INTRODUÇÃO À COMPILAÇÃO

Scan: Alison Miazaki
Agregação para PDF: Bruno Duarte
Crop, OCR e Indexação: Eduardo Russo
Reprodução para fins didáticos autorizada pelo autor

Apresentação da Série

"Engenharia de Computação"

Destinados a um público interessado em adquirir ou consolidar conhecimentos profissionais e acadêmicos, os textos da série "Engenharia de Computação" têm como tônica apresentar, de maneira rigorosa porém acessível, os diversos aspectos relacionados à problemática da área.

Esta série concretiza uma resposta à crescente demanda de textos técnicos de boa qualidade, editados em língua portuguesa, que possam servir como livros-texto em disciplinas ligadas a cursos de graduação, pós-graduação e especialização, em Computação e áreas afins.

Diversos títulos que compõem a coleção tiveram sua origem em notas de aulas e apostilas, tendo tomado a forma de livros somente após uma cuidadosa evolução, resultante da utilização didática de sucessivas revisões do material, ao longo de diversos semestres de maturação.

Muito honrado por ter sido convidado a coordenar esta série, expresso uma grande satisfação por ter constatado, por parte da LTC, dos autores e dos futuros autores, o entusiasmo, o esforço e a boa vontade cujos frutos começam desde já a ser colhidos.

Esperando que esta iniciativa estimule muitos autores nacionais a enriquecer a escassa coleção de textos de categoria ora disponíveis, apresento meus agradecimentos à Escola Politécnica da USP e à Fundação para o Desenvolvimento Tecnológico da Engenharia pelo apoio que possibilitou a realização deste projeto.

João José Neto

Prof. Assistente – Doutor
da Escola Politécnica da USP

Prefácio

O estudo de compiladores em cursos de engenharia apresenta-se como um universo didático peculiar por sua natureza: de um lado, o assunto, tradicionalmente tratado de forma teórica e rigorosa em programas da área das ciências da computação, e de um outro lado, o estudante de engenharia, tradicionalmente avesso a teorias, e preocupado com aplicações dos conhecimentos disponíveis à resolução de problemas reais do seu dia-a-dia profissional. Torna-se necessário vencer esta barreira de conflitos.

Com a experiência adquirida em quinze anos de contato com estudantes de engenharia envolvidos com o problema da compilação de linguagens de programação, e com a sensibilidade amadurecida pela participação na elaboração de diversos projetos e implementações nesta área, surgiu a iniciativa de se efetuar uma pesquisa no sentido de se elaborar uma metodologia de projeto de compiladores voltado a transpor as dificuldades mencionadas.

Foi então desenvolvido, inicialmente, um modelo formal geral para a representação de reconhecedores sintáticos, e, a partir deste modelo, levantado um método para a obtenção de tais reconhecedores, que formarão o núcleo do compilador. O método em questão pode ser facilmente automatizado, já que se compõe de regras aplicáveis mecanicamente aos dados do problema a ser resolvido.

Um objetivo permanente norteou a busca da solução mencionada: a despeito de eventuais complexidades formais, o modelo deveria ser didático, e a sua aplicação, intuitiva. Como resultado do desenvolvimento deste método, surgiu a técnica apresentada neste livro.

Para construir reconhecedores, o engenheiro precisa apenas aprender a especificar a sintaxe da linguagem e a manipular esta especificação. A utilização de máquinas seqüenciais, de domínio do estudante, facilita o aprendizado e torna direta a assimilação do método. Nenhuma outra teoria é exigida.

A utilização desta técnica de ensino tem-se comprovado eficaz, e seu grande mérito tem sido o de desmistificar totalmente o assunto, permitindo que, em apenas alguns meses, o estudante domine a disciplina e aplique fluentemente os seus resultados.

Este livro é o coroamento dos esforços dirigidos pelo autor em direção às metas mencionadas. Resultado do acúmulo do material didático preparado para o ensino de diversas disciplinas da área, ministrado a platéias de diversos tipos de formação, durante mais de dez anos de magistério, e enriquecido pela inclusão de diversos resultados de pesquisa recente, este texto procura ao mesmo tempo dar uma formação básica sobre a problemática da compilação, apresentar um espectro significativo de soluções para os diversos problemas, e oferecer um roteiro de implementação aplicável a compiladores de uma vasta gama de linguagens de alto nível.

Destinado a atender leitores de diversas formações, o livro não exige conhecimentos anteriores da área, sendo entretanto desejável a familiaridade com as principais características de linguagens de alto nível, e, para o aproveitamento da parte teórica, um pouco de raciocínio abstrato e de boa vontade para a interpretação das notações simbólicas.

A leitura deste texto por parte de um iniciante, bem como no caso do seu emprego em cursos de até um semestre, pode ser efetuada de acordo com a seqüência de capítulos apresentada. Já para o leitor com certa prática em notações formais e na parte teórica, e interessado principalmente nas técnicas, os Capítulos 1 e 2, bem como 3.1 e 3.2 podem ser omitidos.

Para o profissional interessado em pôr imediatamente em prática as técnicas de construção de compiladores sem imediata necessidade de conhecer os seus porquês, o Capítulo 5 pode ser utilizado diretamente como um roteiro de projeto, servindo os demais como referências para consulta em casos de dúvidas ou para complementação de conhecimentos. Para o pesquisador, o texto apresenta-se como um ponto de partida para os estudos relacionados com o processamento de linguagens. Um conjunto significativo de textos de referência é incluído com esta finalidade.

A experiência mostrou que o pleno aprendizado do material aqui exposto só pode ser conseguido mediante a execução de um projeto completo, de preferência com o uso do computador. Uma série de exercícios de projeto é apresentada ao final do Capítulo 5, para orientar o interessado na parte prática do estudo dos compiladores.

Agradecimentos

Apesar de ser fruto de muitos anos de esforço, aprendizado e colaboração, este texto só se tornou realidade na forma em que se encontra devido ao apoio e ao estímulo da Escola Politécnica da Universidade de São Paulo, através da iniciativa de patrocinar a criação e publicação de material didático para os cursos do seu convênio com a FDTE (Fundação para o Desenvolvimento Tecnológico da Engenharia) e com o IPT (Instituto de Pesquisas Tecnológicas).

À EPUSP, então, ficam registrados os meus agradecimentos.

Inúmeras pessoas, que não serão citadas para que nenhuma seja omitida, deram valiosas colaborações a este trabalho. Incluem-se colegas de trabalho, alunos, estagiários, orientados de pós-graduação e amigos com quem troquei muitas idéias e a quem devo muito incentivo e muitas boas sugestões, pelas quais agradeço.

Algumas pessoas, em particular, contribuíram de maneira substancial para que a obra tomasse a forma e o conteúdo com que se apresenta. Especiais agradecimentos dirijo ao Engº Mário Eduardo Santos Magalhães, meu amigo e orientado de pós-graduação, pelas muitas horas de sono e de convívio familiar perdidos enquanto desenvolvíamos os modelos teóricos utilizados neste livro. Agradeço também ao meu amigo Engº Armando Santos Barbosa Júnior, pela inestimável colaboração na fase de estruturação do texto, bem como pela paciente leitura e crítica da primeira versão deste material.

Um agradecimento muito especial dirijo a Cheng Mei Lee, minha esposa e colaboradora, não apenas pelas valiosíssimas opiniões técnicas, grandes incentivos e rigorosas críticas que dela recebi, mas principalmente pela infinita paciência com que suportou as muitas horas de lazer que deixamos de usufruir juntos enquanto este texto estava sendo produzido.

Gostaria, finalmente, de agradecer à Sra. Aparecida Conceta Pompeo Tavares de Souza, pelo seu esforço de decifrar os meus manuscritos e transformá-los em um primoroso trabalho de datilografia.

São Paulo, maio de 1986

João José Neto

Sumário

PREFÁCIO, XI

AGRADECIMENTOS, XIII

1 – CONCEITOS BÁSICOS, 1

1.1 – Compiladores, Filtros e Pré-Processadores, 2
1.2 – Atividades Adicionais dos Compiladores, 4
1.3 – A Formalização das Linguagens de Programação – Gramáticas e Reconhecedores, 5
1.4 – Estratégias Para a Obtenção de Compiladores, 6
1.5 – Estruturação Lógica dos Compiladores, 10
1.6 – Organização Física dos Compiladores, 15
1.7 – Considerações Sobre Custos e Benefícios dos Compiladores, 22
1.8 – Leituras Complementares, 23
1.9 – Exercícios, 24

2 – INTRODUÇÃO À TEORIA DE LINGUAGENS, 26

2.1 – Terminologia Básica, 26
2.2 – Cadeias, 27
2.3 – Gramáticas, 29
2.4 – Reconhecedores, 33
2.5 – Gramáticas de Transdução, 38
2.6 – Transdutores, 40
2.7 – Uma Notação Estruturada para os Autômatos de Pilha, 41
2.8 – Notações para a Definição de Linguagens, 47
2.9 – Algumas Propriedades das Gramáticas Livres de Contexto, 56

2.10 – Leituras Complementares, 57
2.11 – Exercícios, 58

3 – TÉCNICAS DE CONSTRUÇÃO DE RECONHECEDORES, 62

3.1 – Manipulação de Definições Formais, 62
3.1.1 – Conversão Entre Notações Gramaticais, 63
3.1.2 – Conversão Entre Formatos de Reconhecedores, 70
3.2 – Algumas Soluções Para o Problema de Construção de Reconhecedores, 73
3.2.1 – Reconhecedores Não-Determinísticos ("backtracking"), 74
3.2.2 – Reconhecedores Determinísticos Descendentes ("top-down"), 75
3.2.3 – Reconhecedores Determinísticos Ascendentes ("bottom-up"), 82
3.3 – O Mapeamento de Gramáticas em Reconhecedores, 88
3.3.1 – Obtenção de Autômatos Finitos, 92
3.3.2 – O Mapeamento de Autômatos Finitos em Gramáticas Regulares, 104
3.3.3 – Obtenção de Autômatos de Pilha, 104
3.4 – Leituras Complementares, 111
3.5 – Exercícios, 112

4 – ESPECIFICAÇÃO DAS FUNÇÕES INTERNAS DO COMPILADOR, 116

4.1 – Análise Léxica, 117
4.1.1 – Funções do Analisador Léxico, 117
4.1.2 – Considerações Sobre a Implementação de Analisadores Léxicos, 121
4.2 – Análise Sintática, 126
4.2.1 – Funções da Análise Sintática, 126
4.2.2 – Considerações Sobre a Implementação de Analisadores Sintáticos, 128
4.3 – Análise Semântica e Geração de Código, 130
4.3.1 – Funções das Ações Semânticas do Compilador, 130
4.3.2 – Considerações Sobre a Implementação das Ações Semânticas, 133
4.4 – Leituras Complementares, 146
4.5 – Exercícios, 146

5 – A CONSTRUÇÃO DE UM COMPILADOR, 151

5.1 – Especificação Gerais do Compilador, 151
5.2 – Especificação da Linguagem-Fonte, 153
5.3 – Especificação da Linguagem de Saída, 155
5.4 – Projeto do Analisador Léxico, 158
5.5 – Projeto do Analisador Sintático, 163
5.6 – Projeto das Ações Semânticas, 186
5.6.1 – Estruturas de Dados do Compilador, 196
5.6.2 – Ações Semânticas, 198
5.7 – O Ambiente de Execução, 205
5.8 – Aspectos de Implementação e Integração do Compilador, 208
5.9 – Leituras Complementares, 213
5.10 – Exercícios de Projeto, 213

6 – BIBLIOGRAFIA, 215

7 – ÍNDICE REMISSIVO, 217

1

Conceitos Básicos

Nos seus primeiros contatos com o computador, todo programador certamente faz uso dos compiladores, na condição de usuário. Nestas circunstâncias, em geral a máquina e os programas de sistema, e em particular os compiladores, são vistos de modo extremamente mistificado, devido ao tipo de funções por eles executadas. No entanto, tais módulos de software, apesar de suas dimensões, muitas vezes consideráveis, não passam de programas semelhantes a tantos outros que existem. Apesar de apresentarem funções via de regra não convencionais, os compiladores baseiam-se em princípios relativamente simples, e que se tornam tanto mais compreensíveis quanto maior for o embasamento teórico do interessado. Felizmente, nenhuma espécie de teoria sofisticada é exigida para a compreensão dos seus mecanismos básicos de operação e de projeto. O presente texto tem como meta básica apresentar-se ao leitor como um material através do qual os fundamentos da teoria dos compiladores se tornem acessíveis como ferramentas para a análise e para o projeto, desmistificando, desta maneira, inúmeros aspectos geralmente considerados de grande dificuldade, e permitindo a compreensão dos seus mecanismos de funcionamento.

Nos computadores mais antigos, em que os conceitos de sistema de programação ainda não haviam evoluído suficientemente, bem como em máquinas experimentais, de uso restrito, os compiladores muitas vezes exerciam o seu papel como programas autônomos. Desta forma, sua interação com o programador era, às vezes, direta, exigindo uma operação específica e manual, através da qual o programa era transformado sucessivamente em formas intermediárias produzidas em meios de armazenamento externo, tais como fitas de papel ou cartões perfurados. O código-objeto final apresentava-se, em geral, também na forma de fitas ou cartões perfurados contendo programas-objeto em linguagem de máquina relocável, ou em formato fonte de linguagem simbólica de baixo nível. Um processamento adicional encarregava-se de converter tais saídas em programas executáveis, através da incorporação das rotinas do ambiente de execução e da transformação dos mesmos em código de máquina absoluto. O produto final desta operação era, no caso, também uma fita de papel ou uma seqüência de cartões perfurados.

Nas máquinas mais modernas, os compiladores são programas que operam de modo integrado aos demais componentes do sistema de programação disponível, fazendo parte do conjunto os recursos oferecido pelo sistema operacional ao qual está subordinado. Sua operação fica, desta maneira, muito simplificada, uma vez que o encadeamento de execução das suas diversas partes fica em geral aos cuidados do sistema operacional, e eventuais entradas e saídas intermediárias são efetuadas nos dispositivos de armazenamento de massa disponíveis no sistema. Assim sendo, torna-se importante que os compiladores exerçam de modo eficaz seu papel de interface com o sistema operacional, quer explorando os recursos por este oferecidos, com a finalidade de automatizar sua operação e facilitar seu uso por parte do programador, quer oferecendo a este, através de recursos lingüísticos da linguagem-fonte, acesso simples, direto e eficiente aos diversos serviços que o sistema operacional é capaz de prestar, dispensando o usuário da tarefa de comunicar-se com o sistema operacional através de linguagens de controle ou de utilitários, externamente aos seus programas de aplicação.

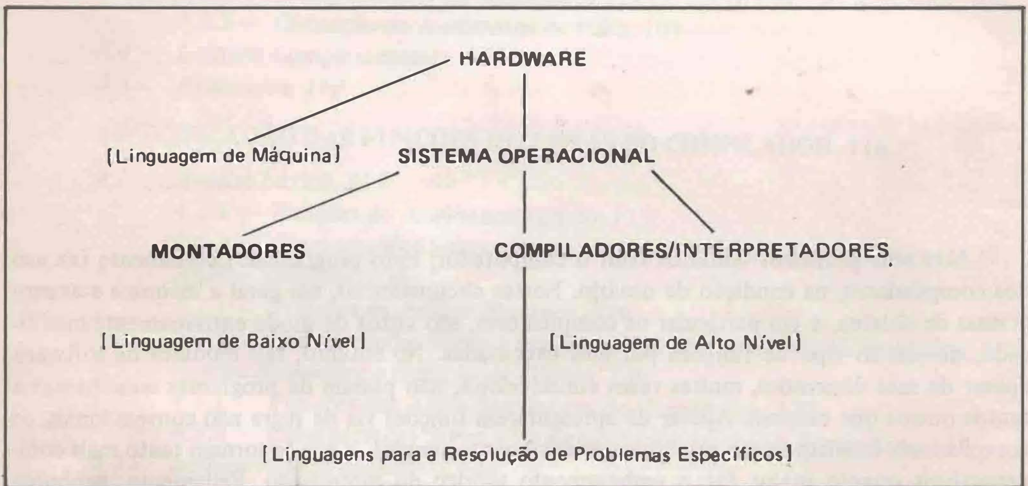


Figura 1.1 – Relacionamento dos compiladores com outros elementos de um sistema de programação.

1.1 – COMPILADORES, FILTROS E PRÉ-PROCESSADORES

Que vem a ser, afinal, um compilador? Trata-se de um dos módulos do software básico de um computador, cuja função é a de efetuar automaticamente a tradução de textos, redigidos em uma determinada linguagem de programação, para alguma outra forma que viabilize sua execução. Em geral esta forma é a de uma linguagem de máquina, embora esta seja apenas uma das inúmeras possíveis alternativas viáveis.

De modo geral, pode-se dizer que um tradutor, como pode ser classificado um programa desta natureza, efetua uma conversão entre duas linguagens: Dada uma linguagem de entrada, denominada *linguagem-fonte*, o compilador tem a função de converter automaticamente textos escritos nesta linguagem (os *textos-fonte*) em textos correspondentes equivalentes (*textos-objeto*), apresentados em uma linguagem de saída, denominada *linguagem-objeto*. Esta conversão deve preservar, no texto-objeto, todas as informações, contidas no texto-fonte, que contribuam para a execução correta do programa. A *Figura 1.2* mostra esquematicamente estas idéias.

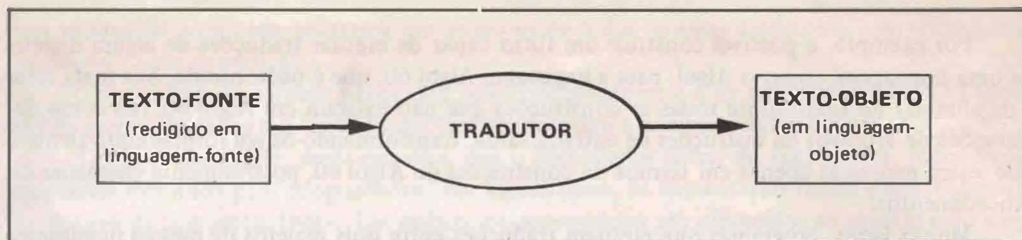


Figura 1.2 – Esquema de conversão efetuado por um tradutor (os textos de entrada e de saída devem ter o mesmo significado).

Denominam-se montadores (*"assemblers"*) aqueles tradutores em que a linguagem-fonte é de baixo nível, como é o caso das linguagens de montagem (*"assembly languages"*). Por tradição, tais tradutores particulares não são chamados de compiladores.

De maneira geral, os tradutores efetuam, portanto, a conversão de textos redigidos em uma linguagem, para formas equivalentes, redigidas em outra linguagem. Se a primeira linguagem for uma linguagem de alto nível, o tradutor receberá o nome de compilador (*Figura 1.3*).

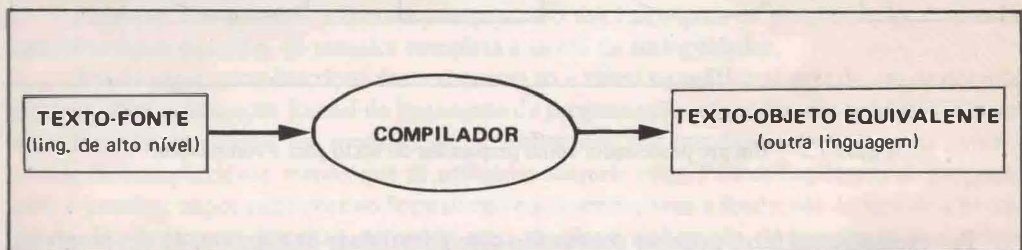


Figura 1.3 – Esquema de um compilador (tradutor de linguagens de alto nível).

Do ponto de vista da linguagem-objeto, esta também pode ser ou não uma linguagem de alto nível. É possível, por exemplo, que o texto-fonte esteja redigido em FORTRAN, e que um tradutor aceite tal texto e o converta para alguma linguagem de baixo nível (linguagem de montagem, ou linguagem de máquina) ou então para alguma outra linguagem de alto nível, tal como PASCAL ou BASIC. Um tradutor que efetue conversões entre duas linguagens de alto nível é denominado *filtro*, se a linguagem objeto for muito semelhante à linguagem fonte (*Figura 1.4*).

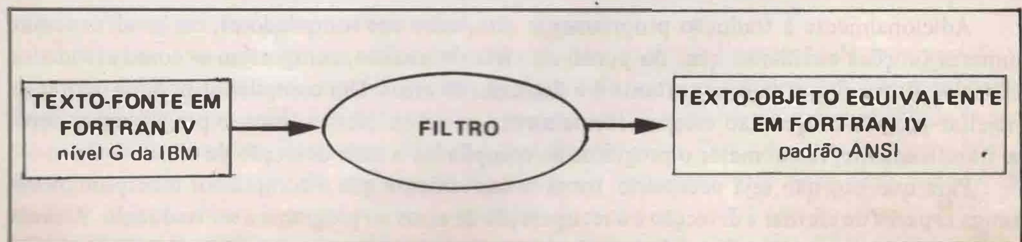


Figura 1.4 – Exemplo de um filtro (tradutor de linguagens próximas entre si).

Por exemplo, é possível construir um filtro capaz de efetuar traduções de algum dialeto de uma linguagem, como o Algol, para a linguagem Algol 60, que é padronizada. Sua meta seria a de eliminar do texto-fonte todas as construções que não existam em Algol 60, tais como declarações de arquivos ou instruções de entrada/saída, transformando-os em formas equivalentes, que sejam expressas apenas em termos de construções do Algol 60, possivelmente chamadas de procedimentos.

Muitas vezes, programas que efetuam traduções entre dois dialetos de mesma linguagem, ou que permitem converter para uma forma padronizada um texto que inclua extensões de uma linguagem disponível, são denominados *pré-processadores*. Basicamente, a função dos pré-processadores, neste caso, consiste na eliminação de construções de nível mais alto que o da linguagem-base, e na correspondente inserção de textos equivalentes, compostos exclusivamente de construções da linguagem-base (*Figura 1.5*).

Figura 1.5 – Um pré-processador como preparador de texto para a compilação.

Por exemplo, um filtro pode ser construído com a finalidade de eliminar, de um programa escrito em FORTRAN 77, todas as construções não disponíveis em FORTRAN IV básico, substituindo-as por combinações equivalentes, das construções desta linguagem.

Outro exemplo de aplicação de pré-processadores é no processamento da definição e utilização de macros em linguagens de alto nível. Em muitos casos, os compiladores destas linguagens não oferecem ao usuário tais recursos, sendo as macros manipuladas externamente, por um pré-processador encarregado de manipular e eliminar tais construções do texto-fonte, para que possa ser traduzido pelo compilador.

1.2 – ATIVIDADES ADICIONAIS DOS COMPILADORES

Adicionalmente à tradução propriamente dita, cabe aos compiladores, em geral, executar inúmeras funções auxiliares, que, do ponto de vista do usuário, comportam-se como atividades utilitárias. Entre elas, a mais importante é a detecção de erros. Um compilador poderia limitar-se a rejeitar programas que não estejam formalmente corretos. Nestes casos, o programador deveria, iterativamente, ressubmeter o programa ao compilador a cada detecção de um novo erro.

Para que isto não seja necessário, torna-se conveniente que o compilador incorpore mecanismos capazes de efetuar a detecção e a recuperação de erros no programa a ser traduzido. Através destes mecanismos, o compilador é reconduzido a uma situação em que possa prosseguir a análise do texto-fonte, a despeito da ocorrência de erros previamente detectados. Isto permite que

seja produzido para cada programa um relatório de todos os erros detectáveis pelo compilador, facilitando a sua depuração.

Outra atividade paralela executada pelos compiladores é a de permitir ao usuário a inclusão de comentários em seu texto-fonte, facilitando assim a documentação do mesmo.

Torna-se necessário, desta maneira, que o compilador filtre, do texto-fonte, todos os comentários incluídos pelo programador. Em alguns casos, os comentários fazem parte integrante da sintaxe da linguagem-fonte. Em outros, os comentários são oferecidos ao usuário pelo compilador e não pela linguagem, não constando oficialmente de sintaxe desta, sendo neste caso manipulados como recursos extra, oferecidos pelo compilador ao programador.

Uma situação semelhante é encontrada nos comandos de controle de compilação. Estes comandos não são passados aos módulos do compilador encarregados da tradução propriamente dita, mas são processados pelos módulos que implementam as funções mais básicas da análise da linguagem. É neste mesmo nível que figuram também, em geral, os pré-processadores encarregados do tratamento de macros.

1.3 – A FORMALIZAÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO – GRAMÁTICAS E RECONHECEDORES

Para que seja possível a correta compreensão das linguagens de programação, é essencial que estas sejam descritas de maneira completa e isenta de ambigüidades.

Via de regra, uma descrição desta natureza só é viável na prática através do uso de notações formais. Para a descrição formal de linguagens de programação, são utilizadas notações matemáticas formais, inspiradas em modelos gerais elaborados por estudiosos de linguagens naturais, porém de complexidade menor que as utilizadas naquele caso. Para as linguagens de programação, é possível impor restrições ao formalismo em questão, com a finalidade de limitar a generalidade das linguagens descritas, tornando mais simples a elaboração de programas analisadores baseados em tais descrições.

Estes programas seriam de custo proibitivo se toda a generalidade das linguagens naturais se mantivesse nas linguagens de programação.

Do ponto de vista de comodidade para o programador, obviamente o ideal seria se a comunicação homem-máquina pudesse ser feita diretamente em linguagem natural. No entanto, apesar dos avanços tecnológicos na área, inegavelmente encabeçados pelas pesquisas na área de inteligência artificial, esta meta ainda não pode ser considerada economicamente viável.

Desta maneira, os programadores têm trabalho até hoje com linguagens de porte substancialmente mais reduzido que o das naturais. Estas linguagens de programação, cuja forma é relativamente mais modesta que a das linguagens naturais, têm sofrido constantes aperfeiçoamentos em seus recursos de expressão, tornando-se cada vez mais elaboradas e sofisticadas, apesar de manterem as restrições acima mencionadas.

Há duas maneiras básicas utilizadas para a formalização de linguagens de programação. A primeira delas consiste em especificar leis de formação para as construções que compõem a linguagem. Ao conjunto de leis de formação que definem de maneira rigorosa o modo de formar textos corretos em uma linguagem dá-se o nome de *gramática*. Partindo-se de uma gramática, é possível efetuar a geração de qualquer texto válido da linguagem, e de nada além disto. A *linguagem* é, pois, definida como sendo o conjunto de todos os textos que podem ser gerados a partir da gramática que define tal linguagem, e nada mais. Gramáticas são, desta maneira, *dispositivos geradores*, através dos quais é feita a síntese de textos pertencentes a uma linguagem.

A segunda maneira através da qual uma linguagem pode ser formalizada consiste em especificar uma regra de teste, através da qual seja possível classificar um texto, proposto como possivelmente pertencente à linguagem, como sendo ou não um texto válido desta linguagem. A esta regra de teste dá-se o nome de *reconhecedor* da linguagem em questão. *Os dispositivos reconhecedores* definem a linguagem como sendo o conjunto de todos os textos classificados como válidos pela regra de teste que implementam, e nada mais. Os dispositivos reconhecedores permitem analisar os textos que pertencem a uma dada linguagem, sendo de grande interesse no estudo dos compiladores, uma vez que muitos compiladores são construídos com base em dispositivos desta natureza.

Gramáticas e reconhecedores são formas de representação que permitem definir formalmente as linguagens de programação. Assim sendo, formam, elas próprias, linguagens, através das quais tal formalização é efetuada. A toda linguagem utilizada como forma de representação ou de definição de outras linguagens dá-se o nome de *metalinguagem*. Pode-se exemplificar metalinguagens, na área das linguagens naturais, observando-se, por exemplo, uma gramática da língua inglesa redigida em francês, para uso de pessoas de língua francesa. Neste caso, a linguagem a ser definida é o inglês, e a metalinguagem, o francês. Na área das linguagens de programação, os exemplos mais conhecidos de metalinguagens são o *BNF* (*"Backus-Naur Form"*), através da qual são construídas gramáticas, e os *diagramas de estados*, que representam autômatos finitos através dos quais são construídos reconhecedores para uma importante classe de linguagens de programação.

1.4 – ESTRATÉGIA PARA A OBTENÇÃO DE COMPILADORES

Através do uso de metalinguagens é possível obter-se, de modo suficientemente rigoroso, uma boa definição da linguagem de programação cujo compilador se deseja construir.

Aplicando-se, sobre tais definições formais, técnicas como as que serão estudadas adiante neste texto torna-se possível construir programas capazes de implementar as funções de compilação que mapeiam o texto-fonte nos programas desejados. Seja qual for a técnica empregada para tal finalidade, o resultado é sempre um programa a ser executado no computador. A *Figura 1.6* exemplifica uma possível estratégia para a obtenção de um compilador.

Sendo um programa, o tradutor (compilador) deverá ser expresso em alguma linguagem de programação disponível, para que seja possível implantá-lo em alguma máquina existente. No pior dos casos, não havendo disponibilidade de qualquer linguagem a não ser a de máquina, esta deveria ser utilizada para a primeira implementação do compilador. Felizmente, proliferam linguagens e compiladores em todas as máquinas comercialmente disponíveis, e portanto algum compilador pode ser suposto existente para permitir a primeira tradução de um compilador para a linguagem que está sendo desenvolvida. Este deverá, portanto, ser escrito na linguagem de programação que o compilador disponível seja capaz de traduzir.

Tem-se duas questões a solucionar: para qual máquina o compilador final deverá gerar código, e em que linguagem tal compilador deverá ser redigido.

Seja L a linguagem a ser compilada para a máquina M, ou seja, deseja-se construir um compilador da linguagem L que gere um código-objeto que seja executável na máquina M.

Pode-se optar por construir este compilador de tal modo que possa ser executado na própria máquina M, ou então em alguma outra máquina M', dita *máquina hospedeira*. No primeiro caso, o compilador é dito *auto-residente*. No segundo, o compilador é chamado *compilador cruzado* (*"cross-compiler"*).

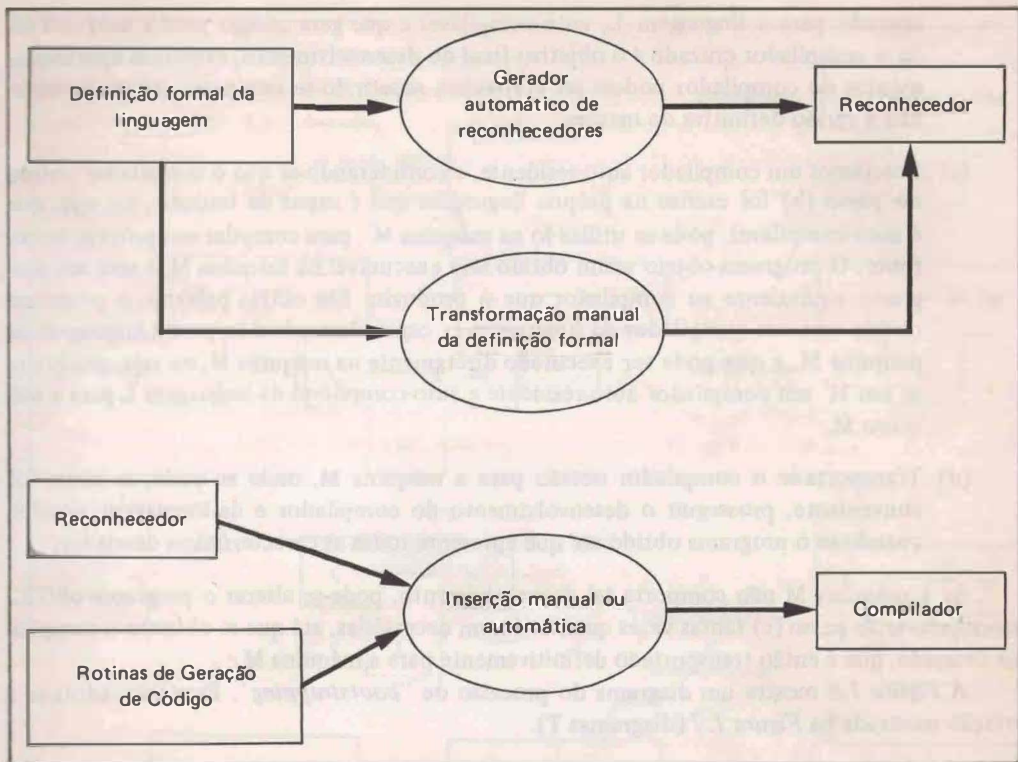


Figura 1.6 – Uma estratégia para a obtenção de compiladores.

Seja L′ uma linguagem disponível a ser utilizada na implementação do compilador, qualquer que seja o esquema adotado. Com estas hipóteses, descreve-se a seguir a técnica de obtenção de compiladores ditos *auto-compiláveis* através do método de "*bootstrapping*". Um compilador é chamado auto-compilável quando a linguagem em que for desenvolvido (L′) é a própria linguagem L que o compilador deve ser capaz de traduzir.

O método de "*bootstrapping*" permite obter, a partir de praticamente nenhuma infraestrutura, uma primeira versão de um compilador auto-compilável, a ser utilizado posteriormente para melhorar e aperfeiçoar, em versão subseqüente, o compilador original. A técnica mencionada pode ser detalhada como segue:

(a) Escreve-se, na linguagem L′, um compilador capaz de traduzir a linguagem L para linguagem de máquina M′ (nesta primeira versão, pode-se implementar apenas os recursos de L que sejam indispensáveis à construção deste primeiro compilador). Obtém-se, assim, na máquina M′, um compilador auto-resistente para a linguagem L, que gera código para a própria máquina M′.

Caso as linguagens L e M sejam a mesma, então o compilador obtido no passo (a) já estaria disponível a priori, sendo assim desnecessário refazê-lo.

(b) De posse do compilador da linguagem L para a máquina M′, reescreve-se o mesmo, na própria linguagem L, eliminando-se deste modo linguagens L′ diferentes, e de modo que o código-objeto que o novo compilador deve produzir se apresente na linguagem da máquina M. Utilizando-se o compilador L disponível, compila-se o novo programa obtido, obtendo-se desta maneira, para ser executado na máquina M′, um compilador

cruzado para a linguagem L, auto-compilável e que gera código para a máquina M. Se o compilador cruzado é o objetivo final do desenvolvimento, eventuais aperfeiçoamentos do compilador podem ser efetuados, repetindo-se este passo até que se obtenha a versão definitiva do mesmo.

(c) Desejamos um compilador auto-residente, e considerando-se que o compilador obtido no passo (b) foi escrito na própria linguagem que é capaz de traduzir, ou seja, que é auto-compilável, pode-se utilizá-lo na máquina M', para compilar seu próprio texto-fonte. O programa-objeto assim obtido será executável na máquina M, e será um programa equivalente ao compilador que o produziu. Em outras palavras, o programa obtido será um compilador da linguagem L, capaz de traduzi-lo para a linguagem da máquina M, e que pode ser executado diretamente na máquina M, ou seja, produziu-se em M' um compilador auto-residente e auto-compilável da linguagem L para a máquina M.

(d) Transporta-se o compilador obtido para a máquina M, onde se pode, se assim for conveniente, prosseguir o desenvolvimento do compilador e da linguagem, aperfeiçoando-se o programa obtido até que apresente todas as características desejadas.

Se a máquina M não comporta tal desenvolvimento, pode-se alterar o programa obtido, retornando-se ao passo (c) tantas vezes quantas forem necessárias, até que se obtenha o compilador desejado, que é então transportado definitivamente para a máquina M.

A *Figura 1.8* mostra um diagrama do processo de *"bootstrapping"*. Para isso, adota-se a notação mostrada na *Figura 1.7* (diagramas T).

