MARCUS VINICIUS MIDENA RAMOS

TEORIA DA COMPUTAÇÃO

Blucher

TEORIA DA COMPUTAÇÃO

Marcus Vinicius Midena Ramos

Teoria da computação

Editora Edgard Blücher Ltda.

Publisher Edgard Blücher
Editor Eduardo Blücher
Coordenador editorial Rafael Fulanetti
Coordenadora de produção Ana Cristina Garcia
Produção editorial Ariana Corrêa e Andressa Lira
Diagramação Marcus Vinicius Midena Ramos
Revisão de texto Maurício Katayama
Capa Lira Editorial
Imagem da capa iStockphoto

Editora Blucher
Rua Pedroso Alvarenga, 1245, 4º andar
CEP 04531-934 – São Paulo – SP – Brasil
Tel.: 55 11 3078-5366
contato@blucher.com.br
www.blucher.com.br

Dados Internacionais de Catalogação na Publicação (CIP)
Heytor Diniz Teixeira, CRB-8/10570

Ramos, Marcus Vinicius Midena
Teoria da computação / Marcus Vinicius Midena Ramos. –
São Paulo : Blucher, 2025.

480 p. : il.

Bibliografia
ISBN 978-85-212-2524-9 (impresso)
ISBN 978-85-212-2523-2 (eletrônico - Epub)
ISBN 978-85-212-2522-5 (eletrônico - PDF)

1. Ciência da Computação. 2. Teoria da Computação. 3. Máquinas Universais. 4. Teoria de Linguagens. 5. Teoria de Decibilidade. 6. Teoria da Complexidade. 7. Cálculo Lambda.
I. Título.

CDU 004

Índice para catálogo sistemático:

1. Ciência da Computação CDU 004

Sumário

Nota da edição

Décadas atrás, não tive a dimensão do inusitado encontro. O prof. Dr. Marcus Vinicius Midena Ramos aproximou-se, entregando-me um texto que dizia ser do meu interesse a respeito do paradigma de programação orientado a objetos. Iniciava-se uma relação que se transformou em uma grande amizade e da qual muito aprendi em diversos campos de conhecimento. Esta obra me faz perceber que sempre existe um amplo e espetacular caminho para novos espaços de aprendizagem com esse brilhante e genial amigo, docente e pesquisador.

Este livro apresenta uma abordagem clara, rigorosa e acessível aos pilares conceituais da Ciência da Computação. Voltada tanto para estudantes de graduação quanto para pesquisadores e profissionais da área, a obra explora os fundamentos teóricos que sustentam o funcionamento dos sistemas computacionais contemporâneos. Os principais tópicos abordados incluem: interconexão entre programas e máquinas; uma ampla e inovadora apresentação de máquinas universais, contextualizando as questões de decidibilidade e complexidade no tempo; e por fim uma discussão sobre o cálculo lambda não tipado.

A estrutura do livro reflete uma trajetória construtiva muito interessante, que pode ser visualizada em três partes. A primeira estabelece a noção de programa de computador como ponto de início para a organização de operações e testes que devem ser executados por uma máquina com comportamento sequencial. Tal máquina, por sua vez, assume uma forma abstrata nesta obra, caracterizando-se por apresentar uma memória capaz de ler e devolver dados, além de atribuir significado para cada ocorrência de identificador, seja de operação, seja de teste em um programa. Com isso, o autor prepara o terreno para discorrer a respeito de algoritmos e máquinas universais, núcleos fundamentais da Teoria da Computação clássica. Na segunda parte, duas importantíssimas questões são discutidas: a decidibilidade (problemas cujas respostas correspondem a SIM ou NÃO) e a complexidade, no tempo, para as soluções de problemas decidiveis (tempo necessário para que a máquina execute o algoritmo). A terceira parte oferece uma via alternativa para a representação de computações que não seja por meio de máquinas; trata-se do Cálculo Lambda (não tipado), fundamentado no conceito de função matemática.

O diferencial do livro está em sua abordagem pedagógica: cada conceito é ilustrado com exemplos práticos, exercícios originais, sem descuidar do rigor matemático, esperado em um texto desta natureza. Além disso, observa-se a preocupação em incorporar discussões históricas sobre os pensadores que moldaram a teoria, enriquecendo intelectualmente o conteúdo.

Em termos de motivação, muito se pode descrever. Entendo que se trata de um livro essencial para quem deseja compreender não apenas "como" os computadores funcionam, mas "o que" eles podem ou não fazer. Em uma era dominada por algoritmos, inteligência artificial e sistemas cada vez mais complexos, torna-se fundamental refletir sobre os limites inerentes à computação. A Teoria da Computação clássica não é apenas um campo acadêmico abstrato; ela fornece as ferramentas conceituais para identificar problemas que não possuem solução algorítmica, prever o custo computacional de soluções e projetar sistemas mais eficientes e seguros.

Leitores de áreas como Ciência da Computação, Matemática, Engenharia de Software e Inteligência Artificial encontrarão neste livro uma base sólida para decisões técnicas informadas. Por exemplo, ao desenvolver um novo algoritmo, saber se o problema subjacente é NP-completo pode poupar meses de esforço infrutífero. Da mesma forma, compreender o que é decidível ajuda a evitar a implementação de sistemas que, por natureza, não podem funcionar corretamente em todos os casos.

Além disso, o livro enfrenta o crescente desconhecimento dos fundamentos teóricos entre profissionais da tecnologia. Muitos programadores utilizam recursos poderosos sem entender os princípios que os sustentam. Este texto busca preencher essa lacuna, promovendo uma formação mais profunda, crítica e rigorosa. Ele também é valioso para professores, pois oferece uma estrutura clara e atualizada para cursos de teoria da computação, com sugestões de atividades, projetos e discussões em sala de aula.

A longa convivência com o autor me fez desenvolver o hábito do pensamento formal, uma das marcas presentes nesta memorável obra. Porém, algo me chamou a atenção e ouso compartilhar neste momento. Fui surpreendido pela inesperada estratégia de apresentar um assunto tão complexo de forma inédita. Para ilustrar a minha percepção, recordo o conceito de programa que, para ser concebido, leva à noção de paradigma, de linguagem de programação, culminando com as características que uma máquina de execução deve apresentar. A coerente sequência de temas estimula o pensamento lógico e a capacidade de abstração, habilidades cada vez mais escassas, entretanto, indispensáveis em um mundo tecnológico acelerado. Ao ler esta obra, não se aprende apenas conceitos técnicos, mas desenvolve-se uma mentalidade científica capaz de questionar, analisar e inovar, efetivamente.

Por fim, o livro inclui um apêndice com respostas comentadas dos exercícios, um glossário de termos técnicos e um índice remissivo. Esses recursos tornam a obra autocontida e adequada tanto para estudo individual quanto para uso em sala de aula.

Para encerrar, *Teoria da computação* é mais do que um livro-texto: é um convite à reflexão profunda sobre os fundamentos da computação. Ao combinar rigor matemático com clareza expositiva e relevância prática, ele se consolida como uma referência indispensável para todos que desejam ir além do código e compreender a alma da computação. Sua leitura não apenas amplia o conhecimento técnico, mas transforma a maneira como enxergamos os computadores, os algoritmos e os próprios limites do pensamento humano.

Sinto-me honrado pelo convite de apresentar mais um grandioso trabalho do prof. Dr. Marcus Vinicius Midena Ramos. Ótima leitura!

Prof. Dr. Italo Santiago Vega
São Paulo, julho de 2025
Faculdade de Ciências Exatas e Tecnologia
Pontifícia Universidade Católica de São Paulo

Prefácio

Por várias décadas, o prof. Dr. Marcus Vinicius Midena Ramos muito me tem honrado com sua amizade, convívio e trabalho, como orientando de mestrado, parceiro de pesquisa e colega de magistério.

Vem atuando há quase quarenta anos no ensino superior em várias universidades brasileiras, destacando-se a Universidade de São Paulo, a Pontifícia Universidade Católica de São Paulo e a Universidade Federal do Vale do São Francisco.

Dentre os assuntos em que se aprofundou, e aos quais dedica sua pesquisa e docência, e nos quais muito investiu através da autoria de diversas obras, receberam sem dúvida especial atenção aqueles ligados à Ciência da Computação, tais como: Linguagens de Programação, Compiladores, Autômatos, Teoria da Computação, Linguagens Formais.

Não posso deixar de mencioná-lo, com muito orgulho, como meu coautor no livro *Linguagens formais: teoria, modelagem e implementação* (Bookman, 2009), ganhador em 2010 do honroso Prêmio Jabuti, na categoria Ciências Exatas, Tecnologia e Informática.

O prof. Marcus posteriormente escreveu *Linguagens formais: exercícios e soluções*, uma primorosa extensão dessa obra, lançada pela editora Novatec em 2021.

Foi ele também o principal responsável pela extensa revisão e ampliação do primeiro livro, ora disponível sob o título de *Linguagens formais: teoria e conceitos*, obra publicada pela editora Blucher, em 2023, e vencedor do 1º Prêmio Jabuti Acadêmico em 2024 na categoria Ciência da Computação.

Observando as obras existentes acerca dos diversos temas da Ciência da Computação, é preocupante constatar que, com o tempo, suas taxas de lançamento estão progressivamente minguando, e que visivelmente muitas delas se mostram cada vez menos abrangentes, rigorosas e científicas, e seus conteúdos, cada vez mais superficiais e incompletos.

Na contramão dessa tendência nefasta, com o presente livro, *Teoria da computação*, o prof. Marcus prossegue cumprindo sua missão de ajudar a reverter esse cenário, insistindo em popularizar e desmistificar em suas obras alguns temas complexos e pouco divulgados desses assuntos teóricos, procurando para isso expô-los, sempre que possível, de maneira intuitiva, acessível e menos artificiosa do que usualmente se encontra na literatura disponível.

O texto compõe-se de cinco capítulos, todos acompanhados das respectivas listas de exercícios. No primeiro, os assuntos estudados são as abstrações de máquinas computacionais, os seus programas, e as equivalências, entre máquinas e entre programas. O segundo capí-

tulo conceitua e aprofunda as máquinas universais e, através da análise de algumas delas, apresenta e aplica técnicas úteis para que essa análise seja realizada.

A decidibilidade de problemas está em foco no capítulo seguinte, que faz um estudo desse conceito, apresenta e aplica alguns métodos e técnicas de verificação da decidibilidade de problemas. No penúltimo capítulo, que fornece uma introdução à complexidade computacional através do estudo da complexidade de tempo, são investigadas as mais importantes classes de complexidade computacional, bem como o conceito e algumas técnicas de redução de problemas.

O quinto capítulo encerra o livro fazendo uma suave introdução à formulação não tipada do Cálculo Lambda, o qual, talvez pela sua relativa aridez, é pouco discutido na literatura. Explora a manipulação e a equivalência de expressões lambda, representações numéricas e de computações, e os importantes conceitos de ponto fixo e de recursão.

É fácil identificar as características deste livro que o distinguem da maior parte das obras recentemente lançadas no mercado: obra didática, fortemente voltada ao uso em ensino; preocupação constante em apresentar os assuntos de maneira palatável ao iniciante; cuidado com a elaboração de um material abrangente, sem ser superficial; coragem de garantir elevado padrão ao conteúdo da obra, através da disponibilização de tópicos avançados, pouco encontrados na literatura; uso de uma linguagem clara, que dá ao leitor uma oportunidade concreta de formação autodidática; cuidado em dar ao leitor, que esteja interessado em aprofundar-se nos temas mais importantes da área, o conhecimento necessário para complementar sua pesquisa em obras mais específicas.

Com todas essas características, tão desejáveis e pouco encontradas, o leitor passa a ter acesso a mais uma obra deste autor, cujo legado, nesta área tão carente, sem dúvida disponibiliza com este livro mais um instrumento pedagógico de alta qualidade na literatura técnica nacional.

Concluo este prefácio parabenizando o prof. Marcus e a Editora Blucher pelo lançamento deste livro, desejando não apenas que seja um sucesso editorial, mas que também estimule os docentes da área a contribuírem para que a qualidade do ensino em nossas escolas não continue a ser determinado apenas pelo sucesso editorial das obras publicadas, mas principalmente pelo alto padrão de seu conteúdo.

Prof. Dr. João José Neto
São Paulo, março de 2025
Departamento de Engenharia de Computação e Sistemas Digitais
Escola Politécnica da Universidade de São Paulo

Apresentação

O conhecimento da Teoria da Computação permite, entre outras coisas, a percepção da computação como uma área com muitas possibilidades, mas também com muitas limitações. Conhecer essas limitações é importante para a formação de profissionais que, em suas vidas, muitas vezes, irão se deparar com problemas intratáveis ou mesmo insoluvéis. Representar computações por meio de modelos matemáticos e atribuir significados precisos para os principais termos usados na área também é importante para a formação de profissionais de qualidade.

