma com 2 registradores, 119

Numerais de Church, 232
 Número de Gödel, 106, 247
Number Theoretic Register Machine ver Máquina
 norma

O

Operação
 e teste, 110-115
 fluxograma, 37, 38, 85
 identificador de, 36
 instrução rotulada, 38, 39, 106
 máquina, 49
 norma, 110
 programa, 35
 vazia, 36
 Operacional (formalismo), 214
 Operador de ponto fixo, 234

P

Palavra, 25-27
 comprimento, 26
 lambda, 224
 tamanho, 26
 vazia, 26
 Palíndromo, 28
 Parada, 184, 190, 191
 fluxograma, 37, 85
 Parêntese externo, 225
 Partida, 184, 190, 191
 fluxograma, 37, 85
 Pascal, 140, 141, 204, 238, 248
 alfabeto, 26
 linguagem, 28, 41
 programa, 27
 Passo de indução, 83
 Pilha, 119, 182, 196
 Poder computacional, 65
 Poderosa (máquina de registrador), 105
 Ponto fixo, 234
 operador, 234
 Post, Emil Leon, 183
 Prefixo, 27
 Primitiva (recursão), 217, 218, 221, 241
 Princípio da redução de problema, 247, 249
 Problema
 completamente insolúvel, 250
 da aceitação da palavra, 270
 da correspondência de Post, 261-265
 da equivalência, 261
 da não parada (*loop*), 267
 da parada, 29, 247, 248, 258, 259
 da palavra constante, 270
 da palavra vazia, 259
 da totalidade, 260

 de autoaplicação, 253-255
 de decisão, 23, 29, 30, 247, 248, 251-252, 259-261
 de reconhecimento de linguagem, 252
 decidível, 250
 não computável, 250
 não decidível, 250
 não solucionável, 250, 255, 258, 260-262
 parcialmente solucionável, 250, 254
 sim/não, 247
 solucionabilidade, 132, 246, 252
 totalmente solucionável, 250
 Procedimento efetivo, 24
 Processador de função, 132
 Programa, 24, 34, 35-77, 133, 134, 167, 184, 190, 191, 196,
 adição, 167
 equivalente, 35, 65
 fortemente equivalente, 35, 58, 67
 iterativo, 34, 40-43, 61, 63
 condicional, 41, 62
 enquanto, 41, 62
 sequencial, 35, 41, 61
 monolítico, 37, 39, 40, 49, 55, 121
 com instrução
 etiquetada composta, 80, 84
 rotulada composta, 84
 rótulo
 final, 39, 84
 inicial, 39, 84
 simplificado, 89
 para uma máquina, 47
 recursivo, 34, 43-45, 51, 56, 62, 63, 121
 subtração, 169
 Programação
 estruturada, 41
 funcional, 238
 Propriedade da solucionabilidade, 266-269
 Pura (linguagem), 238

R

Random Access Stored Programs ver RASP
 RASP, 24
 Reconhecedor de linguagem, 132
 Recursão, 43, 217
 e ponto fixo, 234, 235
 máquina norma, 110
 primitiva, 217, 218, 221, 241
 Recursiva
 parcial (função), 215, 221
 primitiva (função), 241
 Recursivo (programa), 34, 43-45, 51, 56, 62, 63, 121, 215
 Redução
 alfa, 229, 230

beta, 229, 230
 cálculo lambda, 229
 de problema, 247, 252
 iterada, 229, 230
 máquina, 256, 257
 princípio, 247, 249
 Registrador, 109
 máquina, 24
 X, 167
 Y, 167
 Rejeita, 138, 141, 142, 184, 185, 191, 192
 Relação
 de equivalência forte de programa, 57, 58
 equivalência de máquina, 57, 66
 Rotulação de nós (fluxograma), 85
 Rotulada composta, 85, 88
 Rótulo, 38, 39
 antecessor, 84
 consistente, 91
 de instrução, 167
 equivalente fortemente, 91
 final (programa monolítico), 39, 84
 fortemente equivalente, 89
 inicial (programa monolítico), 39, 84
 sucessor, 84

S

Sequencial (composição), 35, 41, 61
 Símbolo, 25, 26
 auxiliar, 184
 de entrada, 143
 branco, 135
 de início (fita), 135
 Simples (máquina de registrador), 105
 Simplificação de ciclo infinito, 94
 Simulação
 de máquina, 66
 forte de máquina, 66
 Sistema
 canônico de Post, 24
 de Post, 261
 operacional, 36
 Snobol (linguagem), 24
 Solucionabilidade (problema), 132, 246, 252
 Subpalavra, 27
 Sub-Rotina, 43
 Substituição
 cálculo lambda, 227
 composicional, 216, 217, 221
 de função, 216
 Subtração (programa), 169
 Sufixo, 27

T

Tamanho (palavra), 26
 Teorema
 da não completude (Gödel), 23, 247
 Teoria
 da computação, 23, 29, 36
 clássica, 159
 Termo lambda, 223, 224, 229
 Tese
 de Church, 24, 158, 175
 de Turing-Church, 175
 Teste, 185, 191
 fluxograma, 37, 38
 identificador, 36
 instrução rotulada, 29, 106
 máquina, 49
 norma, 110
 programa, 35, 170
 Tipo de dado
 básico, 231-233
 booleano
 falso, 231
 verdadeiro, 231
 lógico, 231
 natural, 232
 Topo (pilha), 119
 Traço, 80, 81
 Turing
 Alan, 24, 132, 215, 247
 Church (hipótese), 175
 Church (tese), 175

U

União disjunta, 88, 89
 Unidade de controle, 133, 196

V

Valor
 absoluto, 115
 constante como função, 219
 de entrada, 46
 de memória, 46
 de saída, 46
 numérico, 110, 115, 116
 Variável
 ligada, 226
 livre, 226-228
 X, 184, 190
 Y, 190
 Verdadeiro (tipo de dado booleano), 231
 Verificação da equivalência de programa, 79-101