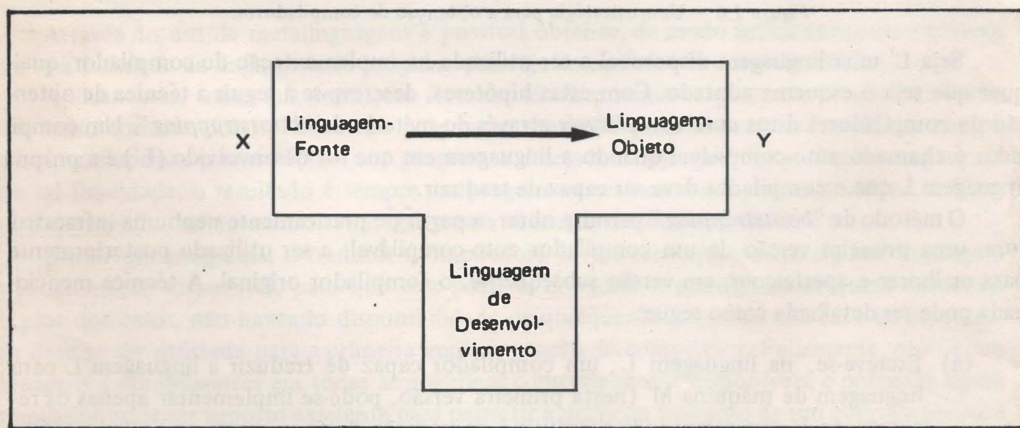


Figura 1.7 – Módulo básico de um diagrama T utilizado na representação do processo de *"bootstrapping"*.

Um texto X, denominado texto-fonte, é submetido ao programa representado no polígono, pela extremidade esquerda da figura.

No braço esquerdo do polígono está indicada a linguagem em que X está escrito (linguagem-fonte). O programa compilador, escrito na linguagem de desenvolvimento indicada na extremidade inferior do polígono, traduz o texto-fonte para a linguagem-objeto, indicada na extremidade direta do polígono, gerando um texto Y equivalente, denotado na linguagem-objeto.

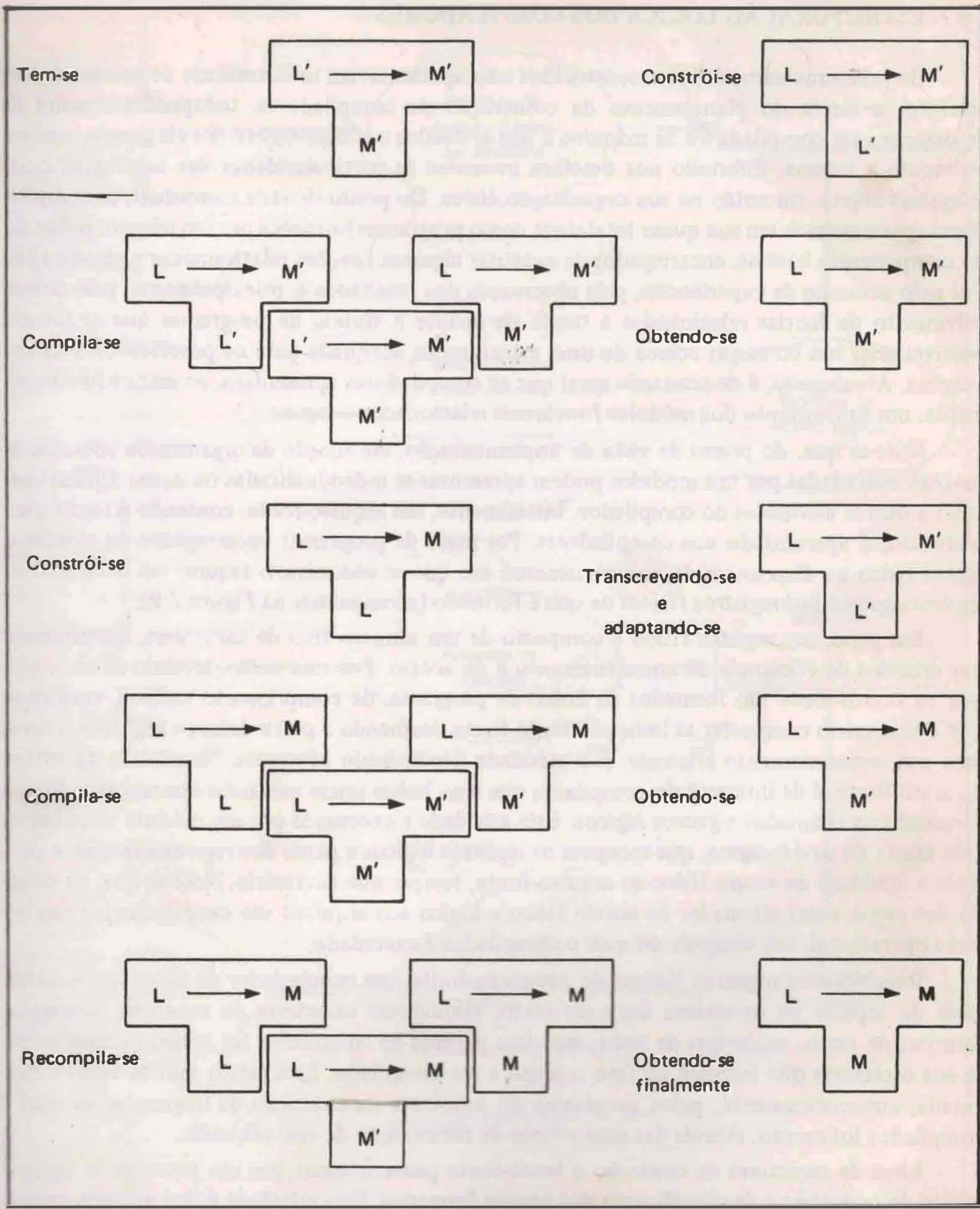


Figura 1.8 – O processo de *"bootstrapping"* esquematizado em diagramas T.

1.5 – ESTRUTURAÇÃO LÓGICA DOS COMPILADORES

Os primeiros compiladores construídos não apresentavam uniformidade de estrutura. Entretanto, a tarefa do planejamento da construção de compiladores, independentemente da linguagem a ser compilada ou da máquina a que se destina o código-objeto por ele gerado, é essencialmente a mesma, diferindo nos detalhes inerentes às particularidades das linguagens e das máquinas-objeto, ou então na sua organização física. Do ponto de vista conceitual, os compiladores apresentam-se em sua quase totalidade como programas formados por um número reduzido de componentes básicos, encarregados de executar algumas funções relativamente padronizadas. Foi pelo acúmulo de experiências, pela observação dos resultados e, principalmente, pelo desenvolvimento de teorias relacionadas à tarefa de análise e síntese de programas que se tornou possível criar um consenso acerca de uma estruturação adequada para os processadores de linguagens. Atualmente, é de aceitação geral que os compiladores apresentem, ao menos funcionalmente, um subconjunto dos módulos funcionais relacionados a seguir.

Note-se que, do ponto de vista de implementação, em função da organização adotada, as funções executadas por tais módulos podem apresentar-se individualizadas ou então diluídas em meio a outras atividades do compilador. Inicialmente, um arquivo-fonte, contendo o texto a ser traduzido, é apresentado aos compiladores. Por meio de programas encarregados de efetuar o acesso físico ao dispositivo de armazenamento em que se encontra, o arquivo em questão deve ser decomposto nos *registros físicos* de que é formado (acompanhar na *Figura 1.9*).

Em geral, um registro físico é composto de um número fixo de caracteres, determinado por critérios de eficiência de armazenamento e de acesso. Por esta razão, levando-se em conta que os textos-fonte são formados de linhas de programa, de comprimento variável, verifica-se que é necessário compactar as linhas do texto-fonte, formando a partir delas os registros físicos, para um armazenamento eficiente. É a atividade denominada *blocagem.* Na ocasião da leitura do texto fonte, é de interesse do compilador que suas linhas sejam extraídas dos registros físicos, formando os chamados *registros lógicos.* Esta atividade é executada por um módulo responsável pela tarefa de *de-blocagem,* que recupera os registros lógicos a partir dos registros físicos, e controla a atividade de acesso físico ao arquivo-fonte, sempre que necessário. Note-se que, na maioria dos casos, estas atividades de acesso físico e lógico aos arquivos são executadas por um sistema operacional, sob controle do qual o compilador é executado.

Recebidos os registros lógicos do programa-fonte, um *manipulador de caracteres* encarrega-se de separar os caracteres úteis do texto, eliminando caracteres de controle, tabulação, retornos de carro, mudanças de linha, etc. Isto permite ao compilador ter acesso exclusivamente aos caracteres que formam de fato o texto a ser compilado. Esta tarefa muitas vezes é executada, automaticamente, pelos programas do ambiente de execução da linguagem na qual o compilador foi escrito, através das suas rotinas de formatação de entrada/saída.

Livre de caracteres de controle, o texto-fonte passa, a seguir, por um processo de agrupamento de caracteres e de classificação dos grupos formados. Esta atividade é denominada *análise léxica,* e produz como saída um texto cujos componentes não são caracteres individuais, mas sim seqüências classificadas. A cada uma destas seqüências, acompanhada de sua classificação, dá-se o nome de *átomo* da linguagem. A atividade de análise léxica muitas vezes incorpora a função de descartar seqüências de caracteres que não contribuam para a análise posterior do programa-fonte (por exemplo, brancos, separadores e comentários). Neste caso, a atividade é executada por módulos denominados *filtros léxicos.* Outras atividades auxiliares são efetuadas nesta fase, sendo que as mais importantes referem-se à conversão numérica, à identificação de palavras reservadas e à criação e manutenção de tabelas de símbolos da compilação.

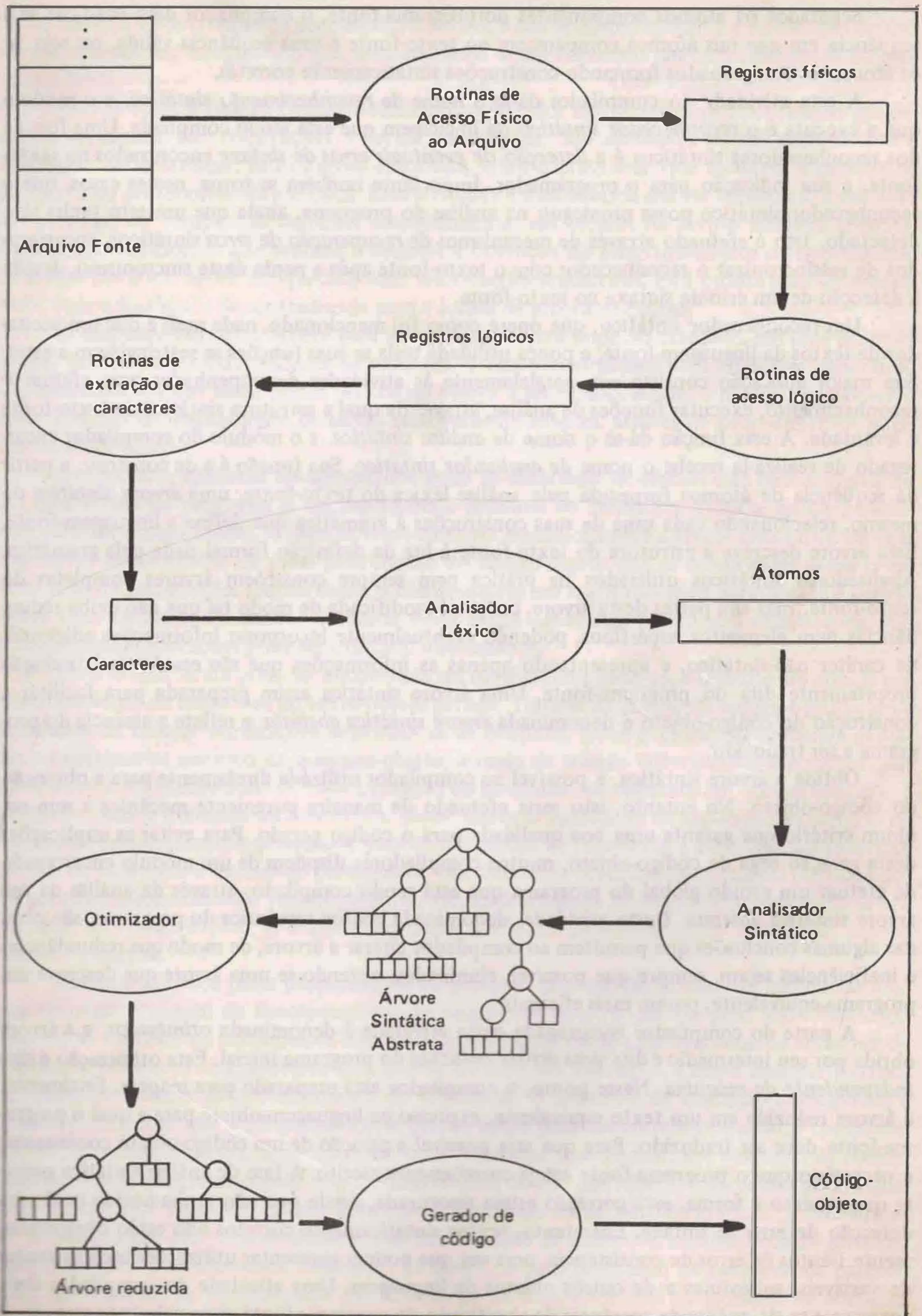


Figura 1.9 – Estruturação Lógica dos Compiladores.

Separados os átomos componentes do programa-fonte, o compilador deve verificar se a seqüência em que tais átomos comparecem no texto-fonte é uma seqüência válida, ou seja, se os átomos estão agrupados formando construções sintaticamente corretas.

A esta atividade do compilador dá-se o nome de *reconhecimento sintático,* e o módulo que a executa é o *reconhecedor sintático* da linguagem que está sendo compilada. Uma função dos reconhecedores sintáticos é a *detecção de eventuais erros de sintaxe* encontrados no texto-fonte, e sua indicação para o programador. Importante também se torna, nestes casos, que o reconhecedor sintático possa prosseguir na análise do programa, ainda que um erro tenha sido detectado. Isto é efetuado através de mecanismos de *recuperação de erros* sintáticos, encarregados de ressincronizar o reconhecedor com o texto-fonte após a perda deste sincronismo, devida à detecção de um erro de sintaxe no texto-fonte.

Um reconhecedor sintático, que opere como foi mencionado, nada mais é que um aceitador de textos da linguagem-fonte, e pouca utilidade teria se suas funções se restringissem a estas. Sua maior aplicação consiste em, paralelamente às atividades desempenhadas para efetuar o reconhecimento, executar funções de análise, através da qual a estrutura sintática do texto-fonte é levantada. A esta função dá-se o nome de *análise sintática,* e o módulo do compilador encarregado de realizá-la recebe o nome de *analisador sintático.* Sua função é a de construir, a partir da seqüência de átomos fornecida pela análise léxica do texto-fonte, uma *árvore sintática* do mesmo, relacionando cada uma de suas construções à gramática que define a linguagem-fonte. Esta árvore descreve a estrutura do texto-fonte à luz da definição formal dada pela gramática. Analisadores sintáticos utilizados na prática nem sempre constróem árvores completas do texto-fonte, mas sim partes desta árvore, em geral modificada de modo tal que não exiba redundâncias nem elementos supérfluos, podendo eventualmente incorporar informações adicionais de caráter não-sintático, e apresentando apenas as informações que são essenciais à tradução propriamente dita do programa-fonte. Uma árvore sintática assim preparada para facilitar a construção do código-objeto é denominada *árvore sintática abstrata,* e reflete a essência do programa a ser traduzido.

Obtida a árvore sintática, é possível ao compilador utilizá-la diretamente para a obtenção do código-objeto. No entanto, isto seria efetuado de maneira puramente mecânica e sem nenhum critério que garanta uma boa qualidade para o código gerado. Para evitar as implicações desta geração cega de código-objeto, muitos compiladores dispõem de um módulo encarregado de efetuar um estudo global do programa que está sendo compilado, através da análise da sua árvore sintática abstrata. Desta atividade, denominada *análise semântica* do programa, são obtidas algumas conclusões que permitem ao compilador alterar a árvore, de modo que redundâncias e ineficiências sejam, sempre que possível, eliminadas, obtendo-se uma árvore que descreve um programa equivalente, porém mais eficiente.

A parte do compilador encarregada desta atividade é denominada *otimizador,* e a árvore obtida por seu intermédio é dita uma *árvore reduzida* do programa inicial. Esta otimização é dita *independente da máquina.* Neste ponto, o compilador está preparado para mapear, finalmente, a árvore reduzida em um texto equivalente, expresso na linguagem-objeto para a qual o programa-fonte deve ser traduzido. Para que seja possível a geração de um código-objeto consistente, é necessário que o programa-fonte esteja corretamente escrito. A fase de análise sintática garante que, quanto à forma, esta correção esteja assegurada, desde que não tenha havido nenhuma detecção de erro de sintaxe. Entretanto, textos sintaticamente corretos não estão obrigatoriamente isentos de erros de consistência, uma vez que podem apresentar utilizações inconsistentes de variáveis, subrotinas e de outros objetos da linguagem. Uma atividade do compilador deve encarregar-se da análise da coerência de significado do programa-fonte, especialmente através da verificação da coerência de utilização dos identificadores dos objetos declarados no programa.

A tal módulo dá-se o nome de *aceitador semântico,* e sua principal função é a verificação do uso de identificadores, assegurando que seus atributos sejam coerentes em cada construção do programa fonte.

Muitas vezes a função desta atividade do compilador não se restringe à verificação da coerência semântica das construções a serem compiladas. São efetuadas outras análises com a finalidade de determinar informações adicionais sobre o programa, que facilitem a geração de código, a ser posteriormente efetuada. Esta atividade é denominada *análise semântica,* e sua ação no processo de compilação consiste, essencialmente, em incluir, na árvore sintática abstrata, informações adicionais que venham a facilitar a obtenção do programa-objeto correspondente. A árvore sintática, agora enriquecida com informações semânticas, está pronta para ser processada, com a finalidade de ser traduzida para a forma de programa-objeto.

O mapeamento desta árvore para a linguagem-objeto pode ser efetuado, em uma primeira etapa, de um modo independente de máquina, e sem preocupações com eficiência, ignorando dependências mútuas entre partes do código-objeto a ser gerado. Assim, é possível inicialmente levar a efeito tal mapeamento de modo relativamente simples, através de procedimentos puramente mecânicos.

Um módulo adicional do compilador pode se encarregar de efetuar esta tarefa, produzindo uma versão do programa já em um formato próximo ao definitivo. Trata-se de um *gerador canônico de código-objeto,* o qual às vezes converte mecanicamente a árvore reduzida para uma *linguagem intermediária* de nível próximo ao do código-objeto que se deseja obter como resultado final da compilação. Outras vezes, o código-objeto em sua forma mais definitiva já é gerado nesta fase. De qualquer modo, obtém-se nesta etapa um código-objeto sem otimizações, mas que em muitas aplicações pode ser utilizado diretamente.

Uma última tarefa pode ser encontrada em muitos compiladores, e consiste em aplicar um nível adicional de melhorias no texto-objeto gerado. Trata-se da análise do código-objeto com a finalidade de efetuar otimizações dependentes de máquina. Estas otimizações procuram explorar ao máximo os recursos da máquina-objeto, através da seleção criteriosa das instruções mais adequadas à implementação de cada operação do programa, no contexto em que tal operação é realizada. Isto é efetuado através de um gerenciamento cuidadoso de registradores, do uso de modos de endereçamento adequados para as instruções de referência à memória, do aproveitamento de endereços e de índices previamente calculados, da reutilização de valores já calculados de subexpressões utilizadas repetitivamente, etc. Esta atividade do compilador denomina-se *otimização do código-objeto,* e o resultado desta fase é um *programa-objeto otimizado,* pronto para ser utilizado.

Embora não faça parte propriamente do processo de compilação, convém mencionar aqui um elemento essencial do funcionamento do programa-objeto, e que em geral não é destacado quando se estudam os mecanismos básicos de compilação: trata-se do denominado *ambiente de execução,* em que o código-objeto fica imerso durante sua execução, e que é uma parte crítica de qualquer sistema de processamento de linguagens. O ambiente de execução corresponde à infraestrutura com que o programa-objeto gerado conta para sua correta execução, e compreende, entre outros recursos, uma interface com o sistema operacional, um conjunto de rotinas chamadas pelo programa-objeto como conseqüência do uso dos recursos oferecidos pela linguagem (como, por exemplo, rotinas aritméticas, trigonométricas, pacotes gráficos, formatadores de entrada/saída, etc.) e procedimentos de manutenção de ambiente, encarregados do gerenciamento do uso (eventualmente dinâmico) da memória pelo programa, da passagem de parâmetros, do controle da recursividade das subrotinas da linguagem, dos mecanismos de acesso às variáveis estruturadas do programa, da comunicação entre módulos paralelos e concorrentes, etc. Como se pode notar facilmente, o ambiente de execução é a peça mais importante do pro-

grama compilado, e que determina os limites até onde podem chegar os recursos das linguagens que são executadas sob sua influência. A *Figura 1.10* esquematiza o processo de execução de uma linguagem de alto nível.

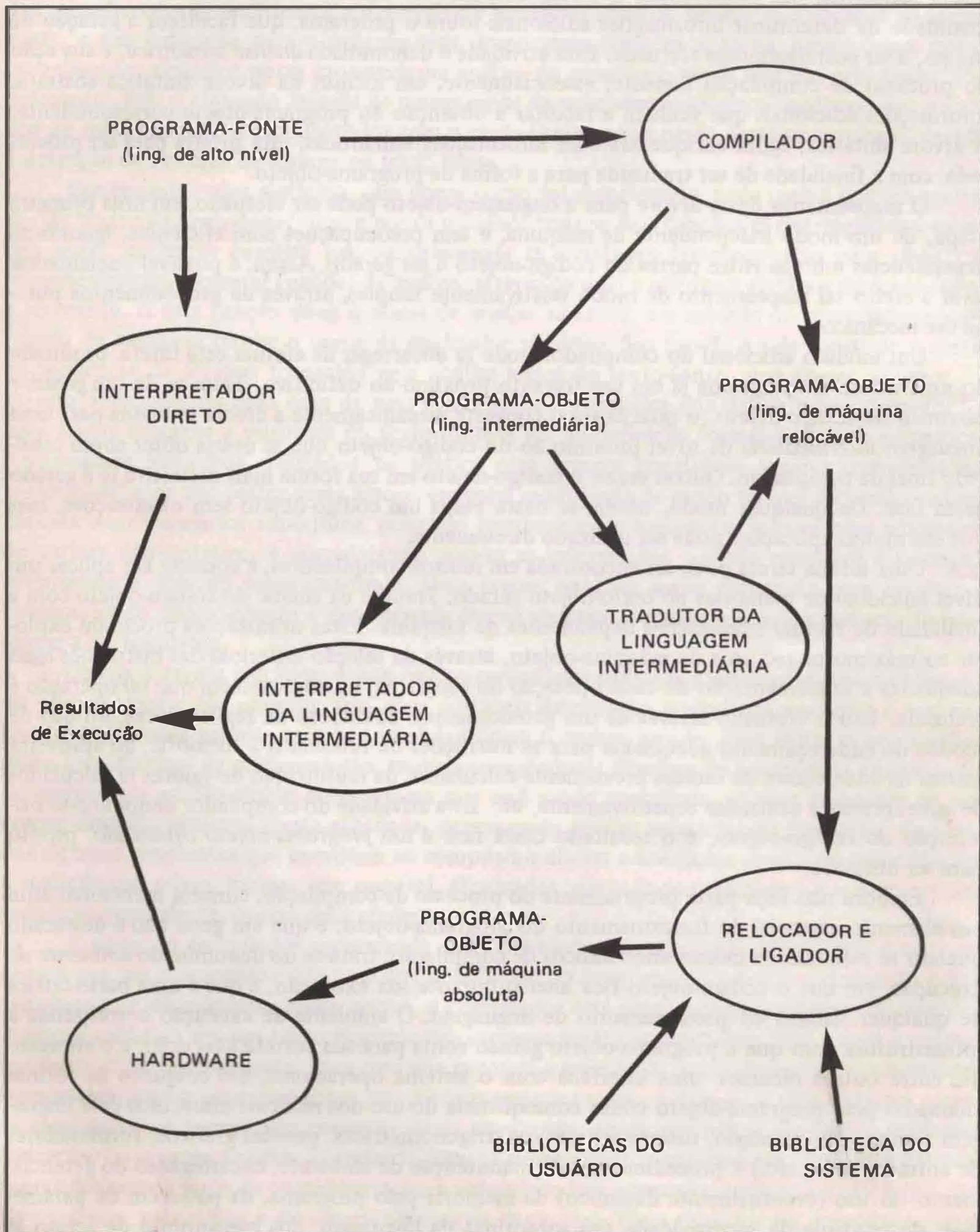


Figura 1.10 – O processo de execução de linguagens de alto nível.

Finalmente, convém lembrar que a compilação não é o único recurso de processamento para linguagens de alto nível. Pelo contrário, os *interpretadores* mostram-se cada vez mais atraentes, especialmente para o desenvolvimento interativo de programas, durante a fase de depuração. Quando o tempo de retorno de resultados é importante, como ocorre na fase de depuração de um programa, o interpretador é, sem dúvida alguma, uma boa opção para o usuário. Note-se que, na classe dos interpretadores, comparece uma importante família de processadores de linguagem. Trata-se daqueles que traduzem a linguagem de alto nível para um código intermediário, o qual, ao invés de ser convertido para linguagem de máquina, é utilizado diretamente para ser executado. Como a máquina hospedeira não reconhece tal linguagem, torna-se necessário criar um simulador que seja capaz de comportar-se como se fosse um hardware, para o qual tal linguagem seria a linguagem de máquina. Em geral, processadores de linguagens que operam desta maneira apresentam-se também em versões capazes de compilar para linguagem de máquina o código intermediário mencionado, aumentando assim a gama de opções do uso da linguagem de alto nível para o usuário.

Antes de prosseguir, convém mencionar que na maioria dos textos sobre compiladores são caracterizadas três fases macroscópicas, na compilação: a *análise léxica*, correspondendo ao conjunto de todas as atividades que culminam com a identificação dos átomos, a *análise sintática*, que inclui todas as tarefas do compilador relativas à forma do texto-fonte, identificação das construções gramaticais da linguagem e obtenção das árvores de sintaxe, e as *atividades semânticas* e de *geração de código*, que englobam todo o restante do compilador, incluindo tarefas bastante heterogêneas, desde as conversões numéricas e a montagem e manutenção de tabelas de símbolos e de atributos até a geração propriamente dita do código-objeto e sua otimização.

1.6 – ORGANIZAÇÃO FÍSICA DOS COMPILADORES

Do ponto de vista lógico, foi visto no parágrafo anterior um conjunto de atividades que compõem tipicamente um programa compilador. A implementação propriamente dita do compilador, no entanto, apresenta-se com um vasto leque de opções para o projetista, configurando escolhas que devem ser feitas com base especialmente nas finalidades a que se destina o compilador e nos recursos disponíveis para a sua implementação. Por questões de simplicidade, as atividades do compilador podem ser consideradas como agrupadas nas três fases macroscópicas citadas anteriormente: análise léxica, análise sintática e geração de código.

Antes de sugerir esquemas de organização física para a implementação dos compiladores, é importante tecer algumas considerações sobre metas a serem atingidas pelo projeto.

Um ponto de grande importância é a eficiência a ser alcançada. Note-se que, sendo o compilador um gerador de programas, é preciso para isto considerar tanto a eficiência do compilador como a do programa-objeto. Conforme a aplicação a que se destina, é mais importante que o compilador seja rápido, do que ser eficiente o código-objeto gerado, ou vice-versa.

Em compiladores para uso em escolas, com finalidades didáticas, por exemplo, é essencial que a compilação seja extremamente rápida, não importando que o código-objeto seja ineficiente. Isto porque, neste tipo de aplicação, os programas a serem compilados serão em geral executados poucas vezes, e compilados inúmeras vezes, até que a versão definitiva seja obtida. Nesta aplicação, as mensagens de erro oferecidas pelo compilador devem ser claras e precisas, escritas por extenso, para facilitar o aprendizado. Portanto, os objetivos a serem alcançados neste tipo de aplicação correspondem basicamente à obtenção rápida de um código-objeto que dispense otimizações. Seria conveniente que o compilador pudesse residir completamente na memória

do computador, para evitar gastos de tempo devido ao gerenciamento da execução de suas partes.

Para compiladores destinados a auxiliar no desenvolvimento e na depuração de programas novos valem, na prática, os mesmos argumentos e considerações efetuadas em relação aos compiladores para uso didático, dada sua freqüência de utilização para programas em fase de criação.

Os fatos são totalmente diferentes, entretanto, no caso de compildores destinados a criar versões definitivas de programas já depurados, e prontos para serem implantados com a finalidade de fornecer serviços com grande freqüência e durante muito tempo. Para tais casos, o compilador pode ser lento, desde que o programa-objeto obtido por seu intermédio seja o mais eficiente possível. Desta maneira, exige-se que o compilador apresente o maior número possível de recursos de otimização para o programa compilado, podendo ser composto de várias fases a serem executadas em seqüência, ocupando vastas áreas de memória e executando algoritmos eventualmente demorados. Todas estas desvantagens são toleráveis, visto que tais compiladores são executados raramente, apenas na ocasião em que se tem a garantia de que o programa a ser compilado já está correto.

Assim sendo, seus recursos de depuração e mensagens de erro podem ser pobres e alguns até suprimidos, sem prejuízo substancial para o programador.

Para aplicações em tempo real, os compiladores devem produzir um código eficiente, otimizado em relação ao tempo de execução. Para aplicações em que o programa-objeto deve ser executado em máquinas de pequena capacidade de memória, exigem-se recursos de otimização da área de código ocupada pelo programa-objeto, ao invés de recursos de aumento de sua velocidade.

Como se pode notar, cada aplicação tem suas exigências peculiares, e o compilador deve ser projetado com tais exigências em mente.

Em relação aos recursos disponíveis na máquina-objeto, deve-se considerar que máquinas com pequena capacidade de memória e sem recursos de armazenamento secundário são pouco adequadas como máquinas hospedeiras de um compilador. Assim, se é para uma máquina desta natureza que se deseja um compilador, convém que tal compilador seja implementado como compilador cruzado, que resida em alguma máquina hospedeira que apresente recursos mais adequados para o desenvolvimento de programas.

Se, no entanto, houver, por qualquer razão, uma forte justificativa para que o compilador seja executado em uma máquina com poucos recursos, torna-se necessário organizá-lo de tal modo que, por exemplo, seja executado por partes que se comuniquem convenientemente através de dados armazenados na memória principal, em arquivos ou, então, em último caso, em algum meio externo de armazenamento, como fitas de papel, cartões perfurados ou fitas cassete. Neste caso o desempenho do processo fica seriamente comprometido, podendo inclusive inviabilizar o uso do compilador em situações práticas, devendo ser encarada tal solução como uma forma de solução de emergência para ser utilizada quando todas as demais forem inviáveis.

Estudados os condicionantes do projeto, pode-se passar à análise das possíveis organizações do compilador que possam satisfazer aos requisitos por eles impostos.

Um ponto a ser considerado é o número de passos com que o compilador deverá ser construído. Denomina-se *passo de compilação* a atividade do compilador que exija, para seu processamento, uma leitura completa do texto-fonte ou de alguma de suas formas intermediárias equivalentes. Um *compilador de um único passo* é aquele que executa todas as suas atividades com uma única leitura do texto-fonte, gerando como saída o código-objeto final a ser executado, por meio de um único programa (*Figura 1.11*).

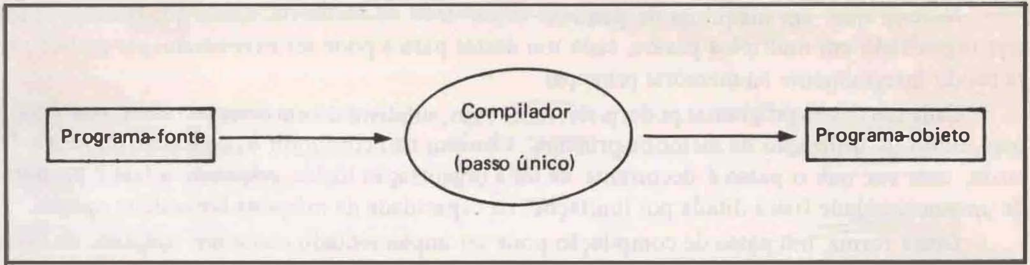


Figura 1.11 – Compilador de um único passo.

Um *compilador de vários passos* efetua a leitura do texto-fonte e sua conversão para uma forma intermediária equivalente, através de um programa que corresponde ao primeiro passo do compilador. A saída deste programa alimenta um outro programa, que é o segundo passo do compilador, que por sua vez gera outro código intermediário que irá abastecer o próximo passo de compilação e assim por diante, até que o código-objeto final seja finalmente produzido pelo último passo do compilador (*Figura 1.12*).

Programa-fonte

Compilador
(1º passo)

1ª forma intermediária
do programa

Compilador
(2º passo)

2ª forma intermediária
do programa

. . .

n-1 ésima forma inter-
mediária do programa

Compilador
(nº passo)

Programa-objeto
final

Figura 1.12 – Compilador de múltiplos passos.

Note-se que, em máquinas de pequena capacidade de memória, ainda que o compilador seja organizado em múltiplos passos, cada um destes passos pode ser excessivamente grande para residir integralmente na memória principal.

Cada um destes programas poderia ser, neste caso, subdividido em diversas fases (*"overlays"*) para efeito de utilização da memória principal. Convém não confundir os conceitos de *fase* e de *passo*, uma vez que o passo é decorrente de uma organização lógica, enquanto a fase é produto de uma necessidade física ditada por limitações na capacidade da máquina hospedeira apenas.

Desta forma, um passo de compilação pode ser implementado como um conjunto de fases executadas na seqüência conveniente, conforme as exigências do programa a ser compilado e as limitações da capacidade da máquina hospedeira.

Escolhido o número de passos a ser utilizado na compilação, cada um deles pode ser caracterizado como sendo ele próprio um compilador de um único passo, encarregado de traduzir sua própria linguagem de entrada para a linguagem de saída correspondente. Para esquemas de um só passo, a linguagem de entrada é a própria linguagem-fonte, e a linguagem de saída, a linguagem do programa-objeto a ser construído. Para esquemas de passos múltiplos, a linguagem-fonte é a linguagem de entrada do primeiro passo do compilador, e a linguagem-objeto corresponde à linguagem de saída de seu último passo. Para cada passo intermediário, a sua linguagem de entrada corresponde à linguagem de saída do passo anterior, e sua linguagem de saída, à linguagem de entrada do passo seguinte.

Assim sendo, são indicados a seguir alguns dos inúmeros possíveis esquemas de organização para um passo de compilação. Deve-se interpretar os termos "análise léxica", "átomo", "análise sintática", "árvore abstrata", "gerador de código" e "código-objeto" no contexto do passo a que se referem. Assim, em um passo genérico, os átomos são obtidos fracionando-se adequadamente o texto de entrada correspondente e não no sentido de agrupar caracteres do texto-fonte, como havia sido definido para a linguagem-fonte inicial. De maneira análoga, cada uma das linguagens intermediárias tem sua própria estrutura e sintaxe, de modo que a análise sintática a ser aplicada a tais linguagens deve referir-se à sintaxe a elas inerentes, e não obrigatoriamente à da linguagem-fonte.

O mesmo ocorre em relação à geração de código, a qual deve, em cada passo, produzir textos equivalentes ao texto de entrada correspondente, porém expressos na linguagem de entrada do passo seguinte.

Os esquemas descritos em seguida refletem algumas das opções mais encontradas para a organização de um passo de compilação. Apesar de diferirem no seu modo de funcionamento e operação, comportam-se de maneira idêntica quando analisados como sistema completo. Além disso, os módulos básicos que os compõem podem ser identificados e reconhecidos em cada um dos esquemas, assim como os elementos de comunicação entre os módulos.

O primeiro e mais usual dos esquemas apresenta o passo de compilação como um programa seqüencial, em que o analisador sintático é o programa principal, e o analisador léxico, bem como o gerador de código, são compostos de subrotinas ativadas pelo analisador sintático conforme as necessidades deste (*Figura 1.13*).