O assunto Teoria da Computação é vasto e a escolha dos principais tópicos a serem abordados em um texto introdutório é um tanto arbitrária, variando de autor para autor. No presente caso, foram escolhidos tópicos formando capítulos relativamente independentes uns dos outros, mas que cobrem os assuntos mais importantes da área.

São, portanto, cinco os capítulos que compõem este livro. Cada um deles pode ser apresentado em cerca de 12 horas-aula, fazendo com que a obra inteira possa ser usada como livro-texto em numa disciplina com cerca de 60 horas-aula, tanto em nível de graduação quanto em nível de pós-graduação, em Ciência da Computação ou áreas relacionadas.

No Capítulo 1 introduzimos algumas das noções mais fundamentais da computação, como é o caso de programas, máquinas, computação e função computada, mostrando as respectivas formalizações matemáticas. Ainda neste capítulo, são estudadas as relações de equivalência forte de programas e de poder computacional que existem entre os vários paradigmas de programação considerados.

O Capítulo 2 introduz as importantes noções de algoritmo e máquina universal. A ideia é deixar claro como as computações podem ser representadas em dispositivos do tipo máquina, e para essa finalidade são discutidos modelos tais como Máquina Norma, Máquina de Post, Máquina com Pilhas, Autômato com Duas Pilhas e Máquina de Turing. Esta última, inclusive, é discutida à luz de diversas extensões e restrições que podem ser feitas sobre o modelo básico, preparando o caminho para os capítulos seguintes.

O uso de linguagens para representar problemas de decisão é apresentado no Capítulo 3. A partir daí, mostra-se que os problemas podem ser classificados em decidíveis e indecidíveis. Após a ilustração de diversos problemas em cada uma dessas categorias, é apresentada uma técnica que permite classificar problemas, de natureza inicialmente desconhecida, em decidíveis ou indecidíveis (Seção 3.9). Por fim, o famoso problema PCP é apresentado, seguido de

aplicações envolvendo a indecidibilidade de diversos problemas relacionados com gramáticas e linguagens livres de contexto.

Problemas decidíveis podem ter soluções que demandem tempo excessivo para a sua execução. Por esse motivo, o Capítulo 4 classifica os problemas decidíveis em tratáveis ou intratáveis. De maneira similar ao que foi feito na Seção 3.9, uma técnica é apresentada na Seção 4.8 para classificar problemas de natureza inicialmente desconhecida em tratáveis (se este for o caso). O capítulo termina com uma discussão sobre uma classe muito especial de problemas, os chamados problemas $\mathcal{NP}$-completos.

A Teoria da Decidibilidade e a Teoria da Complexidade (apresentadas brevemente, respectivamente, nos Capítulos 3 e 4) são dependentes do principal modelo de computação usado desde a década de 1930, a Máquina de Turing (apresentada na Seção 2.6 e depois novamente na Seção 2.15). Se algum dia for descoberto algum modelo de computação mais poderoso que a Máquina de Turing, é provável que os principais resultados dessas teorias tenham que ser revistos. Enquanto isso não acontece, tais resultados são apresentados nos referidos capítulos.

O Capítulo 5 serve a dois propósitos. Em primeiro lugar, ele mostra ao leitor que computações podem ser representadas por intermédio de outros tipos de formalismos, e não apenas por formalismos do tipo máquina, como visto no Capítulo 2. Em segundo lugar, ele apresenta os princípios fundamentais das chamadas linguagens de programação funcionais, baseadas na definição e chamada de funções.

Este livro pressupõe que o leitor esteja familiarizado com conceitos fundamentais da Teoria de Linguagens, assim como das noções e definições mais importantes da Matemática Discreta. Para ambos os casos recomendam-se os títulos [1], [2], [3], [4], [5] ou [6] (em português) ou ainda [7], [8], [9], [10] ou [11] (em inglês), entre outros. O livro [12] é uma excelente referência sobre linguagens formais e autômatos (provavelmente a mais usada em todo o mundo), mas não dispõe de um capítulo introdutório sobre matemática discreta.

Teoremas (e lemas e corolários) e exemplos terminam, respectivamente, com as cadeias "□□□" e "⋆ ⋆ ⋆" no lado direito da página.

Ao término de cada capítulo, um conjunto de exercícios permite ao leitor praticar o conteúdo estudado. São 230 exercícios no total e as soluções de todos eles são apresentadas no final do livro, na forma de um apêndice.

Um conjunto de slides é disponibilizado em `http://www.marcusramos.com.br/univasf/teoriadacomputacao`. Eles são atualizados constantemente e podem ser usados livremente.

A importância da Teoria da Computação para estudantes e profissionais da área é o assunto de [13], cuja leitura é recomendada.

Marcus Vinicius Midena Ramos
Abril de 2025

Capítulo 1

Programas, máquinas e equivalências

Este capítulo tem três objetivos principais. O primeiro é introduzir, de maneira rigorosa (com o emprego da matemática), as diversas noções empregadas habitualmente por estudantes e profissionais de computação. São noções como programa, máquina, computação etc., que são normalmente usadas com elevado grau de informalidade, e que serão revisitadas e definidas de maneira precisa, o que poderá inclusive revelar aspectos pouco conhecidos acerca destas noções.

O segundo objetivo é apresentar algumas noções de equivalência de programas e máquinas, em especial a noção de equivalência forte de programas. Tal noção, que poderá se mostrar útil no dia a dia do profissional de computação, é acompanhada de um algoritmo que permite determinar se dois programas são equivalentes (ou não).

O terceiro objetivo é mostrar que, apesar de existirem diversos paradigmas de programação, o poder computacional deles é o mesmo. Ou seja, a escolha de um particular paradigma não restringe as possibilidades computacionais decorrentes do seu uso. Isto constrasta com a noção de equivalência forte de programas, em que os paradigmas considerados não são equivalentes.

O conteúdo deste capítulo é baseado em [14] e [15].

1.1 Programas

A maioria das pessoas que atuam na computação (como estudantes, profissionais ou usuários) possui alguma familiaridade com a noção de **programa**. Existem centenas de linguagens de programação, e este número não para de crescer. Linguagens de programação permitem a construção de programas, porém a noção que apresentaremos aqui é um pouco diferente da noção que a maioria das pessoas têm.

A noção mais difundida de programa (de computador) refere-se a uma coleção de instruções que são sequenciadas no tempo. As instruções, por sua vez, são formadas por operações (como adição, subtração etc.) e testes (como maior, diferente etc.), além dos elementos de estruturação do fluxo de controle (isto é, a maneira como as operações e testes são sequenciados no tempo, desde o início da execução do programa até o seu término).

Naturalmente, estamos nos referindo aqui apenas ao modelo sequencial de execução, ou seja, ao modelo de computação em que as operações e os testes são executados um de cada vez e um após o outro na ordem determinada pelo programa. Modelos diversos, como é o caso do modelo concorrente (no qual várias instruções compartilham um mesmo processador, porém uma de cada vez) ou do modelo paralelo (no qual as várias instruções são executados simultaneamente em vários processadores), não são discutidos neste texto.

Diferentemente desta noção mais difundida, no entanto, a noção que estamos introduzindo aqui é a noção de programa enquanto elemento de sequencialização de operações e testes no tempo. As operações e os testes propriamente ditos não são definidos pelo programa (ou pela linguagem de programação), mas sim definidos pela máquina subjacente (real ou virtual), na qual o programa é executado. Assim, o programa nada mais faz do que "tomar emprestado" as operações e testes definidos máquina na qual ele executado, determinando a ordem de execução destes. Portanto:

- Um **programa** é um conjunto de instruções que estabelecem a sequência em que certas operações e testes devem ser executados.
- Ele tem por finalidade manipular dados de entrada, produzindo as saídas desejadas.
- A **estrutura de controle** do programa define a maneira como as operações e os testes são sequenciados no tempo.

Em vez de considerarmos operações e testes específicos, como adição, subtração, maior ou igual, por exemplo, usaremos operações e testes denotados por nomes abstratos:

- Identificadores de operações: $F, G, H, ...$
- Identificadores de testes: $T_1, T_2, T_3, ...$
- Operação vazia: ✓ (não executa nenhuma operação).

Quantas linguagens de programação existem? Não se sabe ao certo, mas os números variam de quase 700 ([16]) a quase 9.000. Este número cresce todos os anos, e é bom que seja assim. Um número crescente linguagens de programação costuma indicar que elas estão aderentes a um número cada vez maior de áreas de aplicação, facilitando a construção de soluções para os problemas destas áreas.

Na década de 1960, não havia muitas linguagens para se escolher. A opção tinha que ser entre o COBOL (para aplicações comerciais), o FORTRAN (para aplicações de engenharia ou muméricas de uma forma geral), o ALGOL (muito usada na academia), o BASIC (para aplicações domésticas e usuários "leigos") ou o Lisp (para pesquisas em inteligência artificial). Desde então, novas linguagens surgem todos os anos, voltadas para um grande número de novas áreas de aplicação distintas. Como já dizia o professor João José Neto, em uma das suas aulas, *"quando a única ferramenta que se tem é um martelo, todos os problemas se parecem com um prego"*.[1] Ou seja, o surgimento de novas linguagens de programação é bem-vindo na medida em que permite que os programadores façam escolhas mais aderentes às áreas de aplicação em que trabalham. Caso contrário, as possibilidades do programador ficam limitadas pelas possibilidades da linguagem que ele adota (ou conhece), o que pode ser desastroso. É o caso, por exemplo, de querer colocar um parafuso numa parede usando um martelo: é possível, mas o resultado seria muito melhor se uma chave de fenda fosse usada no lugar de um martelo.

Mas como selecionar, entre um número grande e crescente de linguagens de programação, aquelas que representam as melhores escolhas para cada caso? Parte da resposta está no

[1] Originalmente esta frase é de Abraham Harold Maslow (psicólogo norte-americano, 1908-1970), tendo sido publicada pela primeira vez em [17]. A frase original é *"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail"*. Ela é também conhecida como *Maslow's Hammer* (ou "Martelo de Maslow", em tradução livre).

conceito de paradigma. Um paradigma é um estilo, um modelo de uma linguagem de programação, e descreve um certo conjunto de recursos que pode ser usado para a construção de programas nesta linguagem. Em outras palavras, um paradigma representa uma certa filosofia de programação, correspondendo a uma descrição abstrata de uma linguagem. Normalmente, uma linguagem de programação suporta um único paradigma de programação, mas não é raro encontrar linguagens que suportam mais de um paradigma ao mesmo tempo.

Linguagens de programação que suportam um mesmo paradigma costumam exibir semelhanças (tanto do ponto de vista sintático quando semântico), ao passo que linguagens que suportam paradigmas distintos costumam exibir grandes diferenças entre si.

Felizmente, e diferentemente do que acontece com o número de linguagens de programação, o número de paradigmas de linguagens de programação que existe é pequeno. Portanto, conhecer bem os paradigmas, e sabendo a priori quais linguagens suportam quais paradigmas (uma linguagem pode suportar mais de um paradigma ao mesmo tempo), é o melhor caminho para escolher a linguagem mais indicada para cada caso, ainda que seus detalhes não sejam conhecidos.

Não parece haver muito consenso em relação a quais são os principais paradigmas existentes (o critério varia de autor para autor). Entretanto, é mais ou menos consenso que os principais paradigmas são o imperativo e o declarativo. No primeiro se encontram as linguagens que permitem ao programador expressar a forma como um certo problema deve ser resolvido. No segundo se encontram as linguagens que permitem ao programador concentrar-se no que deve ser feito, em detrimento do como fazê-lo. Portanto:

$$\text{Paradigmas} \left\{ \begin{array}{l} \text{Imperativo} \\ \text{Declarativo} \end{array} \right.$$

O paradigma imperativo costuma ser subdividido em três: o monolítico (linguagens que usam desvios arbitrários), o iterativo (linguagens estruturadas) e o orientado a objetos (no qual objetos abstratos representam objetos do mundo real e se comunicam através de mensagens). Já o paradigma declarativo costuma se dividir em dois: o funcional (no qual a definição e chamada de funções é a base da programação) e o lógico (no qual fatos e regras de inferência são usados para se chegar à solução do problema):

$$\text{Paradigmas} \left\{ \begin{array}{l} \text{Imperativo} \left\{ \begin{array}{l} \text{Monolítico} \\ \text{Iterativo} \\ \text{Orientado a objetos} \end{array} \right. \\ \\ \text{Declarativo} \left\{ \begin{array}{l} \text{Funcional} \\ \text{Lógico} \end{array} \right. \end{array} \right.$$

No que segue, dos cinco paradigmas citados (monolítico, iterativo, orientado a objetos, funcional e lógico), serão estudados neste livro apenas três: o monolítico, o iterativo e o funcional (aqui chamado recursivo). Estes paradigmas serão considerados apenas no modelo sequencial de execução. Os modelos concorrente e paralelo não serão levados em conta, tampouco os paradigmas orientado a objetos e lógico.