Neste esquema, o analisador sintático implementa uma realização de um autômato, cujas transições ativam, sempre que necessário, as rotinas de análise léxica, para a obtenção de novos átomos, que irão promover novas transições no reconhecedor. Como resultado da análise efetuada, o analisador sintático produz uma árvore abstrata do programa ou de parte do mesmo, a qual, sempre que julgado oportuno, é fornecida às rotinas do gerador de código, que produzem, a partir da mesma, o código-objeto correspondente. A escolha da ocasião da chamada das rotinas do gerador de código, bem como a seleção da rotina adequada a cada caso, são também de-

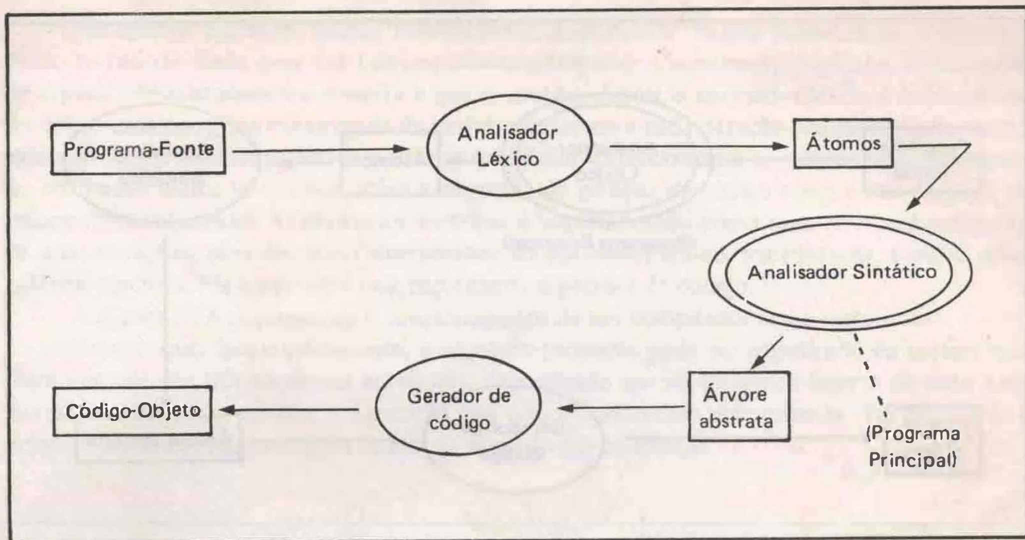


Figura 1.13 – Esquema da organização de um passo do compilador, com o analisador sintático como programa principal.

terminadas pelas transições efetuadas pelo analisador sintático. A este esquema, devido ao fato de que a estrutura sintática do programa-fonte determina a seqüência das atividades de compilação, dá-se o nome de *compilação dirigida pela sintaxe* (*"syntax-driven"*), e corresponde, como foi mencionado, ao esquema mais popular de organização de compiladores.

Um segundo esquema (*Figura 1.14*) promove o analisador léxico à categoria de programa principal. Desta maneira, é ele quem irá comandar as atividades do compilador. Para tanto, recebe como entrada o arquivo que contém o programa-fonte a ser compilado, e o decompõe em seus átomos. Cada átomo extraído é passado ao analisador sintático, como parâmetro da subrotina que, neste esquema, este analisador é. Sendo possível ao analisador sintático consumir o átomo em questão e promover transições em seu estado interno, isto será feito, caso contrário o átomo recebido é armazenado pelo analisador sintático para uso futuro. Em qualquer caso, todas as transições que possam eventualmente ser efetuadas com base nos átomos já recebidos serão de fato executadas. O retorno do comando ao programa principal ocorre sempre que não for possível ao analisador sintático prosseguir em suas transições com base nas informações disponíveis.

O retorno corresponde a uma indicação de que novo átomo deve ser extraído, para que a análise possa prosseguir. Como no esquema anterior, também aqui o analisador sintático decide, em função de transição efetuada, qual das rotinas do gerador de código deve ser acionada, e passa para tal rotina a árvore parcialmente construída até o momento, como informação para promover a geração do código que for possível.

Um terceiro esquema visualiza o gerador de código como sendo o programa principal do compilador. É este módulo, portanto, que toma a iniciativa de controlar e seqüencializar as atividades de compilação. Sempre que não lhe for possível gerar, a partir das informações disponíveis até o momento, mais uma parcela de código-objeto, o programa principal chama, como subrotina, o analisador sintático, solicitando-lhe novo segmento da árvore abstrata do programa.

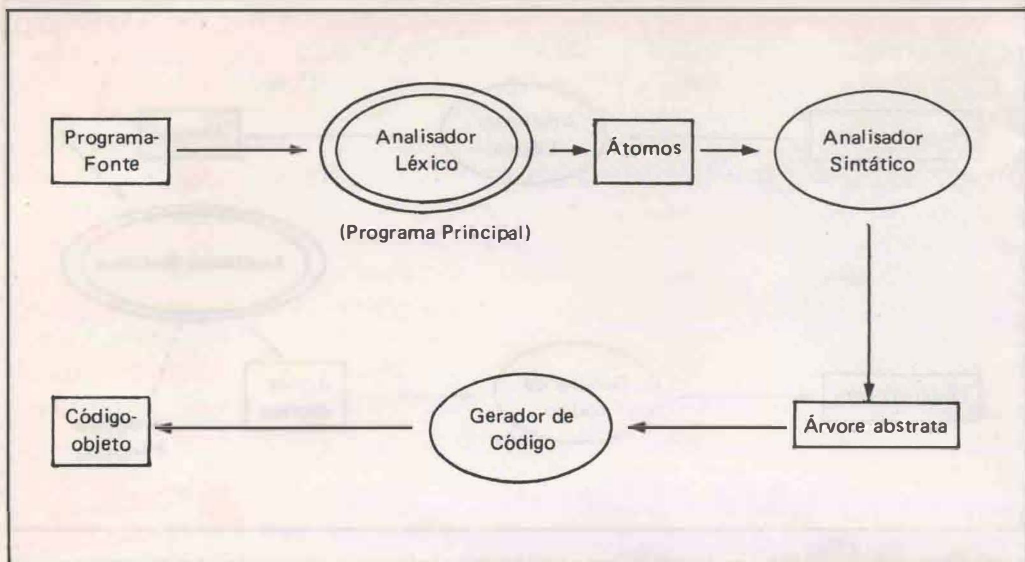


Figura 1.14 – Esquema de organização de um passo de compilação em que o Analisador Léxico é o programa principal.

Se o analisador sintático dispuser de informações suficientes, a geração é efetuada, e o comando retorna ao gerador de código. Caso contrário, o analisador sintático chama como subrotina o analisador léxico, solicitando-lhe mais átomos. De posse de tais átomos, o analisador sintático tenta novamente produzir mais uma parte da árvore abstrata, solicitando ao analisador léxico novos átomos até que isto seja possível, ocasião em que tal árvore é gerada e fornecida ao programa principal. O procedimento se repete até que o texto-fonte se esgote. Ver *Figura 1.15*.

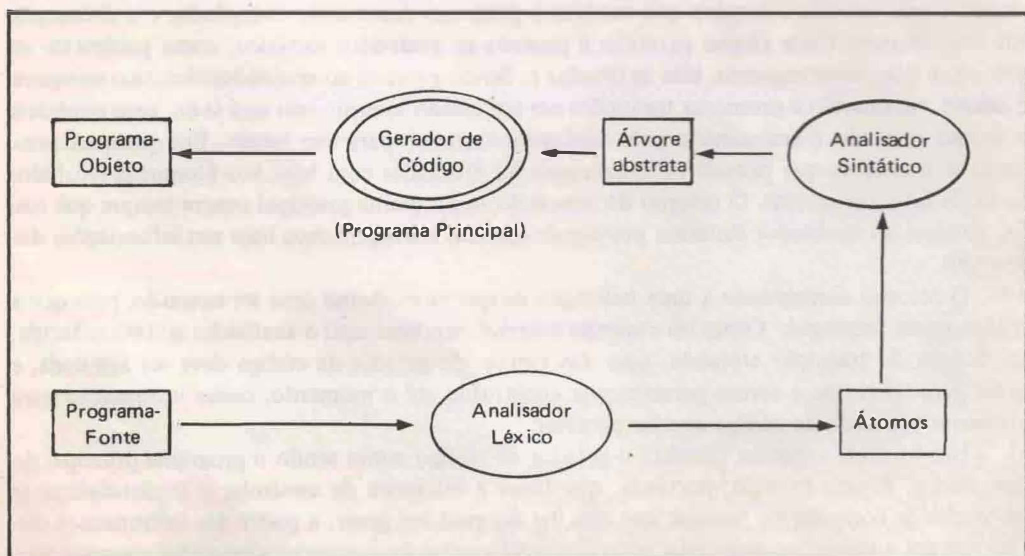


Figura 1.15 – Esquema de organização de um compilador onde o Gerador de Código é o programa principal.

Um quarto esquema, menos convencional, implementa os três módulos de compilador como co-rotinas. Cada uma das três co-rotinas corresponde a uma iteração infinita, encarregada de repetir permanentemente a tarefa a que se destina. Assim, o analisador léxico é implementado como uma co-rotina encarregada de extrair um átomo a cada iteração completa. Cada átomo extraído leva o analisador léxico a ativar o analisador sintático como co-rotina. Este, recebendo do analisador léxico os átomos, ativa a co-rotina de geração de código sempre que dispuser de informações suficientes. O analisador sintático é implementado como uma co-rotina composta de duas iterações, uma das quais sincronizada ao analisador léxico, mais interna, e outra, mais externa, sincronizada à co-rotina que implementa o gerador de código.

A *Figura 1.16* esquematiza o funcionamento de um compilador assim organizado.

Note-se que, hierarquicamente, o esquema proposto pode ser organizado da mesma maneira que um dos três esquemas anteriores, dependendo apenas da lógica interna de cada uma das co-rotinas, uma vez que a execução das três é fortemente sincronizada. Tal hierarquia é definida fundamentalmente pela ordem de ativação das co-rotinas.

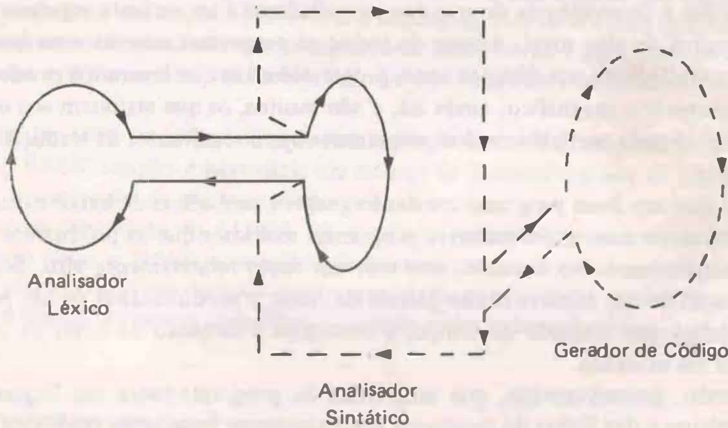


Figura 1.16 – O funcionamento de um compilador em que os três módulos são implementados como co-rotinas.

Num quinto esquema proposto, procura-se explorar as potencialidades de paralelismo do sistema em que o compilador deverá ser executado. Os três blocos básicos do compilador são implementados, neste caso, na forma de três processos paralelos (tarefas assíncronas concorrentes). A comunicação entre os três processos pode ser, por exemplo, efetuada através do uso de mensagens ou de acessos sincronizados a áreas de dados compartilhadas. Em qualquer caso, o veículo de comunicação (mensagem ou área comum) deverá conter os átomos e as subárvores utilizadas para alimentar os módulos de análise sintática e de geração de código, respectivamen-

te. Um processo inicial auxiliar, ou então um dos três processos acima, pode ser utilizado para iniciar a atividade do compilador, dando condições para que cada um dos três processos inicie suas funções. O sistema operacional tem, neste esquema, o importante papel de controlar o bloqueio da execução dos processos que estejam solicitando dados não disponíveis, ou então dos processos que tenham produzido dados além do que os outros dois processos sejam capazes de consumir ou que o sistema seja capaz de armazenar em suas memórias de comunicação.

Naturalmente, os esquemas apresentados não são os únicos existentes ou possíveis de serem implementados. Cabe ao projetista, com base nas informações de que dispõe sobre o seu projeto como um todo, decidir qual esquema adotar ou como combinar entre si os esquemas propostos para formar o arranjo mais adequado ao seu caso particular. Em qualquer caso, uma boa dose de imaginação e criatividade, aliada ao conhecimento das técnicas usualmente adotadas, são essenciais para uma boa escolha e para o adequado encaminhamento do projeto de um compilador.

1.7 – CONSIDERAÇÕES SOBRE CUSTOS E BENEFÍCIOS DOS COMPILADORES

Não se pode afirmar que a tarefa de compilação de uma linguagem de alto nível convencional seja trivial. Ao contrário, há uma razoável complexidade envolvida em sua implementação, exigindo-se, para seu domínio, o estudo de teorias e de métodos especiais aplicáveis à solução dos inúmeros problemas suscitados por atividades a ela relacionadas.

Para defender a conveniência do uso dos compiladores é importante argumentar em favor do uso de linguagens de alto nível. Apesar de todos os progressos ocorridos na área da teoria e construção de compiladores nos últimos anos, progressos estes que levaram à produção de compiladores de desempenho magnífico, ainda há, e são muitos, os que atribuem aos compiladores a culpa total pela alegada ineficiência dos programas-objeto resultantes da tradução automática do programa-fonte.

É inegável que um bom programador das linguagens simbólicas de baixo nível (*"assembly languages"*) certamente conseguirá escrever programas melhores que os produzidos automaticamente pelos compiladores. No entanto, isto tem um custo relativamente alto. Segundo resultados da observação de um número muito grande de casos, a produtividade de um programador, em linhas de código por unidade de tempo, é constante a despeito do nível da linguagem de programação por ele utilizada.

Considerando, grosseiramente, que uma linha de programa-fonte em linguagem de alto nível seja equivalente a dez linhas do programa equivalente em linguagem simbólica, o tempo de desenvolvimento de um programa qualquer será uma ordem de grandeza maior quando desenvolvido em linguagem simbólica do que se for codificado em linguagem de alto nível. Adicionalmente, com respeito à velocidade de execução, constatou-se que, em geral, é desnecessário otimizar um programa em sua totalidade, visto que, também com base na observação de numerosos programas, em média um programa típico permanece 80% do seu tempo de processamento executando apenas 20% de seu código. Assim sendo, torna-se atraente a idéia de desenvolver todo o programa em linguagem de alto nível, acelerando substancialmente o seu desenvolvimento e reduzindo a equipe envolvida, e portanto os custos do projeto. De posse do programa obtido, podem ser, se necessário, efetuados testes com a finalidade de localizar os 20% do código que são responsáveis por quase todo o tempo de execução, e, somente sobre esta parte do código, investir um esforço de otimização, e eventualmente até de recodificação em linguagem simbólica, no caso de aplicações mais críticas.

Outro fato freqüentemente constatado, na análise de programas considerados ineficientes, é a má estruturação dos dados que utilizam, o que acarreta a necessidade de algoritmos complicados e ineficientes para efetuar o acesso a tais dados. Nestes casos, o melhor a fazer é rever todo o projeto de estruturação dos dados e da lógica do programa, antes de recorrer a ações drásticas tais como o abandono do uso de linguagens de alto nível na codificação dos programas.

Um aspecto importante deve ser considerado quando se estuda o papel da linguagem de alto nível e de seu compilador no desenvolvimento de software de boa qualidade: as linguagens de alto nível, através de poderosas construções sintáticas, reduzem o número de linhas de código de um programa, aumentando assim a legibilidade ou capacidade de entendimento do programa-fonte, o que acarreta um aumento da confiabilidade do programa resultante. Isto promove ainda a possibilidade de utilização do conceito de modularidade nos programas desenvolvidos, facilitando a aplicação de metodologias modernas recomendadas pela Engenharia de Software, fornecendo ao programador um suporte para aumentar a clareza e melhorar o estilo de seus programas, facilitando, como decorrência, a análise do programa por um não-especialista, bem como a manutenção do programa após a sua implementação. Pelo fato de, ainda, uma linguagem de alto nível ser relativamente independente de máquina ou do sistema em que é utilizada, os programas desenvolvidos neste tipo de linguagem tornam-se mais versáteis, o que permite seu transporte e implantação em outras máquinas ou sistemas a custos ordens de grandeza menores que os necessários para efetuar o mesmo trabalho em linguagens simbólicas.

Adicionalmente, é conveniente considerar que realmente pode ocorrer uma perda de eficiência, especialmente quando a linguagem de alto nível e a máquina em que esta linguagem irá ser executada não forem compatíveis entre si. Isto é contornado por alguns fabricantes através do projeto de um hardware aderente à linguagem. Outros utilizam máquinas convencionais e incorporam otimizadores aos compiladores desenvolvidos para elas. Deve-se levar em conta que, em geral, otimizadores incorporados a um compilador no mínimo dobram sua complexidade.

Para se ter idéia de ordens de grandeza, pode-se mencionar que para uma linguagem do porte de um BASIC simples é necessário um esforço de desenvolvimento de 3 homens-mês treinados para a confecção de seu compilador, enquanto que, para linguagens de grande porte, como PL/I com otimização ou ALGOL 68, o esforço necessário é de cerca de 30 homens-ano. Como se pode notar, é grande a variedade de porte entre os compiladores existentes. É grande também o atrativo que tais compiladores apresentam, consideradas as vantagens que demonstram sobre o uso de linguagens simbólicas na grande maioria das aplicações.

1.8 – LEITURAS COMPLEMENTARES

Bons textos introdutórios são encontrados na literatura, e que podem ser utilizados como referências para a complementação de informações básicas sobre compiladores. Destacam-se particularmente Gries (1971), Tremblay (1985), Cunin (1980) e Bauer (1976).

Adicionalmente, pode-se encontrar algumas referências sobre a evolução da tecnologia da área de construção de compiladores. São relevantes, entre outros:

KNUTH, D. E. – *A History of Writing Compilers.* Computers and Automation, vol. 11, nº 12. Dec. 1962, p. 8-14.

McCLURE, R. M. – *An Appraisal of Compiler Technology.* Proc. Spring Joint Computer Conference (AFIPS), 1972, p. 1-9.

BAUER, F. L. – *Historical Remarks on Compiler Construction* in Bauer (1976), p. 603-621.

AHO, A. V. – *Translator Writing Systems: Where do They Now Stand?*. Computer, vol. 13, n.º 8. Aug. 1980, p. 9-15.

FELDMAN, J. A. & GRIES, D. – *Translator Writing Systems*. Communications of the ACM, vol. 11, n.º 12. Feb. 1968, p. 77-113.

1.9 – EXERCÍCIOS

1 – Quais os tipos de tradutores que você conhece? Enumere-os, defina-os e exemplifique com casos práticos do seu dia-a-dia.

2 – Identifique as funções executadas por um pré-processador ou por um compilador, em relação ao oferecimento de mecanismos de extensão de uma linguagem de programação.

3 – Liste as principais atividades de um compilador, classificando-as em essenciais ou acessórias. Justifique.

4 – Quais são os métodos que você conhece que permitem a formalização de linguagens de programação? Defina-as e compare-as quanto às semelhanças e diferenças conceituais.

5 – Que são metalinguagens? Exemplifique com alguns casos conceitualmente variados.

6 – Que são compiladores: auto-residentes, auto-compiláveis e cruzados? Justifique a sua aplicabilidade prática e a razão de existência dos três tipos de compiladores.

7 – Descreva pormenorizadamente os passos necessários à obtenção de um compilador ALGOL, para ser executado no microprocessador INTEL 8085. Dispõe-se, como sistema de desenvolvimento, um computador DEC PDP-11, onde a linguagem indicada pelo CPD para ser utilizada é o PASCAL. O compilador resultante deverá ser auto-compilável.

8 – Repita a questão 7, eliminando a especificação de que o compilador resultante deva ser auto-compilável. E se for também relaxada a necessidade de que o compilador seja auto-residente?

9 – Descreva algumas possíveis organizações físicas dos programas compiladores, comparando-as em relação às situações em que podem ser mais adequadamente utilizadas.

10 – Que é um passo de compilação? Qual a diferença entre os compiladores de um e de vários passos?

11 – Que é compilação dirigida pela sintaxe?

12 – Qual é o papel dos otimizadores no processo de compilação? Qual é o impacto de sua utilização, do ponto de vista do programador?

13 – Faça um levantamento dos compiladores aos quais você tem acesso. Verifique, para cada um, a linguagem que compila, o número de passos, a quantidade de memória e de disco utilizados, eventuais linguagens intermediárias empregadas, sua organização física, os recursos de compilação oferecidos ao programador, o seu valor de compra, etc.

14 – Procure ler, na literatura técnica da área, alguns artigos sobre os conceitos básicos de compilação.

15 – Situe o compilador em um sistema de programação completo, mostrando sua relação com os outros módulos do software básico, e descrevendo em detalhes sua operação neste ambiente.

16 – Utilize o esquema que você construiu na questão 15, e mostre como a operação varia conforme os métodos e técnicas utilizados na compilação. Leve em consideração o nível da linguagem, e número de passos, a(s) linguagem(ns) intermediária(s) adotada(s), a quantidade de memória principal disponível, e a aplicação a que se destina o compilador.

17 – Que estratégia você adotaria para construir um compilador para uma dada linguagem em prazo mínimo? Não são importantes, no caso, as considerações de eficiência.

18 – Suponha que você disponha, em um computador, dos recursos de compilação oferecidos por um montador e por um compilador Pascal, apenas. Se for necessário desenvolver, para este computador, um compilador para uso em aplicações de tempo real, como poderia tal compilador ser organizado e construído?

19 – Que são interpretadores? Quais as semelhanças e diferenças que eles guardam em relação aos compiladores?

20 – Faça um esquema de projeto estrutural e um plano de implementação para um interpretador voltado para aplicações didáticas. Como seria um compilador voltado para a mesma classe de aplicações?

21 – Quais as dificuldades encontradas quando se deseja transportar um compilador de uma máquina para outra? Como contorná-las?

22 – Estude o compromisso entre o tempo de desenvolvimento do compilador, a velocidade do compilador e a eficiência do código-objeto por ele gerado. Faça uma tabela indicando as estratégias utilizadas para cada combinação, e uma indicação dos casos em que podem ser aplicadas estas estratégias.

23 – Estude as implicações da adoção das diversas possíveis estratégias da construção de compiladores, em diferentes situações. Considere neste estudo a aplicação a que se destina o compilador; a complexidade da linguagem a ser compilada; o porte e os recursos disponívies na(s) máquina(s) utilizada(s); os prazos e recursos relativos do projeto; a eficiência do compilador; a eficiência do código gerado; a decisão entre implementar um compilador e um interpretador; a portabilidade do produto.

24 – Que são pré-processadores? Esboce uma estratégia para a inclusão de um pré-processador para uma linguagem disponível na sua máquina.

25 – Em que condições pode um expansor de macros ser utilizado como pré-processador de linguagem de alto nível? Esboce a operação do conjunto.

2

Introdução à Teoria de Linguagens

Neste capítulo são introduzidos os conceitos, relacionados aos aspectos formais das linguagens de programação, que são relevantes ao estudo das técnicas e métodos de projeto e construção de compiladores. Pretende-se cobrir apenas de maneira superficial este vasto assunto, de modo tal que sejam evidenciados os resultados mais importantes das teorias que envolvem a formalização das linguagens de programação.

Não serão desenvolvidas justificativas nem demonstrações, mas procurar-se-á apresentar as notações e as terminologias mais importantes, bem como os resultados mais significativos para o estudo a ser realizado.

2.1 – TERMINOLOGIA BÁSICA

O presente estudo tem como meta criar mecanismos e formalismos através dos quais seja possível descrever, analisar e sintetizar linguagens de programação. O centro das atenções é portanto o estudo de linguagens. *Linguagem* é uma coleção de cadeias de símbolos, de comprimento finito. Estas cadeias são denominadas *sentenças* da linguagem, e são formadas pela justaposição de elementos individuais, os *símbolos* ou *átomos* da linguagem.

Uma linguagem pode ser representada através de três mecanismos básicos: por *enumeração* das cadeias de símbolos que formam as suas sentenças (só linguagens finitas podem ser representadas através deste método), através de um conjunto de *leis de formação* das cadeias (ao conjunto de leis de formação dá-se o nome de *gramática*), ou então através de *regras de aceitação* de cadeias (à regra de aceitação dá-se o nome de *reconhecedor*).

No primeiro caso, todas as sentenças da linguagem aparecem explicitamente na enumeração, e a decisão acerca da pertinência ou não de uma cadeia à linguagem se faz por meio de uma busca da cadeia em questão no conjunto de sentenças da linguagem.

No caso de se dispor de uma descrição da linguagem através de leis de formação (gramáticas), dada uma cadeia de símbolos, só é possível afirmar que tal cadeia pertence à linguagem

se for possível, aplicando-se as leis de formação que compõem a gramática da linguagem, sintetizar a cadeia em questão. Ao processo de obtenção de uma sentença a partir da gramática dá-se o nome de *derivação* da sentença.

No terceiro caso, dispõe-se de um conjunto de regras de aceitação. Para decidir se uma cadeia é uma sentença da linguagem, basta aplicar a ela as regras de aceitação, as quais deverão fornecer a decisão desejada acerca da pertinência da cadeia à linguagem.

Note-se que as gramáticas e os reconhecedores são modelos da linguagem que procuram formalizar, sendo que as primeiras são dispositivos generativos, enquanto os últimos são dispositivos de aceitação. Para as atividades ligadas à construção de compiladores, o uso direto de gramáticas não é, geralmente, prático, sendo em geral necessário obter reconhecedores que descrevem a mesma linguagem, para então completar a construção do compilador. Esta será uma das metas a serem atingidas no decorrer deste texto.

2.2 – CADEIAS

Inicialmente convém definir alguns conceitos e notações relacionadas ao formalismo de cadeias e, por esta razão, o estudo das cadeias merece algum destaque ao se introduzir as noções sobre o formalismo de linguagens. Seja Σ um *alfabeto*, conjunto finito não vazio de *símbolos* de que são formadas as cadeias. Define-se a *cadeia vazia* ϵ como sendo uma cadeia que não contém nenhum símbolo. Diz-se que o comprimento da cadeia vazia é zero. Define-se uma *cadeia elementar*, de comprimento unitário, como sendo uma cadeia formada por um único símbolo $\sigma \in \Sigma$. Uma cadeia arbitrária α de comprimento n pode ser obtida através da justaposição de um número n arbitrário de símbolos de Σ. Denota-se o fato como $|\alpha| = n$.

Exemplos:

Alfabeto: $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Símbolos: *1, 7, 2, etc.*

Cadeias Elementares: *1, 8, 0, 3, etc.*

Cadeias: *123094, 109, 001, 6,* ϵ

Comprimento: $|123094| = 6$
$|109| = 3$
$|\epsilon| = 0$

Define-se *concatenação* β de duas cadeias α e α' como sendo a cadeia formada pela justaposição das seqüências de símbolos α e α', nesta ordem. Denota-se como $\beta = \alpha \cdot \alpha'$ ou simplesmente $\beta = \alpha\alpha'$. Obviamente o comprimento de β será igual à soma dos comprimentos de α e α': $|\beta| = |\alpha| + |\alpha'|$.

Define-se concatenação β de um símbolo $\sigma \in \Sigma$ a uma cadeia α como sendo a cadeia formada pela concatenação de cadeia elementar, formada pelo símbolo σ, à cadeia α. Esta concatenação pode ser feita à direita ou à esquerda de α:

$$\beta = \sigma\alpha \quad \text{ou} \quad \beta = \alpha\sigma$$

Neste caso, $|\beta| = |\alpha| + 1$. É importante que a operação de concatenação é associativa:

$$\alpha(\beta\,\gamma) = (\alpha\,\beta)\,\gamma = \alpha\,\beta\,\gamma \quad (\alpha, \beta, \gamma, \text{são cadeias})$$

Não é entretanto, comutativa, pois $\alpha\,\beta$ nem sempre forma a mesma cadeia que $\beta\,\alpha$ (α, β são cadeias).

Vale ainda observar que ϵ funciona como elemento neutro desta operação:

$$\alpha\,\epsilon = \epsilon\,\alpha = \alpha \quad (\alpha \text{ é cadeia})$$

Exemplos:

Concatenação: $(109) \cdot (6) = 1096$
$|1096| = |109| + |6| = 3_{10} + 1_{10} = 4_{10}$

Concatenação c/símbolo: $0 \cdot (109) = 0109$
$|0109| = 1_{10} + |109| = 1_{10} + 3_{10} = 4_{10}$

Associatividade: $(123094) \cdot ((109) \cdot (001)) =$
$= ((123094) \cdot (109)) \cdot (001) =$
123094109001

Não Comutatividade: $(109) \cdot (6) = 1096$
$(6) \cdot (109) = 6109$
$1096 \neq 6109$

Elem. Neutro: $(001) \cdot \epsilon = \epsilon \cdot (001) = 001$

Dá-se o nome de *fechamento recursivo e transitivo* do conjunto Σ ao conjunto Σ^* formado por todas as cadeias, de qualquer comprimento, que se possa construir com os elementos de Σ. Este conjunto inclui a cadeia vazia ϵ. Note-se que Σ^* é infinito, já que, por definição, Σ é finito e não vazio. Denota-se como Σ^+ (*fechamento transitivo*) o conjunto formado retirando-se a cadeia vazia do conjunto Σ^*: $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$, onde $\epsilon \notin \Sigma^+$ (observe-se que Σ^* e Σ^+ são conjuntos extremamente gerais e englobam toda sorte de possíveis linguagens que se possam construir a partir do conjunto Σ).

Exemplos: Para o conjunto $\Sigma = \{0, 1, \ldots, 9\}$ *tem-se:*

Fechamento transitivo:

$$\Sigma^+ = \{0, 1, \ldots, 9, 00, 01, \ldots, 99, 000, \ldots, 999, \ldots\}$$

Fechamento recursivo e transitivo:

$$\Sigma^* = \{\epsilon, 0, 1, \ldots, 9, 00, \ldots, 99, 000, \ldots, 999, \ldots\}$$

As operações de fechamento são utilizadas com muita freqüência nos estudos de linguagens formais, aplicando-se a diversos tipos de operadores. No caso mencionado, o operador sobre o qual as operações de fechamento atuam é o operador de concatenação. Para um operador genérico $\odot$, denota-se como $\odot^*$ e $\odot^+$, respectivamente, as operações de fechamento mencionadas, significando que o operador $\odot$ deve ser aplicado um número qualquer de vezes sobre os elementos do conjunto sobre o qual $\odot$ é definido. No caso do fechamento transitivo $\odot^+$ é necessário que o operador $\odot$ seja aplicado no mínimo uma vez, restrição esta que não recai sobre o fechamento recursivo e transitivo $\odot^*$, em que $\odot$ pode eventualmente não ser aplicado. Para relações $\odot$ binárias em que $\odot$ opera sobre dois argumentos α e β, $\odot^*$ e $\odot^+$ denotam aplicações sucessivas do operador sobre α e β, segundo a definição dos fechamentos respectivos.

Uma notação adicional, muito usada em teoria de linguagens, é a da *concatenação de linguagens:* Sejam X e Y duas linguagens. Denota-se por XY a linguagem formada através da concatenação de cada sentença de X com cada uma das sentenças de Y.

Com estas informações sobre as cadeias, pode-se passar, a seguir, ao estudo das gramáticas e dos reconhecedores que sobre elas operam.

2.3 – GRAMÁTICAS

Formalmente as gramáticas, dispositivos de geração de sentenças das linguagens que definem, podem ser caracterizadas como quádruplas ordenadas

$$G = (V, \Sigma, P, S)$$

onde V representa o *vocabulário* da gramática G. Este vocabulário corresponde ao conjunto de todos os elementos simbólicos dos quais a gramática se vale para definir as leis de formação das sentenças da linguagem.

Σ representa alguns dos elementos de V, que são exatamente os símbolos ou átomos dos quais as sentenças da linguagem são constituídas. Dá-se o nome de *terminais* aos elementos de Σ, no contexto das gramáticas. Os elementos de V que não pertencem a Σ são usados pela gramática para definir construções auxiliares ou intermediárias na formação das sentenças. A estes elementos dá-se o nome de *não-terminais*, e seu conjunto N obedece à relação $V = \Sigma \cup N$, com $\Sigma \cap N$ vazio.

P representa o conjunto de todas as leis de formação utilizadas pela gramática para definir a linguagem. Para tanto, cada construção parcial, representada por um não-terminal, é definida como um conjunto de regras de formação relativas à definição do não-terminal a ela referente. A cada uma destas regras de formação que compõem o conjunto P dá-se o nome de *produção* da gramática. Cada produção de P tem a forma:

$$\alpha \to \beta$$

onde α é, no caso geral, uma cadeia contendo no mínimo um não-terminal, ou seja: $\alpha \in V^* N V^*$ e β é uma cadeia, eventualmente vazia, de terminais e não-terminais: $\beta \in V^*$.

S é um elemento de N, cuja propriedade é o de ser o não-terminal que dá início ao processo de geração de sentenças. S é dito o *símbolo inicial* da gramática.

As gramáticas devem ser vistas como *sistemas de substituição,* nos quais as produções indicam as substituições possíveis para os não-terminais. Partindo-se do símbolo inicial da gramáti-

ca, que é um não-terminal, e aplicando-se, sucessivamente, substituições aos não-terminais remanescentes, de acordo com as regras definidas pelas produções, até que todos os não-terminais sejam eliminados, o resultado é uma sentença da linguagem. Mais rigorosamente, pode-se definir este procedimento através da introdução de alguns conceitos auxiliares:

Dá-se o nome de *forma sentencial* a qualquer cadeia de elementos de V obtida a partir de S por substituição dos não-terminais, efetuada conforme as produções do conjunto P:

a) S é uma forma sentencial

b) Se $\alpha\ \beta\ \gamma$ é uma forma sentencial e $\beta \rightarrow \delta$ for uma produção pertencente ao conjunto P, então $\alpha\ \delta\ \gamma$ também será uma forma sentencial ($\alpha, \delta, \gamma \in V^*, \beta \in V^* N\ V^*$).

Dá-se o nome de *sentença* a uma forma sentencial F particular em que todos os elementos que a compõem forem terminais, ou seja, $F \in \Sigma^*$.

Define-se uma *derivação direta* como sendo uma substituição em que apenas uma produção é aplicada: se $\alpha\ \beta\ \gamma \in V^*$ e $\beta \rightarrow \delta \in P$, denota-se como $\alpha\ \beta\ \gamma \underset{G}{\rightarrow} \alpha\ \delta\ \gamma$ a derivação que substitui β por δ, segundo a gramática G.

Uma derivação não-trivial corresponde a uma aplicação de no mínimo uma derivação direta, e é denotada como $\underset{G}{\Rightarrow^+}$.

Denomina-se simplesmente *derivação* à aplicação de zero ou mais derivações diretas denotando-se tal operação como $\underset{G}{\Rightarrow^*}$.

Obviamente, quando não houver dúvidas quanto a qual gramática se refere uma derivação, os símbolos $\underset{G}{\Rightarrow}$, $\underset{G}{\Rightarrow^+}$ e $\underset{G}{\Rightarrow^*}$ podem ser substituídos por $\Rightarrow$, $\Rightarrow^+$ e $\Rightarrow^*$ respectivamente, sem perda de informação.

Com as definições acima, torna-se possível finalmente definir a *linguagem* gerada pela gramática G como sendo o conjunto de todas as possíveis sentenças por ela geradas através de derivações a partir do símbolo inicial S:

$$\pounds(G) = \left\{ F \in \Sigma^* \mid S \underset{G}{\Rightarrow^*} F \right\}$$

Exemplos:

Gramática

$G1 = (V1, \Sigma 1, P1, S1)$

onde $V1 = \{A, B, 0, 1\}$

$\Sigma 1 = \{0, 1\}$

$P1 = \{A \rightarrow 0A, A \rightarrow B, B \rightarrow 1B, B \rightarrow \epsilon\}$

$S1 = A$

Obtenção de Sentença:	*aplicando-se*
(1) $A \Rightarrow 0A$	$A \rightarrow 0A$
(2) $0A \Rightarrow 00A$	$A \rightarrow 0A$
(3) $00A \Rightarrow 00B$	$A \rightarrow B$
(4) $00B \Rightarrow 001B$	$B \rightarrow 1B$
(5) $001B \Rightarrow 0011B$	$B \rightarrow 1B$
(6) $0011B \Rightarrow 0011$	$B \rightarrow \epsilon$

(1), (2), ... , (6) : derivações diretas
0011 : sentença
A, 0A, 00A, 00B, 001B, 0011B, 0011 : formas sentenciais
$0A \Rightarrow^{*} 00A$: *derivação*
$00A \Rightarrow^{+} 0011B$: *derivação não trivial*
$S \Rightarrow^{+} 0011$: *derivação da sentença*
$0^{*}1^{*}$: *linguagem gerada pela gramática G1*

Note-se que até este ponto não foi imposta qualquer restrição sobre a gramática ou sobre as produções que denotam as leis de formação da linguagem que está sendo definida. As gramáticas gerais têm limitações em relação à sua aplicabilidade no contexto do estudo dos compiladores, devido às dificuldades que acarretam em seu tratamento, sendo que as linguagens de programação de interesse não exigem toda a generalidade que as gramáticas gerais definidas acima são capazes de oferecer. Torna-se atraente o estudo de casos particulares, de aplicação mais restrita, porém suficiente para resolver os problemas levantados ao se projetar compiladores para as linguagens de interesse.

Conforme as restrições impostas ao formato das produções de uma gramática, a classe de linguagens que tal gramática gera varia correspondentemente. A teoria mostra que há quatro classes de gramáticas, capazes de gerar quatro classes correspondentes de linguagens, de acordo com a denominada *hierarquia de Chomsky:*

Gramáticas Irrestritas – São aquelas às quais nenhuma limitação é imposta. Obviamente, todo o universo das linguagens que se podem definir através dos mecanismos generativos definidos pelas gramáticas, corresponde exatamente ao conjunto das linguagens que esta classe de gramáticas é capaz de gerar. A esta classe de gramáticas, a hierarquia de Chomsky classifica como sendo a das *gramáticas irrestritas,* ou do *tipo 0.* Chamam-se linguagens do tipo 0 todas as linguagens que podem ser geradas por alguma gramática do tipo 0. Para gramáticas do tipo 0, as produções são todas da forma

$$\alpha \rightarrow \beta, \text{ com } \alpha \in V^{*} N V^{*} \text{ e } \beta \in V^{*}.$$

Exemplo:

$$G = (\{A, B, C\}, \{a, b\}, \{A \rightarrow BC, BC \rightarrow CB, B \rightarrow b, C \rightarrow a\}, A)$$

derivações:

$A \Rightarrow BC \Rightarrow CB \Rightarrow aB \Rightarrow ab$
$A \Rightarrow BC \Rightarrow bC \Rightarrow ba$

Gramáticas Sensíveis ao Contexto – Se às regras de substituição for imposta a restrição de que nenhuma substituição possa reduzir o comprimento da forma sentencial à qual a substituição é aplicada, cria-se uma classe de gramáticas, ditas sensíveis ao contexto.

As produções devem ser todas da forma $\alpha \rightarrow \beta$, com $|\alpha| \leqslant |\beta|$ onde $\alpha \in V^{*} N V^{*}$ e $\beta \in V^{*}$. As gramáticas que obedecem a estas restrições pertencem, na hierarquia de Chomsky, ao conjunto das chamadas gramáticas *sensíveis ao contexto,* ou do *tipo 1.* Chamam-se linguagens do tipo 1 todas aquelas que podem ser geradas por alguma gramática do tipo 1.

Note-se que todas as linguagens do tipo 1 podem ser geradas através de gramáticas irrestritas do tipo 0, embora o inverso não seja verdade. Assim, surge a seguinte relação de inclusão: as linguagens irrestritas incluem todas as linguagens sensíveis ao contexto. O mesmo ocorre com as correspondentes gramáticas.

Observe-se ainda que, no caso geral, uma produção de uma gramática sensível ao contexto se apresenta como regra para substituir uma cadeia da forma $\gamma\ B\ \delta$, com $\gamma, \delta \in V^*$. Em outras palavras, trata-se de regras para substituição, determinada pelo não-terminal B, condicionada à presença das cadeias γ e δ, respectivamente à sua esquerda e à sua direita.

Exemplo:

$$G = (\{A, B, C\}, \{a, b\}, \{A \rightarrow AB, AB \rightarrow AC, C \rightarrow abA\}, A).$$

derivações:

$$A \Rightarrow AB \Rightarrow AC \Rightarrow AabA \Rightarrow \ldots$$
$$A \Rightarrow AB \Rightarrow AAB \Rightarrow AAC \Rightarrow \ldots$$

Gramáticas Livres de Contexto – Conceituam-se gramáticas livres de contexto como sendo aquelas em que é levantado o condicionamento das substituições impostas pelas regras definidas pelas produções. Este condicionamento é eliminado impondo às produções uma restrição adicional, que restringe as produções à forma geral $A \rightarrow \alpha$, onde $A \in N$, $\alpha \in V^*$, ou seja, o lado esquerdo da produção é um não-terminal isolado e α é a cadeia pela qual A deve ser substituído ao ser aplicada esta regra de substituição, independentemente do contexto em que A está imerso. Daí o nome "livre de contexto" aplicado às gramáticas que obedecem a esta restrição.

Segundo a hierarquia de Chomsky, esta classe de gramáticas é classificada como *livre de contexto* ou do *tipo 2*. Denominam-se linguagens do tipo 2 aquelas que podem ser definidas através de gramáticas do tipo 2.

Notar novamente a relação de inclusão: toda gramática ou linguagem do tipo 2 também é do tipo 1, e portanto do tipo 0.

Exemplo:

$$G = (\{S\}, \{a, +, *, (,)\}, \{S \rightarrow S * S, S \rightarrow S + S, S \rightarrow (S), S \rightarrow a\}, S)$$

derivações:

$$S \Rightarrow S * S \Rightarrow S * S + S \Rightarrow S * (S) + S \Rightarrow$$
$$\Rightarrow S * (S) + a \Rightarrow S * (a) + a \Rightarrow a * (a) + a$$

Gramáticas Lineares à Direita (esquerda) – Aplicando-se mais uma restrição sobre a forma das produções, pode-se criar uma nova classe de gramáticas, de grande importância no estudo dos compiladores por possuírem propriedades adequadas para a obtenção de reconhecedores simples. Nesta classe de gramáticas, as produções são restritas à formas seguintes:

$$A \rightarrow \alpha B \qquad (A \rightarrow B\alpha)$$
$$A \rightarrow \alpha$$

onde $\alpha \in \Sigma^*$; $A, B \in N$.

Em palavras, uma gramática linear à direita (esquerda) admite apenas regras de substituição de um não-terminal por uma cadeia de terminais, seguida (precedida) ou não por um não-terminal único.

À classe de gramáticas assim definida dá-se o nome de *gramáticas lineares à direita (esquerda)* ou do *tipo 3*. Denominam-se *linguagens do tipo 3*, na hierarquia de Chomsky, as linguagens que podem ser definidas através de alguma gramática do tipo 3. Assim sendo, as gramáticas e as linguagens do tipo 3 também podem ser classificadas como sendo dos tipos 2, 1 ou 0.

Exemplo: $G = (\{S\}, \{a, b\}, \{S \to aS, S \to b\}, S)$

Derivações: $S \Rightarrow a\,S \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaab$

Em termos gerais, para $n \in \{0, 1, 2, 3\}$ pode-se afirmar que uma gramática de qualquer tipo pode ser classificada também como sendo de tipo menor, se for o caso. Analogamente, uma linguagem do tipo n é caracterizada pela existência de alguma gramática do tipo n que a descreva, podendo ser também classificada como sendo do tipo menor, se for o caso.

Conceituadas e classificadas as gramáticas e suas relações com as linguagens que definem, passa-se a um estudo similar, relativo aos reconhecedores.

2.4 – RECONHECEDORES

Como alternativa para a definição de uma linguagem, é possível a utilização de dispositivos aceitadores, denominados *reconhecedores* da linguagem. Através dos reconhecedores é possível submeter uma cadeia de símbolos a um teste de aceitação capaz de determinar se tal cadeia pertence ou não à linguagem em questão.

Um reconhecedor (*Figura 2.1*) é um dispositivo conceitual que pode ser visualizado, formalmente, através da identificação de seus componentes fundamentais: um *texto de entrada*, representado por uma cadeia de símbolos. Esta cadeia é percorrida por um *cursor* encarregado de efetuar a leitura dos símbolos da cadeia por ele apontados. Este cursor é controlado por uma *máquina finita de controle de estados*, que se encarrega de movimentá-lo e de consumir o produto de suas leituras.

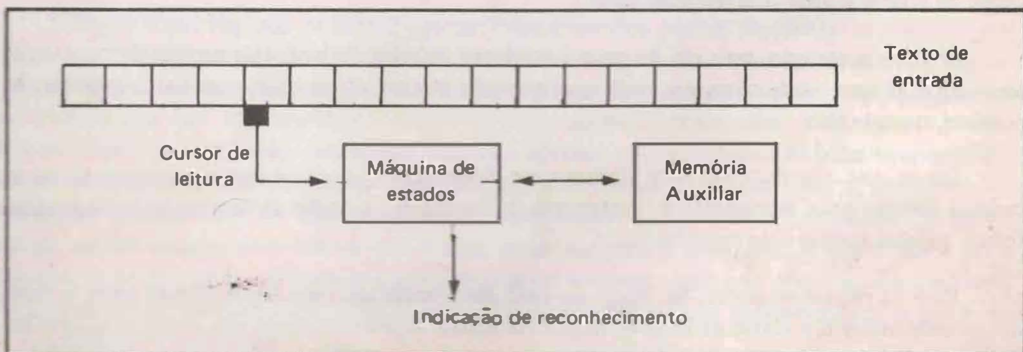


Figura 2.1 – Esquema de um reconhecedor genérico.

Esta máquina pode utilizar-se eventualmente de uma *memória auxiliar*, normalmente uma pilha, para memorizar eventos ocorridos durante o reconhecimento.

Nesta memória, são armazenadas informações codificadas conforme um *alfabeto de memória*. A máquina finita de controle de estados efetua movimentos, os quais podem determinar a leitura (ou não) de um elemento do texto de entrada, o *armazenamento* (ou não) de um elemento na memória auxiliar, e a *mudança de estado da máquina*. Em qualquer instante pode-se caracterizar a *configuração* do reconhecedor como sendo a tripla formada pelo *estado corrente* da máquina, pela *posição do cursor e conteúdo do texto de entrada*, e pelo *conteúdo da memória auxiliar*.

Do ponto de vista de funcionamento, os reconhecedores dependem fundamentalmente da estrutura e da operação da máquina de estados em que se baseia. Dois grandes grupos de reconhecedores podem ser então identificados: *os reconhecedores determinísticos*, para os quais qualquer possível configuração admite um único movimento no máximo, e os *reconhecedores não-determinísticos*, em que cada configuração admite um conjunto finito de movimentos possíveis. Naturalmente, os reconhecedores não-determinísticos operam por tentativa e erro, o que determina um desempenho pobre, desqualificando-os, na grande maioria das aplicações, como base para a construção de compiladores eficientes. Felizmente a teoria oferece uma série de recursos que permitem obter reconhecedores determinísticos para uma ampla classe de linguagens de interesse, garantindo, desta forma, a obtenção de compiladores com boa eficiência.

Para a operação de um reconhecedor, é necessário levá-lo, antes de mais nada, a uma *configuração inicial*, em que a máquina de estados é posicionada em um estado inicial bem definido, o cursor aponta o início da cadeia de entrada a ser analisada e a memória é preenchida com um conteúdo inicial conhecido e conveniente.

Define-se como *configuração final do reconhecedor* aquela em que o cursor aponta imediatamente além do último símbolo da cadeia de entrada, a máquina de estados está posicionada em um dos estados de um conjunto adequado de *estados finais* ou *estados de aceitação* da cadeia de entrada, e o conteúdo da memória auxiliar satisfaz um critério pré-estabelecido.

Com estas definições em mente, pode-se afirmar que ocorre a *aceitação* de uma cadeia de entrada *W* pelo reconhecedor quando, partindo-se de uma configuração inicial, com a cadeia *W* na entrada do reconhecedor, este efetuar alguma seqüência de movimentos que seja capaz de levá-lo a uma configuração final.

Deve-se notar que, no caso de reconhecedores determinísticos, esta seqüência, se existir, será única, já que, neste caso, em cada configuração intermediária existe um único movimento possível, no máximo.

Diz-se que um *reconhecedor define uma linguagem* quando, e apenas quando, todas as cadeias aceitas pelo reconhecedor pertencem à linguagem, e todas as sentenças da linguagem forem cadeias aceitas pelo reconhecedor.

Para os reconhecedores, há, como no caso das gramáticas, uma forte relação entre a classe do reconhecedor e a classe da linguagem por ele aceita. Apesar de ainda não terem sido detalhados os vários tipos de reconhecedores, são apresentadas a seguir e sumarizadas na *Tabela 2.1* as diversas classes de linguagens estudadas, e a indicação dos reconhecedores que lhes correspondem:

Linguagens do tipo 0, ditas *conjuntos recursivamente enumeráveis*, são aceitas por reconhecedores denominados *Máquinas de Turing*, que podem ser vistas como dispositivos de reconhecimento com número finito de estados, e com uma memória auxiliar que toma a forma de uma fita de comprimento finito, porém não limitado, onde podem ser efetuadas operações de leitura e de gravação.

Linguagens do tipo 1, ou linguagens sensíveis ao contexto, são aceitas por máquinas de Turing limitadas nas dimensões de sua fita de rascunho.

Linguagens do tipo 2, ou livres de contexto, são aceitas por uma classe importante de reconhecedores, denominados *autômatos de pilha*. A memória de rascunho deste tipo de reconhecedor é organizada em pilha (LIFO = "last-in, first-out").

Linguagens do tipo 3, ou conjuntos regulares, são aceitos pela classe mais simples dos reconhecedores, denominados *autômatos finitos*.

LINGUAGEM	GRAMÁTICA	RECONHECEDOR
tipo 0: conjuntos recursivamente enumeráveis	tipo 0: gramáticas irrestritas	Máquina de Turing
tipo 1: sensíveis ao contexto	tipo 1: gramáticas sensíveis ao contexto	Máquinas de Turing com memória limitada
tipo 2: livres de contexto	tipo 2: gramáticas livres de contexto	Autômatos de Pilha
tipo 3: conjuntos regulares	tipo 3: gramáticas lineares à direita	Autômatos Finitos

Tabela 2.1 – Relação de correspondência entre as classes de linguagens, gramáticas e reconhecedores

Cumpre comentar que os reconhecedores mencionados podem apresentar-se como dispositivos determinísticos ou não-determinísticos, sendo que, para alguns casos, como o das linguagens do tipo 2, os reconhecedores não-determinísticos são capazes de reconhecer linguagens particulares que não são reconhecíveis por reconhecedores determinísticos da mesma classe. Para este caso, por exemplo, reconhecedores que operam com autômatos de pilha determinísticos são capazes de aceitar a classe de linguagens livres de contexto determinísticas, definidas por gramáticas denominadas $LR\ (k)$, que descrevem linguagens que podem ser reconhecidas da esquerda para a direita, com leitura antecipada de no máximo k símbolos. É esta a classe mais importante de linguagens determinísticas pesquisadas até hoje, uma vez que engloba a maioria das linguagens de programação, embora seja suficientemente restrita para permitir a construção de reconhecedores eficientes.

Passa-se, a seguir, à particularização dos diversos casos de interesse, caracterizando os reconhecedores correspondentes de modo mais formal.

Autômatos Finitos – São reconhecedores definidos através de quíntuplas da forma:

$$M = (Q, \Sigma, P, q_0, F)$$

onde

Q é um conjunto finito não vazio de *estados* do autômato finito.

Σ é denominado o *alfabeto de entrada* do autômato e corresponde a um conjunto finito não vazio dos *símbolos de entrada* ou *átomos* indivisíveis que compõem a cadeia de entrada submetida ao autômato para aceitação.

P é uma *função de transição* de estados do autômato e seu papel é o de indicar as transições possíveis em cada configuração do autômato. Esta função mapeia o produto cartesiano $Q \times (\Sigma \cup \{\epsilon\})$ em Q, ou seja, fornece para cada par (*estado, símbolo de entrada*) um novo estado para onde o autômato deverá mover-se.

q_0 é denominado o *estado inicial* do autômato finito, e corresponde a um elemento do conjunto Q. É o estado para o qual o reconhecedor deve ser levado antes de iniciar suas atividades ($q_0 \in Q$).

F é um subconjunto do conjunto Q dos estados do autômato, e contém todos os *estados de aceitação* ou *estados finais* do autômato finito. Estes estados são aqueles em que o autômato deve terminar o reconhecimento das cadeias de entrada que pertencem à linguagem que o autômato define. Nenhuma outra cadeia deve ser capaz de levar o autômato a qualquer destes estados ($F \subseteq Q$),

---

Exemplo:

$$M = (\{A, B\}, \{0, 1\}, \{(A, 0) \rightarrow A, (A, 1) \rightarrow B, (B, 1) \rightarrow B, (B, 0) \rightarrow A\}, A, \{B\})$$

Para este autômato finito, reconhecem-se os seguintes elementos:

estados do autômato: A e B
símbolos do alfabeto de entrada: 0 e 1
estado final: B
estado inicial: A
linguagem reconhecida: cadeias de dígitos binários iniciados obrigatoriamente por um dígito 0 e terminadas obrigatoriamente por um dígito 1.

Prova-se que o conjunto de todas as linguagens reconhecíveis através de autômatos finitos é exatamente o mesmo que o das linguagens geradas por gramáticas lineares à direita (esquerda).

Dá-se às linguagens desta classe o nome de *linguagens regulares*.

Prova-se ainda que os autômatos finitos reconhecem sempre linguagens regulares, quer sejam determinísticos ou não-determinísticos. Isto é um importante resultado, uma vez que se baseia em uma garantia teórica de equivalência de poder de reconhecimento entre os dois grupos de autômatos finitos. Há também a certeza teórica de que, para qualquer autômato finito não-determinístico, existe sempre outro autômato, determinístico, que reconhece a mesma linguagem. Adicionalmente, a teoria mostra os caminhos para a obtenção de tais autômatos, bem como métodos para a sua redução e minimização. O resultado mais importante destes estudos é que, qualquer que seja a linguagem regular apresentada, é possível obter um autômato finito determinístico mínimo que a reconheça, e que, além disto, tal autômato mínimo é único.

Autômatos de Pilha – São reconhecedores definidos através de uma sétupla da forma

$$M = (Q, \Sigma, \Gamma, P, q_0, Z_0, F)$$

onde

Q é um conjunto finito não vazio de *estados* do autômato de pilha.

Σ é um conjunto finito não vazio de *símbolos de entrada* ou *átomos*, denominado *alfabeto de entrada* do autômato de pilha. Os símbolos de entrada são os elementos de que são formadas as cadeias de entrada, que são submetidas ao autômato para aceitação.

Γ é um conjunto finito não vazio de símbolos de pilha, e forma o *alfabeto de pilha.* Os símbolos de pilha são os códigos armazenados pelo autômato em sua memória auxiliar. Esta memória, no caso do autômato de pilha, é organizada na forma de uma pilha, ou seja, os últimos dados armazenados são os primeiros a serem lidos da pilha, e vice-versa.

P é a chamada *função de transição* do autômato de pilha, e é composta de um conjunto de *produções* que definem as regras de movimentação do autômato de pilha. Esta função mapeia o produto cartesiano $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ no produto cartesiano $Q \times \Gamma^*$. Em palavras, dado um estado, um símbolo de entrada e um símbolo de pilha contido no topo da memória auxiliar, esta função determina um novo estado do autômato e o novo conteúdo do topo de pilha (de comprimento qualquer).

q_0 é denominado o *estado inicial* do autômato de pilha, e é um elemento do conjunto Q. É o estado em que deve se encontrar o autômato de pilha imediatamente antes do início do reconhecimento de uma cadeia de entrada ($q_0 \in Q$).

Z_0 é um elemento do conjunto Γ, distinto dos demais pela convenção de que sua presença, no topo da pilha que implementa a memória do autômato, indica a ausência de outros elementos na mesma. É um *marcador de pilha vazia* ($Z_0 \in \Gamma$).

F é o subconjunto do conjunto de estados Q do autômato, que contém todos os chamados *estados finais* ou *estados de aceitação* do autômato de pilha. Tais estados correspondem àqueles nos quais o autômato de pilha deve encerrar o reconhecimento de todas as cadeias de entrada que sejam sentenças da linguagem definida pelo autômato de pilha. Nenhuma outra cadeia deve levar o autômato a qualquer destes estados.

Exemplo:

$$\begin{aligned} M = (&\{A, B, C\},\ \{0, 1\},\ \{x, y\},\\ &\{(A, 1, y) \rightarrow (A, yx),\\ &(A, 1, x) \rightarrow (A, xx),\\ &(A, 0, y) \rightarrow (B, y),\\ &(A, 0, x) \rightarrow (B, x),\\ &(B, 0, x) \rightarrow (B, x),\\ &(B, 1, xx) \rightarrow (C, x),\\ &(B, 1, yx) \rightarrow (C, y),\\ &(C, 1, xx) \rightarrow (C, x),\\ &(C, 1, yx) \rightarrow (C, y)\},\\ &A, y, \{C\}) \end{aligned}$$

Este autômato reconhece cadeias binárias da forma $1^n\ 0^{1+m}\ 1^n$, onde m e n são inteiros não-negativos e α^n simboliza uma cadeia de n símbolos α seguidos.

Prova-se que o conjunto de todas as linguagens reconhecíveis através de autômatos de pilha é exatamente o mesmo das linguagens geradas por gramáticas livres de contexto. A esta classe de linguagens dá-se o nome de *linguagens livres de contexto*. Conseqüentemente, esta classe de reconhecedores é capaz de reconhecer qualquer linguagem regular. Um subconjunto dos autômatos de pilha é de particular interesse: trata-se dos *autômatos de pilha determinísticos*, os quais se caracterizam por apresentarem funções de transição de estado tais que, qualquer que seja a configuração do reconhecedor, a produção aplicável é única. Em outras palavras, um único movimento é permitido para o autômato no máximo, qualquer que seja a sua configuração. Linguagens livres de contexto reconhecíveis por autômatos de pilha determinísticos são denominadas *linguagens livres de contexto determinísticas.*

A classe das gramáticas $LR\ (k)$, mencionada anteriormente, é, dentre os tipos de gramática conhecidas que são classificáveis como sendo do tipo 2, aquela que menos restrições impõe, e que maior espectro de linguagens do tipo 2 consegue descrever, representando atualmente a mais importante e poderosa classe teórica conhecida de gramáticas utilizadas para a representação de linguagens livres de contexto determinísticas. Todas as demais linguagens livres de contexto, não reconhecíveis por autômatos de pilha determinísticos, são denominadas simplesmente *livres de contexto,* e são reconhecidas por *autômatos de pilha não-determinísticos.* Tais autômatos são caracterizados por funções não-determinísticas de transição de estado, nas quais é permitido, dada uma configuração do reconhecedor, haver duas ou mais produções indicando possíveis diferentes movimentos a partir desta configuração.

Para qualquer linguagem livre de contexto a teoria assegura a existência de reconhecedores do tipo dos autômatos de pilha. Infelizmente, os resultados teóricos obtidos para esta classe de linguagens não são tão abrangentes e definitivos como ocorre para as linguagens regulares, embora grandes progressos tenham sido conseguidos nas pesquisas conhecidas, o que torna também resolvido o problema da análise de linguagens livres de contexto.

Máquinas de Turing – As denominadas máquinas de Turing são reconhecedores em que uma máquina de estados utiliza como memória auxiliar uma fita com possibilidade de leitura e gravação nos dois sentidos do percurso. Não serão formalizadas neste texto as máquinas de Turing, por não serem de interesse imediato no estudo dos compiladores. Entretanto, algumas observações são significativas a respeito destes reconhecedores: Máquinas de Turing podem ser, como os demais reconhecedores, determinísticas ou não-determinísticas. Ambas as classes de máquinas de Turing reconhecem o mesmo conjunto de linguagens, que é a classe das linguagens do tipo 0 da hierarquia de Chomsky. Estas linguagens, também denominadas conjuntos recursivamente enumeráveis, são geradas pelas gramáticas do tipo 0, ou seja, pelas gramáticas irrestritas. Limitando-se a potência das máquinas de Turing, através da imposição de uma limitação nas dimensões da sua fita de trabalho, obtém-se uma classe de reconhecedores denominados *Máquinas de Turing linearmente limitadas em espaço.* Não se sabe ao certo se o fato de tais máquinas serem determinísticas ou não altera a classe de linguagens que são capazes de reconhecer. No entanto, sabe-se que todas as linguagens do tipo 1, ou linguagens sensíveis ao contexto, são aceitas por máquinas de Turing linearmente limitadas em espaço. Correspondentemente, tais linguagens são geradas por gramáticas sensíveis ao contexto, ou do tipo 1.

2.5 – GRAMÁTICAS DE TRANSDUÇÃO

As gramáticas e os reconhecedores são, como foi visto, mecanismos através dos quais são definidas as linguagens de programação. No estudo dos compiladores, um aspecto importante,

não considerado por tais mecanismos, corresponde ao da especificação formal de um mapeamento da linguagem definida para uma outra forma sintática. Isto pode ser feito com base na gramática que formaliza a linguagem, obtendo-se neste caso as chamadas *gramáticas de transdução*. A gramática em que se apóia uma gramática de transdução é chamada *gramática básica*, e corresponde aos dispositivos geradores já descritos. As gramáticas de transdução apresentam-se com um formalismo semelhante, diferindo apenas por apresentar um *alfabeto de saída*, e também no formato das suas produções, as quais associam, às regras de substituição que compõem as produções da gramática básica, uma cadeia de elementos do alfabeto de saída, correspondente à saída gerada pelo transdutor quando da aplicação da regra em questão.

Formalmente, pode-se definir uma gramática de transdução G associada à gramática básica $G' = (V, \Sigma, P, S)$, como sendo a quíntupla $G = (V, \Sigma, \Lambda, P', S)$, onde

V, Σ, P e S formam as mesmas construções da gramática básica.

Λ é o *alfabeto de saída* da gramática de transdução.

P' é um conjunto dos pares (p_i, λ_i) para todo $p_i \in P$ onde p_i é uma regra de substituição e

$\lambda_i \in \Lambda^*$ é a cadeia de saída que deve ser gerada quando a produção p_i for aplicada.

Pode-se dizer que G mapeia a linguagem básica $\pounds(G) = \pounds(G')$, gerada pela gramática básica G', em uma linguagem de saída $\pounds'(G)$, cujas cadeias são formadas por elementos do alfabeto de saída Λ, sendo que $\pounds'(G)$ é o conjunto de todas as cadeias geradas por G para as sentenças de $\pounds(G)$.

Sendo $\omega \in \pounds(G) \subseteq \Sigma^*$, e $q_1, q_2, \ldots, q_n$ a seqüência dos índices das produções $p'_i = (p_i, \lambda_i)$ utilizadas para a geração de ω por G', então a cadeia $\lambda = \lambda_{q_1} \lambda_{q_2} \ldots \lambda_{q_n} \in \Lambda^*$ corresponderá à transdução de ω, levada a efeito pela gramática de transdução G.

Assim, $\pounds'(G) = \{\lambda = \lambda_{q_1} \ldots \lambda_{q_n} \in \Lambda^* \mid \omega \in \pounds(G')$ é derivável a partir de S pela aplicação da seqüência de produções $p_{q_1} \ldots p_{q_n} \in P^*\}$.

Exemplo:

$$G = (\{S, a, b\}, \{a, b\}, \{a, c\},$$
$$\{(S \to a, a),$$
$$(S \to b, c),$$
$$(S \to Sa, a),$$
$$(S \to Sb, c)\}, S)$$

Esta gramática traduz cadeias formadas pelos símbolos a e b, mantendo as ocorrências de a e substituindo b por c.

Sua gramática básica é:

$$G' = (\{S, a, b\}, \{a, b\}, \{S \to a, S \to b, S \to Sa, S \to Sb\}, S)$$

As gramáticas de transdução, fortemente vinculadas às gramáticas básicas correspondentes, definem a mesma linguagem que estas, de modo que a generalidade quanto ao tipo de linguagens tradutíveis desta maneira é exatamente a mesma que a das gramáticas, ou seja, restrições quanto

à classe de linguagens tratáveis só dependem do formato permissível para as produções da gramática básica.

2.6 – TRANSDUTORES

Uma relação análoga à existente entre as gramáticas usuais e as gramáticas de transdução ocorre entre os reconhecedores e os denominadores *transdutores*. Trata-se de reconhecedores usuais, enriquecidos com um alfabeto de saída e com uma função de geração de saídas. A tais reconhecedores dá-se o título de *reconhecedor básico* do transdutor. Os transdutores podem, como as gramáticas, representar as linguagens dos diversos tipos descritos anteriormente. Conforme a classe de linguagens a que se propõem, os transdutores são formados com base em reconhecedores da classe correspondente. Às transições do reconhecedor são associadas cadeias de saída a serem geradas sempre que tais transições forem executadas. Desta maneira, enquanto o reconhecedor básico consome átomos da cadeia de entrada, uma cadeia de saída vai sendo simultaneamente construída, e, ao final do reconhecimento da cadeia de entrada, é completada a geração do texto de saída, que representa sua transdução, efetuada segundo as regras impostas pelo transdutor. A classe de transdutores mais usualmente encontrada no estudo das técnicas de construção de compiladores corresponde aos *transdutores finitos*, cujos reconhecedores básicos são autômatos finitos. Transdutores para linguagens livres de contexto ou mesmo para outras classes mais complexas comportam-se de maneira similar e não serão apresentados neste texto.

Um transdutor seqüencial T baseado no autômato finito

$$A = (Q, \Sigma, \delta, q_0, F)$$

pode ser formalizado como

$$T = (Q, \Sigma, \Lambda, \delta', q_0, F)$$

onde os elementos comuns a A e T possuem a mesma conotação, sendo introduzidos:

Λ – alfabeto de saída do transdutor

δ – é um mapeamento do produto cartesiano $Q \times (\Sigma \cup \{\epsilon\})$ no produto cartesiano $Q \times \Lambda^*$, ou seja, δ' é uma função que, estando o transdutor T no estado $q \in Q$, e sendo $\sigma \in \Sigma$ o próximo átomo a ser consumido da cadeia, promove a mudança do estado do transdutor T para $q' \in Q$, emitindo como saída uma cadeia $\lambda \in \Lambda^*$.

Através de um transdutor, a linguagem básica $\pounds(A) = \pounds(T)$ reconhecida pelo autômato A é mapeada na *linguagem de saída* $\pounds'(T)$, cujas cadeias são compostas de elementos do alfabeto de saída Λ, sendo $\pounds'(T)$ o conjunto de todas as cadeias geradas por T a partir de sentenças de $\pounds(T)$.

Sendo $\omega \in \pounds(T) \subseteq \Sigma^*$ e $\omega = \omega_1 \omega_2 \ldots \omega_n$, $\omega_j \in \Sigma$, se $\delta'(\bar{q}, \bar{\omega}) = (\bar{q}', \bar{\lambda})$ para $\bar{q}, \bar{q}' \in Q$, $\bar{\omega} \in (\Sigma \cup \{\epsilon\})$ e $\bar{\lambda} \in \Lambda^*$ então

$$\delta'(q_0, \omega_1) = (\bar{q}_1, \bar{\lambda}_1)$$

$$\delta'(\bar{q}_1, \omega_2) = (\bar{q}_2, \bar{\lambda}_2)$$

$$\delta'(\bar{q}_{n-1}, \omega_n) = (\bar{q}_n, \bar{\lambda}_n) \text{ com } \bar{q}_n \in F$$

logo a cadeia obtida a apartir de ω pelo transdutor T será

$$\overline{\lambda}_1 \overline{\lambda}_2 \ldots \overline{\lambda}_n \in \Lambda^*$$

Deste modo, $£'(T) = \{\lambda = \overline{\lambda}_1 \overline{\lambda}_2 \ldots \overline{\lambda}_n \in \Lambda^* | \ \omega \in £(A)$ é reconhecida por A através da aplicação da seqüência dos movimentos indicados acima$\}$.

Exemplo:

$$T = (\{A, B\}, \{0, 1\}, \{1\}, \{(A, 0) \rightarrow (A, 11), (A, 1) \rightarrow (B, 1), (B, 1) \rightarrow (B, 1), (B, 0) \rightarrow (A, 11)\}, A, \{B\})$$

Este transdutor finito, baseado no autômato finito visto anteriormente

$$M = (\{A, B\}, \{0, 1\}, \{(A, 0) \rightarrow A, (A, 1) \rightarrow B, (B, 1) \rightarrow B, (B, 0) \rightarrow A\}, A, \{B\})$$

traduz cadeias binárias, iniciadas por um dígito e terminadas por um dígito 1, em cadeias binárias de dígitos 1, procedendo à substituição de cada 0 da cadeia original por 11, e preservando os dígitos 1 originais.

Os transdutores, baseados nos correspondentes reconhecedores básicos, são capazes de definir e reconhecer a linguagem por estes definida, sendo a forma dos reconhecedores básicos utilizados a responsável pela determinação da classe de linguagens tratáveis pelo transdutor em questão.

Um importante resultado teórico é fornecido pelo *teorema do transdutor seqüencial*: "Se L é uma linguagem livre de contexto ou regular, então, sendo S um transdutor seqüencial e $S(L)$ sua linguagem de saída (ou seja, o conjunto das cadeias geradas por S para todas as sentenças de L), então $S(L)$ será também livre de contexto ou regular, respectivamente". Em outras palavras, a ação de um transdutor finito sobre uma linguagem qualquer gera sempre uma linguagem de complexidade não-maior que a da linguagem original.

2.7 – UMA NOTAÇÃO ESTRUTURADA PARA OS AUTÔMATOS DE PILHA

O formalismo estudado para os autômatos de pilha é extensivamente utilizado em textos teóricos sobre linguagens formais, e é através dele que foram estudadas as propriedades das linguagens livres de contexto e de suas implementações, na literatura. Porém, o mapeamento de uma gramática livre de contexto para a forma de um reconhecedor baseado em tais autômatos não é direta. Neste livro, será apresentado adiante um método de construção de reconhecedores, através do mapeamento direto de uma gramática para uma forma equivalente, dada em produções de um autômato de pilha, expresso através de um formalismo ligeiramente modificado. É possível provar que qualquer autômato de pilha denotado convencionalmente pode ser representado através desta notação alternativa, e vice-versa, o que demonstra a equipotência das duas notações, validando a utilização desta última em aplicações à construção de reconhecedores sintáticos, para linguagens livres de contexto, de um modo geral. Estuda-se a seguir, um pouco mais detalhadamente, esta notação.

Um autômato de pilha pode ser representado através de uma abstração da forma:

$$M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$$

onde A é um conjunto de *submáquinas* a_i, tais que:

$$a_i = (Q_i, \Sigma_i, P_i, E_i, S_i)$$

onde:

$Q_i \subseteq Q$ é o conjunto dos *estados* q_{ij} da submáquina a_i.

$\Sigma_i \subseteq \Sigma$ é o conjunto dos *símbolos de entrada* da submáquina a_i.

$E_i \subseteq Q_i$ contém o (único) *estado de entrada* da submáquina a_i ($E_i = \{q_{io}\}$).

$S_i \subseteq Q_i$ o conjunto dos *estados de saída* da submáquina a_i.

$P_i \subseteq P$ é o *conjunto de produções* da submáquina a_i, cujos elementos assumem uma das três formas seguintes:

(.) $\gamma\, q_{ik}\, \beta \rightarrow \gamma\, q_{ij}\, \alpha$, representando *transições internas* à submáquina a_i.