Cada um dos três paradigmas considerados será apresentado por meio de uma linguagem de programação pequena, de baixa complexidade e que suporta apenas o paradigma em questão. Aqui, o objetivo não é apresentar linguagens de programaçao reais, mas apenas linguagens que ilustram os respectivos paradigmas e que sirvam aos propósitos de estudo que serão conduzidos mais adiante. Cada um destes paradigmas oferece uma forma diferente de sequenciar operações e testes ao longo do tempo, ou seja, de estruturar o fluxo de controle.

Esta forma de estruturar o fluxo de controle também é conhecida por estrutura de controle da linguagem de programação. Os paradigmas de programação que serão considerados no presente texto são, portanto:

- Monolítico (*flowchart programs*).
- Iterativo (*while programs*).
- Recursivo (*procedure programs*).

Cada um destes paradigmas será estudado por meio de uma mini-linguagem de programação. Concebida com finalidades exclusivamente didáticas, estas mini-linguagens são pequenas por natureza e incorporam apenas os recursos relevantes ao seu estudo. Não é o que acontece com as linguagens de programação comerciais, nas quais é frequente a incorporação de um ou mais paradigmas na sua definição, além de uma quantidade grande de detalhes que são irrelevantes para os nossos objetivos.

Em todos estes paradigmas, os quais serão estudados em detalhes mais adiante, a composição sequencial é utilizada como elemento de estruturação do fluxo. Ela determina que as instruções do programa devem ser executadas em uma certa sequência, de tal forma que a instrução seguinte só pode ser executada após o término da execução da anterior. Tal tipo de composição está presente em praticamente todas as linguagens de programação e costuma ser denotada pelo símbolo ";" (ponto e vírgula). Os demais elementos de estruturação são específicos dos paradigmas considerados.

1.1.1 Programas monolíticos

O paradigma **monolítico** é aquele em que os elementos de estruturação do fluxo são, além da composição sequencial, o desvio incondicional e o desvio condicional (além de algumas variações destes). O desvio incondicional e condicional correspondem, respectivamente, à transferência do controle de uma instrução qualquer para uma outra rotulada para essa finalidade (de forma incondicional) e à transferência do controle de forma similar, porém apenas se uma certa condição for verificada (portanto, de forma condicional). Tais instruções são normalmente encontradas nas linguagens de programação com os nomes GOTO ou JUMP e servem, portanto, para trasnferir o controle para uma outra instrução do programa, rompendo o modelo sequencial de execução, no qual a instrução seguinte é sempre executada ao término da execução da instrução corrente.

No que segue, ⟨rótulo⟩ identifica uma instrução qualquer do programa e ⟨exp⟩ denota uma expressão qualquer que, ao ser avaliada, retorna os valores *verdadeiro* ou *falso*:

Desvio incondicional (para uma instrução anterior):

```
...
⟨rótulo⟩ ...
...
GOTO ⟨rótulo⟩
...
```

Desvio incondicional (para uma instrução posterior):

```
...
```

```
GOTO ⟨rótulo⟩
...
⟨rótulo⟩ ...
```

Desvio condicional (para uma instrução anterior):

```
...
⟨rótulo⟩ ...
...
IF ⟨exp⟩ GOTO ⟨rótulo⟩
...
```

Desvio incondicional (para uma instrução posterior):

```
...
IF ⟨exp⟩ GOTO ⟨rótulo⟩
...
⟨rótulo⟩ ...
```

Programas monolíticos, portanto, são programas que fazem uso de desvio arbitrários (condicionais ou incondicionais) e são normalmente constituídos por um único bloco.[2]

O paradigma monolítico é provavelmente o paradigma de programação mais antigo que se conhece. De fato, os primeiros computadores eram programados por meio do uso exclusivo de desvios condicionais e incondicionais, tornando esta forma de programação bastante popular nas décadas de 1950 e 1960, em particular nas primeiras versões do Fortran e do Basic.[3,4,5] Diversos problemas foram identificados neste paradigma de programação, os quais serão discutidos mais adiante.

Um **programa monolítico** pode ser representado de forma gráfica ou textual:

- Gráfica, por meio de um **fluxograma**.
- Textual, por meio de um conjunto de **instruções rotuladas**.

Os componentes de um fluxograma são os seguintes:

Início (único em cada fluxograma, ele indica o ponto de início da execução do fluxograma). Veja Figura 1.1.

Término (pode ocorrer em qualquer quantidade no fluxograma, indicando os pontos de término da execução do fluxograma). Veja Figura 1.2.

Operação (indica o local em que ocorre, no fluxograma, a execução de uma operação, como F, G, H etc.). Veja Figura 1.3.

Desvio condicional (indica a execução de um teste, como T_1, T_2 etc., seguido da ramificação do fluxo de controle conforme o seu resultado). Veja Figura 1.4.

[2] Um bloco corresponde a uma sequência de declarações e comandos que pode ser considerado como uma abstração independente dentro do programa.

[3] Sigla de FORmula TRANslator, foi uma linguagem de programação bastante popular nesse período e que continua

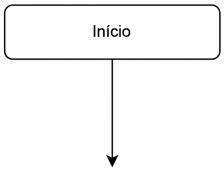


Figura 1.1 Componente “Início” de um fluxograma.

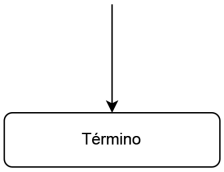


Figura 1.2 Componente “Término” de um fluxograma.

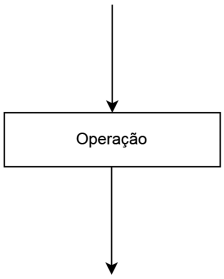


Figura 1.3 Componente “Operação” de um fluxograma.

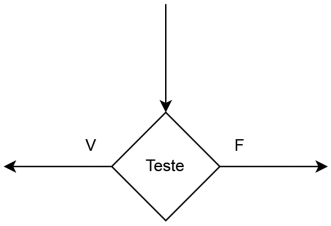


Figura 1.4 Componente “Desvio condicional” de um fluxograma.

Estes componentes são interligados de forma arbitrária por linhas direcionadas, e dessa forma o fluxo de execução fica estabelecido. Note que o desvio incondicional é representado apenas por uma linha direcionada.

Exemplo 1.1 A Figura 1.5 ilustra o fluxograma de um programa monolítico. ⋆⋆⋆

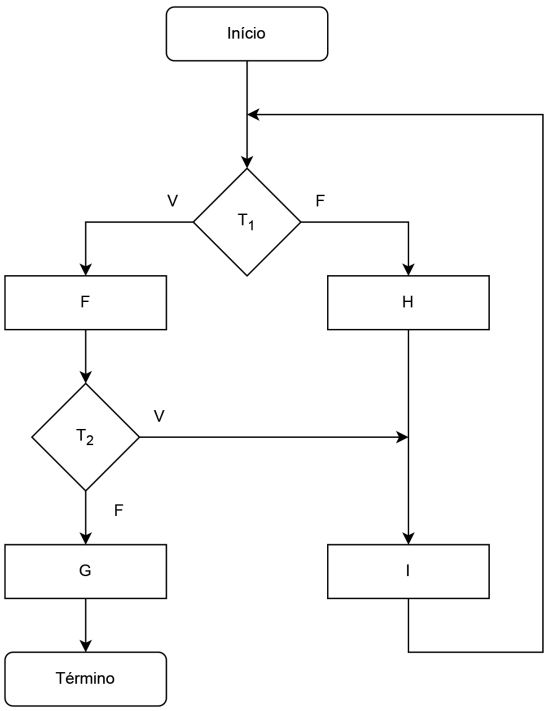


Figura 1.5 Exemplo de fluxograma para programa monolítico.

Programas monolíticos também podem ser representados de forma textual. Formalmente, uma **instrução rotulada** é uma cadeia finita de caracteres que possui um dos seguintes formatos:

- Desvio incondicional com execução de operação:

  r_1: faça F vá_para r_2

- Desvio incondicional sem execução de operação:

popular até hoje, especialmente em aplicações científicas nos grandes computadores.

[4] Sigla de Beginners All-Purpose Simbolic Instruction Code, também foi uma linguagem de programação bastante popular nesse período.

[5] Tanto o FORTRAN II (1958) como o BASIC (1964) tinham outras estruturas de controle, além do GOTO. Em FORTRAN, o comando DO...CONTINUE funcionava para designar repetições, e o IF aritmético fazia desvios triplos conforme o valor de um inteiro. Também havia o comando FOR, para repetições. Em BASIC, havia comandos dessa mesma natureza.

r_1: faça ✓ vá_para r_2

- Desvio condicional:

r_1: se T então vá_para r_2 senão vá_para r_3

onde r_1, r_2 e r_3 são rótulos numéricos, F é um identificador de operação e T é um identificador de teste.

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

$$P = (I, r)$$

onde:

- I é um conjunto finito de instruções rotuladas.
- r é o rótulo inicial.

Observações importantes:

- Duas instruções não podem ter o mesmo rótulo.
- Rótulos finais são aqueles que são referenciados mas não estão associados a nenhuma instrução.

Exemplo 1.2 O programa P abaixo é um exemplo de programa monolítico:

$P = (I, 1)$, onde $I = \{$
1: se T_1 então vá_para 2 senão vá_para 3,
2: faça F vá_para 4,
3: faça H vá_para 5,
4: se T_2 então vá_para 5 senão vá_para 6,
5: faça I vá_para 1,
6: faça G vá_para 7}

Note que P é uma representação textual do fluxograma do Exemplo 1.1. Observe que o rótulo final de P é 7 (pois ele não é definido no programa). ⋆⋆⋆

Exemplo 1.3 São outros exemplos de programas monolíticos:

- Este programa não faz nada e termina. O rótulo final neste caso é 2.
  ({1: faça ✓ vá_para 2}, 1)

- Este programa executa a operação F até que o teste T seja *falso*. F é executada pelo menos uma vez. O rótulo final neste caso é 3.
  ({1: faça F vá_para 2, 2: se T vá_para 1 senão vá_para 3}, 1)

⋆⋆⋆

Para mais informações sobre o comando GOTO, acesse [18].

1.1.2 Programas iterativos

O paradigma **iterativo**[6] representa programas estruturados sem subprogramas. A programação estruturada é aquela em que os desvios arbitrários (como no paradigma monolítico) são eliminados e, no seu lugar, são usados apenas dois elementos de estruturação (além, é claro, da composição sequencial):

- Execução condicional (com uma única entrada e uma única saída).
- Execução iterativa (com uma única entrada e uma única saída).

A programação estruturada ([19]) surgiu no final da década de 1960[7] como alternativa à programação não estruturada (ou baseada exclusivamente em desvios arbitrários). Ela foi criada como uma solução para os problemas apresentados pela programação monolítica e que foram relatados numa carta ao editor da revista *Communications of the ACM* publicada por Dijkstra em 1968 ([20]). Neste carta, talvez um dos documentos mais famosos da história da computação (e cuja polêmica se estendeu por anos, veja [21] e [22]), Dijkstra alega que o comando GOTO deveria ser abolido em função dos seus malefícios. Segundo ele:

> *"The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."*

ou, em tradução livre:

> *"O comando goto, da forma como ele se apresenta, é muito primitivo; é um convite para fazer uma bagunça do programa de alguém."*

De fato, o uso de desvios arbitrários produz um efeito nocivo, especialmente no que se refere à análise e à manutenção de programas de computador. Tal efeito está relacionado com a distância que existe entre a representação estática do programa (texto ou fluxograma) e o seu comportamento dinâmico (execução). Quando esta distância é reduzida (como no caso da programação iterativa), considera-se que a análise e a manutenção sejam facilitadas. Quando ela aumenta (como no caso da programação monolítica), considera-se que a análise e a manutenção sejam dificultadas.

Para entender o que acontece num certo trecho de programa (condição necessária para sua análise e manutenção), o programador deve entender como se chega até ele. Na programação estruturada, a chegada a uma instrução pode acontecer tendo como ponto de partida o comando anterior (por isso todas as instruções possuem uma única entrada e uma única saída) ou o próprio comando (no caso do comando iterativo). Já na programação monolítica, a chegada a uma instrução pode acontecer tendo como ponto de partida qualquer outra instrução do programa. Considerado de forma recorrente, o processo mental necessário à compreensão de um programa monolítico pode ser extremamente complexo. Esta complexidade, no entanto, é bastante reduzida quando se faz uso da programação estruturada.