(..) $\gamma\, q_{ik}\, \alpha \rightarrow \gamma\, (i, j)\, q_{mo}\, \alpha$, representando *transações de chamada* da submáquina a_m, através do seu estado de entrada q_{mo}. O estado de retorno será q_{ij}, pois esta informação fica armazenada na pilha após a aplicação de produções deste tipo.

(...) $\gamma\, (m, n)\, q_{ik}\, \alpha \rightarrow \gamma\, q_{mn}\, \alpha$, representando *transições de retorno* para o estado indicado na pilha, ou seja, q_{mn}. Esta produção encerra a operação da submáquina a_i, retornando para a submáquina a_m no seu estado q_{mn}. O estado q_{ik} é um estado de saída da submáquina a_i.

Nestas produções, valem as seguintes convenções:

– $\gamma \in \Gamma^*$ representa o conteúdo da pilha. Na notação, o valor assumido por γ é irrelevante, ou seja, o conteúdo da pilha não é visível quando denotado por γ.

– $(m, n) \in \Gamma$ representa o antigo conteúdo do topo da pilha, antes da aplicação de uma produção indicativa da transição de retorno. Este par de números simboliza um estado de retorno q_{mn} da submáquina a_m, que tenha sido previamente empilhado por uma transição de chamada de submáquina.

– $(i, j) \in \Gamma$ representa a alteração da pilha, ou seja, o elemento empilhado por uma transição de chamada de submáquina, executada pela submáquina a_i. O estado de retorno deverá ser q_{ij}, e o retorno ocorrerá quando for executada, na submáquina chamada, a transição de retorno correspondente.

– $q_{xy} \in Q_x$ representa um estado qualquer. O índice x relaciona-se com a submáquina a_x a que o estado pertence, e o índice y indica a qual dos estados da submáquina q_{xy} se refere.

- $\beta \in \Sigma^*$ representa a situação da cadeia de entrada antes da aplicação da produção. β pode assumir uma das duas formas seguintes:

  $\sigma\alpha$ – explicitando $\sigma \in \Sigma$ como o próximo átomo da cadeia

  α – denotando uma cadeia qualquer $\alpha \in \Sigma^*$, irrelevante à análise.

- $\sigma \in \Sigma$ explicita o símbolo de entrada a ser consumido pela aplicação da produção.

As demais componentes da óctupla que denota o autômato têm as seguintes conotações:

Q – é o conjunto de todos os estados do autômato M. É o conjunto união de todos os conjuntos Q_i dos estados das submáquinas a_i.

$$Q = \{q_{ij} \in Q_i \mid a_i \in A\}$$

Σ – é o alfabeto de entrada do autômato M. É o conjunto união de todos os conjuntos Σ_i dos átomos de entrada das submáquinas a_i.

$$\Sigma = \{\sigma \in \Sigma_i \mid a_i \in A\}$$

Γ – é o *alfabeto de pilha* do autômato M. Inclui todos os possíveis símbolos empilháveis de M. No caso, são empilháveis todos os estados de retorno das submáquinas a_i:

$$\Gamma = \{Z_0\} \cup \{\gamma_{ij} = (i, j) \mid \gamma\, q_{ik}\, \alpha \to \gamma\, (i, j)\, q_{mn}\, \alpha \in P_i,\ a_i \in A\}$$

P – é o conjunto de produções do autômato M, e corresponde à união de todos os conjuntos P_i de produções das submáquinas a_i:

$$P = \{p \in P_i \mid a_i \in A\}$$

Z_0 – é o *símbolo inicial* de pilha do autômato M ($Z_0 \in \Gamma$)

q_0 – simboliza o *estado inicial* do autômato M. O estado q_0 é certamente o estado inicial de uma submáquina de M, dita a *submáquina inicial* de M, e denotada como a_0. Logo, $q_0 \in Q_0$, e $q_0 \in E_0$. Por convenção, este estado é denotado sempre como q_{00}, o que é consistente com a notação adotada para os nomes dos estados das submáquinas:

$$q_0 = q_{00} \in E_0 \subseteq Q_0$$

F – é o conjunto de *estados finais* do autômato M, e corresponde a um subconjunto dos estados finais da submáquina inicial de M:

$$F \subseteq \{q_{0j} \in S_0 \mid a_0 \in A\}$$

Exemplo:

$$M = (\{A, B, C, D\}, \{S\}, \{a, b\}, \{x, y\},$$

$$\begin{aligned}
\{ &(A, a, x) \rightarrow (D, x), \\
&(A, a, y) \rightarrow (D, y), \\
&(A, b, x) \rightarrow (B, x), \\
&(A, b, y) \rightarrow (B, y), \\
&(B, \epsilon, x) \rightarrow (A, xx), \\
&(B, \epsilon, y) \rightarrow (A, yx), \\
&(C, b, x) \rightarrow (D, x), \\
&(C, b, y) \rightarrow (D, y), \\
&(D, \epsilon, xx) \rightarrow (C, x), \\
&(D, \epsilon, yx) \rightarrow (C, y)\},
\end{aligned}$$

$$y, A, \{D\})$$

Este autômato reconhece cadeias do tipo b^n a b^n, com uma unica submáquina S auto-recursiva. As produções foram denotadas na forma convencional de autômatos de pilha. Na notação apresentada para esta forma de autômato de pilha, o conjunto de produções pode ser simplificado para:

$$\begin{aligned}
\{ &\gamma A a \alpha \rightarrow \gamma D\alpha, \\
&\gamma A b \alpha \rightarrow \gamma B\alpha, \\
&\gamma C b \alpha \rightarrow \gamma D\alpha, \\
&\gamma (S, C) D\alpha \rightarrow \gamma C\alpha, \\
&\gamma B \alpha \rightarrow \gamma (S, C) A\alpha\}
\end{aligned}$$

Note-se que (S, C) corresponde ao símbolo x de pilha na notação anterior, explicitando C como estado de retorno da submáquina S.

O funcionamento deste modelo de autômato de pilha segue o padrão usualmente encontrado em outras formas de representação:

Definindo-se t como sendo uma *situação do autômato M*, pode-se representá-lo como: $t = (\gamma, q, \alpha \Psi)$ onde

$\gamma \in \Gamma^*$ representa o conteúdo da pilha.

$q \in Q$ representa o estado de M,

$\alpha \in \Sigma^*$ representa a parte da cadeia de entrada ainda não analisada,

$\Psi \notin \Sigma$ representa uma marca especial de fim de texto de entrada.

Define-se a *situação inicial* do autômato M como

$$t_0 = (Z_0, q_0, \alpha_0 \Psi)$$

onde

$Z_0 \in \Gamma$ indica que a pilha está vazia

$q_0 \in Q_0 \subseteq Q$ indica que M está em seu estado inicial

$\alpha_0 \in \Sigma^*$ é a cadeia de entrada completa, a ser analisada

$\Psi \notin \Sigma$ é o marcador de final de entrada

Denota-se uma transição de M, de uma situação t_i para outra situação t_{i+1}, devida à aplicação de uma produção $p \in P$, como

$$t_i \Rightarrow t_{i+1}$$

Uma sucessão de transições que leva M da situação t_i para a situação t_j, pode ser abreviada como

$$t_i \Rightarrow^* t_j$$

Diz-se que o autômato M reconhece uma cadeia de entrada $\alpha_0 \in \Sigma^*$ se, e somente se, partindo-se da situação inicial t_0 de M, for possível, através da aplicação sucessiva de produções $p \in P$, levar M a uma situação final t_n dada por:

$$t_n = (Z_0, q_F, \Psi), \text{ em que}$$

Z_0 indica novamente a ausência de dados na pilha

q_F indica que M se encontra em um estado final $q_F \in F$

Ψ indica que a cadeia de entrada foi esgotada

Assim, M reconhece α_0 se e somente se:

$$t_0 = (Z_0, q_0, \alpha_0\Psi) \Rightarrow^* t_n = (Z_0, q_F, \Psi)$$

Demonstra-se que:

(a) Um autômato de pilha é capaz de simular este autômato, no seu caso geral
(b) Este autômato é capaz de simular um autômato de pilha, no seu caso geral
(c) As duas afirmações anteriores implicam na equipotência dos dois modelos
(d) Conseqüentemente, o autômato proposto é capaz de implementar reconhecedores para qualquer linguagem livre de contexto.

Note-se que, devido a tais propriedades, a aplicabilidade do modelo é grande, como será verificado adiante.

Pela sua estruturação, os estados deste autômato ficam agrupados funcionalmente, dando a possibilidade de se construirem submáquinas a_i coesas. As submáquinas a_i podem operar como autômatos finitos durante todo o tempo em que executam apenas transições internas, uma vez que o conteúdo da pilha é irrelevante nestas situações, e não se altera nestas transições.

Assim sendo, a pilha só é utilizada quando da transição entre submáquinas, o que pode ser reduzido ao mínimo através de um projeto criterioso. Desta forma, reconhecedores sintáticos extremamente eficientes podem ser obtidos se o uso de pilha for restringido apenas aos casos em que seja realmente necessário utilizar submáquinas e suas chamadas. Isto é uma tarefa relativamente simples, como será visto adiante.

A intuição do funcionamento do modelo apresentado para os autômatos de pilha é estudada em seguida. É de extrema importância assimilá-la, já que os reconhecedores sintáticos a serem construídos baseiam-se nos autômatos de pilha que o modelo representa.

Como se sabe, os autômatos finitos são capazes de reconhecer qualquer linguagem regular. Para linguagens livres de contexto, porém, os autômatos finitos falham sempre que a linguagem apresentar as chamadas *construções auto-recursivas centrais,* em que um não-terminal X pode derivar uma cadeia de tipo $\alpha \; X \; \beta$, com α e β não vazios. Deve-se observar, porém, que nem todos os não-terminais da gramática são obrigatoriamente auto-recursivos centrais. Para o reconhecimento dos não-terminais cuja definição seja regular, a pilha do autômato não precisa ser utilizada, pois autômatos finitos não precisam de memória auxiliar, e portanto podem ser simulados, no modelo proposto, através de submáquinas definidas por um conjunto de produções que denotam apenas transições internas à submáquina.

As construções auto-recursivas centrais podem, por sua vez, ser decompostas em três partes: uma parte não obrigatoriamente auto-recursiva, que consta à esquerda da auto-recursão central propriamente dita, e uma outra parte, também não obrigatoriamente auto-recursiva, que figura à direita da auto-recursão. Assim, em $X \Rightarrow^* \alpha \; X \; \beta$ tem-se: α e β não obrigatoriamente auto-recursivos em X, e X, a auto-recursão propriamente dita.

Para tal tipo de construções, o reconhecimento pode ser efetuado analisando-se inicialmente a parte esquerda α, suspendendo-se temporariamente a análise até que uma construção central completa X seja totalmente analisada, e retomando-se, finalmente, a análise através do reconhecimento da parte direita β.

Deve-se notar que, ao ser analisada a construção X central, novas instâncias de α e de β surgem eventualmente. A pilha do autômato exerce neste caso um papel decisivo no reconhecimento desta classe de construções, pois viabiliza a memorização, por parte do autômato, do estado em que se encontra o reconhecedor global do texto de entrada.

Observe-se que, tratando-se desta maneira o reconhecimento deste tipo de construções, tudo se passa como se todo o texto de entrada, que implementa a construção representada por X como recursão central, fosse invisível ao analisador na instância corrente da análise, sendo o reconhecimento, da cadeia representada por X, efetuado em uma nova instância da análise.

A pilha do modelo é utilizada exatamente para efetuar este trabalho: iniciado um reconhecimento, um estado de retorno é empilhado, e a submáquina que se encarrega do reconhecimento da construção auto-recursiva central é ativada a partir de seu estado inicial. Terminado o reconhecimento desta construção, o movimento contrário é executado pelo autômato, retornando-se à instância anterior do reconhecimento, no estado indicado pela informação empilhada anteriormente (estado de retorno).

Ao movimento de chamada de submáquina, acompanhado pelo empilhamento do estado de retorno, seguido do reconhecimento completo de uma parcela da cadeia de entrada corrente, atingindo-se um estado final da submáquina, e do retorno ao estado anteriormente empilhado, dá-se o nome de *transição com submáquina.* Nos diagramas de estados, estas transições complexas podem ser identificadas na forma de transições com os não-terminais que a submáquina em questão implementa.

Note-se que é possível criar reconhecedores em que são identificadas transições com submáquinas que não sejam obrigatoriamente correspondentes a auto-recursões centrais. Nestes casos, existe a possibilidade de tais transições serem eliminadas, através da incorporação, na submáquina chamadora, das transições da submáquina chamada. Em contraste, as auto-recursões centrais exigem chamadas de submáquinas, não podendo ser eliminadas em reconhecedores de linguagens livres de contextos gerais.

2.8 – NOTAÇÕES PARA A DEFINIÇÃO DE LINGUAGENS

As linguagens podem ser representadas, como foi visto, através de dois mecanismos básicos: dispositivos geradores, ou gramáticas, e dispositivos aceitadores, ou reconhecedores, também conhecidos como autômatos. Para ambos os mecanismos existem inúmeras formas através das quais a representação pode ser efetuada. A tais notações dá-se o nome de *metalinguagens*, já que elas próprias são linguagens, através das quais as linguagens são especificadas. Como, em geral, as linguagens de interesse são conjuntos infinitos, torna-se conveniente que, através da escolha de uma metalinguagem adequada, seja possível descrevê-las de um modo compacto, através de um número finito de regras, expressas com clareza e legibilidade suficientes.

Nesta seção é feita, de modo conciso, uma apresentação de algumas das formas mais utilizadas na definição de linguagens.

Gramáticas Lineares à Direita (Esquerda) – Esta forma de representação é em geral implementada através de metalinguagens tais como o BNF, a notação de Wirth ou alguma outra forma de expressão de gramáticas. Como tais metalinguagens são de alcance mais geral, destinadas à definição de linguagens livres de contexto, pode-se por meio delas representar também as *linguagens regulares*, através da restrição, definida anteriormente, para o formato de suas produções. Esta classe de representações é um dispositivo de geração para linguagens do tipo 3.

Exemplo:

$$G = (\{A, B, C\}, \{a, b, c\}, \{A \rightarrow Ba, A \rightarrow Cb, A \rightarrow Ac, B \rightarrow Cc, B \rightarrow Ba, C \rightarrow Cb, C \rightarrow c\}, A)$$

Expressões Regulares – Esta é outra maneira muito difundida, especialmente em textos mais teóricos, para a representação de linguagens. As expressões regulares correspondem a formas gerais das sentenças das linguagens que representam, as quais são expressas através do uso exclusivo dos terminais (átomos ou símbolos) da linguagem, sem o recurso da utilização de não-terminais.

Pode-se definir recursivamente esta notação da seguinte maneira:

- um terminal x forma uma expressão regular, e representa o conjunto que contém apenas a cadeia formada pelo terminal x isolado.
- o símbolo ϵ forma uma expressão regular, e representa o conjunto que contém apenas a cadeia vazia.
- dadas duas expressões regulares a e b, tem-se:

(a) a concatenação ab é uma expressão regular e representa o conjunto cujos elementos são formados pela concatenação de todos os pares ordenados obtidos pelo produto cartesiano dos conjuntos que as expressões regulares a e b representam, respectivamente.

(b) a alternância $a | b$ é uma expressão regular, e representa o conjunto união dos conjuntos representados pelas expressões regulares a e b.

(c) o fechamento recursivo e transitivo $(a)^*$ é uma expressão regular (podendo, caso não haja ambigüidade, ser denotado como a^*), e representa o conjunto obtido pela união dos conjuntos formados por todas as possíveis concatenações da expressão regular a, e incluindo a cadeia vazia. Em outras palavras, $(a)^*$ representa o conjunto dado pela expressão regular $\epsilon \mid a \mid aa \mid aaa \mid \ldots$.

(d) o fechamento transitivo $(a)^+$, ou a^+, caso não haja ambigüidade, é uma expressão regular, e denota o conjunto obtido eliminando-se a cadeia vazia ϵ do conjunto definido por $(a)^*$ ou a^*.

Exemplos: *São expressões regulares:*

1) $((c^* b\, c)^* aa \mid c^* bb)^* c$
2) $a^+ b \mid c^* \mid \epsilon$

As expressões regulares representam uma notação poderosa aplicável à definição de uma importante classe de linguagens, a das linguagens regulares, sendo por suas propriedades uma boa alternativa, que pode ser utilizada para a definição de linguagens do tipo 3 através de mecanismos de geração. Isto é ainda mais verdadeiro se se levar em consideração a facilidade com que se podem obter reconhecedores a partir de expressões regulares, como será estudado em outra parte deste livro.

"Backus-Naur Form" (BNF) – Talvez a forma mais popular de expressão da sintaxe de linguagens de programação, a forma normal de Backus, ou forma de Backus-Naur é uma metalinguagem que tem sido utilizada com sucesso para a especificação de linguagens de programação, desde que foi publicada pela primeira vez no relatório de especificação da linguagem Algol 60. Trata-se de uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentenças. Para tanto, cada produção corresponde a uma regra de substituição, em que a um símbolo da metalinguagem são associadas uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição. Os símbolos em questão correspondem a não-terminais da gramática que está sendo especificada. As cadeias podem ser formadas de terminais e/ou não-terminais, e do símbolo ϵ, que representa a cadeia vazia. A simbologia adotada é a seguinte:

$<x>$ – representa um não-terminal, cujo nome é dado por uma cadeia x de caracteres quaisquer. Os caracteres $<$ e $>$ são usados para delimitar o nome do não-terminal.

: := – é o símbolo da metalinguagem que associa a um não-terminal um conjunto de cadeias de terminais e/ou não-terminais, incluindo o símbolo da cadeia vazia. O não-terminal em questão é escrito à esquerda deste símbolo, e as diversas cadeias, à sua direita. Lê-se "define-se como".

| – é o símbolo da metalinguagem que separa as diversas cadeias que constam à direita do símbolo : :=. Lê-se "ou".

X – representa um *terminal* da linguagem que está sendo definida, e pertence ao conjunto de todos os átomos que compõem as sentenças da linguagem. Deve ser denotado tal como figura nas sentenças da linguagem, e não entre os caracteres $<$ e $>$, como ocorre no caso da denotação escolhida para os não-terminais.

ϵ – representa a cadeia vazia na notação BNF.

yz – representa uma cadeia construída pela concatenação dos elementos y e z nesta ordem. Estes dois elementos podem, por sua vez, ser símbolos de terminais, de não-terminais, da cadeia vazia, ou mesmo outras cadeias.

Exemplo:

<A> ::= <B> *a* | <C> *b* | <A> *c*
<B> ::= <C> *c* | <B> *a*
<C> ::= <C> *b* | *c*

Através do uso da notação BNF, e restringindo-se o lado esquerdo das produções à forma de um não-terminal isolado, é possível representar, de modo recursivo, qualquer linguagem livre de contexto, sendo uma metalinguagem muito utilizada para a especificação de gramáticas do tipo 2. Note-se que, bastando impor restrições adicionais de linearidade à direita (esquerda) às opções do lado direito da produção BNF, é possível representar, com facilidade, gramáticas do tipo 3. Através da extensão do lado esquerdo das produções à forma de uma cadeia de terminais e/ou não-terminais, que contenha ao menos um não-terminal, é possível ainda impor dependências de contexto na substituição de não-terminal, obtendo-se também desta maneira, recursos para a especificação de gramáticas do tipo 1 e 0, o que mostra a grande versatilidade desta metalinguagem.

Notação de Wirth – Uma outra metalinguagem de uso prático para a especificação de gramáticas é a notação de Wirth, variante da forma normal de Backus (BNF) que procura torná-la mais legível através do oferecimento de mecanismos para a substituição de recursões por iterações. Em alguns textos esta notação é denominada *BNF estendido*, embora na realidade não seja exatamente uma extensão da notação em questão. Tendo em mente a legibilidadde, a notação de Wirth elimina os incômodos símbolos < e > como delimitadores dos nomes dos não-terminais. Em compensação, os terminais aparecem denotados entre aspas. Parênteses são utilizados no agrupamento de opções de cadeias, o sinal ::= do BNF é simplificado para =, e as produções são terminadas pelo símbolo . (ponto).

Assim, nesta metalinguagem, os elementos constituintes das regras tornam-se:

X – simboliza um nome de não-terminal. *X* é uma cadeia qualquer de caracteres que identifica uma classe de construções da linguagem.

"*y*" – simboliza um terminal, *y* é uma cadeia de caracteres que pertence ao conjunto dos símbolos terminais (átomos) que compõem as sentenças da linguagem definida.

[*z*] – simboliza o fato de que *z* é uma construção opcional. Equivale a $z|\epsilon$ onde *z* é um conjunto de opções de cadeias de terminais e/ou não-terminais.

(*z*) – simboliza o mesmo que *z*, ou seja, um conjunto de opções de cadeias de terminais e/ou não-terminais. Esta notação é utilizada para propósito de agrupamento.

{ *z* } – simboliza iterações arbitrárias da construção *z*, a qual representa um conjunto de opções entre cadeias de terminais e/ou não-terminais.

zt – simboliza a concatenação de uma cadeia representada pela construção *z* com uma cadeia representada pela construção *t*, nesta ordem. Tanto *z* como *t* podem ser quaisquer das cinco construções definidas acima, ou um agrupamento, entre parênteses, de tais construções, separadas pelo sinal de alternância se necessário.

z|*t* – simboliza que as cadeias representadas pelas construções *z* e *t* são alternativas válidas no contexto em que aparecem. Tanto *z* como *t* podem ser quaisquer das

seis construções já definidas, ou um agrupamento, entre parênteses ou chaves, de tais construções, separadas por sinais de alternância se necessário.

= — é o símbolo que separa o não-terminal, cuja substituição está sendo definida, do lado direito da produção, que representa o conjunto de opções de cadeias válidas para substituir o não-terminal em questão.

. — é o símbolo delimitador do final de uma regra de substituição.

Note-se que a notação de Wirth cria uma forma mais direta de representação de construções repetitivas, permitindo, através da possibilidade de agrupamento de opções, e concatenação subseqüente de tais grupos, reduzir, às vezes drasticamente, o número de não-terminais necessários à definição da linguagem.

Exemplos:

1) *A* = "*c*" | *B* "*a*" *A* | *C* "*b*" *A*.
B = "*a*" | *C* "*c*" *B*.
C = "*b*" | "*c*" *C*.

2) *O mesmo exemplo, denotado na notação de Wirth sem recursões, torna-se*

A = { *B* "*a*" | *C* "*b*" } "*c*".
B = { *C* "*c*" } "*a*".
C = { "*c*" } "*b*".

3) *X* = ["*a*" { "*a*" } | { "*c*" }].

Como o BNF, a notação de Wirth também é um dispositivo gerador, que permite a especificação de gramáticas do tipo 2 quando o lado esquerdo de suas produções constar de um único não-terminal. Gramáticas dos demais tipos são também representáveis, desde que obedecidas as convenções correspondentes, como foi mencionado no caso do BNF.

Notação usada para a definição de COBOL — Outra metalinguagem, muito semelhante à notação de Wirth, foi utilizada no relatório que define a linguagem COBOL e, posteriormente, na definição da gramática do PL/I. Esta metalinguagem difere da anterior nos seguintes aspectos principais:

— os terminais se distinguem dos não-terminais por serem representados em letras maiúsculas, sublinhados quando obrigatórios, enquanto os não-terminais são denotados em letras minúsculas.

— colchetes são utilizados para agrupar construções obrigatórias alternativas. Cada opção figura em uma linha separada.

— repetições indefinidas são denotadas por elementos seguidos de reticências.

— chaves envolvem construções opcionais.

Não trazendo grandes contribuições conceituais, e sendo apenas uma alternativa adicional de representação, não serão discutidos maiores detalhes desta metalinguagem. Permite representar linguagens do tipo 2, podendo ser adaptada para a definição de linguagens de outros tipos de modo análogo ao que foi discutido anteriormente.

Exemplos:

1) A: $\left\{\begin{matrix} B & \underline{a} \\ C & \underline{b} \end{matrix}\right\} \ldots \underline{c}$

B: $\{C\ \underline{c}\} \ldots \underline{a}$

C: $\{\underline{c}\} \ldots \underline{b}$

2) X: $\left\{\begin{matrix} [\underline{a}] \ldots \\ \{\underline{c}\} \ldots \end{matrix}\right\}$

Expressões Regulares Estendidas – Uma pequena extensão na notação, estudada anteriormente, para expressões regulares, permite ampliar sua capacidade de representação para englobar linguagens do tipo 2. Trata-se de introduzir, na notação, o conceito de não-terminal, cuja finalidade é a de dar um nome ao conjunto denotado através de uma expressão regular estendida. Os não-terminais são também incluídos entre os símbolos elementares que podem compor as expressões regulares estendidas. O sinal de igualdade = associa o não-terminal à expressão regular estendida a que se refere.

Exemplos:

1) $A = (Ba \mid Cb)^* c$

$B = (Cc)^* a$

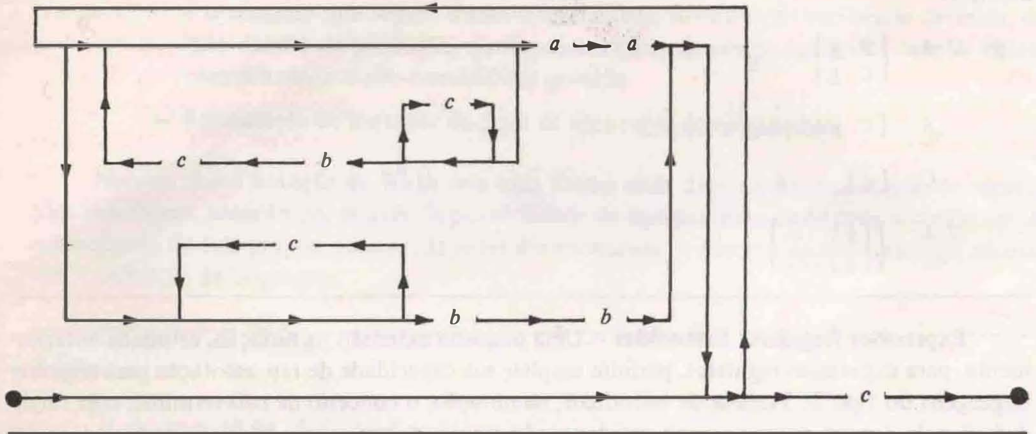
$C = c^* b$

2) $X = aa^* \mid c^* \mid \epsilon$

Diagramas de Sintaxe – Uma notação gramatical que tem ganho muitos adeptos ultimamente, por suas características de legibilidade, obtidas devido ao aspecto gráfico que apresenta, é a dos diagramas de sintaxe, às vezes denominados *notação ferroviária.* Estes diagramas oferecem um meio alternativo de representação para as expressões regulares (estendidas ou não). Caracterizam-se por apresentarem dois pontos diferenciados: o ponto de partida e o de chegada. Estes dois pontos são ligados entre si por um grafo orientado, cujos nós representam pontos onde eventualmente pode haver uma escolha de caminho, e cujos ramos representam caminhos de percurso. Cada ramo pode incluir um terminal (ou não-terminal). As cadeias formadas pelas seqüências de terminais encontrados em qualquer caminho que percorre o grafo desde o seu ponto de partida até o seu ponto de chegada, e somente estas, são sentenças da linguagem definida por estes diagramas.

Diagramas de sintaxe que não utilizam não-terminais são capazes de representar linguagens do tipo 3 apenas. Utilizando-se também de não-terminais, a classe de linguagens representáveis se amplia, englobando também as de tipo 2. A linguagem de programação Pascal, em sua definição, utilizou-se desta notação, e a tornou popular. Trata-se de uma ferramenta muito cômoda para a documentação e o estudo da sintaxe de linguagens de programação, oferecendo possibilidade de obtenção de reconhecedores eficientes a partir da gramática, mediante um esforço muito reduzido.

Exemplo:



Produções Gramaticais – Esta forma de representação deve ser incluída, pelo fato de ser utilizada com freqüência em textos teóricos sobre linguagens. Para aplicações práticas, sua notação é excessivamente pouco compacta, o que reduz relativamente sua inteligibilidade. Essencialmente, apresenta-se em uma forma próxima da notação BNF, com as seguintes diferenças:

– nem sempre é usada uma convenção para distinguir os terminais dos não-terminais.

– a associação da parte esquerda à parte direita de uma produção é em geral feita usando-se o símbolo →.

– uma única alternativa é permitida em cada produção, não existindo sinal de alternância como abreviatura.

Valem para esta notação as observações mencionadas no caso do BNF.

Exemplo:

$A \rightarrow c$ $\quad A \rightarrow B\,a\,A$ $\quad A \rightarrow C\,b\,A$
$B \rightarrow a$ $\quad B \rightarrow C\,c\,B$
$C \rightarrow b$ $\quad C \rightarrow c\,C$

Gramáticas de Dois Níveis – Representadas significativamente pelas *Gramáticas W*, estes dispositivos de geração diferem dos apresentados até aqui em um ponto fundamental: o número de regras gramaticais que as demais notações permitem é finito, enquanto as gramáticas *W* utilizam-se do artifício dos dois níveis para criar um número infinito de produções. Isto é feito através de dois grupos de regras. O primeiro é um conjunto finito de regras livres de contexto através das quais as regras de um segundo conjunto de produções são geradas. Este segundo conjunto pode ser infinito, e é ele o responsável pela geração das sentenças da linguagem que se deseja definir. O formalismo destas gramáticas foge ao escopo desta publicação, podendo ser encontrado na literatura. Cabe mencionar que através das gramáticas *W*, devidas a van Wijngaarden, e utilizadas na definição da linguagem Algol 68, é possível representar formalmente de maneira rigorosa as dependências de contexto inerentes às linguagens usuais de programação, criando

uma ferramenta poderosa para a descrição da sintaxe e da chamada "semântica estática" das linguagens. A semântica estática caracteriza aspectos de dependência de contexto relativos à utilização de identificadores, tais como o uso coerente dos mesmos nas diversas construções da linguagem, as regras de escopo a que estão sujeitos, à verificação do número e do tipo de parâmetros de procedimentos, e muitos outros aspectos não tratáveis de maneira econômica em notações gramaticais livres de contexto. É importante salientar o fato de que as gramáticas *W* não são as únicas gramáticas de dois níveis conhecidas, sendo as *gramáticas de atributos* um exemplo de outra classe de tais gramáticas, de uso corrente no estudo da semântica de linguagens, bem como na especificação formal de transdutores sintáticos, núcleos de compiladores gerados automaticamente, o que também não será detalhado neste trabalho.

Como foi mencionado, as gramáticas *W* apresentam potencial para a representação de linguagens do tipo 1, para cuja definição foram projetadas. Obviamente, linguagens mais simples, dos tipos 2 e 3, são facilmente definidas através deste tipo de metalinguagem.

Até este ponto foram apresentadas algumas notações gramaticais. Há um conjunto também significativo de notações para a representação de reconhecedores, como será exemplificado a seguir.

Diagramas de Estados – São notações gráficas, semelhantes aos diagramas de sintaxe. Enquanto estes dão ênfase aos terminais que compõem as sentenças cuja forma geral representam, os diagramas de estado dão ênfase aos estados internos do autômato que representam, indicando em que situações ocorrem mudanças no estado do mesmo, e o correspondente consumo de átomos da cadeia de entrada. Trata-se, portanto, de uma forma de representação gráfica para reconhecedores baseados em máquinas de estados finitos.

Os diagramas de estados possuem um *estado inicial*, a partir de onde iniciam qualquer reconhecimento. Um subconjunto dos seus estados é formado por estados denominados *estados finais* ou *estados de aceitação*. Qualquer sentença da linguagem representada pelo autômato denotado através dos diagramas de estados, e nada além delas, leva o autômato a um destes estados após o seu reconhecimento. Arcos orientados interligam os estados, e a cada um é possível associar-se um átomo a ser consumido sempre que o autômato executa a transição que tais arcos representam. O reconhecimento é feito partindo-se do estado inicial, e promovendo-se sucessivas transições entre os estados do autômato, baseadas no consumo de átomos retirados seqüencialmente da cadeia de entrada. Ao ser esgotada a cadeia de entrada, se o estado atingido pertencer ao conjunto dos estados de aceitação, e somente nesta situação, diz-se que a cadeia de entrada é uma sentença da linguagem.

A notação utilizada para representar autômatos finitos através de diagramas de estados é a seguinte:

- círculos denotam estados.
- estados representados por dois círculos concêntricos representam estados finais.
- o estado inicial é aquele (único) para onde se dirige um arco que não provém de nenhum estado.
- um arco orientado, ligando o estado X ao estado Y, nesta ordem, e rotulado com o terminal Z, indica uma transição do estado X para o estado Y, com o consumo simultâneo do terminal Z.
- arcos sem rótulo denotam transições em vazio, isto é, sem consumo de terminais. Às vezes o rótulo ϵ, representando a cadeia vazia, é utilizado neste caso.

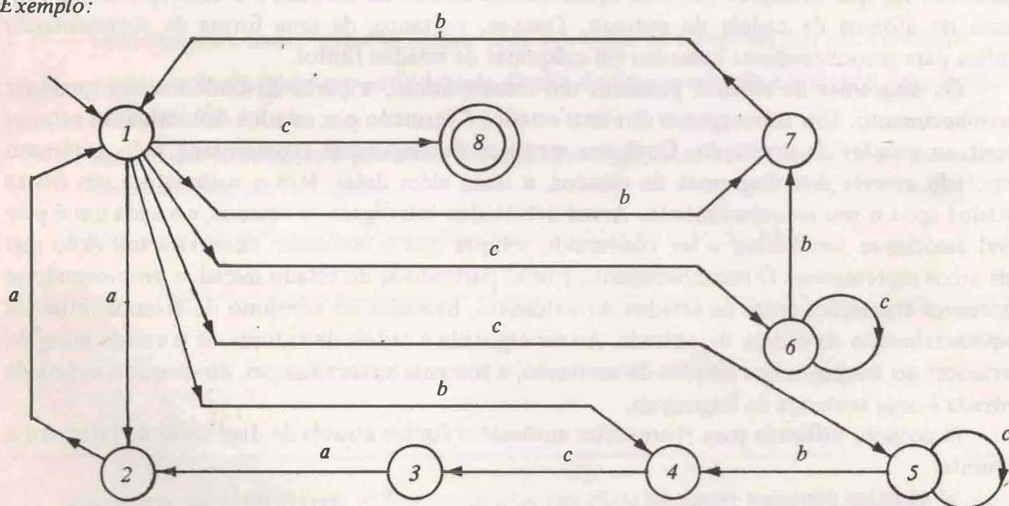
Com esta notação é possível representar qualquer autômato finito, e, conseqüentemente, qualquer linguagem do tipo 3.

Mediante uma pequena extensão, a classe de linguagens representáveis pelos diagramas de estados pode ser ampliada para as do tipo 2. Basta para isto que seja possível utilizar os seguintes recursos adicionais:

- um nome pode ser associado a um diagrama. Este nome passa a ser interpretado como símbolo não-terminal que denota qualquer cadeia que o diagrama representa.
- uma linguagem é representada não por um diagrama apenas, mas por um conjunto de diagramas de estado.
- arcos orientados, rotulados com o nome de um diagrama, representam uma transição composta, em que é totalmente consumida a maior cadeia que o diagrama, cujo nome foi utilizado, seja capaz de reconhecer. Dá-se, às vezes, o nome de *transição com não-terminal* a este tipo de abreviatura.

A extensão mencionada dá à notação a potência suficiente para representar qualquer linguagem livre de contexto, ou seja, do tipo 2, e tem uma grande utilidade na definição da sintaxe desta classe de linguagens.

Exemplo:



Tabelas de Transição – São formas alternativas de notação para os autômatos (reconhecedores sintáticos), mais adequadas para uso pelo computador. Trata-se de notações tabulares, com o mesmo poder de expressão dos diagramas de estados. Nesta notação, cada linha da tabela refere-se a um estado, e cada coluna, a um terminal ou não-terminal.