Entre outros efeitos negativos, Dijkstra alega, em seu artigo, que a qualidade dos programadores é inversamente proporcional à quantidade de comandos GOTO usados nos seus programas (ou seja, segundo Dijkstra, quanto mais comandos GOTO o programador usasse em seus programas, pior programador ele seria). Ao final do seu artigo, Dijkstra cita a programação estruturada como uma possível alternativa a programação com desvios arbitrários,

[6] Não confundir "iterativo" com "interativo". No primeiro caso a palavra se refere àquilo que se repete; no segundo caso a palavra se refere a dois elementos (humanos ou máquinas) que trocam ações entre si.

[7] Portanto, não muito antes do surgimento das ideias que levaram ao estabelecimento da Engenharia de Software.

eliminando os seus efeitos nocivos. Neste momento, portanto, começava o movimento que conduziria à substituição do paradigma monolítico pelo paradigma iterativo (ou programação estruturada).

O uso excessivo e indiscriminado do comando GOTO deu origem, inclusive, a um termo pejorativo para se referir a este tipo de programação ("spaghetti code", veja mais em [23]).

Já que a programação estruturada apresentava uma grande vantagem em relação à programação não estruturada, a questão a ser considerada passou a ser: até que ponto será possível resolver os mesmos problemas se a programação estruturada for adotada no lugar da programação não estruturada? Haverá alguma limitação na nossa capacidade de resolução de problemas?

Felizmente a resposta para esta última questão é negativa. Conforme demonstrado por Böhm e Jacopini, no seu famoso Teorema de Böhm-Jacopini ([24], Teorema 1.6)), também conhecido como Teorema do Programa Estruturado ([25]), todo programa monolítico pode ser transformado num programa iterativo equivalente ([26], [27]). Este teorema possui grande valor teórico, pois estabelece que a mudança de paradigma de programação não introduz nenhum prejuízo na nossa capacidade de resolução de programas. Ou seja, o comando GOTO é desnecessário e pode ser substituído pelos comandos estruturados.

Apesar de o teorema mostrar como esta conversão pode ser feita, na prática ela não é realizada, pois o programa iterativo construído a partir do programa monolítico tende a ser maior e mais complexo. Para evitar isso, sugere-se, em vez de fazer programas monolíticos e depois convertê-los para as versões iterativas correspondentes, que se adote um raciocínio estruturado desde o início, dando origem a programas estruturados que não precisam ser convertidos posteriormente. Desta maneira, evitam-se os custos adicionados pela conversão apresentada no artigo e obtém-se um programa com as características desejadas.

A programação estruturada posteriormente se desenvolveu bastante, tornando-se a base não apenas das principais linguagens de programação daquela época (como a família Algol), mas também da maioria das linguagens usadas até os dias de hoje (Java, C etc.). Quatro trabalhos de três importantes pesquisadores foram fundamentais no estabelecimento do que hoje conhecemos como programação estruturada: um relatório de Dijkstra (publicado em [28]), dois artigos de Wirth ([29] e [30]) e um artigo de Knuth ([31]). Neste último, inclusive, Knuth procurou retirar um pouco da culpa que recaía sobre os programadores que usavam o comando GOTO, e que fora lançada por Dijkstra anos antes. A começar pelo título do seu artigo ("Structured programming with Goto statements", ou "Programação estruturada usando o comando Goto"), Knuth sustenta que o problema da programação não é o comando GOTO em si, mas a maneira como é utilizado. Dessa forma, Knuth prevê a possibilidade de que um programa possa manter-se estruturado, ainda que fazendo uso do comando GOTO. Em outras palavras, Knuth defendia uma abordagem estruturada ao problema da programação, sem se importar muito com os comandos utilizados para a sua implementação.

Assim como no caso do paradigma monolítico, no paradigma iterativo um programa pode ser representado de forma gráfica ou textual. Na forma gráfica, ele é representado por meio de um fluxograma estruturado. Na forma textual, ele é representado por meio de um conjunto de instruções que realizam os elementos de estruturação.

A diferença entre um fluxograma qualquer (usado na construção de programas monolíticos) e o fluxograma estruturado citado acima é que, neste último, toda instrução deve ter uma única entrada e uma única saída (veja Figura 1.6). Trata-se, pois, de uma restrição à forma como fluxogramas podem ser obtidos. Como resultado, todo fluxograma estruturado é também um fluxograma, mas nem todo fluxograma é também um fluxograma estruturado.

Exemplo 1.4 A Figura 1.7 ilustra o fluxograma de um programa iterativo. ⋆⋆⋆

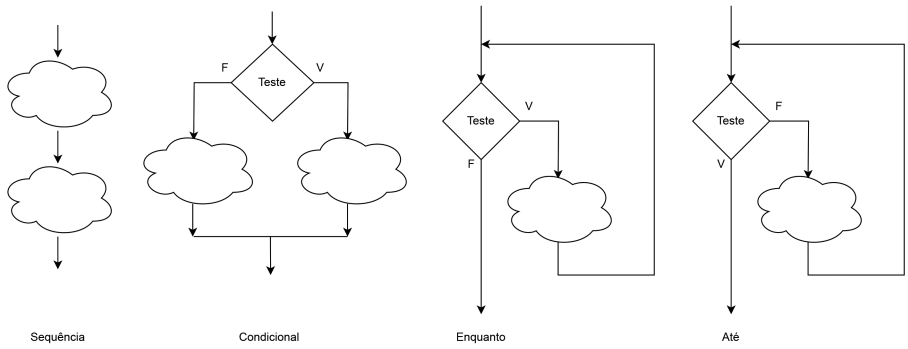


Figura 1.6 Componentes de um fluxograma estruturado.

Um **programa iterativo** é definido de forma indutiva como segue (observe que esta definição também permite a obtenção de representações textuais para programas iterativos):

1. A operação vazia ✓ e os identificadores de operação são programas iterativos.
2. Se `V` e `W` são programas iterativos, então `V;W` é um programa iterativo (execução sequencial).
3. Se `V` e `W` são programas iterativos, e `T` é um identificador de teste, então `se T então V senão W` é um programa iterativo (execução condicional).
4. Se `V` é um programa iterativo, e `T` é um identificador de teste, então `enquanto T faça V` é um programa iterativo (execução iterativa do tipo "enquanto").
5. Se `V` é um programa iterativo, e `T` é um identificador de teste, então `até T faça V` é um programa iterativo (execução iterativa do tipo "até", substitui o "enquanto" com a condição negada).
6. Se `V` é um programa iterativo, então `(V)` é um programa iterativo.

Exemplo 1.5 A seguir apresentamos um exemplo de programa iterativo. Observe que a versão textual do programa iterativo deste exemplo corresponde ao fluxograma do Exemplo 1.4 (Figura 1.7). Inicialmente em múltiplas linhas, para facilitar o entendimento:

se T_1
então enquanto T_2 faça até T_3 faça (F;G)
senão ✓

Em uma única linha, para facilitar a verificação:

se T_1 então enquanto T_2 faça até T_3 faça (F;G) senão ✓

Verificação:

- Pela regra 1, ✓, `F` e `G` são programas iterativos.
- Pela regra 2, `F;G` é um programa iterativo.
- Pela regra 6, `(F;G)` é um programa iterativo.

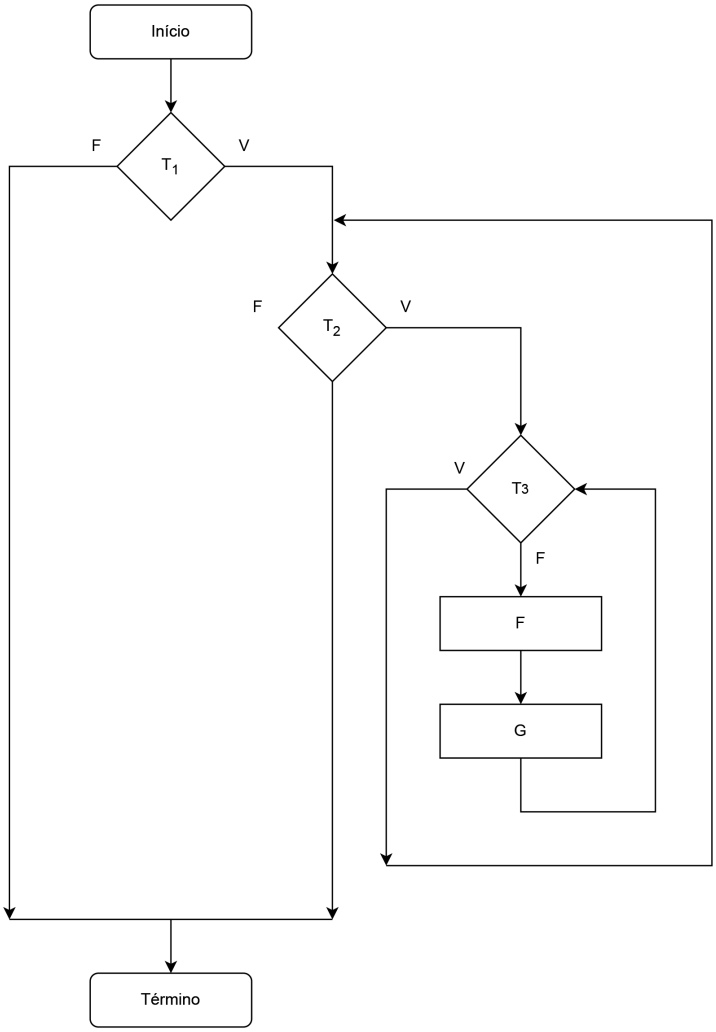
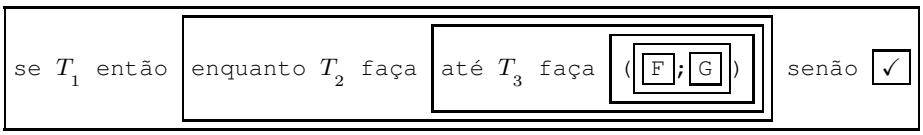


Figura 1.7 Exemplo de fluxograma para programa iterativo.

- Pela regra 5, até T_3 faça (F;G) é um programa iterativo.
- Pela regra 4, enquanto T_2 faça até T_3 faça (F;G) é um programa iterativo.
- Pela regra 3, se T_1 então enquanto T_2 faça até T_3 faça (F;G) senão ✓ é um programa iterativo.



⋆⋆⋆

Exemplo 1.6 São outros exemplos de programas iterativos:

- Este programa não faz nada e termina.
  ✓

- Este programa executa a operação F até que o teste T seja *falso*. F é executada pelo menos uma vez.
  F; enquanto T faça F

⋆⋆⋆

1.1.3 Programas recursivos

O paradigma **recursivo** representa programas por meio da definição de subprogramas e de chamadas (eventualmente recursivas) de subprogramas. Cada subprograma está associado a uma expressão que, depois de avaliada, define o valor a ser retornado pelo correspondente subprograma. Não há comando iterativo, e toda repetição deve ser expressa por meio da recursão. Os elementos de estruturação do fluxo são, portanto (além da composição sequencial), a execução condicional (com uma única entrada e uma única saída, como no paradigma iterativo), a definição de subprograma e a chamada de subprograma.

Este paradigma, também conhecido como **funcional**, representa computações usando apenas os elementos descritos anteriormente, em particular a definição e a chamada de subprogramas. Um subprograma representa uma sequência de declarações e comandos e serve para implementar novas operações ou comandos na linguagem de programação. No primeiro caso o subprograma recebe o nome de função, no segundo ele é chamado de procedimento. Subprogramas são encontrados em linguagens modernas e antigas, eventualmente combinados com outros paradigmas. A linguagem Lisp,[8] usada desde a década de 1960 para pesquisas em inteligência artificial, é uma destas linguagens. Outras, mais modernas, incluem Haskell e Erlang, OCaml. No entanto, a maioria das linguagens de programação modernas oferece pelo menos algum tipo de suporte a este paradigma através da definição e uso de subprogramas, ainda que não de forma exclusiva.

Diferentemente dos paradigmas monolítico e iterativo, o paradigma recursivo não possui representação gráfica, apenas textual. Assim, considere que $R_1, R_2, ...$ são "identificadores de subprogramas". Então, a noção de **expressão**, fundamental para a definição de programas interativos, pode ser definida de maneira indutiva:

[8] Sigla de List Processing Language.

1. A operação vazia ✓ e os identificadores de operação são expressões.
2. Todos os identificadores de subprograma são expressões.
3. Se `V` e `W` são expressões, então `V;W` é uma expressão (composição sequencial).
4. Se `V` e `W` são expressões, e `T` é um identificador de teste, então `se T então V senão W` é uma expressão (execução condicional).
5. Se `V` é uma expressão, então `(V)` é uma expressão.

Uma expressão denota uma fórmula matemática que é avaliada em tempo de execução para produzir um valor ou algum efeito colateral (por exemplo, a mudança no valor da memória). Formalmente, um **programa recursivo** é definido da seguinte maneira:

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

sendo que $R_1, R_2, ..., R_n$ são identificadores de subprogramas, $E_1, E_2, ..., E_n$ são as expressões que definem, respectivamente, os subprogramas identificados por $R_1, R_2, ..., R_n$ e E_0 é a expressão inicial, correspondente ao programa principal `P`. Além disso, todos os identificadores de subprograma referenciados em `P` devem ser definidos em `P`.

Exemplo 1.7 Exemplo de programa recursivo:

P é $\boxed{R;Z}$ onde
R def $\boxed{\texttt{F; se T então } R \texttt{ senão (G;}S\texttt{)}}$
S def $\boxed{\texttt{se T então ✓senão (F;}R\texttt{)}}$
Z def $\boxed{\texttt{se T então G senão F}}$

Este programa pode ser visto como:

P é $\boxed{E_0}$ onde
R_1 def $\boxed{E_1}$
R_2 def $\boxed{E_2}$
R_3 def $\boxed{E_3}$

onde as expressões E_0, E_1, E_2 e E_3 são analisadas a seguir.

Expressão $\boxed{E_0}$ (correspondente a $R;Z$):

- Pela regra 2, R e Z são expressões.
- Pela regra 3, $R;Z$ é uma expressão.

Expressão $\boxed{E_1}$ (correspondente a `F; se T então R senão (G;`S`)`):

- Pela regra 1, `F` e `G` são expressões.
- Pela regra 2, S é uma expressão.

- Pela regra 3, `G;` S é uma expressão.
- Pela regra 5, `(G;` S`)` é uma expressão.
- Pela regra 4, `se T então` R `senão (G;` S`)` é uma expressão.
- Pela regra 3, `F; se T então` R `senão (G;` S`)` é uma expressão.

Expressão $\boxed{E_2}$ (correspondente a `se T então` ✓`senão (F;` R`)`):

- Pela regra 1, ✓ e `F` são expressões.
- Pela regra 2, R é uma expressão.
- Pela regra 3, `F;` R é uma expressão.
- Pela regra 5, `(F;` R`)` é uma expressão.
- Pela regra 4, `se T então` ✓`senão (F;` R`)` é uma expressão.

Expressão $\boxed{E_3}$ (correspondente a `se T então G senão F`):

- Pela regra 1, `F` e `G` são expressões.
- Pela regra 4, `se T então G senão F` é uma expressão.

⋆⋆⋆

Exemplo 1.8 São outros exemplos de programas recursivos:

- Este programa não faz nada e termina.
  `P é` R `onde`
  R `def` ✓

- Este programa executa a operação `F` até que o teste `T` seja *falso*. `F` é executada pelo menos uma vez.
  `P é T;` R `onde`
  R `def se T então F;` R `senão` ✓

⋆⋆⋆

1.2 Máquinas

Máquinas são dispositivos que executam programas. Elas podem ser reais (implementadas por meio de circuitos eletrônicos) ou virtuais (implementadas por software). Neste último caso elas são também chamadas de abstratas e a sua execução acontece por meio de emuladores (interpretadores que executam a máquina abstrata na máquina real). Neste texto, a particular forma de implementação da máquina é irrelevante, uma vez que o objetivo é construir um modelo matemático para tal dispositivo.

O conceito de **máquina**, introduzido na presente seção, também é definido de forma um pouco diferente da usual. Normalmente, entende-se que máquina é um dispositivo eletrônico (portanto material), capaz de executar programas. Aqui, no entanto, máquinas são definidas de forma abstrata, por meio de formulações matemáticas dos processos de computação.

Uma máquina deve possuir as três seguintes características: (i) estrutura para armazenamento de dados (memória); (ii) capacidade de ler (entrada) e devolver dados para o meio externo (saída); (iii) atribuir significado para os identificadores de operação e de teste usados nos programas.

Conforme antecipado na Seção 1.1, as operações e testes usados num programa devem ser definidos na máquina na qual o programa é executado. Isto é feito por intermédio da definição de funções, associadas à definição da máquina, as quais atribuem significado para os identificadores de operação e de teste.

No caso das operações, estas funções mapeiam o valor corrente da memória (anterior à execução da operação) no valor futuro da memória (posterior à execução da operação).

No caso dos testes, as funções associam o valor corrente da memória aos valores lógicos *verdadeiro* ou *falso*. A memória não é alterada pela execução dos testes.

Formalmente, uma máquina é uma 7-upla:

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

onde:

- V é o conjunto de valores que podem ser armazenados na memória.
- X é o conjunto de valores que podem ser lidos na entrada.
- Y é o conjunto de valores que podem ser escritos na saída.
- π_X é a função de entrada, tal que $\pi_X : X \to V$.
- π_Y é a função de saída, tal que $\pi_Y : V \to Y$.

Sejam O_M e T_M, respectivamente, os conjuntos de identificadores de operações e testes definidos por M. Então as seguintes convenções serão usadas:

- Π_O, representando o conjunto de interpretações de operações tal que:

$$\forall o \in O_M, (\pi_o : V \to V) \in \Pi_O$$

- Π_T, representando o conjunto de interpretações de testes tal que:

$$\forall t \in T_M, (\pi_t : V \to \{\textit{verdadeiro}, \textit{falso}\}) \in \Pi_T$$

Note, pelas definições acima, que uma função que implementa a interpretação de uma operação mapeia sempre o valor total da memória em um novo valor total (mesmo que apenas uma parte da memória sofra modificação). No caso de uma função que implementa a interpretação de um teste, o mapeamento é feito sempre entre o valor total da memória (mesmo que apenas uma parte dela seja usado) e os valores lógicos.

Exemplo 1.9 O significado de valor total de memória é exemplificado a seguir. Suponha $V = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ em uma máquina com três registradores (a, b e c). Então:

- A operação inc_a, que incrementa o valor do registador a, deve ter como domínio $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ e não apenas $\mathbb{N}$. Ou seja, $inc_a(x, y, z) = (x + 1, y, z)$. Note que o registrador a é incrementado (passa do valor x para o valor $x + 1$), ao passo que os demais registradores permanecem com seus valores originais (respectivamente b e c, com y e z).
- A operação $zero_b$, que testa se o valor do registador b é zero, deve ter como domínio $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ e não apenas $\mathbb{N}$. Ou seja, $zero_b(x, y, z) =$ *verdadeiro* se $y = 0$ ou *falso* caso contrário.

⋆ ⋆ ⋆

Exemplo 1.10 A Máquina de Um Registrador $M_{_U} = (V, X, Y, \pi_{_X}, \pi_{_Y}, \Pi_{_O}, \Pi_{_T})$ possui as seguintes características:

- $V = \mathbb{N}$
- $X = \mathbb{N}$
- $Y = \mathbb{N}$
- $\pi_{_X} = id_{_\mathbb{N}}$
- $\pi_{_Y} = id_{_\mathbb{N}}$
- $O_{_M} = \{\text{add, sub}\}$
- $T_{_M} = \{\text{zero}\}$

Funções que implementam entrada, saída, operações e testes em $M_{_U}$:

- $id_{_\mathbb{N}} : \mathbb{N} \to \mathbb{N}$
  $\forall n \in \mathbb{N}, id_{_\mathbb{N}}(n) = n$
- add: $\mathbb{N} \to \mathbb{N}$
  $\forall n \in \mathbb{N},\ \text{add}(n) = n + 1$
- sub: $\mathbb{N} \to \mathbb{N}$
  $\forall n \in \mathbb{N}, n > 0,\ \text{sub}(n) = n - 1$
  se $n = 0,\ \text{sub}(n) = 0$
- zero: $\mathbb{N} \to \{\textit{verdadeiro}, \textit{falso}\}$
  se $n = 0,\ \text{zero}(n) = \textit{verdadeiro}$
  se $n \neq 0,\ \text{zero}(n) = \textit{falso}$

⋆ ⋆ ⋆

Exemplo 1.11 A Máquina de Dois Registradores: $M_{_D} = (V, X, Y, \pi_{_X}, \pi_{_Y}, \Pi_{_O}, \Pi_{_T})$ possui as seguintes características:

- $V = \mathbb{N}^2$
- $X = \mathbb{N}$
- $Y = \mathbb{N}$
- $\pi_{_X}$ =armazena_a
- $\pi_{_Y}$ =retorna_b
- $O_{_M} = \{\text{subtrai\_a, adiciona\_b}\}$
- $T_{_M} = \{\text{a\_zero}\}$

Funções que implementam entrada, saída, operações e testes em $M_{_D}$:

- armazena_a: $\mathbb{N} \to \mathbb{N}^2$
  $\forall n \in \mathbb{N},\ \text{armazena\_a}(n) = (n, 0)$
- retorna_b: $\mathbb{N}^2 \to \mathbb{N}$
  $\forall (n, m) \in \mathbb{N}^2,\ \text{retorna\_b}(n, m) = m$
- adiciona_b: $\mathbb{N}^2 \to \mathbb{N}^2$
  $\forall (n, m) \in \mathbb{N}^2,\ \text{adiciona\_b}(n, m) = (n, m + 1)$
- subtrai_a: $\mathbb{N}^2 \to \mathbb{N}^2$
  $\forall (n, m) \in \mathbb{N}^2, n > 0,\ \text{subtrai\_a}(n, m) = (n - 1, m)$
  se $n = 0,\ \text{subtrai\_a}(n, m) = (0, m)$
- a_zero: $\mathbb{N}^2 \to \{\textit{verdadeiro}, \textit{falso}\}$
  $\forall (n, m) \in \mathbb{N}^2,\ \text{se } n = 0,\ \text{a\_zero}(n, m) = \textit{verdadeiro}$
  $\forall (n, m) \in \mathbb{N}^2,\ \text{se } n \neq 0,\ \text{a\_zero}(n, m) = \textit{falso}$

⋆ ⋆ ⋆

1.3 Programa para uma máquina

Nas Seções 1.1 e 1.2 definimos, respectivamente, e de forma isolada, as noções de programa e de máquina. Na presente seção é investigada a condição que deve ser satisfeita para que um programa possa ser executado por uma máquina.

Sejam uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_O, \Pi_T)$ e P um programa onde O_P e T_P são os respectivos conjuntos de identificadores de operações e testes. Diz-se que P é um **programa para a máquina** M se, e somente se:

- $\forall o \in O_P$, existe uma única função $(\pi_o : V \to V) \in \Pi_O$.
- $\forall t \in T_P$, existe uma única função $(\pi_t : V \to \{\textit{verdadeiro}, \textit{falso}\}) \in \Pi_T$.
- A operação vazia ✓ é sempre interpretada em qualquer máquina.

Portanto, P é um **programa para uma máquina** M se todos os identificadores de operações e testes utilizados em P estiverem definidos, em M, através das correspondentes funções de operações e testes, respectivamente.

Exemplo 1.12 Exemplos de programas para a Máquina de Um Registrador. Começamos com um programa monolítico:

```
1:  faça add vá_para 2
2:  faça add vá_para 3
```

Este programa monolítico adiciona 2 no registrador. Exemplo de programa iterativo que faz a mesma coisa:

```
(add;add)
```

Exemplo de programa recursivo que dobra o valor da entrada:

```
P é R onde
R def se zero então ✓senão (sub;R;add;add)
```

⋆⋆⋆

Exemplo 1.13 Exemplos de programas para a Máquina de Dois Registradores. Começamos com um programa monolítico:

```
1:  se a_zero vá_para 4 senão vá_para 2
2:  faça subtrai_a vá_para 3
3:  faça adiciona_b vá_para 1
```

Programa iterativo:

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

Programa recursivo:

```
P é R onde
R def se a_zero então ✓senão (S;R),
```

```
S def (subtrai_a; adiciona_b)
```

Os três programas deste exemplo fazem a mesma coisa: eles copiam o dado da entrada para a saída.
⋆ ⋆ ⋆

Um programa não precisa usar todas as operações e testes definidos pela máquina. Para que um programa possa ser executado numa máquina é suficiente que as operações e testes do programa sejam implementados na máquina. O conjunto de operações e testes do programa pode ser um subconjunto das operações e testes implementados na máquina.