Um cabeçalho associa a cada coluna da tabela o elemento (terminal ou não-terminal) a que se refere. No caso geral, uma coluna adicional pode ser utilizada para transições em vazio. Uma coluna especial é empregada, em muitas implementações, para designar qualquer dos terminais não especificados no cabeçalho. Na célula correspondente a um estado e um terminal

ou não-terminal, em geral figura um código indicativo do estado seguinte para onde o autômato deve movimentar-se. Para autômatos não-determinísticos, este estado pode não ser único, e, neste caso, a célula codificará o conjunto dos possíveis estados seguintes para aquela célula.

Nas linhas correspondentes a estados não finais, células em branco caracterizam transições impossíveis, e portanto a detecção de um erro na cadeia de entrada. No caso de células em branco, relativas às linhas correspondentes a estados finais, isto é indicativo de que a cadeia até então consumida está correta e forma uma sentença, e o átomo correntemente analisado é considerado um delimitador, não sendo consumido pela análise.

Exemplo:

	a	*b*	*c*
→ *1*	*2*	*4, 7*	*5, 6, 8*
2	*1*	–	–
3	*2*	–	–
4	–	–	*3*
5	–	*4*	*5*
6	–	*7*	*6*
7	–	*1*	–
(*8*)	–	–	–

Como os diagramas de estados são mapeáveis em tabelas de transição e vice-versa, ambos representam a mesma abstração, que é o reconhecedor que denotam. Assim, têm a capacidade de representar linguagens dos tipos 2 e 3, conforme foi concluído anteriormente.

Produções – São notações algébricas, utilizadas principalmente em textos teóricos sobre linguagens, para a representação de reconhecedores.

Seu formato é variável de autor para autor, na literatura, porém constituem uma ferramenta que mapeia diretamente os conceitos matemáticos ligados à idéia dos reconhecedores. Assim, cada produção é uma expressão que associa uma configuração possível do autômato, indicando uma transição que este pode realizar neste caso. A cada transição está associada uma produção, onde ficam explicitados o estado atual, a entrada corrente, a situação da memória auxiliar, o estado seguinte, a nova situação da memória auxiliar e a nova situação de reconhecimento da entrada corrente.

Para o caso de autômatos finitos, não é utilizada nenhuma memória auxiliar. Neste caso, são representáveis linguagens do tipo 3 através destas produções.

Exemplo:

(1, a) → 2	*(1, c) → 8*	*(5, c) → 5*
(1, b) → 4	*(2, a) → 1*	*(6, b) → 7*
(1, b) → 7	*(3, a) → 2*	*(6, c) → 6*
(1, c) → 5	*(4, c) → 3*	*(7, b) → 1*
(1, c) → 6	*(5, b) → 4*	

Para a representação de autômatos de pilha, a memória auxiliar utilizada tem uma organização de pilha, sendo indicado o conteúdo corrente do seu topo e o novo conteúdo, após a aplicação da transição indicada. São representáveis neste caso as linguagens do tipo 2.

Linguagens dos tipos 1 e 0 são também representáveis através das produções que representam suas máquinas de Turing.

2.9 – ALGUMAS PROPRIEDADES DAS GRAMÁTICAS LIVRES DE CONTEXTO

Nesta seção, são apresentados, embora sem demonstração, alguns dos principais resultados da teoria das linguagens livres de contexto, que são úteis para o desenvolvimento de reconhecedores e algoritmos ligados à construção de compiladores. Muita teoria existe a respeito deste assunto. Não se pretende cobri-la, mas fornecer apenas algumas das inúmeras informações que são mais significativas ao objetivo deste texto.

O primeiro destes resultados refere-se ao caráter localizado das estruturas geradas por gramáticas livres de contexto. Sendo $G = (V, \Sigma, P, S)$ uma gramática livre de contexto, e sendo α e β formas sentenciais tais que $S \Rightarrow^* \alpha \Rightarrow \beta$, então, se α puder ser decomposto em duas cadeias α_1 e α_2 tais que $\alpha = \alpha_1\ \alpha_2$, então é garantida a existência de duas cadeias β_1 e β_2 tais que $\beta = \beta_1\ \beta_2$ e que, ou $\alpha_1 \Rightarrow \beta_1$ com $\alpha_2 = \beta_2$ ou então $\alpha_2 \Rightarrow \beta_2$ com $\alpha_1 = \beta_1$.

Este resultado é básico para as duas generalizações seguintes:

a) Sendo $G = (V, \Sigma, P, S)$ uma gramática livre de contexto, e sendo α e β formas sentenciais tais que $S \Rightarrow^* \alpha \Rightarrow^* \beta$, então, se α puder ser decomposta em duas cadeias α_1 e α_2 tais que $\alpha = \alpha_1\ \alpha_2$, então existem cadeias β_1 e β_2, tais que $\beta = \beta_1\ \beta_2$, com $\alpha_1 \Rightarrow^* \beta_1$ e $\alpha_2 \Rightarrow^* \beta_2$.

b) Sendo $G = (V, \Sigma, P, S)$ uma gramática livre de contexto, e sendo α e β formas sentencias tais que $S \Rightarrow^* \alpha \Rightarrow^* \beta$, então se α puder ser decomposta em n cadeias $\alpha_1, \alpha_2, \ldots, \alpha_n$, tais que $\alpha = \alpha_1\ \alpha_2\ \ldots\ \alpha_n$, então existirá para β uma decomposição $\beta = \beta_1 \beta_2\ \ldots\ \beta_n$ tal que $\alpha_i \Rightarrow^* \beta_i$ para $1 \leqslant i \leqslant n$.

Estas importantes observações acerca das gramáticas livres de contexto não se aplicam a gramáticas mais gerais. O fato de que as gramáticas livres de contexto guardam tais propriedades permite analisar partes do texto de entrada sem que as demais partes sejam sequer conhecidas, o que simplifica dramaticamente a tarefa de reconhecimento para tais casos.

Um segundo resultado, bastante significativo para a análise de linguagens livres de contexto, também é decorrente do resultado anterior. Trata-se do fato de que qualquer derivação, em linguagens livres de contexto, não depende da ordem de aplicação das produções.

Sendo $G = (V, \Sigma, P, S)$ uma gramática livre de contexto, α e β formas sentenciais tais que $S \Rightarrow^* \alpha \Rightarrow^* \beta$, onde $\alpha \Rightarrow^* \beta$ corresponde à aplicação da seguinte seqüência de n produções de P:

$$p_1, p_2, \ldots, p_k, p_{k+1}, \ldots, p_n \ (p_i \in P, 1 \leqslant i \leqslant n)$$

Supondo-se a decomposição $\alpha = \alpha_1 \alpha_2$, o que implica na existência correspondente de $\beta = \beta_1 \beta_2$, e que p_k seja aplicada na derivação parcial $\alpha_1 \Rightarrow^* \beta_1$, e que p_{k+1} ocorra na derivação parcial $\alpha_2 \Rightarrow^* \beta_2$, prova-se que a seqüência de produções $p_1, p_2, \ldots, p_{k-1}, p_{k+1}, p_k, p_{k+2}, \ldots, p_n$ também implementa a derivação $\alpha \Rightarrow^* \beta$.

Em outras palavras, a permutação da ordem de aplicação das regras de substituição não altera a sentença gerada pela gramática.

Como conseqüência, existem em geral inúmeras seqüências possíveis de aplicação das produções para a derivação de uma sentença. Para os compiladores, convém que uma destas seqüências seja sempre utilizada, o que simplifica os algoritmos de reconhecimento. Denomina-se *seqüência canônica de produções* à seqüência obtida impondo-se alguma regra de escolha do não-terminal a ser substituído, entre os que figuram em uma forma sentencial qualquer. Geralmente, as regras utilizadas para tal escolha baseiam-se em dois critérios triviais de escolha:

(*a*) selecionar para substituição o não-terminal mais à esquerda que figura na forma sentencial. A este tipo de critério correspondem as análises denominadas *descendentes* ("top-down") com varredura da cadeia de entrada da esquerda para a direita.

(*b*) selecionar para substituição o não-terminal mais à direita que figura na forma sentencial. Nesta categoria podem ser classificados os analisadores *ascendentes* ("bottom-up"), com varredura da cadeia de entrada da esquerda para a direita e derivação mais à direita, em ordem invertida.

Dada uma seqüência de produções $p_1, p_2, \ldots, p_n$, aplicada à forma sentencial α para gerar a forma sentencial β, ou seja, $\alpha \Rightarrow^* \beta$, aplicando-se os resultados descritos anteriormente, α e β podem ser decompostas em $\alpha = \alpha_1 \alpha_2$, e $\beta = \beta_1 \beta_2$, tais que $\alpha_1 \Rightarrow^* \beta_1$ e $\alpha_2 \Rightarrow^* \beta_2$.

Se p_i figurar em $\alpha_2 \Rightarrow^* \beta_2$, e p_{i+1} em $\alpha_1 \Rightarrow^* \beta_1$, troca-se p_i com p_{i+1}, e repete-se o procedimento, até que não haja mais trocas possíveis na ordem de aplicação das produções. A seqüência de produções obtida nesta situação será canônica.

Apesar de se poder garantir a existência de uma seqüência canônica de produções, o mesmo não se pode afirmar de sua unicidade. Se houver alguma sentença na linguagem, para a qual haja mais de uma seqüência canônica de produções, diz-se que a gramática que gera esta sentença é *ambígua*. Observa-se que, se uma gramática é ambígua, a linguagem por ela gerada pode ser gerada por muitas outras gramáticas, ambíguas ou não. Ambigüidade é, portanto, uma propriedade da gramática, e não da linguagem.

Seqüências de produções, sendo aplicáveis em ordens diversas sobre as formas sentenciais geradas pela gramática, podem ser consideradas como um procedimento paralelo de análise das sentenças, visto que gramáticas livres de contexto apresentam a propriedade da independência relativa entre partições diferentes da mesma cadeia, do ponto de vista da aplicação das produções. Isto cria um conceito de localidade, que permite o tratamento independente e simultâneo de partes diversas da cadeia. A representação mais natural deste paralelismo é feita através do uso de *árvores de derivação*, que são formas bidimensionais de representação da análise ou de síntese de uma cadeia, com base em uma gramática.

O trabalho básico de um compilador, ao efetuar a análise das sentenças da linguagem a que se refere, consiste em escolher, em cada etapa da derivação, uma produção gramatical a ser aplicada, já que a escolha do não-terminal a ser substituído não é suficiente, devido ao fato de que o mesmo não-terminal pode corresponder a diversas produções distintas. Não há solução trivial para o problema, nem no caso da análise ascendente nem no caso descendente, tendo a solução geral deste problema ficado em aberto por muitos anos desde que foi enunciado.

2.10 – LEITURAS COMPLEMENTARES

Há, na literatura, uma infindável coleção de textos publicados na área da teoria de linguagens e de compiladores. Aho (1972) faz um excelente apanhado de toda a teoria essencial ao estudo e à implementação de compiladores e linguagens de programação.

Mais específicos, e cobrindo aspectos teóricos mais detalhados, podem ser encontrados os textos de Harrison (1978), Hopcroft (1979 a, 1979 b) e Salomaa (1978). Harrison (1978) destaca-se pela apresentação e clareza de condução da teoria apresentada. Hopcroft (1979 b) é um clássico do assunto. Uma publicação mais recente, e de caráter mais introdutório, indicada para iniciantes, é Révész (1983), dedicado inteiramente às linguagens formais, e não, também, à teoria de autômatos, como os demais textos mencionados.

São recomendadas, como material de pesquisa, revistas técnicas da área, especialmente publicações de altíssimo nível tais como: *Information and Control, Mathematical Systems Theory, Acta Informatica,* etc. Não pode deixar de ser indicada, adicionalmente, a excelente série publicada pela Springer-Verlag, intitulada *Lecture Notes in Computer Science,* da qual vários números são dedicados ao estudo da teoria de autômatos e linguagens formais.

A notação alternativa apresentada em 2.6 para os autômatos de pilha foi introduzida e desenvolvida nas seguintes publicações:

JOSÉ NETO, J. & MAGALHÃES, M. E. S. – *Reconhecedores Sintáticos – Uma alternativa didática para o uso em cursos de engenharia.* Anais XIV Congresso Nacional de Processamento de Dados. São Paulo, 1981.

JOSÉ NETO, J. & MAGALHÃES, M. E. S. – *Um gerador automático de reconhecedores sintáticos para o SPD.* Anais do VIII SEMISH. Florianópolis, 1981.

MAGALHÃES, M. E. S. & JOSÉ NETO, J. – *Um gerador automático de núcleos eficientes para compiladores dirigidos por sintaxe.* Anais do X SEMISH. Campinas, 1983.

2.11 – EXERCÍCIOS

1 – Conceitue, intuitivamente, os seguintes termos. Exemplifique.

– linguagem	– alfabeto
– sentença	– cadeia
– átomo	– cadeia vazia
– gramática	– concatenação de cadeias
– reconhecedor	– concatenação de um átomo com uma cadeia
– derivação	– comprimento de uma cadeia

2 – Identifique cada um dos conceitos acima considerando a linguagem onde todas as sentenças são os números inteiros decimais, com ou sem sinal.

3 – Para o conjunto $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ apresente exemplos de cadeias que ilustrem os conceitos de fechamento transitivo e de fechamento recursivo e transitivo.

4 – Para a linguagem dos identificadores, formados de uma letra no mínimo, ou de uma cadeia iniciada por uma letra, seguida de letras e/ou dígitos, pede-se definir formalmente uma gramática $G = (V, \Sigma, P, S)$, identificando os quatro elementos que a compõem.

5 – Uma possível solução (entre as infinitas existentes) para o exercício 4, apresenta as seguintes produções:

$S \rightarrow L \mid L\,D$

$D \rightarrow N \mid L \mid DN \mid DL \mid \epsilon$

$L \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid \ldots \mid z$

$N \rightarrow \emptyset \mid 1 \mid 2 \mid \ldots \mid 9$

Pede-se: *a*) dar exemplos de sentenças com 1, 2, . . . , 10 átomos
b) derivar cada uma das sentenças fornecidas
c) identificar nas derivações, a sentença, derivações diretas, e derivações não triviais
d) identificar as formas sentenciais
e) montar árvores de derivações para as sentenças
f) ϵ pertence à linguagem gerada pela gramática?

6 – Conceitue cada um dos tipos de gramática da hierarquia de Chomsky.

7 – Dê exemplos de gramáticas lineares à direita e à esquerda.

8 – Dê exemplos de gramáticas livres de contexto.

9 – Verifique se as gramáticas do exercício 8 não são lineares à direita ou à esquerda.

10 – As gramáticas do exercício 7 podem ser respostas à pergunta 8? Em que situação?

11 – As gramáticas do exercício 8 podem ser respostas à pergunta 7? Em que situação?

12 – Que são reconhecedores?

13 – A máquina de estados definida na tabela abaixo reconhece a linguagem definida pela gramática do exercício 5:

Estado Atual	Entrada	Estado Seguinte
1	*a, b, . . . , z*	2
2	*a, b, . . . , z*	2
2	0, 1, . . . , 9	2

(Estado inicial = 1. Estado de aceitação = 2. Memória: não existe)

Para o texto *a1b2c3d4e* pede-se:

a) configuração inicial do reconhecedor
b) configuração final do reconhecedor
c) os movimentos do reconhecedor necessários à aceitação do texto

14 – O reconhecedor do exercício 13 é determinístico? Por quê?

15 – Que são autômatos finitos?

16 – Formalize um autômato finito $M = (Q, \Sigma, P, q_0, F)$ para o reconhecedor do exercício 13.

17 – Justifique: "Como a linguagem que define os identificadores pode ser reconhecida por um autômato finito, então esta linguagem é do tipo 3".

18 – Que são autômatos de pilha?

19 – A linguagem que define expressões envolvendo elementos *a*, somas e produtos, e subexpressões entre parênteses, pode ser definida através da gramática cujas produções são:

$S \rightarrow S + T \mid T$

$T \rightarrow T \cdot F \mid F$

$F \rightarrow a \mid (S)$

Pergunta-se: *a*) esta gramática é linear à direita/esquerda? Por quê?
b) esta gramática é livre de contexto? Por quê?
c) derive a sentença: $a + (a + a * a + (a + a)) * (a + a)$

20 – O reconhecedor seguinte reconhece a linguagem definida no exercício 19:

Estado Corrente	*Entrada*	*Próximo Estado*	*Pilha Corrente*	*Pilha Seguinte*
1	a	2		
1	(	1	λ	λ, 2
2	$*, +$	3		
2	)	2	λ, 2	λ
3	a	2		
3	(	1	λ	λ, 2

estado inicial: 1 estado final: 2 pilha inicial: vazia

Identifique os elementos que compõem o reconhecedor. Para o texto $a + (a + a * a + (a + a)) * (a + a)$, pede-se:

a) a configuração inicial do reconhecedor
b) a configuração final do reconhecedor
c) os movimentos do reconhecedor para a aceitação do texto

21 – Formalize um autômato de pilha $M = (Q, \Sigma, \Gamma, P, q_0, Z_0, F)$ que represente o reconhecedor do exercício 20. Identifique seus elementos.

22 – Que são gramáticas de transdução? Que são transdutores?

23 – A linguagem do exercício 19 pode ser reconhecida pelo seguinte conjunto (unitário) de submáquinas:

Submáquina S:

Estado Corrente	*Entrada*	*Próximo Estado*	*Pilha Corrente*	*Pilha Seguinte*
1	a	4		
1	(	2		
2	ϵ	1	λ	λ, 3
4	$+, *$	1		
4	ϵ	3	λ, 3	λ
3	)	4		

estado inicial: 1 estado final: 4 pilha inicial: vazia

Pode-se formalizar este autômato de pilha segundo a notação estruturada $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$, identificando cada uma das submáquinas $a_i = (Q_i, \Sigma_i, P_i, E_i, S_i)$ e os elementos do modelo formal. Interpretar o significado de cada produção.

24 – Para o texto $a + (a + a * a + (a + a)) + (a + a)$ já utilizado nos exercícios anteriores, identificar a situação inicial do autômato, as transições utilizadas no reconhecimento e a situação final.

25 – Usando o autômato de pilha do exercício 23, mostrar que o texto $a * (a + a * a)) + a$ não pertence à linguagem reconhecida pelo autômato. Faça o mesmo usando o autômato do exercício 20. Use, em uma terceira tentativa, a gramática do exercício 19.

26 – Represente identificadores (ver exercícios 4, 5 e 13) através das diversas notações para a definição de linguagens: expressões regulares, BNF, notação de Wirth, diagramas de sintaxe, diagramas de estado, tabelas de transição. Observe semelhanças e diferenças entre as notações.

27 – Crie uma gramática para a representação de números reais, como os dos seguintes exemplos:

a) 5.35
b) + 53.
c) – 029
d) 50
e) 8 E 10
f) – 5.E – 10
g) + .5 E + 5
h) 5.3 E 5

Utilize cada uma das notações formais estudadas.

28 – Crie reconhecedores para a aceitação de sentenças da linguagem definida no exercício 27.

29 – Utilize cada um dos exemplos do exercício 27, e teste com eles cada uma das gramáticas/reconhecedores dos exercícios 27 e 28.

30 – Mostre que os textos 5 + E – 5, – 5 –, 4.3 E– + 3 não são reconhecidos pelos autômatos do exercício 28, nem gerados pelas gramáticas do exercício 27.

3

Técnicas de Construção de Reconhecedores

62

Vistos os principais conceitos, definições e notações relativos à formalização de linguagens, pode-se passar, em seguida, ao estudo da maneira como os mecanismos de análise por eles representados podem ser explorados no sentido de auxiliar o projetista a construir o seu compilador.

Como se sabe, um dos grandes problemas suscitados pela construção de compiladores decorre do fato de que as linguagens de alto nível são quase sempre especificadas através de gramáticas. É fato, por outro lado, que os compiladores são, em grande número, construídos utilizando-se como estrutura de apoio um reconhecedor sintático da linguagem, que funciona como esqueleto do compilador.

Há uma certa distância conceitual e física entre as gramáticas e os reconhecedores de uma dada linguagem, o que exige que mecanismos, por vezes elaborados, sejam criados com a finalidade de permitir ao projetista o mapeamento cômodo da gramática de que dispõe em algum reconhecedor que seja capaz de definir a mesma linguagem. Em geral, em um primeiro mapeamento, o reconhecedor construído não é o mais adequado para utilização direta no compilador, exigindo uma certa manipulação, com a finalidade de se obter um outro reconhecedor equivalente, porém mais eficiente e compacto. Este capítulo tem como finalidade apresentar algumas técnicas usuais de mapeamento de gramáticas em reconhecedores e de otimização dos reconhecedores obtidos, bem como sugerir uma sistemática para a obtenção de reconhecedores eficientes a partir da descrição gramatical da sintaxe de uma linguagem de programação. São apresentadas alternativas para a obtenção manual e automática de tais reconhecedores, dando assim ao projetista diversas maneiras através das quais possa ser elaborado o compilador desejado.

3.1 – MANIPULAÇÃO DE DEFINIÇÕES FORMAIS

Nesta seção, são estudadas maneiras através das quais as definições formais podem ser transformadas de uma notação para outra. Isto é apresentado com a finalidade de permitir que,

no texto restante desta publicação, seja utilizada uma técnica de obtenção de reconhecedores baseada em uma notação formal única. Para aplicar esta técnica, uma seqüência de transformações são efetuadas sobre a descrição formal inicialmente fornecida, obtendo-se uma definição equivalente, da linguagem em questão, redigida na notação única para a qual está descrita a técnica de obtenção de reconhecedores. Isto permite, finalmente, que sejam aplicados os métodos a serem apresentados, e obtido, conseqüentemente, o reconhecedor desejado para a linguagem em estudo.

Há inúmeras combinações possíveis de pares de notações, para as quais de poderia desejar estudar a conversão de notação. Serão solucionados alguns destes pares, de modo que se possa, a partir dos métodos apresentados para os pares selecionados, dispor de ferramentas cuja aplicação sucessiva sobre uma definição de linguagem permita efetuar todas as transformações de interesse para a construção de compiladores através dos métodos a serem estudados.

Não serão tratadas, neste texto introdutório, formalizações mais complexas como gramáticas W e gramáticas de atributos, por não serem do escopo deste trabalho.

3.1.1 – Conversão entre Notações Gramaticais

Para iniciar, são estudadas as conversões entre alguns pares de notações gramaticais. São vistos informalmente métodos para a conversão entre os pares de notações nos dois sentidos, o que permite que, dada uma gramática expressa em qualquer das notações, seja possível, através da aplicação encadeada das regras adequadas, a obtenção da mesma em qualquer outra notação.

A primeira das notações a ser estudada é das produções gramaticais e suas conversões de formato, para BNF e vice-versa. Observe-se que entre as duas notações não há diferenças formais muito significativas, diferindo principalmente pelo fato de a notação BNF permitir o agrupamento de alternativas para um mesmo não-terminal. A conversão das produções gramaticais para BNF é trivial:

(a) Rebatizam-se todos os não-terminais, conforme as regras do BNF.
(b) Agrupam-se as opções de definição de cada não-terminal, separadas pelo símbolo de alternância, à direita do nome do não-terminal que definem. Entre o não-terminal e as opções, deve figurar o meta-símbolo ::= ao invés da seta →.

A conversão inversa é também trivial, bastando-se desmembrar as definições múltiplas em produções gramaticais separadas, substituindo-se o sinal ::= por → e rebatizando-se os não-terminais, através do uso de nomes simples, de preferência de um único caractere.

Exemplo:

Produções ←————→ *BNF*

Produções	BNF
$S \rightarrow x$	<S> ::= <M> | x
$S \rightarrow M$	<M> ::= <M> <N>
$M \rightarrow MN$	<N> ::= y
$N \rightarrow y$	

Uma notação muito útil na metodologia de construção de reconhecedores a ser apresentada, é a notação de Wirth, cuja forma é semelhante à do BNF, mas que apresenta meta-símbolos adicionais, que possibilitam a eliminação de algumas definições recursivas.

A conversão da notação BNF para a de Wirth pode ser efetuada com ou sem a eliminação de recursão. Para o presente estudo, porém, há uma forte conveniência em tal eliminação, razão pela qual é detalhada a seguir uma regra que permite executar tal operação:

Seja um não-terminal <X>, definindo em BNF através de uma definição múltipla, em que possa ocorrer, no caso geral, referenciado recursivamente à esquerda, à direita ou no centro de uma cadeia de terminais e/ou não-terminais:

$$<X> ::= <X>\alpha_1 \mid <X>\alpha_2 \mid \ldots \mid <X>\alpha_m \mid$$

$$\beta_1 <X> \gamma_1 \mid \beta_2 <X> \gamma_2 \mid \ldots \mid \beta_n <X> \gamma_n \mid$$

$$\delta_1 <X> \mid \delta_2 <X> \mid \ldots \mid \delta_p <X> \mid$$

$$\theta_1 \mid \theta_2 \mid \ldots \mid \theta_q$$

onde α_i, β_i, γ_i, δ_i, θ_i são cadeias não vazias de terminais e/ou não-terminais, sendo que θ_1 pode assumir a forma ϵ, designando a cadeia vazia.

Chamando-se, na notação de Wirth, X ao não-terminal <X> definido acima, a regra geral de conversão desta definição para uma forma não recursiva dada em notação de Wirth consiste em reescrever esta definição como segue:

$$X = \{\delta_1 \mid \delta_2 \mid \ldots \mid \delta_p\} (\theta_1 \mid \theta_2 \mid \ldots \mid \theta_q \mid \beta_1 X \gamma_1 \mid \beta_2 X \gamma_2 \mid \ldots \mid \beta_n X \gamma_n) \{\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m\}$$

Caso $\theta_q = \epsilon$, então X pode ser escrito na forma:

$$X = \{\delta_1 \mid \delta_2 \mid \ldots \mid \delta_p\} [\theta_1 \mid \theta_2 \mid \ldots \mid \theta_{q-1} \mid \beta_1 X \gamma_1 \mid \beta_2 X \gamma_2 \mid \ldots \mid \beta_n X \gamma_n] \{\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m\}$$

A transformação acima pode ser justificada com base nos seguintes fatos:

(*a*) A linguagem gerada por uma produção do tipo $<X> ::= <X> \alpha \mid \theta$ é o conjunto $\{\theta, \theta\alpha, \theta\alpha\alpha; \theta\alpha\alpha\alpha, \ldots\}$, que pode ser representado como $\theta\ \{\alpha\}$ na notação de Wirth

(*b*) Por raciocínio semelhante, $<X> ::= \gamma <X> \mid \theta$ gera a linguagem $\{\gamma\}\ \theta$

(*c*) Para opções do tipo $<X> ::= <X>\alpha_1 \mid <X>\alpha_2 \mid \ldots \mid <X>\alpha_m \mid \theta$ pode-se passar para a notação de Wirth inicialmente como

$$X = X\alpha_1 \mid X\alpha_2 \mid \ldots \mid X\alpha_m \mid \theta$$

Colocando-se X em evidência, vem

$$X = X\,(\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m) \mid \theta$$

que é da forma $X = X\alpha \mid \theta$.
Pode-se, portanto, escrever $X = \theta\ \{\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m\}$

(*d*) Analogamente, $\langle X\rangle ::= \delta_1 \langle X\rangle \mid \delta_2 \langle X\rangle \mid \ldots \mid \delta_p \langle X\rangle \mid \theta$ pode ser escrito como $X = \{\delta_1 \mid \delta_2 \mid \ldots \mid \delta_p\}\theta$

(*e*) Para o caso composto: $\langle X\rangle ::= \langle X\rangle\, \alpha \mid \delta \langle X\rangle \mid \theta$ a linguagem gerada é o conjunto:

$$X = \{\theta, \theta\alpha, \delta\theta, \theta\alpha\alpha, \delta\theta\alpha, \delta\delta\theta, \ldots\}$$

$$= \{\theta, \theta\alpha, \theta\alpha\alpha, \ldots, \delta\theta, \delta\delta\theta, \ldots, \delta\theta\alpha, \delta\theta\alpha\alpha, \ldots\}$$

Note-se que θ aparece precedido de uma cadeia de δ's, e sucedido por uma cadeia de α's. Isto pode ser reescrito como

$$X = \{\delta\}\ \theta\ \{\alpha\}$$

(*f*) Chamando-se de θ o conjunto das opções independentes de X, e das que apresentam X em recursões centrais, tem-se:

$$\theta = (\theta_1 \mid \theta_2 \mid \ldots \mid \theta_q \mid \beta_1 X \gamma_1 \mid \ldots \mid \beta_n X \gamma_n)$$

Pode-se compor esta definição com os resultados anteriores, chegando-se à forma final desejada:

$$X = \{\delta_1 \mid \delta_2 \mid \ldots \mid \delta_p\}\ (\theta_1 \mid \theta_2 \mid \ldots \mid \theta_q \mid \beta_1 X \gamma_1 \mid \ldots \mid \beta_n X \gamma_n)\ \{\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m\}$$

Exemplo:

BNF ⟵⟶	*Wirth*
<X> ::= <X> *a* | <X> *b* | <X> <Y>	X = X ("a" | "b" | Y)
<Y> ::= <Y> *a* | *b* <Y> *c* | *c* <Y> *d*	Y = ("b"Y"c" | "c"Y"d") {"a"}
<Z> ::= <Z> *a* | *b* | *c* <Z> | ε	Z = {"c"} "b" {"a"}
<T> ::= <T> *abc* | <T> <Z> <Y> |	T = {"a""b""c" | "d"}
ab <T> *c* | *a* <T> *bc* |	("e" | "f" "g" X |
abc <T> | *d* <T> |	"a""b"T"c" |
e | *fg* <X>	"a"T"b""c")
	{"a""b""c" | ZY}

A conversão inversa, da notação de Wirth para BNF, pode ser feita de várias maneiras. A mais simples é a conversão canônica:

(*a*) rebatizam-se os não-terminais, de acordo com a notação BNF

(*b*) criam-se novos não-terminais para cada agrupamento de opções entre parênteses, colchetes ou chaves.

(*c*) criam-se produções BNF para cada um dos não-terminais originais, substituindo-se os agrupamentos eventualmente existentes por referências ao não-terminal correspondente criado.

(*d*) criam-se produções BNF para cada agrupamento entre parênteses, definindo o não-terminal correspondente como sendo o conjunto das opções agrupadas.

(*e*) criam-se produções BNF para cada agrupamento entre colchetes, definindo o não-terminal correspondente como sendo o conjunto das opções agrupadas, e mais a cadeia vazia ϵ.

(*f*) criam-se produções BNF para cada agrupamento entre chaves, definindo recursivamente o não-terminal $\langle X\rangle$ criado para designar tal agrupamento, como segue:

Seja $\{ \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \}$ o agrupamento.

Sendo $\langle X\rangle$ o nome do não-terminal que irá representá-lo,

$$\langle X\rangle ::= \alpha_1 \langle X\rangle \mid \alpha_2 \langle X\rangle \mid \ldots \mid \alpha_n \langle X\rangle \mid \epsilon$$

ou então

$$\langle X\rangle ::= \langle X\rangle \alpha_1 \mid \langle X\rangle \alpha_2 \mid \ldots \mid \langle X\rangle \alpha_n \mid \epsilon$$

que, como se pode verificar facilmente, geram a mesma linguagem que o agrupamento inicial representa.

(*g*) aplicam-se as regras acima até que toda a transformação tenha sido realizada.

Exemplos:

Wirth ←——————→ *BNF*

1) { "a" | "b" Y "c" | "d" "e" } <X> ::= a <X> |
b <Y> c <X> |
d e <X> | ϵ

(introduz-se o não-terminal X)

ou então <X> ::= <X> a |
<X> b <Y> c |
<X> d e | ϵ

2) ("a" | "b" C | D "e") <Z> ::= a | b <C> | <D> e

3) ["a" | "b" C "d"] <T> ::= a | b <C> d | ϵ

4) "a" ("b" | ["c" | "d"] { "e" }) <S> ::= a <K>
<K> ::= b | <L> <M>
<L> ::= c | d | ϵ
<M> ::= <M> e | ϵ

Uma outra notação próxima da notação de Wirth é a notação utilizada na definição das linguagens Cobol e PL/I. A diferença básica das notações reside apenas na maneira de denotar os agrupamentos e as repetições, o que indica que a operação de conversão para a notação de Wirth e vice-versa não é complexa:

(*a*) Identificam-se as palavras reservadas (escritas com letras maiúsculas).
(*b*) Identificam-se as palavras-chave (sublinhadas) como obrigatórias, e as demais como opcionais. Para convertê-las para a notação de Wirth, as palavras sublinhadas permanecem como são, enquanto as não sublinhadas devem ser denotadas entre colchetes, indicando que são opcionais.
(*c*) Identificam-se nomes escritos em letras minúsculas como sendo os não-terminais, rebatizando-se tais nomes segundo a convenção da notação de Wirth, se necessário.
(*d*) Identificar os agrupamentos de opções entre chaves. Uma das opções deve ser obrigatoriamente escolhida. Isto corresponde, na notação de Wirth, a um agrupamento equivalente de opções entre parênteses. As opções, relacionadas uma em cada linha, devem ser separadas pelo sinal de alternância utilizado na notação de Wirth (|).
(*e*) Os agrupamentos entre colchetes sofrem um tratamento idêntico, porém sem alterar os colchetes, que também simbolizam opcionalidade na notação de Wirth.
(*f*) Elementos seguidos de reticências (...) podem ser repetidos quantas vezes se desejar. Se o elemento for obrigatório, a conversão não é direta, exigindo o seguinte artifício: Seja X o elemento obrigatório. Neste caso, X ... é convertido para $(X)\{X\}$ na notação de Wirth, para garantir que X ocorra no mínimo uma vez. Se o elemento for opcional, então basta denotá-lo entre chaves na notação de Wirth. Assim, $[X]$... é convertido para $\{X\}$, e $\{X\}$... para $(X)\{X\}$. As alternativas que X representa deverão, naturalmente, ser separadas por barras. Os parênteses podem ser omitidos caso tal omissão não altere o significado da notação.

A conversão oposta é também muito simples:

(*a*) todos os não-terminais devem ser denotados em letras minúsculas.
(*b*) palavras reservadas devem ser denotadas em maiúsculas, sendo sublinhadas quando obrigatórias.
(*c*) agrupamentos entre parênteses devem ser reescritos entre chaves, com as opções listadas uma por linha. Os separadores usados como sinal de alternância são suprimidos.
(*d*) agrupamentos entre colchetes continuam entre colchetes, também com as opções listadas uma por linha, e sem separadores de alternância.
(*e*) agrupamentos entre chaves devem ser reescritos entre colchetes, seguidos de reticências. Os separadores de alternância devem também ser eliminados, listando-se uma opção por linha.
(*f*) uma seqüência do tipo $(X)\{Y\}$, onde o conjunto das opções X e conjunto das opções Y são iguais, pode ser reescrita como $\{X\}$..., também eliminados os separadores e listadas as opções uma por linha.

Exemplo:

Notação Cobol ⟷	*Wirth*
$\left\{\begin{matrix} a \\ b \\ c \\ XYZ \end{matrix}\right\}$	*(A | B | C | "XYZ")*
$\left[\begin{matrix} a \\ b \\ XYZ \end{matrix}\right]$	*[A | B | "XYZ"]*
$\{\ a\ \}\ldots$	$A\ \{A\}$
$[\ a\ ]\ldots$	$\{A\}$
$\left\{\begin{matrix} a \\ b \end{matrix}\right\}\ldots$	*(A | B) {A | B}*
$\left[\begin{matrix} a \\ XY \end{matrix}\right]\ldots$	*{A | "XY"}*
$\left[\begin{matrix} \left\{\begin{matrix} a \\ XY \end{matrix}\right\}\ [b]\ldots\ XY \\ XYZ\ \ [a]\ldots \end{matrix}\right]\ldots$	*{ (A | "XY") B {B} "XY" | "XYZ" {A} }*

Outra notação, que guarda uma grande semelhança com a notação de Wirth, é a das expressões regulares estendidas. A conversão entre as duas notações baseia-se em mera substituição de meta-símbolos. Dada uma gramática denotada através de expressões regulares estendidas, obtém-se uma gramática equivalente na notação de Wirth do seguinte modo:

(*a*) Substitui-se um agrupamento do tipo $(X \mid \epsilon)$ por $[X]$.
(*b*) Substitui-se um agrupamento do tipo $(X)^*$ por $\{X\}$.
(*c*) Substitui-se um agrupamento do tipo $(X)^+$ por $(X)\ \{X\}$.
(*d*) Mantêm-se os agrupamentos do tipo (X).