1.4 Computação

A noção de computação também é familiar para quem é da área homônima. Aqui, no entanto, vamos definir **computação** como sendo a sequência de configurações que a execução de um programa assume desde o seu início até o seu término (se ele terminar). A noção de **configuração** será, portanto, específica para cada tipo paradigma considerado. Uma computação é dita **finita** quando a sequência de configurações é finita. Caso contrário, ela é dita **infinita**.

1.4.1 Computação de programa monolítico

Sejam uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_O, \Pi_T)$ e um programa monolítico $P = (I, r)$ para M, e suponha que R é o conjunto de rótulos de P. Uma **computação do programa monolítico** P na máquina M é uma cadeia de elementos de $R \times V$:

$$(r_0, v_0)(r_1, v_1)(r_2, v_2)...$$

onde $r_0 = r$ é o rótulo inicial de P e v_0 é o conteúdo (total) inicial da memória de M.

- Essa cadeia indica a sequência de configurações que são assumidas pela máquina M durante a execução do programa P.
- Uma computação pode ser finita ou infinita.

Os pares $(r_{k+1}, v_{k+1}), k \geq 0$, são obtidos a partir dos pares (r_k, v_k), a partir da análise do tipo da instrução rotulada por r_k:

- r_k : `faça F vá_para` r'
  $(r_{k+1}, v_{k+1}) = (r', \pi_F(v_k))$
- r_k : `faça` ✓`vá_para` r'
  $(r_{k+1}, v_{k+1}) = (r', v_k)$
- r_k : `se T então vá_para` r' `senão vá_para` r''
  se $\pi_T(v_k)$=*verdadeiro*, então $(r_{k+1}, v_{k+1}) = (r', v_k)$
  se $\pi_T(v_k)$=*falso*, então $(r_{k+1}, v_{k+1}) = (r'', v_k)$

Exemplo 1.14 Considere o programa monolítico P:

```
1:  se a_zero vá_para 4 senão vá_para 2
2:  faça subtrai_a vá_para 3
```

```
3:  faça adiciona_b vá_para 1
4:  faça adiciona__b vá_para 5
```

A computação de P na Máquina de Dois Registradores M_D é (supondo que o valor inicial da memória é $(3, 0)$):

(1,(3,0)) (2,(3,0)) (3,(2,0)) (1,(2,1)) (2,(2,1)) (3,(1,1)) (1,(1,2)) (2,(1,2)) (3,(0,2)) (1,(0,3)) (4,(0,3)) (5, (0,4))

e, portanto, finita. ⋆⋆⋆

Exemplo 1.15 Considere o programa monolítico Q:

```
1:  faça adiciona_b vá_para 2
2:  faça adiciona_b vá_para 1
```

A computação de Q na Máquina de Dois Registradores M_D é (supondo que o valor inicial da memória é $(3, 0)$):

(1,(3,0)) (2,(3,1)) (1,(3,2)) (2,(3,3)) (1,(3,4)) (2,(3,5)) (1,(3,6)) (2,(3,7)) (1,(3,8)) (2,(3,9)) ...

e, portanto, infinita. ⋆⋆⋆

1.4.2 Computação de programa iterativo

Sejam uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_O, \Pi_T)$ e um programa iterativo P para M. Uma **computação do programa iterativo** P na máquina M é uma cadeia de elementos de $I \times V$:

$$(i_0, v_0)(i_1, v_1)(i_2, v_2)...$$

onde I é um conjunto de programas iterativos, $i_0 = P$; ✓ e v_0 é o conteúdo (total) inicial da memória de M.

- Essa cadeia indica a sequência de configurações que são assumidas pela máquina M durante a execução do programa P.
- Uma computação pode ser finita ou infinita.

Os pares $(i_{k+1}, v_{k+1}), k \geq 0$, são obtidos a partir dos pares (i_k, v_k), a partir da análise do tipo da instrução inicial de i_k. Considere que U, W e Z são programas iterativos, F é identificador de operação e T é identificador de teste. Então:

- i_k=✓
  A computação termina com o valor v_k na memória.
- i_k=`F;U`
  $(i_{k+1}, v_{k+1}) = (U, \pi_F(v_k))$
- i_k=`se T então U senão W;Z`
  se $\pi_T(v_k)$=*verdadeiro*, $(i_{k+1}, v_{k+1}) = (U; Z, v_k)$
  se $\pi_T(v_k)$=*falso*, $(i_{k+1}, v_{k+1}) = (W; Z, v_k)$

- i_k=`enquanto T faça U;W`
  se $\pi_T(v_k)$=*verdadeiro*, $(i_{k+1}, v_{k+1}) = (U;$ enquanto T faça $U; W, v_k)$
  se $\pi_T(v_k)$=*falso*, $(i_{k+1}, v_{k+1}) = (W, v_k)$

- i_k=`até T faça U;W`
  se $\pi_T(v_k)$=*falso*, $(i_{k+1}, v_{k+1}) = (U;$ até T faça $U; W, v_k)$
  se $\pi_T(v_k)$=*verdadeiro*, $(i_{k+1}, v_{k+1}) = (W, v_k)$

Observe que, diferentemente do que acontece com os programas monolíticos, no caso de um programa iterativo não é possível registrar o seu estado atual de execução apenas anotando o rótulo da próxima instrução a ser executada (pois não existem rótulos em programas iterativos). Assim, a única forma de se registrar o estado do programa é anotando toda a parte do programa que ainda resta ser executada (e que também é um programa iterativo).

Exemplo 1.16 Considere o programa iterativo P:

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

A computação de P na Máquina de Dois Registradores M_D é (supondo que o valor inicial da memória é $(2, 0)$):

(até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(2,0))
(subtrai_a; adiciona_b; adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(2,0))
(adiciona_b; adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(1,0))
(adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(1,1))
(até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(1,2))
(subtrai_a; adiciona_b; adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(1,2))
(adiciona_b; adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(0,2))
(adiciona_b; até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(0,3))
(até a_zero faça (subtrai_a; adiciona_b; adiciona_b);✓,(0,4))
(✓,(0,4))

A computação é, portanto, finita. ⋆⋆⋆

Exemplo 1.17 Considere o programa iterativo Q:

```
enquanto a_zero faça ✓
```

A computação de Q na Máquina de Dois Registradores M_D é (supondo que o valor inicial da memória é $(0, 0)$):

(enquanto a_zero faça ✓,(0,0))
(✓; enquanto a_zero faça ✓,(0,0))
(enquanto a_zero faça ✓,(0,0))
(✓; enquanto a_zero faça ✓,(0,0))
(enquanto a_zero faça ✓,(0,0))
(✓; enquanto a_zero faça ✓,(0,0))
...
A computação é, portanto, infinita. ⋆⋆⋆

1.4.3 Computação de programa recursivo

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

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

Uma **computação do programa recursivo** P na máquina M é uma cadeia de elementos de $J \times V$:

$$(j_0, v_0)(j_1, v_1)(j_2, v_2)...$$

onde J é um conjunto de expressões, $j_0 = E_0$; ✓ e v_0 é o conteúdo (total) inicial da memória de M.

- Essa cadeia indica a sequência de configurações que são assumidas pela máquina M durante toda a execução do programa P.
- Uma computação pode ser finita ou infinita.

Os pares $(j_{k+1}, v_{k+1}), k \geq 0$, são obtidos a partir dos pares (j_k, v_k), a partir da análise do tipo da instrução inicial de j_k. Considere que U, W e Z são expressões, F é identificador de operação, T é identificador de teste e R_i é identificador de subprograma. Então:

- j_k=✓;U
  $(j_{k+1}, v_{k+1}) = (U, v_k)$
- j_k=F;U
  $(j_{k+1}, v_{k+1}) = (U, \pi_F(v_k))$
- j_k=R_i;U
  $(j_{k+1}, v_{k+1}) = (E_i; U, v_k)$
- j_k=se T então U senão W;Z
  se $\pi_T(v_k)$=*verdadeiro*, $(j_{k+1}, v_{k+1}) = (U; Z, v_k)$
  se $\pi_T(v_k)$=*falso*, $(j_{k+1}, v_{k+1}) = (W; Z, v_k)$

A computação de um programa recursivo corresponde à avaliação da expressão inicial (E_0). Durante essa avaliação (que corresponde ao processo de determinar o valor da expressão ou gerar seus efeitos colaterais), se um identificador de subprograma for usado, então a correspondente expressão deve ser empregada em seu lugar (ocorre uma substituição, como acontece com a expansão de macros). A avaliação continua até que a expressão se torne apenas ✓.

Exemplo 1.18 Considere o programa recursivo para a Máquina de Um Registrador do Exemplo 1.12. Avaliar a expressão R (expressão inicial deste programa) significa substituir o nome do subprograma (R) pela sua definição. Assim, gera-se a nova expressão:

se zero então ✓ senão (sub;R;add;add)

Se o conteúdo da memória no instante da avaliação do teste for zero, então esta expressão se transforma em ✓. Caso contrário, se transforma em sub;R;add;add. A avaliação desta última, por sua vez,

produz um efeito colateral ao decrementar o registrador (operação `sub`). Em seguida, procede-se com a avaliação de `R;add;add` (o que irá provocar uma nova substituição) e assim por diante até que a expressão seja apenas ✓. ⋆⋆⋆

Observe que, diferentemente do que acontece com os programas monolíticos, e de forma similar ao que acontece com os programas iterativos, no caso de um programa recursivo não é possível registrar o seu estado atual de execução apenas anotando o rótulo da próxima instrução a ser executada (pois não existem rótulos em programas recursivos). Assim, a única forma de se registrar o estado do programa é anotando a expressão que ainda resta ser avaliada.

Exemplo 1.19 Considere o programa recursivo P:

```
P é R onde
R def se a_zero então ✓senão (S;R),
S def subtrai_a; adiciona_b
```

A computação de P na Máquina de Dois Registradores M_D é (supondo que o valor inicial da memória é $(2, 0)$):

(R; ✓,(2,0))
((se a_zero então ✓ senão (S; R));✓,(2,0))
(S; R;✓,(2,0))
(subtrai_a; adiciona_b;R;✓,(2,0))
(adiciona_b;R;✓,(1,0))
(R; ✓,(1,1))
((se a_zero então ✓ senão (S; R));✓,(1,1))
...
(S; R;✓,(1,1))
(subtrai_a; adiciona_b;R;✓,(1,1))
(adiciona_b;R;✓,(0,1))
(R; ✓,(0,2))
((se a_zero então ✓ senão (S; R));✓,(0,2))
(✓;✓,(0,2))
(✓,(0,2))

A computação é, portanto, finita. ⋆⋆⋆

Exemplo 1.20 Considere o programa recursivo Q:

```
Q é R onde R def R
```

A computação de Q na Máquina de Dois Registradores M_D (supondo que o valor inicial da memória é $(2, 0)$):

(R; ✓,(2,0)) (R; ✓,(2,0)) (R; ✓,(2,0)) (R; ✓,(2,0)) (R; ✓,(2,0)) (R; ✓,(2,0)) (R; ✓,(2,0) ...

A computação é, portanto, infinita. ⋆⋆⋆

Exemplo 1.21 Considere o programa recursivo S:

```
P é R onde R def se zero então ✓senão (sub;R;add;add)
```

A computação de S na Máquina de Um Registrador M_U é (supondo que o valor inicial da memória é 2):

(R; ✓,2)
(se zero então ✓ senão (sub;R;add;add);✓,2)
(sub;R;add;add;✓,2)
(R;add;add;✓,1)
(se zero então ✓ senão (sub;R;add;add);add;add;✓,1)
(sub;R;add;add;add;add;✓,1)
(R;add;add;add;add;✓,0)
(se zero então ✓ senão (sub;R;add;add);add;add;add;add;✓,0)
(✓;add;add;add;add;✓,0)
...
(add;add;add;add;✓,0)
(add;add;add;✓,1)
(add;add;✓,2)
(add;✓,3)
(✓,4)

A computação é, portanto, finita. ⋆⋆⋆

1.5 Função computada

A noção de computação, apresentada na Seção 1.4, envolve apenas a obtenção de um valor de memória final a partir de um valor de memória inicial por intermédio da execução sequencial das operações de um programa. A noção de função computada é mais abrangente e engloba a composição da função de entrada com a computação e com a função de saída, nesta ordem. Ela exprime o significado do programa, aquilo que ele gera na saída a partir dos dados de entrada.

A noção de **função computada** corresponde à combinação da função de entrada, com a computação e com a função de saída de um programa. Portanto, função computada é uma noção que associa valores de saída com valores de entrada. Os valores de saída correspondem a uma computação realizada sobre os dados de entrada. Naturalmente, a função computada é definida apenas quando a computação é finita. Além disso, ela é definida de maneiras diferentes (mas nem tanto) para os paradigmas considerados. Via de regra, a função computada por um programa P corresponde, nesta ordem, à:

- Aplicação da função de entrada π_X ao dado de entrada.
- Execução da computação (finita).
- Aplicação da função de saída π_Y ao valor final de memória.

1.5.1 Função computada de programa monolítico

Sejam uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_O, \Pi_T)$ e um programa monolítico P para M. A **função computada pelo programa monolítico** P na máquina M, denotada:

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

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

$$(r_0, v_0)(r_1, v_1)...(r_n, v_n)$$

é uma computação finita de P em M, onde:

- r_0 é o rótulo inicial de P
- $v_0 = \pi_X(x)$

A imagem de x, denotada $\langle P, M\rangle(x)$, é dada por $\pi_Y(v_n)$.

Exemplo 1.22 Considere a Máquina de Dois Registradores M_D e o programa monolítico P abaixo:

```
1:  se a_zero vá_para 5 senão vá_para 2
2:  faça subtrai_a vá_para 3
3:  faça adiciona_b vá_para 4
4:  faça adiciona_b vá_para 1
```

Então:

- $\langle P, M_D\rangle : \mathbb{N} \to \mathbb{N}$.
- $\forall n \in \mathbb{N}, \langle P, M_D\rangle(n) = 2 * n$.
- P produz na saída o dobro do dado de entrada
- Exemplo:
  - $\pi_X(3) = (3, 0)$
  - A computação de P em M_D produz o valor final $(0, 6)$
  - $\pi_Y(0, 6) = 6$
  - Portanto, $\langle P, M_D\rangle(3) = 6$

⋆ ⋆ ⋆

1.5.2 Função computada de programa iterativo

Sejam uma máquina $M = (V, X, Y, \pi_X, \pi_Y, \Pi_O, \Pi_T)$ e um programa iterativo P para M. A **função computada pelo programa iterativo** P na máquina M, denotada:

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

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

$$(i_0, v_0)(i_1, v_1)(i_2, v_2)...(i_n, v_n)$$

é uma computação finita de P em M, onde:

- $i_0 = P; \checkmark$
- $v_0 = \pi_X(x)$

A imagem de x, denotada $\langle P, M\rangle(x)$, é dada por $\pi_Y(v_n)$.

Exemplo 1.23 Considere a Máquina de Dois Registradores M_D e o programa iterativo P abaixo:

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

Então:

- $\langle P, M_{D} \rangle : \mathbb{N} \to \mathbb{N}$
- $\forall n \in \mathbb{N}, \langle P, M_{D} \rangle(n) = 3 * n$
- P produz na saída o triplo do dado de entrada
- Exemplo:
  - $\pi_{X}(3) = (3, 0)$
  - A computação de P em M_{D} produz o valor final $(0, 9)$
  - $\pi_{Y}(0, 9) = 9$
  - Portanto, $\langle P, M_{D} \rangle(3) = 9$

⋆ ⋆ ⋆

1.5.3 Função computada de programa recursivo

Sejam uma máquina $M = (V, X, Y, \pi_{X}, \pi_{Y}, \Pi_{O}, \Pi_{T})$ e um programa recursivo P para M. A **função computada pelo programa recursivo** P na máquina M, denotada:

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

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

$$(j_0, v_0)(j_1, v_1)(j_2, v_2)...(j_n, v_n)$$

é uma computação finita de P em M, onde:

- $j_0 = E_0; \checkmark$
- $v_0 = \pi_{X}(x)$

A imagem de x, denotada $\langle P, M \rangle(x)$, é dada por $\pi_{Y}(v_n)$.

Exemplo 1.24 Considere a Máquina de Um Registrador M_{U} e o programa recursivo P abaixo:

```
P é R onde R def se zero então ✓senão (sub;R;add;add;add;add)
```

Então:

- $\langle P, M_{U} \rangle : \mathbb{N} \to \mathbb{N}$
- $\forall n \in \mathbb{N}, \langle P, M_{U} \rangle(n) = 4 * n$
- P quadruplica na saída o dado de entrada
- Exemplo:
  - $\pi_{X}(3) = 3$
  - A computação de P em M_{U} produz o valor final 12
  - $\pi_{Y}(12) = 12$
  - Portanto, $\langle P, M_{U} \rangle(3) = 12$

⋆ ⋆ ⋆

1.6 Equivalência forte de programas

Dois programas P e Q, de quaisquer tipos, são ditos **fortemente equivalentes**, denotado $P \equiv Q$ se, e somente se, $\forall M, \langle P, M\rangle = \langle Q, M\rangle$, ou seja, P e Q são fortemente equivalentes se, e somente se, as respectivas funções computadas coincidem para qualquer máquina M que se possa considerar.

- Essa relação induz a uma partição do conjunto universo dos programas em classes de equivalências.
- Ela permite analisar, de forma comparativa, propriedades exibidas pelos programas, como é o caso da sua complexidade estrutural ou do consumo de recursos como tempo e espaço.

A fim de prosseguir, antecipamos um resultado que será demonstrado mais adiante (na Seção 1.9): as computações de programas (de quaisquer tipos) fortemente equivalentes executam as mesmas operações na mesma ordem.

Exemplo 1.25 Os programas monolíticos P_1 e P_2 abaixo são fortemente equivalentes:

P_1:
```
1:  se T vá_para 2 senão vá_para 3
2:  faça F vá_para 1
```

P_2:
```
1:  se T vá_para 2 senão vá_para 4
2:  faça F vá_para 3
3:  se T vá_para 1 senão vá_para 4
```

Computação de P_1 com a entrada x:

$(1, \pi_X(x))$
$(2, \pi_X(x))$
$(1, \pi_F(\pi_X(x)))$
$(2, \pi_F(\pi_X(x)))$
$(1, \pi_F^2(\pi_X(x)))$
$(2, \pi_F^2(\pi_X(x)))$
...
$(1, \pi_F^n(\pi_X(x)))$
$(3, \pi_F^n(\pi_X(x)))$

Portanto, $\langle P_1, M\rangle(x) = \pi_Y(\pi_F^n(\pi_X(x)))$.

Computação de P_2 com a entrada x:

$(1, \pi_X(x))$
$(2, \pi_X(x))$
$(3, \pi_F(\pi_X(x)))$
$(1, \pi_F(\pi_X(x)))$
$(2, \pi_F(\pi_X(x)))$
$(3, \pi_F^2(\pi_X(x)))$
...

$(1, \pi_F^{n-1}(\pi_X(x)))$
$(2, \pi_F^{n-1}(\pi_X(x)))$
$(3, \pi_F^{n}(\pi_X(x)))$
$(4, \pi_F^{n}(\pi_X(x)))$

Portanto, $\langle P_2, M\rangle(x) = \pi_Y(\pi_F^n(\pi_X(x)))$.

Supondo que T retorna *falso* após n execuções da operação F em qualquer caso, então $P_1 \equiv P_2$. $\star\star\star$

Exemplo 1.26 Os programas P_3 (iterativo) e P_4 (recursivo) abaixo são fortemente equivalentes:
P_3:
```
enquanto T
faça F
```

P_4:
```
P4 é R onde
R def se T então (F;R) senão ✓
```

Computação de P_3 com a entrada x:

(enquanto T faça F; ✓, $\pi_X(x)$)
(F; enquanto T faça F; ✓, $\pi_X(x)$)
(enquanto T faça F; ✓, $\pi_F(\pi_X(x))$)
(F; enquanto T faça F; ✓, $\pi_F(\pi_X(x))$)
(enquanto T faça F; ✓, $\pi_F^2(\pi_X(x))$)
(F; enquanto T faça F; ✓, $\pi_F^2(\pi_X(x))$)
...
(enquanto T faça F; ✓, $\pi_F^n(\pi_X(x))$)
(✓, $\pi_F^n(\pi_X(x))$)

Portanto, $\langle P_3, M\rangle(x) = \pi_Y(\pi_F^n(\pi_X(x)))$.

Supondo que T retorna *falso* após n execuções da operação F em qualquer caso, então $P_1 \equiv P_2 \equiv P_3$.

Computação de P_4 com a entrada x:

(R; ✓, $\pi_X(x)$)
((se T então (F; R) senão ✓);✓, $\pi_X(x)$)
(F; R;✓, $\pi_X(x)$)
(R;✓, $\pi_F(\pi_X(x))$)
((se T então (F; R) senão ✓);✓, $\pi_F(\pi_X(x))$)
(F; R;✓, $\pi_F(\pi_X(x))$)
(R;✓, $\pi_F^2(\pi_X(x))$)
...
(R;✓, $\pi_F^n(\pi_X(x))$)
((se T então (F; R) senão ✓);✓, $\pi_F^n(\pi_X(x))$)
(✓;✓, $\pi_F^n(\pi_X(x))$)
(✓, $\pi_F^n(\pi_X(x))$)

Portanto, $\langle P_4, M\rangle(x) = \pi_Y(\pi_F^n(\pi_X(x)))$.

Supondo que T retorna *falso* após n execuções da operação F em qualquer caso, então $P_1 \equiv P_2 \equiv P_3 \equiv P_4$. ⋆⋆⋆

1.6.1 Iterativos ⊆ Monolíticos

O Teorema 1.1 mostra que todo programa iterativo possui um programa monolítico fortemente equivalente.

Teorema 1.1 (Iterativos ⊆ Monolíticos) "Seja P_I um programa iterativo. Então, existe um programa monolítico P_M tal que $P_M \equiv P_I$."

A obtenção de um programa P_M monolítico a partir de P_I é direta, a partir do mapeamento das construções elementares de um programa iterativo em sequências de construções equivalentes em um programa monolítico. Afinal, lembre-se que todo fluxograma estruturado é também um fluxograma geral (o contrário, no entanto, não é verdadeiro). Em outras palavras, todo programa iterativo é também um programa monolítico (mas o contrário não é verdadeiro). A Figura 1.8 ilustra os componentes de um fluxograma estruturado (conforme visto na Seção 1.1), os quais, como é fácil perceber, podem ser representados diretamente como um fluxograma qualquer. Desta maneira, dado P_I qualquer, o P_M que pode ser construído a partir de P_I é o próprio P_I. □□□

Exemplo 1.27 Considere o seguinte programa iterativo P_I:

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

O programa monolítico P_M abaixo, obtido por mapeamento direto, é fortemente equivalente a P_I:

```
1:  se a_zero vá_para 4 senão vá_para 2
2:  faça subtrai_a vá_para 3
3:  faça adiciona_b vá_para 1
```

⋆⋆⋆

1.6.2 Iterativos ≠ Monolíticos

O Teorema 1.2 prova que a relação de inclusão mostrada no Teorema 1.1 é própria.

Teorema 1.2 (Iterativos ≠ Monolíticos) "Existe um programa monolítico que não possui um iterativo fortemente equivalente."

Ou seja, deseja-se provar que, dado um programa monolítico P_M qualquer, não necessariamente existe um programa iterativo P_I tal que $P_I \equiv P_M$. É suficiente mostrar que existe pelo menos um programa monolítico que, para uma determinada máquina, não apresente nenhum programa iterativo que seja fortemente equivalente.