A conversão oposta é trivial:

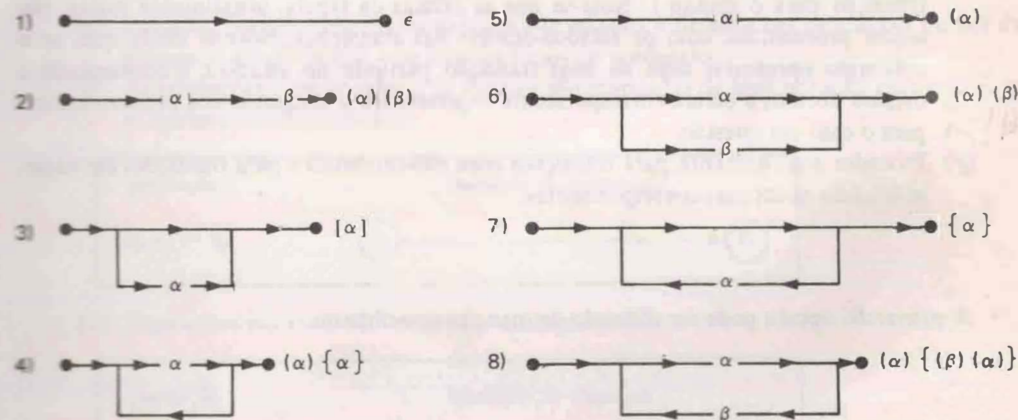
(*a*) Substitui-se um agrupamento entre colchetes $[X]$ por um agrupamento idêntico entre parênteses, incluindo adicionalmente como opção a cadeia vazia ϵ: $(X \mid \epsilon)$.
(*b*) Substitui-se um agrupamento entre chaves $\{X\}$ por um agrupamento idêntico entre parênteses seguido do sinal * indicativo de fechamento recursivo e transitivo: $(X)^*$.
(*c*) Mantêm-se os agrupamentos entre parênteses (X).
(*d*) Para seqüências do tipo $(X)\ \{Y\}$ com os conjuntos X e Y de opções iguais, reescreve-se na forma de expressões regulares estendidas como $(X)^+$.

Exemplo:

Expressões Regulares Estendidas ⟷	*Wirth*
$(a \mid b \mid \epsilon)$	$[a \mid b]$
a^*	$\{a\}$
$(a \mid b)^*$	$\{a \mid b\}$
$(a \mid b)^+$	$(a \mid b)\ \{a \mid b\}$
$(a \mid b)$	$(a \mid b)$

Os diagramas de sintaxe tornam-se dia a dia mais utilizados para a descrição da sintaxe de linguagens de programação. Sua conversão para alguma das outras notações torna-se interessante, assim como a conversão oposta, visto que os diagramas de sintaxe correspondem a representações gráficas, que dificultam sua manipulação pelo computador, embora facilitem a compreensão da sintaxe da linguagem, quando usadas em suas documentações.

A conversão, nos dois sentidos, é feita pela aplicação de substituições de notações, de acordo com as correspondências seguintes. A tabela apresenta a conversão entre os diagramas de sintaxe e a notação de Wirth.



Nestes diagramas, α e β podem assumir a forma de terminais, não-terminais, ou quaisquer das oito formas de diagramas apresentadas.

Os parênteses podem ser dispensados (ou ignorados) nos casos em que isto não alterar o significado da expressão obtida.

Nas expressões da notação de Wirth, os símbolos α e β podem corresponder a terminais, não-terminais, ou agrupamentos (entre parênteses, colchetes ou chaves), de expressões na notação de Wirth, separadas por meta-símbolos de alternância (|) se necessário.

Com este conjunto de regras, tem-se um grupo de metalinguagens entre as quais é possível efetuar conversões duas a duas. Encadeando a aplicação das regras, pode-se facilmente converter uma gramática, de qualquer destas notações para qualquer outra deste conjunto. Outras notações existem, sendo relativamente simples criar novas regras de conversão para elas.

3.1.2 – Conversão entre Formatos de Reconhecedores

Nesta seção será efetuado para os reconhecedores um estudo semelhante ao realizado anteriormente para as gramáticas. São tratadas três notações: as produções dos autômatos, as tabelas de transições e os diagramas de estados.

Muitas vezes, os diagramas de estados aparecem como primeira forma de representação para os autômatos, por causa de sua apresentação gráfica e facilidade de leitura pelo projetista. Para o computador, entretanto, formas tabulares são indiscutivelmente mais cômodas e eficientes. Assim, a conversão de uma notação para a outra torna-se conveniente para efeito de implementação e de documentação. Para converter diagramas de estados em tabelas de transições, pode-se seguir as seguintes regras:

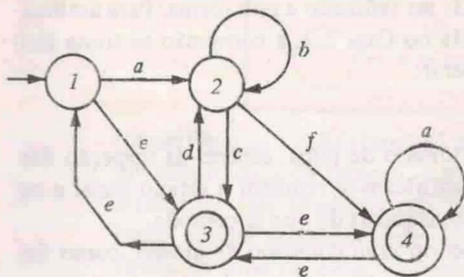
(*a*) Cria-se uma linha na tabela de transições para cada estado do diagrama.
(*b*) Marcam-se os estados finais (de aceitação) e o estado inicial.
(*c*) Cria-se uma coluna para cada terminal e não terminal utilizados no diagrama.
(*d*) Cria-se uma coluna adicional para representar quaisquer outros terminais da linguagem que não estejam sendo explicitamente utilizados no diagrama.
(*e*) Cria-se uma coluna adicional para transições em vazio.
(*f*) Para cada transição do estado *i* para o estado *j*, consumindo um átomo α, insere-se, na coluna correspondente a α, da linha correspondente ao estado *i*, a indicação de transição para o estado *j*. Note-se que as células da tabela, inicialmente vazias, vão sendo preenchidas com os estados-destino das transições. Note-se ainda que, se o autômato apresentar mais de uma transição partindo do estado *i*, e consumindo o mesmo átomo, a célula correspondente representará o conjunto dos estados-destino para o caso em questão.
(*g*) Proceder analogamente para transições com não-terminais e para transições em vazio, utilizando as colunas correspondentes.

A conversão oposta pode ser efetuada de maneira semelhante:

(*a*) Cria-se, para cada linha da tabela, um estado no diagrama.
(*b*) Marcam-se os estados inicial e finais.
(*c*) Para cada célula da tabela, correspondente ao estado *i*, e ao consumo de um terminal α, que indique transições para um conjunto de estados-destino $\{j_1, \ldots, j_n\}$, criar, no diagrama de estados, caminhos orientados do estado *i* para os estados $j_1, \ldots, j_n$, com indicação do consumo do terminal α. Se o conjunto for vazio, nada há a fazer.
(*d*) Tratar de modo similar as células pertencentes a colunas correspondentes a não-terminais e à transição em vazio.
(*e*) Para uma eventual célula não vazia do estado *i*, pertencente à coluna correspondente a todos os demais terminais não explícitos na tabela, criar para cada um destes terminais (α) caminhos orientados partindo do estado *i* para cada um dos estados-destino indicados na célula.

Exemplo:

Diagramas de Estados ⟷ *Tabelas de Transições*



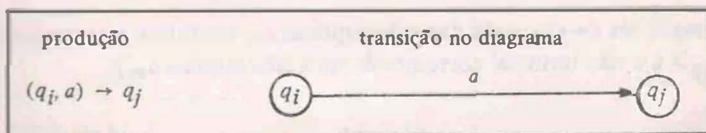
	a	*b*	*c*	*d*	*e*	*f*	*ε*
→ *1*	*2*						*3*
2		*2*	*3*			*4*	
⓷				*2*	*1,4*		
⓸	*4*				*3*		

O formato de produções, para a denotação dos autômatos, é muito conveniente do ponto de vista teórico, explicitando cada possível transição do mesmo. Neste trabalho, será apresentado um método através do qual é possível converter gramáticas em reconhecedores expressos através de produções. Esta é a principal razão da inclusão desta notação neste estudo de conversões de formatos. Há dois casos a considerar que são de interesse: o dos autômatos finitos e o dos autômatos de pilha.

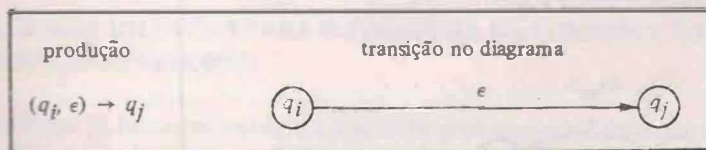
A conversão da notação de autômatos finitos, de produções para diagramas de estados, bem como a conversão oposta, é trivial, e baseia-se nas regras seguintes:

(*a*) Levantam-se inicialmente: o conjunto de estados utilizados nas produções ou nos diagramas, o estado inicial e os estados finais do autômato.

(*b*) As transições com consumo de átomos são mapeadas de uma notação para outra com a seguinte correspondência:



(*c*) Transições em vazio são mapeadas analogamente:



Exemplo:

O autômato do exemplo anterior pode ser convertido para o seguinte conjunto de produções, de acordo com as regras acima:

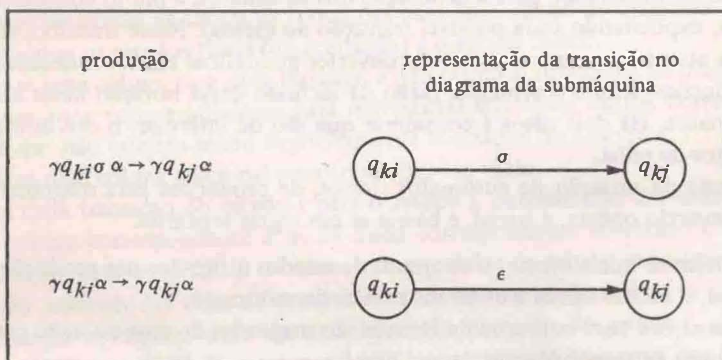
$(1, a) \to 2$	$(2, b) \to 2$	$(3, d) \to 2$	$(4, a) \to 4$
$(1, \epsilon) \to 3$	$(2, c) \to 3$	$(3, e) \to 1$	$(4, e) \to 3$
	$(2, f) \to 4$	$(3, e) \to 4$	

estado inicial: 1 *estados finais: 3, 4*

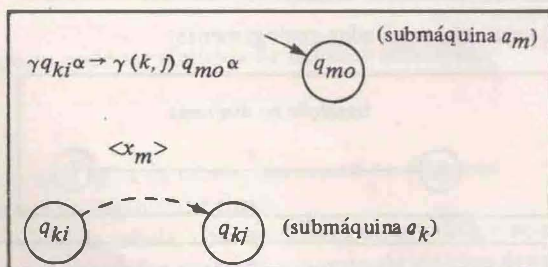
Para o caso dos autômatos de pilha, o mapeamento, no caso geral, é bastante complexo e não será estudado por não contribuir para as finalidades deste trabalho. Em lugar disto, serão considerados apenas os autômatos de pilha em sua forma mais adequada ao método de construção de reconhecedores a ser apresentado. Isto não tira a generalidade do estudo, uma vez que está demonstrado que qualquer autômato de pilha pode ser reduzido a esta forma. Para autômatos de pilha expressos na forma estrutural apresentada no Cap. 2.7, a conversão se torna simples, embora com um número maior de casos a considerar:

(*a*) Levantam-se inicialmente os estados do autômato de pilha, através da inspeção das produções ou dos diagramas de estados. Identificam-se também o estado incial e os estados finais do autômato, bem como as submáquinas de que se compõe.

(*b*) Mapeiam-se as transições internas, em vazio ou com consumo de átomo, como foi feito no caso dos autômatos finitos:

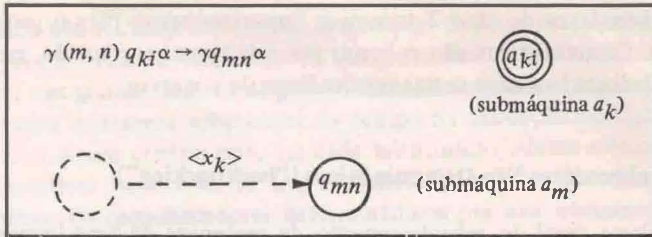


(*c*) Mapeiam-se transições de chamada das submáquinas a_m conforme a correspondência seguinte ($<x_m>$ é o não-terminal correspondente à submáquina a_m):



(Observar que q_{mo} é o estado inicial da submáquina a_m, e é atingido por transições incondicionais de chamada de $<x_m>$ em vazio, empilhando o estado de retorno desta transição com não-terminal).

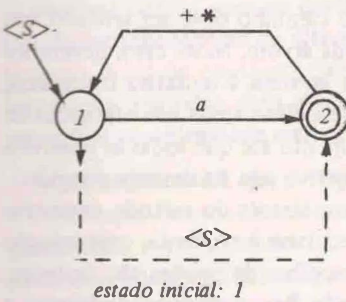
(*d*) Mapeiam-se as transições de retorno da submáquina corrente a_k, para a submáquina chamadora a_m, segundo a correspondência seguinte ($<x_k>$ é o não-terminal correspondente à submáquina a_k):



(Observar que a cada transição de chamada de $\langle x_k \rangle$ na submáquina a_m corresponde um estado q_{mn} de retorno, indicado no topo de pilha pelo par (m, n). Pares que não satisfazem esta condição não formam, na realidade, estados de retorno, e por esta razão não há necessidade de serem explicitados. Por outro lado, na notação, (m, n) figura no topo da pilha, simbolizando qualquer par válido, e q_{ki} figura como estado final, e portanto, simboliza uma transição incondicional, em vazio, para o estado indicado no topo da pilha, qualquer que seja ele. Para autômatos bem construídos, existe um casamento entre os estados empilhados e as transições de retorno, o que impede a ocorrência de inconsistências).

Exemplo:

Diagramas de submáquinas ←——→ *Produções*



$$\gamma q_{s1}\, a\alpha \rightarrow \gamma q_{s2}\, \alpha$$

$$\gamma q_{s1}\, \alpha \rightarrow \gamma\, (S, 2)\, q_{s1}\, \alpha$$

$$\gamma q_{s2} + \alpha \rightarrow \gamma q_{s1}\, \alpha$$

$$\gamma q_{s2} * \alpha \rightarrow \gamma q_{s1}\, \alpha$$

$$\gamma (m, n)\, q_{s2}\, \alpha \rightarrow \gamma q_{mn}\, \alpha$$

estado inicial: 1 *estado final: 2* *pilha inicial: vazia*

3.2 – ALGUMAS SOLUÇÕES PARA O PROBLEMA DA CONSTRUÇÃO DE RECONHECEDORES

Conforme já foi mencionado, a solução do problema geral da construção de reconhecedores, para linguagens irrestritas, é de uma complexidade muito grande, razão pela qual os estudiosos das técnicas de construção de compiladores procuraram reduzir tal complexidade através da introdução de restrições nas linguagens a serem compiladas. Naturalmente nem todas as restrições são aceitáveis, e surgiram desta forma compromissos entre a complexidade dos reconhecedores e a limitação das linguagens por eles aceitas. Desta maneira, através da observação de características das linguagens de programação usuais e das diversas classes de linguagens formais estudadas, chegou-se à conclusão que as linguagens do tipo 2 são as que maiores contribuições oferecem ao estudo dos compiladores, a custo moderado, visto que quase todas as linguagens de programação de interesse apresentam a propriedade de poderem ser modeladas com boas aproximações através de linguagens livres de contexto. Assim, o estudo das gramáticas, lingua-

gens e reconhecedores do tipo 2 tornou-se importantíssimo para o projetista e construtor de compiladores. Características não cobertas por tais aproximações são, em geral, tratadas a parte, em compiladores baseados nestas versões livres de contexto.

3.2.1 – Reconhecedores Não-Determinísticos ("backtracking")

O problema geral do reconhecimento de sentenças de uma linguagem de programação, com base em uma gramática, consiste em determinar uma seqüência de produções da gramática que seja capaz de gerar a sentença em questão. Uma solução ampla para este problema consiste na aplicação do método geral de solução de problemas baseado em buscas exaustivas, da solução desejada, no universo de combinações alternativas oferecidas pela gramática. Iniciando-se tal busca em um ponto de partida (no caso, partindo-se do não-terminal correspondente à raiz da gramática), e tendo-se em vista um objetivo (no caso, a construção da cadeia proposta para reconhecimento), procura-se montar uma árvore, cujos nós correspondem a pontos de decisão, em que uma escolha entre a aplicação de diversas produções deve ser feita, e cujos ramos indicam o caminho que foi tomado, ou seja, a escolha que foi realizada (o número de opções é obviamente finito em cada caso, o que garante que o procedimento termina em tempo finito).

Se, após uma escolha, um novo nó for atingido, diz-se que a tentativa teve sucesso. Caso contrário, diz-se que a tentativa falhou. Em função da seqüência de escolhas realizadas com sucesso, é possível que o objetivo seja atingido após uma escolha. Neste caso, o reconhecimento teve sucesso, e a sentença pode ser gerada a partir da gramática mediante a aplicação das produções escolhidas. Caso contrário, se a tentativa falhou, um novo caminho deve ser tentado, em função das escolhas ainda não realizadas nos nós já montados da árvore. Neste caso, devem ser eliminados da árvore já construída os últimos nós e ramos que levaram à tentativa fracassada, substituindo-se esta parte da árvore por outra, gerada a partir de escolhas ainda não efetuadas de produções nos nós superiores da árvore, repetindo-se o procedimento até que todas as possíveis substituições tenham sido tentadas sem sucesso, ou até que o objetivo seja finalmente atingido.

O procedimento encarregado do reconhecimento sintático, através do método exaustivo é, portanto, capaz de encontrar, para qualquer gramática não recursiva à esquerda, uma solução do problema, caso ela exista, uma vez que todas as possíveis escolhas de opções são tentadas, até que as opções se esgotem ou que uma solução seja encontrada. Para gramáticas ambíguas, a solução eventualmente encontrada pode não ser única. Neste caso, o procedimento pode ou terminar assim que a primeira solução é encontrada, ou então, prosseguir em busca de novas soluções. Caso o procedimento esteja preparado para prosseguir após a obtenção de uma solução, então tal procedimento será capaz de encontrar todas as soluções do problema, já que se utiliza de uma técnica exaustiva. De qualquer forma, como qualquer gramática de interesse apresenta um número finito de produções para cada não-terminal, o número de combinações será finito para qualquer cadeia de entrada de comprimento finito. No entanto, prova-se que o tempo gasto por um procedimento exaustivo cresce exponencialmente com o comprimento da cadeia de entrada, o que destrói significativamente as vantagens da generalidade por ela exibida, uma vez que são conhecidos algoritmos de reconhecimento para linguagens livres de contexto gerais cujo tempo de processamento é proporcional ao cubo do comprimento da cadeia de entrada. Para linguagens descritas por gramáticas livres de contexto, não-ambíguas, há algoritmos ainda mais eficientes, que dispensam totalmente o uso dos métodos exaustivos na grande maioria dos casos de interesse.

O método exaustivo descrito é classificado como *não-determinístico,* uma vez que não é seguido nenhum critério para a escolha de uma entre as várias produções disponíveis para reger

a substituição de um dado não-terminal em um ponto qualquer do reconhecimento da sentença. Não havendo critério, há uma alta probabilidade de que a escolha efetuada não seja a correta, especialmente se houver um grande número de possibilidades de escolha em cada caso. O modo óbvio de se promover uma economia substancial de tempo na execução do algoritmo de reconhecimento é o de criar algum critério que, em cada substituição, efetue a escolha da opção correta, dispensando tentativas desnecessárias. Um reconhecedor construído segundo esta diretriz é dito *determinístico*. Os reconhecedores determinísticos práticos utilizam-se do estado corrente do reconhecimento e de uma parte limitada da cadeia de entrada em torno do último átomo já utilizado, para efetuar a decisão acerca da escolha a ser realizada em cada situação. Há duas grandes classes de reconhecedores: os descendentes e os ascendentes; existem também métodos que mesclam as duas técnicas. A seguir são descritas algumas das principais opções de realização de reconhecedores determinísticos, para estas duas classes.

3.2.2 – Reconhecedores Determinísticos Descendentes ("top-down")

Uma categoria importante de reconhecedores determinísticos refere-se aos reconhecedores descendentes. Tais reconhecedores constroem a árvore de derivação das sentenças partindo do não-terminal que constitui a raiz da gramática, em direção à cadeia de átomos de que a sentença é composta. É possível construir reconhecedores descendentes determinísticos muito eficientes para uma classe de gramáticas livres de contexto suficientemente ampla para atender às necessidades da maioria das linguagens de programação usuais, para as quais permite a obtenção de reconhecedores de alta eficiência e grande simplicidade. Infelizmente nem todas as linguagens livres de contexto permitem a construção de reconhecedores descendentes. Para tais linguagens, métodos mais gerais devem ser utilizados, acarretando porém um custo maior, em termos de complexidade e eficiência do reconhecedor produzido.

Gramáticas LL (k) – Um subconjunto das gramáticas livres de contexto, que permite a construção de reconhecedores determinísticos descendentes para a classe mais ampla de tais gramáticas, é denominado $LL\ (k)$. Esta sigla designa gramáticas que permitem construir reconhecedores determinísticos que analisam a cadeia de entrada da esquerda para a direita, segundo o método descendente, ou seja, construindo a árvore a partir da raiz da gramática em direção aos terminais, e efetuando todas as decisões acerca da escolha da produção correta a ser utilizada em qualquer substituição de não-terminais com base no estado corrente do reconhecimento e em uma subcadeia de entrada, à direita do último átomo analisado, de comprimento máximo igual a k.

Mais rigorosamente, pode-se definir que uma gramática $G = (V, \Sigma, P, S)$ é $LL\ (k)$ quando todas as suas produções forem $LL\ (k)$. Introduzindo-se um novo não-terminal $\overline{S}$, definido através da produção $\overline{S} \rightarrow S\theta$, com θ sendo uma cadeia de comprimento k, diz-se que uma produção $A \rightarrow \alpha$ é $LL\ (k)$ se

$$\overline{S} \Rightarrow^* \beta A \gamma_1 \Rightarrow \beta \alpha \gamma_1 \Rightarrow^* \beta \alpha_1\ \delta_1$$

e

$$\overline{S} \Rightarrow^* \beta A \gamma_2 \Rightarrow \beta \alpha' \gamma_2 \Rightarrow^* \beta \alpha_1\ \delta_2$$

com $|\alpha_1| = k$, e $\alpha_1 \in \Sigma^*$, implicar em $\alpha = \alpha'$.

Um importante resultado da teoria de linguagens formais garante que, se uma gramática for $LL(k)$, então é possível, a partir dela, construir um reconhecedor determinístico descendente, e que isto é suficiente para indicar que a gramática não é ambígua. Outra propriedade das gramáticas $LL(k)$ refere-se ao fato de que nenhum de seus não-terminais é recursivo à esquerda, ou seja, não há produções na gramática que permitam que um não-terminal A possa derivar $A \Rightarrow^+ A\,\alpha$ para qualquer α.

É possível implementar reconhecedores descendentes de várias maneiras alternativas, das quais a mais conhecida e aplicada é a denominada *análise descendente recursiva.* Nesta implementação, o reconhecedor é construído basicamente na forma de uma coleção de procedimentos, eventualmente mutuamente recursivos, cada qual encarregado do reconhecimento de um dos não-terminais da gramática. Cada um dos procedimentos em questão comporta-se como uma função booleana, que retorna o valor lógico true caso seja reconhecida corretamente uma construção representada pelo não-terminal a que se refere, retornando o valor false caso contrário. A construção dos procedimentos em questão é feita com base em um mapeamento direto da gramática para um programa que efetua o reconhecimento. Através do teste do início da parte de cadeia de entrada a ser analisada, o procedimento decide qual das várias opções de substituição do não-terminal em questão deve ser aplicada. Os átomos iniciais desta parte da cadeia são consumidos, até que um novo não-terminal seja esperado. Nesta ocasião, o procedimento correspondente a tal não-terminal é chamado. Um teste do valor de retorno deste procedimento indica se houve sucesso ou não na tentativa de reconhecimento do não-terminal, e portanto, define se a análise foi bem sucedida ou não.

Note-se que, em tal filosofia de análise, uma produção recursiva à esquerda, porventura existente na gramática, leva o analisador a um ciclo interminável de recursões. Assim sendo, não podem ser admitidas produções recursivas à esquerda. Como, às vezes, uma gramática é apresentada com produções deste tipo, torna-se necessário eliminar tais recursões, através da substituição das mesmas por outras equivalentes, porém sem recursões à esquerda. Para isto, há dois casos a considerar: recursões explícitas à esquerda e recursões indiretas, através de outros não-terminais. No caso geral, uma recursão explícita assume a forma:

$$A \rightarrow A\theta_1 \mid A\theta_2 \mid \ldots \mid A\theta_n \mid \Psi_1 \mid \Psi_2 \mid \ldots \mid \Psi_m$$

onde: A é o não-terminal explicitamente recursivo à esquerda

θ_i são cadeias quaisquer de terminais e/ou não-terminais, não vazias

Ψ_i são também cadeias de terminais e/ou não-terminais, não iniciados por A.

Para eliminar a recursão explícita à esquerda, cria-se um novo não-terminal $\bar{A}$, e efetua-se a substituição do grupo de produções acima pelos seguintes:

$$A \rightarrow \Psi_1 \bar{A} \mid \Psi_2 \bar{A} \mid \ldots \mid \Psi_m \bar{A}$$

$$\bar{A} \rightarrow \theta_1 \bar{A} \mid \theta_2 \bar{A} \mid \ldots \mid \theta_n \bar{A} \mid \epsilon$$

A eliminação de recursões indiretas é mais complicada. Para uma gramática que não apresente ciclos (derivações possíveis do tipo $A \Rightarrow^* A$) ou produções do tipo $A \rightarrow \epsilon$, deve-se obter gramáticas equivalentes sem recursões indiretas através de uma transformação que envolve as seguintes operações (Sejam $A_1, \ldots, A_k$ os não-terminais da gramática):

(*a*) Para cada produção do tipo $A_i \to A_j\theta$ onde $A_j \to \Psi_1 \mid \Psi_2 \mid \ldots \mid \Psi_n$ efetuar a substituição $A_i \to \Psi_1\ \theta \mid \Psi_2\ \theta \mid \ldots \mid \Psi_n\theta$ para todo $j \leqslant i$.

(*b*) Eliminar as recursões explícitas à esquerda que aparecerem após a substituição.

(*c*) Repetir até que todos os A_i sejam considerados.

Este procedimento substitui, sucessivamente, as referências a não-terminais mais à esquerda das produções, pelas opções possíveis de substituição daqueles não-terminais, dadas pela gramática, até que uma recursão explícita à esquerda se manifeste. Nesta ocasião, a técnica de eliminação de tal recursão é aplicada.

Uma vez alterada desta maneira a gramática, é possível que ainda estejam presentes produções da forma

$$A \to \alpha\theta_1 \mid \alpha\theta_2$$

onde α é uma cadeia não vazia qualquer.

Neste caso, a análise do não-terminal A não é imediata, já que a presença de α na cadeia de entrada não é suficiente para que se possa decidir entre $\alpha\theta_1$ ou $\alpha\theta_2$ como a opção correta de substituição para o não-terminal A. Para controlar esta dificuldade, é possível alterar mais uma vez a gramática, substituindo-se o grupo acima pelos seguintes:

$$A \to \alpha\bar{A}$$

$$\bar{A} \to \theta_1 \mid \theta_2$$

Isto equivale a "fatorar" a cadeia α, simplificando a análise em troca do aumento do número de não-terminais na gramática. A essa operação dá-se o nome de *fatoração à esquerda* da gramática.

Neste ponto, a gramática está preparada para a implementação de reconhecedores descendentes recursivos. Cada não-terminal está definido através de um conjunto de opções cujos primeiros terminais permitem decidir qual das opções é aplicável em cada estado do reconhecimento. É possível ainda que uma das opções para a substituição de um não-terminal seja a cadeia vazia (ϵ). A decisão pela substituição do não-terminal por ϵ pode ser efetuada por exclusão: se os primeiros símbolos a analisar da cadeia de entrada não forem adequados às substituições explícitas por cadeias não vazias, então decide-se pela substituição do não-terminal pela cadeia vazia.

Exemplo:

Dada a gramática

$$S \to S + T$$
$$S \to T$$
$$T \to F * T$$
$$T \to F$$
$$F \to a$$
$$F \to (S)$$

Eliminam-se inicialmente as recursões à esquerda:

$$
\begin{aligned}
S &\rightarrow T\,X \\
X &\rightarrow +\,T\,X \\
X &\rightarrow \epsilon \\
T &\rightarrow F * T \\
T &\rightarrow F \\
F &\rightarrow a \\
F &\rightarrow (S)
\end{aligned}
$$

Mapeiam-se em seguida os diversos não-terminais nos procedimentos correspondentes às suas definições:

```
Boolean Procedure S;
   S := if not T
            then false
            else if not X
                 then false
                 else true;

Boolean Procedure X;
   if entrada = "+"
      then begin
           avança;
           X := S;
           end
      else X := true;

Boolean Procedure T;
   if not F
      then T := false
      else if entrada = "*"
           then begin
                avança;
                T := T
                end
           else T := true;

Boolean Procedure F;
   if entrada = "a"
      then begin
           F := true;
           avança;
           end
      else if entrada = "("
                 then begin
                      avança;
                      if S
                      then if entrada = ")"
                                then begin
                                     F := true;
                                     avança;
                                     end
                                     else F := false;
                           end
                      else F := false;

Programa Principal:
if S
   then compilação correta
   else erro de sintaxe;
```

Deve-se levar em consideração que nem todos os sistemas de programação incluem linguagens que implementam eficientemente procedimentos recursivos. Isto, entretanto, não inviabiliza o uso da idéia do método descendente recursivo, já que é possível implementá-lo tabularmente, através do uso de pilhas explícitas, controladas pelo programa reconhecedor, ao invés do das pilhas do sistema, utilizadas pela linguagem. Dispondo-se, todavia, de uma boa linguagem de programação, que ofereça procedimentos recursivos eficientes, o método pode ser utilizado diretamente, com a vantagem adicional de ter uma implementação extremamente simples e direta.

Outro grande atrativo dos reconhecedores descendentes recursivos é o fato de que, nos compiladores baseados em tais reconhecedores, a análise sintática e os procedimentos de geração de código aparecem juntos, o que torna relativamente auto-explicativo o programa assim construído.

Uma forma alternativa de implementação de reconhecedores descendentes recursivos explora o uso de tabelas, denominadas *tabelas de análise* ("parsing tables"). Estas tabelas exibem uma linha para cada não-terminal da gramática, e uma coluna para cada terminal, com uma coluna adicional para um marcador de final de cadeia de entrada.

Além da tabela de análise, o reconhecedor dispõe também de uma pilha explícita, através da qual a recursão pode ser simulada. O mesmo marcador utilizado para indicar o final da cadeia de entrada é utilizado também para indicar a condição de pilha vazia. O programa reconhecedor opera sobre essas informações através da análise do símbolo de entrada corrente, e do símbolo contido no topo da pilha.

A pilha, no início do processamento, deve conter apenas o símbolo indicador do não-terminal que corresponde à raiz da gramática. Nesta ocasião, o símbolo de entrada corrente é o primeiro terminal da cadeia de entrada. As células da tabela de análise indicam, para cada par (não-terminal, símbolo de entrada corrente), uma condição de erro ou então a produção a ser aplicada nesta situação.

O funcionamento do reconhecedor baseia-se em três movimentos possíveis. Seja A o elemento no topo da pilha, e σ o símbolo de entrada corrente:

(*a*) Se $A = \sigma$ e σ não for o marcador de pilha vazia (ou de final de cadeia de entrada), A é eliminado da pilha, e σ é consumido, sendo substituído pelo próximo símbolo da cadeia de entrada (movimento de consumo de átomo).

(*b*) Se $A = \sigma$ e σ for o marcador de pilha vazia (ou de final de cadeia de entrada), considera-se que o reconhecimento foi completado com sucesso, e o processamento é encerrado (movimento de finalização).

(*c*) Se A for um não-terminal, é consultada a linha correspondente da tabela de análise. Na célula correspondente à coluna relativa ao átomo de entrada corrente σ, encontra-se um indicador de erro (neste caso, o processamento termina) ou então a indicação da produção a ser utilizada para substituir o não-terminal A no topo da pilha. São empilhados os símbolos, que compõem a produção escolhida, em ordem reversa, de modo tal que o símbolo mais à esquerda fique no topo da pilha ao final da substituição. Nesta situação, o processamento deve prosseguir, repetindo-se os testes acima para a nova configuração do reconhecimento. Nos compiladores, assim que um movimento de substituição como este é executado, uma ação associada pode ser executada, com finalidade de efetuar, entre outras atividades, a geração de código associada à derivação em questão.

Exemplo:

(Este exemplo foi adaptado de Barrett e Couch 79)

$$V \to S\,R\,\bot$$

$$S \to + \mid - \mid \epsilon$$

$$R \to .dN \mid d\,N.N$$

$$N \to dN\,\epsilon$$

A tabela de análise correspondente é a seguinte:

	+	–	.	d	⊥
V	SR⊥	SR⊥	SR⊥	SR⊥	
S	+	–	ϵ	ϵ	
R			.dN	dN.N	
N			ϵ	dN	ϵ

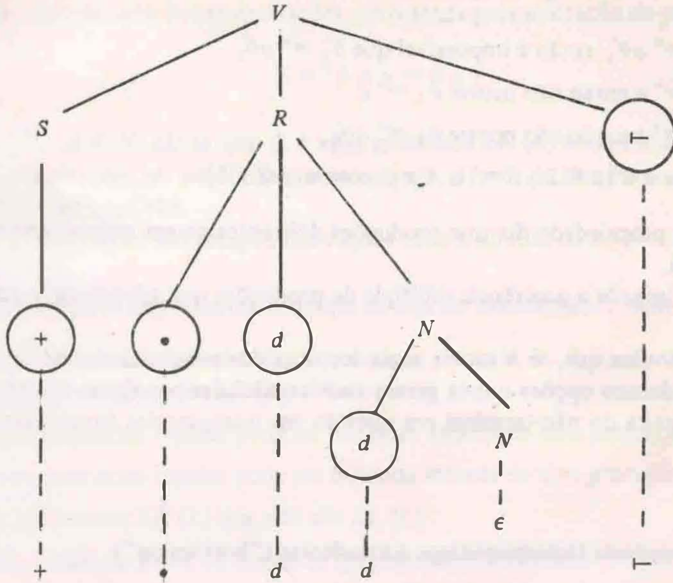
Para a seguinte cadeia de entrada:

$$+\,.\,d\,d\,\bot$$

a análise descendente se processa de acordo com a seqüência abaixo:

pilha	*cadeia*	*produção aplicada*
V ↑	+ . d d ⊥ ↑	V → S R ⊥
S R ⊥ ↑	+ . d d ⊥ ↑	S → +
R ⊥ ↑	. d d ⊥ ↑	R → . d N
. d N ⊥ ↑ ↑	. d d ⊥ ↑ ↑	
N ⊥ ↑	d ⊥ ↑	N → d N
d N ⊥ ↑	d ⊥ ↑	
N ⊥ ↑	⊥ ↑	N → ϵ
⊥ ↑	⊥ ↑	*aceitação da sentença*

O resultado desta análise é a seguinte árvore de derivações para a sentença fornecida:



Um reconhecedor descendente, baseado em tabelas de análise como estas, pode ser construído para qualquer gramática. Entretanto, algumas dessas gramáticas podem gerar tabelas de análise em que, em uma mesma célula, figurem duas ou mais produções ao invés de uma única. Isto ocorre sempre que a gramática em questão for ambígua ou então recursiva à esquerda.