- Considere o programa monolítico P_M da Figura 1.9 e a máquina de um registrador M_U.
- $\langle P_M, M_U \rangle(n) = 1$ se n é par ou 0 se n é ímpar.
- A sequência de operações contida na computação de P_M para um valor de entrada n é:

$$\underbrace{sub; sub; ...; sub}_{n}, \text{ se } n \text{ é ímpar}$$

$$\underbrace{sub; sub; ...; sub}_{n}; add, \text{ se } n \text{ é par}$$

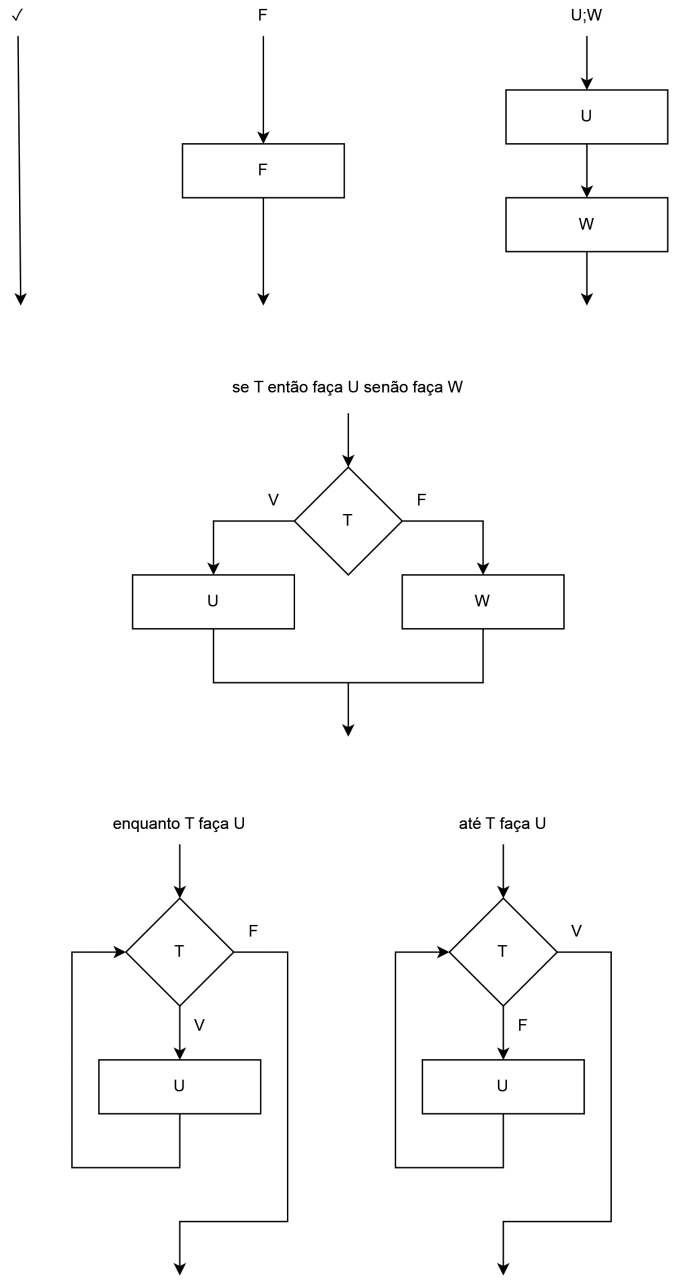


Figura 1.8 Componentes de um programa iterativo.

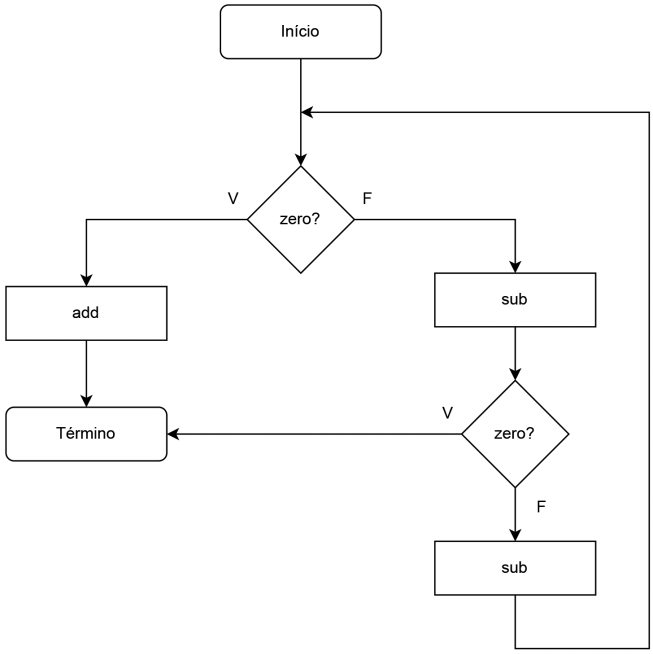


Figura 1.9 Programa monolítico que determina se um número natural é par ou ímpar.

- Suponha que P_I contém k operações sub.
- Suponha um valor de entrada $n > k$.
- Como, por hipótese, $P_I \equiv P_M$, a mesma sequência de operações é executada por P_I.
- Como $n > k$, pelo menos uma instrução sub é executada mais de uma vez na computação de P_I.
- Logo, existe uma instrução iterativa do tipo "enquanto" ou "até" que controla a execução dessa operação sub.
- Ao término da execução dessa instrução, no entanto, não é possível contabilizar a quantidade de execuções da operação sub que foram executadas no loop e, consequentemente, distinguir a condição "par" ou "ímpar" do valor n de entrada.
- Note que em P_M a avaliação do primeiro (segundo) teste implica a execução de um número par (ímpar) de subtrações.
- Logo, P_I não é capaz de produzir o resultado desejado.
- Portanto, não existe P_I que tal que $P_I \equiv P_M$.
- Não é possível construir um programa iterativo para M_U, que determine se a entrada é par.

□□□

1.6.3 Monolíticos $\subseteq$ Recursivos

O Teorema 1.3 mostra que todo programa monolítico possui um programa recursivo fortemente equivalente.

Teorema 1.3 (Monolíticos $\subseteq$ Recursivos) "Seja P_M um programa monolítico. Então, existe um programa recursivo P_R tal que $P_R \equiv P_M$."

A prova consiste na construção de um programa recursivo P_R que execute as mesmas operações na mesma ordem que P_M. Esta construção é explicada a seguir.

Suponha que $L = \{r_1, r_2, ..., r_n\}$ seja o conjunto de rótulos de P_M, que r_1 seja o rótulo inicial e que r_n seja o (único) rótulo final de P_M. Então:

P_R é R_1 onde
R_1 def E_1,
R_2 def E_2,
...
R_n def ✓

e, $\forall k, 1 \leq k < n$, E_k é definido da seguinte forma:

- Se r_k : faça F vá_para r_i então:

  E_k=F; R_i

- Se r_k : se T então vá_para r_i senão vá_para r_j então:

  E_k=(se T então R_i senão R_j)

é fortemente equivalente a P_M, ou seja, $P_R \equiv P_M$ □□□

Exemplo 1.28 Considere o programa monolítico Q:

```
1:  se a_zero vá_para 4 senão vá_para 2
2:  faça subtrai_a vá_para 3
3:  faça adiciona_b vá_para 1
```

O programa recursivo R abaixo, obtido por mapeamento direto, é fortemente equivalente a Q:

R é R_1 onde
R_1 def (se a_zero R_4 senão R_2)
R_2 def (subtrai_a; R_3)
R_3 def (adiciona_b; R_1)
R_4 def ✓

⋆⋆⋆

A relação equivalência forte é transitiva, como mostra o Corolário 1.1.

Corolário 1.1 (Iterativos $\subseteq$ Recursivos) "Seja P_I um programa iterativo. Então, existe um programa recursivo P_R tal que $P_R \equiv P_I$."

Prova: conforme demonstrado no Teorema 1.1, todo programa iterativo possui um programa monolítico fortemente equivalente. De acordo com o Teorema 1.3, todo programa monolítico possui um programa recursivo fortemente equivalente. Logo, para qualquer programa iterativo P_I existe um programa recursivo P_R tal que $P_R \equiv P_I$. □□□

1.6.4 Monolíticos $\neq$ Recursivos

O Teorema 1.4 prova que a relação de inclusão mostrada no Teorema 1.3 é própria.

Teorema 1.4 (Monolíticos $\neq$ Recursivos) "Existe um programa recursivo que não possui um programa monolítico que lhe seja fortemente equivalente."

Deseja-se provar que, dado um programa recursivo P_R qualquer, não necessariamente existe um programa monolítico P_M tal que $P_M \equiv P_R$. Para isso, é suficiente mostrar que existe pelo menos um programa recursivo que, para uma determinada máquina, não apresente nenhum programa monolítico que lhe seja fortemente equivalente.

- Considere o programa recursivo P_R abaixo e a máquina de um registrador M_U:

```
P_R é R onde
R def se zero então ✓senão (sub;R;add;add)
```

- $\langle P_R, M_U \rangle(n) = 2n$.
- A sequência de operações contida na computação de P_R para um valor de entrada n é:

$$\underbrace{sub; sub; ...; sub}_{n}; \underbrace{add; add; ...; add}_{2n}$$

- Suponha que P_M contém k operações add (cada uma numa instrução diferente).
- Suponha um valor de entrada $n > k/2$.
- Como, por hipótese, $P_M \equiv P_R$, a mesma sequência de operações é executada por P_M.
- Como $2n > k$, pelo menos uma mesma instrução add é executada mais de uma vez na computação de P_M.
- Isso significa que há um desvio incondicional que permite a execução repetida dessa instrução, pois não seria possível, com um único registrador, controlar a execução do loop e ainda assim dobrar o valor da entrada.
- Há, portanto, um ciclo infinito em P_M envolvendo essa instrução add.
- Logo, a computação de P_M não pode ser finita e isso contradiz a hipótese da existência de P_M.
- Não existe P_M tal que $P_M \equiv P_R$.
- Não é possível construir um programa monolítico para M_U que dobre o valor da entrada.

□□□

1.6.5 Conclusões

A Figura 1.10 resume os resultados dos teoremas anteriores, lembrando que, em cada caso, a relação de inclusão é própria.

Observações importantes:

- Equivalência forte de programas $\neq$ poder computacional.
- Os três formalismos (monolítico, iterativo e recursivo) possuem o mesmo poder computacional.

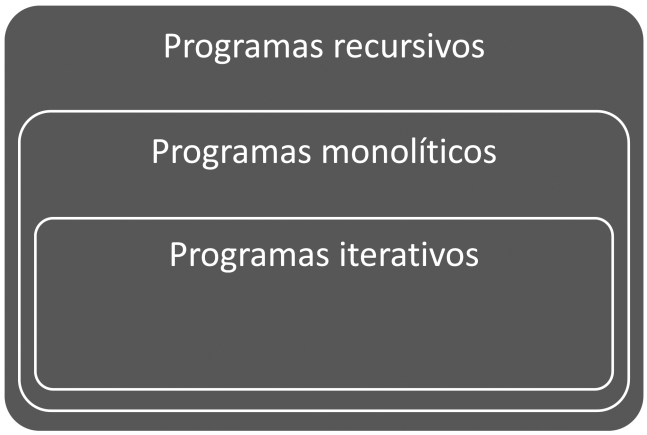


Figura 1.10 Relação de equivalência forte.

- Para qualquer programa recursivo e para qualquer máquina, existe um programa monolítico e uma máquina tal que as funções computadas coincidem.

- Para qualquer programa monolítico e para qualquer máquina, existe um programa iterativo e uma máquina tal que as funções computadas coincidem.

Logo, do ponto de vista do poder computacional, os três paradigmas são equivalentes. Em outras palavras, dada uma máquina M e um programa P para essa máquina, sempre existem uma máquina M' e um programa P' tal que as funções computadas coincidem. O Teorema de Böhm-Jacopini mostra como gerar programas iterativos equivalentes a programas monolíticos dados como entrada. Não necessariamente as mesmas operações e a mesma ordem são usadas, tampouco o resultado vale para qualquer máquina. Portanto, o fato de existir um programa monolítico que não possui um programa iterativo fortemente equivalente (conforme o Teorema 1.2) não invalida o Teorema de Böhm-Jacopini. Para mais detalhes, veja a Seção 1.10, Teorema 1.6.

1.7 Equivalência de programas em uma máquina

A noção de equivalência forte de programas, introduzida na Seção 1.6, exige que dois programas quaisquer possuam a mesma função computada em qualquer máquina. Uma noção mais fraca é a noção de equivalência em uma máquina, a qual exige apenas que os programas em questão possuam a mesma função computada na máquina considerada.

Dois programas P e Q, de quaisquer tipos, são ditos **equivalentes na máquina** M, denotado $P \equiv_M Q$, se, e somente se, as correspondentes funções computadas na máquina M são iguais, ou seja $\langle P, M\rangle = \langle Q, M\rangle$. P e Q, nesse caso, são ditos programas M-equivalentes ou, simplesmente, programas equivalentes na máquina M.

O estudo da Teoria da Computação permite, entre outras coisas, a percepção da computação como uma área com muitas possibilidades, mas também com muitas limitações. Conhecer essas limitações é importante para a formação de profissionais que, em suas vidas, muitas vezes, irão se deparar com problemas intratáveis ou mesmo insolúveis.

De forma introdutória, este livro apresenta os tópicos mais importantes da Teoria da Computação, como Programas, máquinas e equivalências, Máquinas universais, Decidibilidade, Complexidade no tempo e Cálculo Lambda não tipado, que podem ser estudados de forma relativamente independente uns dos outros. Além disso, o livro traz um conjunto de 230 exercícios com as suas respectivas soluções, para ajudar na fixação do conteúdo.

ISBN 978-8-521-22524-9

Blucher

www.blucher.com.br

Clique aqui e:

VEJA NA LOJA

Teoria da computação

Marcus Vinicius Midena Ramos

ISBN: 9788521225249
Páginas: 480
Formato: 17 x 24 cm
Ano de Publicação: 2025