A construção de uma tabela de análise para uma gramática $G = (V, \Sigma, P, S)$ baseia-se no seguinte princípio: Seja $A \rightarrow \theta$ uma produção de G. Seja σ um terminal pertencente ao conjunto dos possíveis terminais gerados mais à esquerda por θ, ou seja:

$$\sigma \in \{x \in \Sigma \mid \theta \Rightarrow^* x\theta', \theta' \in V^*\}.$$

Neste caso, sempre que o topo da pilha contiver o não-terminal A, e o símbolo de entrada corrente for σ, a célula correspondente da tabela de análise deverá conter a produção $A \rightarrow \theta$. Para os casos de exceção, em que $\theta \Rightarrow^* \epsilon$, a substituição $A \rightarrow \theta$ deverá ocorrer sempre que σ for um dos possíveis terminais que podem figurar à direita de uma construção em alguma forma sentencial, ou seja, as células da linha A, correspondentes às colunas relativas aos terminais σ tais que:

$$\sigma \in \{x \in \Sigma \mid S \Rightarrow^* \alpha A x \beta, \text{com } \alpha, \beta \in V^*\}.$$

Isto também deve ser feito para a coluna correspondente à marca de final de cadeia desde que exista $S \Rightarrow^* \alpha A$ para $\alpha \in V^*$, ou seja, quando A figurar como símbolo mais à direita de alguma forma sentencial.

Todas as células não preenchidas após estas inserções corresponderão a situações de erro.

Dá-se a designação de LL (1) às gramáticas cujas tabelas de análise não apresentem nenhuma célula com mais de uma produção. Prova-se que as tabelas de análise constituem propriamente o reconhecedor das linguagens geradas pelas gramáticas que lhes deram origem. Provam-se

ainda as seguintes propriedades para gramáticas LL (1):

Sendo $A \rightarrow \theta_1$ e $A \rightarrow \theta_2$ duas produções distintas de G, tem-se:

(*a*) Se $\theta_1 \Rightarrow^* \sigma\theta'_1$ então é impossível que $\theta_2 \Rightarrow^* \sigma\theta'_2$

(*b*) Se $\theta_1 \Rightarrow^* \epsilon$ então não ocorre $\theta_2 \Rightarrow^* \epsilon$

(*c*) Se $\theta_1 \Rightarrow^* \epsilon$ então não ocorre $\theta_2 \Rightarrow^* \sigma\theta'_2$,

onde $\sigma \in \{x \in \Sigma \mid S \Rightarrow^* \alpha A x \beta$, com $\alpha, \beta \in V^*\}$

A primeira propriedade diz que produções diferentes geram cadeias distinguíveis por seu primeiro terminal.

A segunda impede a ocorrência múltipla de produções que substituem o não-terminal pela cadeia vazia.

A terceira indica que, se a cadeia vazia for uma das possibilidades de substituição de um não-terminal, as demais opções nunca geram cadeias iniciadas por algum dos terminais que possam ocorrer à direita do não-terminal em questão em qualquer das formas sentenciais geradas pela gramática.

3.2.3 – Reconhecedores Determinísticos Ascendentes ("bottom-up")

Em contraste com o que ocorre no reconhecimento descendente, os reconhecedores ascendentes procuram construir a árvore de derivação de uma sentença partindo dos terminais da mesma, em direção à raiz da árvore. O objetivo do reconhecedor é o de construir uma árvore única, cuja raiz seja a raiz da gramática que gera a linguagem a que a sentença pertence, e cujas folhas representem os terminais que compõem a sentença, dispostos na ordem em que aparecem na sentença.

Deve-se notar que, apesar de se estar construindo uma árvore de derivações, o processo de construção não utiliza derivações, e sim *reduções*: enquanto no reconhecimento descendente um não-terminal é expandido em uma cadeia de símbolos através de uma derivação, especificada pela aplicação de uma produção da gramática, no reconhecimento ascendente parte-se de uma forma sentencial que representa a sentença em um dado estado da análise, na qual se procura localizar uma subcadeia, que seja idêntica ao lado direito de alguma das produções da gramática. Esta subcadeia é então *reduzida* ao não-terminal correspondente, através da aplicação da produção que indica a substituição de tal subcadeia pelo não-terminal. Note-se que, desta maneira, as produções são aplicadas em sentido reverso ao inicial, no qual o não-terminal é substituído pelo lado direito da produção, derivando a subcadeia em questão.

Entre todas as classes de reconhecedores determinísticos conhecidos, a dos reconhecedores ascendentes é aquela que maior abrangência exibe, tendo a capacidade de reconhecer um amplo espectro de linguagens, incluindo todas aquelas que são passíveis de reconhecimento por meio de reconhecedores descendentes. Assim sendo, dada uma gramática livre de contexto qualquer, é muito mais provável que seja possível construir para ela um reconhecedor determinístico ascendente do que qualquer outro.

Gramáticas LR (k) – O conceito de gramática *LR* (*k*) está muito ligado ao de "derivação mais à direita". Dada uma forma sentencial qualquer, dá-se o nome de *derivação mais à direita* à substituição do não-terminal mais à direita, encontrado nesta forma sentencial, por uma cadeia especificada através de alguma produção da gramática.

Seja $G = (V, \Sigma, P, S)$ uma gramática, e θ uma sentença da linguagem definida através de G. A sentença θ pode ser gerada a partir de S através de uma seqüência de derivações mais à direita. Observando-se uma destas derivações, provocada pela aplicação da produção $A \rightarrow \alpha$:

$$S \Rightarrow^* \beta A \gamma \Rightarrow \beta \alpha \gamma$$

onde $\beta\alpha\gamma \in V^*$, $A \in N$, diz-se que G é uma gramática $LR(k)$ se a produção $A \rightarrow \alpha$ puder, em cada passo de derivação, ser escolhida univocamente através da simples análise da cadeia $\beta\alpha\gamma_1$, onde $\gamma = \gamma_1 \gamma_2$ com $|\gamma_1| \leqslant k$.

Como as derivações são sempre derivações mais à direita, então $\gamma \in \Sigma^*$ obrigatoriamente.

Alguns resultados importantes da teoria devem ser mencionados neste ponto:

(*a*) Dado um inteiro positivo k arbitrário, se $G = (V, \Sigma, P, S)$ for uma gramática $LL(k)$ então G será também $LR(k)$

(*b*) Toda gramática regular pode ser expressa através de uma gramática $LL(1)$

(*c*) Toda gramática regular pode ser definida através de uma gramática $LR(1)$

(*d*) Há gramáticas $LR(k)$ que não são $LL(k)$

(*e*) Há gramáticas livres de contexto que não são $LL(k)$ nem $LR(k)$

Do ponto de vista prático, a construção de reconhecedores $LR(k)$ baseia-se na aplicação das derivações mais à direita, acima mencionadas, na ordem inversa da utilizada na derivação da sentença, já que o reconhecimento é efetuado partindo-se da sentença em direção à raiz da gramática. Para que isto seja viabilizado, lança-se mão de uma tabela de análise, e de uma pilha.

A cadeia de entrada é varrida da esquerda para a direita, e é nesta ordem que seus átomos são consumidos pelo reconhecedor, um por vez.

A pilha registra uma seqüência de símbolos, cada qual corresponde a um par (A_i, q_i) onde A_i é algum dos terminais ou não-terminais de gramática, e q_i é um estado do reconhecedor. O estado q_i de alguma forma representa, de modo compacto, toda a informação relevante sobre o histórico do reconhecimento, registrado na pilha. Com base em q_i, e no símbolo da cadeia de entrada a ser analisado em seguida, o reconhecedor decide a operação a ser executada, através de uma consulta à tabela de análise.

As tabelas de análise LR são apresentadas em uma linha para cada estado, e uma coluna para cada terminal, não-terminal ou símbolo de final de cadeia de entrada. As células da tabela indicam as ações a serem executadas pelo reconhecedor, e o próximo estado do reconhecedor.

Inicialmente, o reconhecedor deve ser levado à sua *situação inicial* $(q_0, \sigma_1\ \sigma_2 \ldots \sigma_n \phi)$, onde q_0 é o *estado inicial* do reconhecedor, $\sigma_1\ \sigma_2 \ldots \sigma_n$ é a cadeia de entrada, de comprimento n, $\sigma_i \in \Sigma$, e $\phi \notin \Sigma$ é um marcador do final da cadeia de entrada.

Em um instante qualquer do reconhecimento, a situação do reconhecedor se torna: $(q_0\ A_1\ q_1\ A_2\ q_2 \ldots A_p\ q_p, \sigma_r \sigma_{r+1} \ldots \sigma_n \phi)$ onde (A_i, q_i) são os pares empilhados, $\sigma_r \sigma_{r+1} \ldots \sigma_n$ é a parte da cadeia de entrada ainda não consumida, e σ_r é o próximo símbolo a ser analisado na cadeia de entrada.

Nesta situação, o reconhecedor deve analisar o próximo átomo de cadeia (σ_r) e o estado corrente (q_p), através de consulta à célula da tabela de análise correspondente ao estado q_p e ao átomo σ_r.

Há quatro informações que podem ser encontradas nesta célula:

(a) *Consumir* o átomo σ_r, e *empilhar* o par (σ_r, q), onde q é obtido consultando-se, na tabela de análise, o próximo estado do reconhecedor. A nova situação do reconhecedor torna-se

$$(q_0 A_1 q_1 A_2 q_2 \ldots A_p q_p \sigma_r q, \; \sigma_{r+1} \sigma_{r+2} \ldots \sigma_n \phi)$$

(b) Efetuar uma *redução* segundo uma produção da gramática. Neste caso, supondo-se que a produção seja da forma $A \rightarrow A_m A_{m+1} \ldots A_p$, então a nova situação do reconhecedor será:

$$(q_0 A_1 q_1 A_2 q_2 \ldots A_{m-1} q_{m-1} A q, \; \sigma_r \sigma_{r+1} \ldots \sigma_n \phi)$$

e a cadeia de entrada não é alterada. O novo estado q é obtido na tabela de análise, na coluna correspondente ao não-terminal A, e na linha relativa ao estado q_{m-1}.

(c) *Aceitar* a cadeia de entrada. Neste caso o reconhecimento é dado por encerrado com êxito.

(d) *Erro detectado.* Neste caso a cadeia de entrada está incorreta, sendo usual a execução de algum procedimento de recuperação para permitir a ressincronização do reconhecedor com a cadeia de entrada.

Mediante a aplicação iterativa destes movimentos do reconhecedor, é possível, partindo de uma situação inicial, montar a árvore da derivação da sentença com base nos movimentos de redução do reconhecedor. O movimento final de aceitação terá indicado que a árvore está completa e que a cadeia de entrada foi reconhecida corretamente.

Exemplo:

Ainda para a gramática utilizada no exemplo anterior:

$$V \rightarrow S R \perp$$

$$S \rightarrow + \mid - \mid \epsilon$$

$$R \rightarrow . d N \mid d N . N$$

$$N \rightarrow d N \mid \epsilon$$

usando-se as seguintes convenções de simbologia:

r = *"read" (ler e consumir novo símbolo de entrada)*

ℓ = *"look ahead" (consultar o símbolo seguinte sem consumir)*

a = *"apply" (aplicar uma redução)*

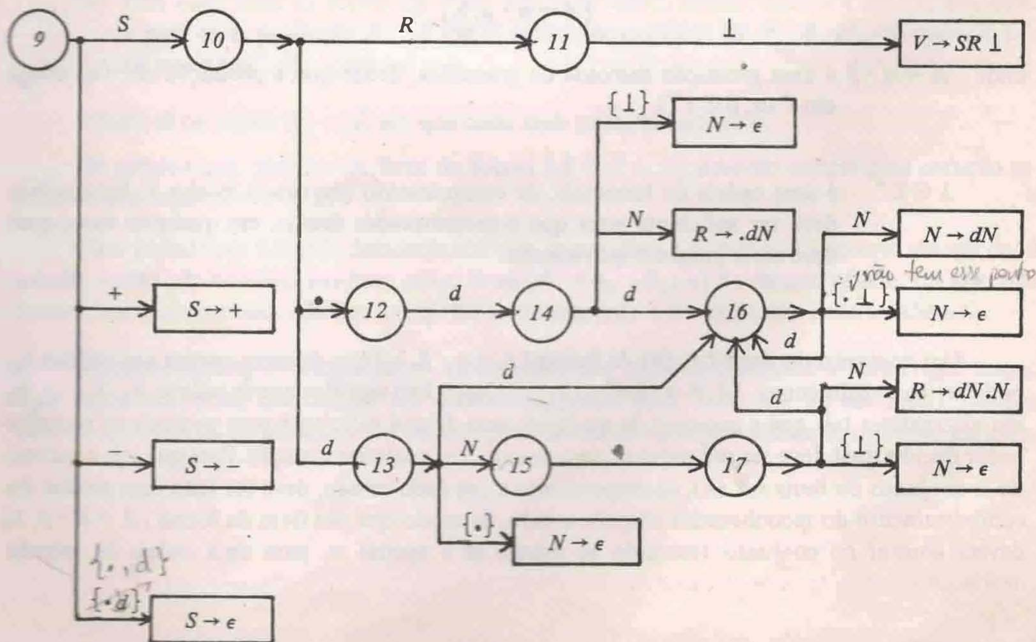
* = *aceitar a cadeia*

– = *erro*

uma possível tabela de análise LR para esta gramática será:

estado \ elemento	S	R	N	V	$+$	$-$	$.$	d	$\perp$
9	r 10	–	–	–	a $S \to +$	a $S \to -$	ℓ $S \to \epsilon$	ℓ $S \to \epsilon$	–
10	–	r 11	–	–	–	–	r 12	r 13	–
11	–	–	–	–	–	–	–	–	a * $V \to SR\perp$
12	–	–	–	–	–	–	–	r 14	–
13	–	–	r 15	–	–	–	ℓ $N \to \epsilon$	r 16	–
14	–	–	a $R \to .dN$	–	–	–	–	r 16	ℓ $N \to \epsilon$
15	–	–	–	–	–	–	r 17	–	–
16	–	–	a $N \to dN$	–	–	–	ℓ $N \to \epsilon$	r 16	ℓ $N \to \epsilon$
17	–	–	a $R \to dN.N$	–	–	–	–	r 16	ℓ $N \to \epsilon$

Graficamente, pode-se desenhar o seguinte reconhecedor em correspondência a esta tabela:



Para a mesma cadeia de entrada do exemplo descendente:

$$+ \, . \, d \, d \, \perp$$

o reconhecimento se processa de acordo com a seguinte seqüência (note-se a formação automática da árvore invertida):

Situação:	*Caminho*	*Produção*
$+ \bullet d \; d \perp$	9	$S \rightarrow +$
$S \bullet d \; d \perp$	9-10-12-14-16	$N \rightarrow \epsilon$
$S \bullet d \; d \; N \perp$	9-10-12-14-16	$N \rightarrow dN$
$S \bullet d \; N \perp$	9-10-12-14	$R \rightarrow .dN$
$S \; R \perp$	9-10-11	$V \rightarrow SR\perp$
V		

Um reconhecedor *LR* (*k*), correspondente a uma gramática $G = (V, \Sigma, P, S)$ apresenta-se como um conjunto de estados, entre os quais o reconhecedor evolui segundo indica a tabela de análise. Os estados de um reconhecedor *LR* (*k*) caracterizam-se por um conjunto de *itens LR* (*k*), da forma

$$[A \rightarrow \alpha \cdot \beta, \lambda]$$

onde $A \rightarrow \alpha \cdot \beta$ é uma *produção marcada* da gramática, desde que a produção $A \rightarrow \alpha\beta$ esteja em $P \, (\alpha, \beta \in V^*)$.

$\lambda \in \Sigma^*$ é uma cadeia de terminais, de comprimento (máximo) igual a *k*. Esta cadeia deve ser suficiente para que o reconhecedor decida, em qualquer caso, qual deve ser o próximo movimento.

Um conjunto de itens *LR* (*k*) da forma $[A \rightarrow \alpha \cdot \beta, \lambda_i]$ que diferem apenas nas cadeias λ_i, pode ser abreviado como $[A \rightarrow \alpha . \beta, \{\lambda_1, \lambda_2, \ldots, \lambda_n\}]$. Isto significa que as cadeias $\lambda_1, \lambda_2, \ldots, \lambda_n$ são alternativas tais que a presença de qualquer uma delas é suficiente para permitir ao reconhecedor decidir qual deve ser seu próximo movimento, em qualquer situação. Para que seja construído o conjunto de itens *LR* (*k*), correspondente a um dado estado, deve ser feita uma análise do comportamento do reconhecedor naquele estado, de modo que um item da forma $[A \rightarrow \alpha \cdot \beta, \lambda]$ deverá constar no conjunto associado ao estado se e apenas se, para uma cadeia de entrada

$\theta = \theta_1 \theta_2$ ainda não analisada, α estiver contido no topo da pilha, $\lambda = \theta_1$, com $|\lambda| = k$. Neste caso, eventualmente, para alguma cadeia de entrada, deverá ser possível a derivação $\beta \Rightarrow^* \theta$.

Reduções podem ser indicadas, no reconhecedor, para itens em que $\beta = \epsilon$. Neste caso, a cadeia α é reduzida ao não-terminal A se $\theta_1 = \lambda$.

Para a obtenção dos conjuntos de itens, que definem os estados, devem ser aplicados os seguintes passos, para a gramática $G = (V, \Sigma, P, S)$:

(*a*) Cria-se o estado inicial do reconhecedor, ao qual se incorpora o item *LR* (*k*)

$$[S \rightarrow \cdot\, \theta, \epsilon]$$

para cada produção $S \rightarrow \theta$ em *P*.

(*b*) Para cada item da forma $[S \rightarrow \cdot\, \theta, \lambda]$ do estado inicial, se $\theta = A\theta_1$ onde A é um não-terminal, incluir um conjunto de itens $[A \rightarrow \cdot\, \Psi, \mu]$ ao estado inicial para cada produção $A \rightarrow \Psi$ em *P*, e para cada cadeia $\mu \in \Sigma^*$ tal que $\theta_1\, \lambda \Rightarrow^* \mu\, \theta_2$, com $|\mu| = k$. Itera-se o item (*b*) até que não mais sejam acrescentados novos itens.

(*c*) Sendo $[A \rightarrow \alpha \cdot \sigma\beta, \lambda]$ um item *LR* (*k*) de algum estado já existente, onde $\sigma \in V$, cria-se um novo estado para cada σ encontrado nos itens do estado original. Em cada um destes novos estados, incluir os itens $[A \rightarrow \alpha\sigma \cdot \beta, \lambda]$, obtidos dos itens originais pelo deslocamento da marca para a direita do símbolo σ. Isto representa uma transição em que o reconhecedor passa do estado original para o novo estado, consumindo σ. Note-se que, em cada estado, todos os itens inicialmente incluídos referem-se ao mesmo símbolo σ. Caso seja construído algum conjunto já existente, este deverá ser mantido intacto.

(*d*) Para cada item da forma $[A \rightarrow \alpha \cdot B\beta, \lambda]$ de algum estado, onde B é um não-terminal e para cada produção $B \rightarrow \theta$ em *P*, incluir novos itens $[B \rightarrow \cdot\, \theta, \mu]$, em que $\mu \in \Sigma^*$, tal que $\beta\lambda \Rightarrow^* \mu\lambda_1$, com $|\mu| = k$.

Iteram-se os passos (*c*) e (*d*) até que nada mais possa ser alterado.

Os estados que contiverem itens da forma $[A \rightarrow \theta\, \cdot, \lambda]$, deverão indicar uma redução segundo a produção $A \rightarrow \theta$ de *P*.

Para gramáticas *LR* (*k*), demonstra-se que se um item $[A \rightarrow \theta\, \cdot, \lambda]$ ocorrer em um dado estado, então não ocorrerá nenhum outro item $[A' \rightarrow \theta_1 \cdot \theta_2, \lambda]$ no mesmo estado, ou seja, não haverá, em nenhum caso, dúvidas a respeito do movimento a ser utilizado, dada a cadeia λ.

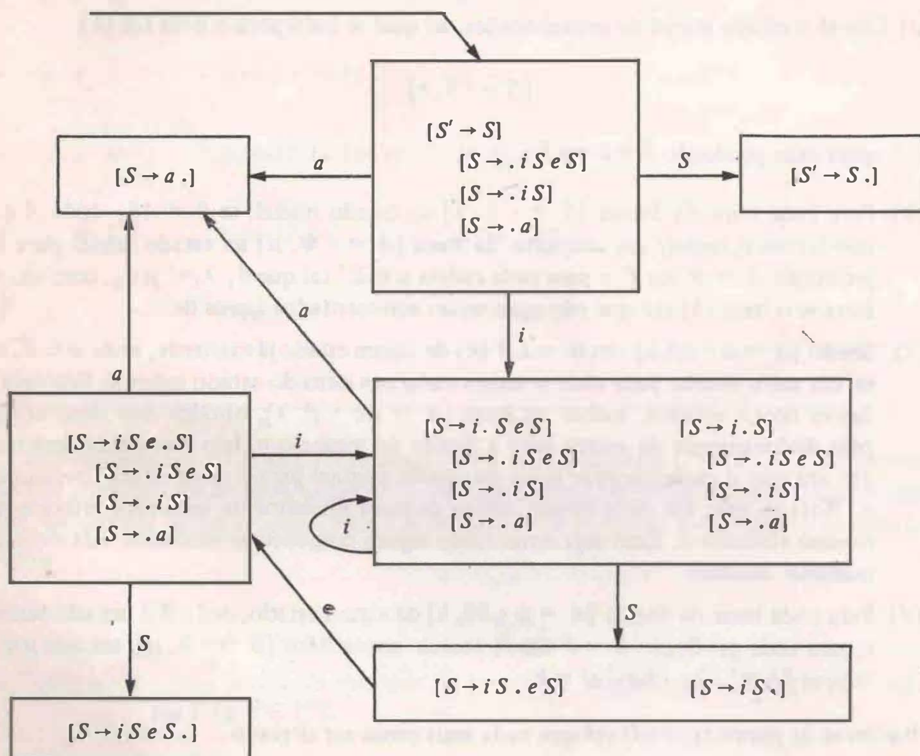
Reconhecedores determinísticos ascendentes são obtidos aplicando-se o método acima. Tais reconhecedores, entretanto, são canônicos, permitindo uma série de melhorias, obtidas através de reduções da tabela de análise que pode ser construída pela aplicação direta do método.

Exemplo:

(Baseado na gramática seguinte, extraída de Aho e Ullman 77)

$$S' \to S$$

$$S \to i\,S\,e\,S \mid i\,S \mid a$$



3.3 – O MAPEAMENTO DE GRAMÁTICAS EM RECONHECEDORES

Nesta seção, é descrita a técnica de mapeamento de descrições formais gramaticais de linguagens de programação, em descrições equivalentes, voltadas ao reconhecimento das linguagens em questão. Estes reconhecedores são de grande utilidade na construção dos compiladores. Não serão estudadas as inúmeras alternativas descritas na literatura, optando-se, ao invés disto, pelo detalhamento de um método único, a ser utilizado, neste texto, nas aplicações à construção de compiladores. Serão tratados dois importantes casos particulares deste método, aplicáveis respectivamente às linguagens regulares e às linguagens livres de contexto. Através destes dois casos, ficam disponíveis ferramentas suficientes para a construção da grande maioria dos compiladores de linguagens de interesse, servindo o reconhecedor como esqueleto do compilador propriamente dito. Procedimentos auxiliares são utilizados, em geral, para completar os reconhecedores em questão em relação a aspectos da dependência de contexto geralmente exibida

pelas linguagens de programação. Tais procedimentos, entretanto, não são incluídos como parte do reconhecedor livre de contexto em que o compilador se apóia, e sim como rotinas anexas aos procedimentos de interpretação do programa e de geração de código objeto.

Nos dois casos, a serem expostos a seguir, a gramática é sempre manipulada na notação de Wirth.

Exemplo:

O exemplo seguinte será utilizado como base para aplicação das regras detalhadas no texto que se segue. Trata-se de uma gramática para expressões aritméticas simplificadas (os operadores possíveis são produtos e subtrações), e são permitidos agrupamentos de subexpressões através de delimitadores $<e>$ *(à guisa de parênteses). Cadeias vazias são consideradas sentenças válidas.*

Este exemplo baseia-se na gramática seguinte:

$S \to E$	$T \to F$	$F \to P$
$E \to \epsilon$	$T \to T - F$	$P \to n$
$E \to T$	$F \to F * P$	$P \to <T>$

É necessário preparar esta gramática para que seja possível utilizá-la na metodologia de construção de reconhecedores a ser exposta.

Note-se que, no caso de linguagens regulares, para as quais se deseja obter um autômato finito a partir da gramática, será necessário eliminar todos os não-terminais que não sejam a raiz da gramática. Em qualquer caso, todas as definições recursivas devem ser eliminadas, e todos os demais não-terminais devem ser substituídos pelas expressões que os definem, repetindo-se este procedimento até que não reste mais nenhum não-terminal na gramática. No caso de linguagens livres de contexto, ou mesmo regulares, para as quais se deseja construir um autômato de pilha, são tolerados não-terminais nas expressões que definem os diversos não-terminais. Estas ocorrências de não-terminais deverão, entretanto, ser preferencialmente limitadas aos não-terminais auto-recursivos centrais, que não podem ser eliminados da gramática. Se a raiz da gramática S for auto-recursiva central, acrescenta-se uma produção $S' \to S$ à gramática, passando S' a ser a nova raiz da gramática.

Uma vez assim preparada a gramática, obtém-se uma definição formal em que os não-terminais são todos definidos através da notação de Wirth.

Exemplo:

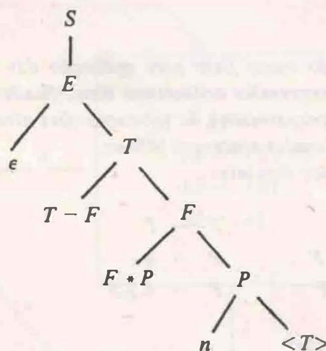
Retornando ao caso da gramática exemplificada acima, o primeiro passo é efetuar o agrupamento das expressões que definem o mesmo não-terminal, obtendo-se:

$$S \to E$$
$$E \to T \mid \epsilon$$
$$T \to F \mid T - F$$
$$F \to P \mid F * P$$
$$P \to n \mid <T>$$

O passo seguinte consiste em determinar os não-terminais auto-recursivos centrais. Para isso, monta-se uma árvore da gramática, pesquisando-se auto-recursões centrais.

Exemplo:

Para a gramática apresentada, pode-se montar a seguinte árvore:



Notar que T, F e P são auto-recursivos pois $T \Rightarrow^* \langle T \rangle$, $F \Rightarrow^* \langle F \rangle$ *e* $P \Rightarrow^* \langle P \rangle$.

Como F e P podem ser expressos em função de T (que encabeça a seqüência, na árvore), pode-se concluir que apenas T seja um não-terminal auto-recursivo central essencial.

Outro não-terminal essencial é obviamente S, a raiz da gramática. Todos os demais não o são, podendo ser eliminados por substituição.

A seqüência da preparação da gramática inclui, a seguir, portanto, a eliminação dos não-terminais que não sejam essenciais, através de substituições sucessivas dos mesmos pelas expressões que os definem, até que seja obtida uma expressão composta apenas de terminais e de não-terminais essenciais.

Exemplo:

Para a gramática acima, deve-se efetuar as manipulações dos não-terminais S e T, já na notação de Wirth:

a) $S = E.$
$S = T \mid \epsilon.$
$S = [T].$

b) $T = T - F \mid F.$

Eliminando-se a recursão de T obtém-se:

$$T = F\{-F\}. \qquad (1)$$

Mas $F = F * P \mid P.$

Eliminando-se a recursão em F nesta expressão obtém-se:

$$F = P \{ * P \}.$$

Substituindo-se em (1) tem-se:

$$T = P \{ * P \} \{ - P \{ * P \} \}.$$

Como $P = n \mid <T>.$, *uma última substituição leva a:*

$$T = (n \mid <T>) \{ * (n \mid <T>) \} \{ - (n \mid <T>) \{ * (n \mid <T>) \} \}.$$

Nesta expressão, T é definido apenas em função do próprio T e de terminais, o que completa a manipulação.

A gramática assim preparada reduz-se, portanto, a:

$$S = [T].$$
$$T = (n \mid <T>) \{ * (n \mid <T>) \} \{ - (n \mid <T>) \{ * (n \mid <T>) \} \}.$$

Partindo-se de uma definição do tipo $A_i = \theta_i.$, na notação de Wirth, em que θ_i é uma expressão envolvendo terminais (e eventualmente não-terminais) e os meta-símbolos de agrupamento, alternância, opcionalidade e iteração, o passo seguinte é o de definir os estados que o reconhecedor deve apresentar. Para isto, as expressões θ_i são inicialmente marcadas com números que identificarão os estados do reconhecedor de acordo com as seguintes regras (a variável j é um contador de estados atribuídos):

(*a*) Ao início de cada cadeia que define as diversas opções de que θ_i é formado, associar o estado 0; fazer $j = 1$.

$$A_i = \underset{0}{\bullet}\ \theta_{i1} \mid \underset{0}{\bullet}\ \theta_{i2} \mid \ldots \mid \underset{0}{\bullet}\ \theta_{in} \bullet$$

(*b*) Para agrupamentos de opções entre parênteses ou colchetes, à esquerda dos quais esteja atribuído previamente o estado x, propagar o estado x para o início de cada uma das cadeias de que é formada cada uma das opções. Atribuir o estado j ao ponto imediatamente à direita do agrupamento, e incrementar j:

$$\underset{x}{\bullet}\ (\underset{x}{\bullet}\ \theta_{i1} \mid \underset{x}{\bullet}\ \theta_{i2} \mid \underset{x}{\bullet} \ldots \mid \underset{x}{\bullet}\ \theta_{in}) \underset{j}{\bullet} \qquad \underset{x}{\bullet}\ [\underset{x}{\bullet}\ \theta_{i1} \mid \underset{x}{\bullet}\ \theta_{i2} \mid \underset{x}{\bullet} \ldots \mid \underset{x}{\bullet}\ \theta_{in}] \underset{j}{\bullet}$$

(*c*) Para agrupamentos de opções entre chaves, à esquerda dos quais tenha sido previamente atribuído o estado x, atribuir ao ponto imediatamente à esquerda de cada uma das cadeias de que é formada cada uma das opções, bem como ao ponto imediatamente à direita do agrupamento, o estado j, e incrementar j:

$$\underset{x}{\bullet}\ \{ \underset{j}{\bullet}\ \theta_{i1} \mid \underset{j}{\bullet}\ \theta_{i2} \mid \underset{j}{\bullet} \ldots \mid \underset{j}{\bullet}\ \theta_{in} \} \underset{j}{\bullet}$$

(*d*) Para terminais ou não-terminais σ isolados que componham as diversas cadeias, e que tenham o estado x previamente atribuído à sua esquerda, atribuir o estado j à sua direita, e incrementar j:

$$\underset{x}{\cdot}\ \sigma\ \underset{j}{\cdot}$$

Estas quatro regras, aplicadas sucessivamente, permitem que sejam atribuídos números aos estados correspondentes a cada ponto entre dois símbolos que compõem as regras gramaticais, expressas na notação de Wirth.

Exemplo:

Aplicando as regras acima ao exemplo que está sendo apresentado, chega-se às seguintes expressões, que voltarão a ser utilizadas em 3.3.2 para ilustrar a construção de autômatos de pilha:

```
S = • [ • T • ] •
    0   0   2   1

T = • ( • n • | • < • T • > • ) • { • * • ( • n • | • < • T • > • ) • } •
    0   0   2   0   3   4   5   1   6   7   7   9   7  10  11  12   8   6

• { • – • ( • n • | • < • T • > • ) •
6  13  14  14  16  14  17  18  19  15

• { • * • ( • n • | • < • T • > • ) • } • } •
15  20  21  21  23  21  24  25  26  22  20  13
```

Uma vez marcadas todas as definições da gramática, de acordo com as regras acima, passa-se à construção propriamente dita dos autômatos correspondentes.

3.3.1 – Obtenção de Autômatos Finitos

De grande importância no estudo da construção dos compiladores, as linguagens regulares são caracterizadas por sua simplicidade e pela facilidade com que são obtidos seus reconhecedores, na forma de autômatos finitos. A grande maioria das linguagens de programação não pode ser modelada exclusivamente através de mecanismos regulares. No entanto, as construções básicas de que são formadas as suas sentenças (identificadores, palavras reservadas, comentários, números decimais, etc.) formam uma linguagem regular, e em geral os compiladores optam por efetuar um processamento prévio do texto-fonte, encarando-o como constituído de uma seqüência de tais elementos básicos regulares antes de iniciar a análise estrutural propriamente dita das sentenças livres de contexto que tais elementos formam.

Uma outra importante aplicação dos autômatos finitos é encontrada na construção de reconhecedores sintáticos ascendentes, nos quais uma das atividades mais freqüentemente

executadas é a da localização de padrões, na cadeia de símbolos contidos na pilha do reconhecedor, em busca de uma cadeia que seja igual ao lado direito de alguma produção da gramática. Esta atividade é executada, via de regra, através do uso de um autômato finito.

Para construir autômatos finitos a partir de gramáticas expressas na notação de Wirth, inicialmente devem ser efetuadas as atribuições de estados, descritas acima, sobre a gramática devidamente preparada como foi mencionado.

A seguir, constrói-se o autômato finito aplicando-se às diversas construções da notação de Wirth as regras seguintes:

(*a*) identifica-se o estado inicial do autômato como sendo o estado marcado com o número 0 na atribuição de estados realizada.

(*b*) identificam-se os estados finais (de aceitação) do autômato como sendo aqueles correspondentes aos números associados aos pontos que seguem (à direita) cada uma das opções θ_k na expressão marcada:

$$A = \underset{a_1}{\theta_1 \ \bullet} \mid \underset{a_2}{\theta_2 \ \bullet} \mid \underset{a_{n-1}}{\ldots \ \bullet} \mid \underset{a_n}{\theta_n \ \bullet}$$

Assim, os estados finais do autômato correspondente à definição acima serão os estados $a_1, a_2, \ldots, a_{n-1}, a_n$.

(*c*) associam-se transições com consumo de átomos σ_k para cada ocorrência de terminais σ_k na expressão marcada. O conjunto de todos os σ_k encontrados corresponderá ao alfabeto Σ de entrada do autômato.

$$\ldots \underset{a_i}{\bullet} \ \sigma_k \ \underset{a_j}{\bullet} \ldots$$

Estas transições partem dos estados a_i marcados à esquerda do terminal σ_k na expressão, e se destinam aos estados correspondentes ao ponto, marcado à direita do terminal σ_k na expressão, com o número a_j. Assim deve-se, para cada uma destas transições, criar uma produção da forma:

$$(a_i, \sigma_k) \rightarrow a_j$$

(*d*) associam-se transições em vazio partindo dos estados correspondentes aos números que seguem cada uma das opções θ_k de um agrupamento entre parênteses ou colchetes na expressão marcada e levando ao estado atribuído ao ponto imediatamente à direita do agrupamento:

$$(\underset{a_1}{\theta_1 \ \bullet} \mid \underset{a_2}{\theta_2 \ \bullet} \mid \underset{a_{n-1}}{\ldots \ \bullet} \mid \underset{a_n}{\theta_n \ \bullet}) \underset{a_j}{\bullet}$$

$$[\underset{a_1}{\theta_1 \ \bullet} \mid \underset{a_2}{\theta_2 \ \bullet} \mid \underset{a_{n-1}}{\ldots \ \bullet} \mid \underset{a_n}{\theta_n \ \bullet}] \underset{a_j}{\bullet}$$

As produções a serem construídas apresentam-se na forma

$$(a_i, \epsilon) \rightarrow a_j \text{ para } i = 1, 2, \ldots, n$$

(*e*) Para agrupamentos entre chaves, associam-se transições em vazio partindo dos estados correspondentes aos números que seguem cada uma das opções θ_k, com destino ao estado (único) que corresponde ao número que marca o início de cada uma destas opções:

$$\underset{a_j}{\{\ \cdot}\ \theta_1 \underset{a_1}{\cdot} \mid \underset{a_j}{\cdot}\ \theta_2 \underset{a_2}{\cdot} \mid \underset{a_j}{\cdot} \cdots \underset{a_{n-1}}{\cdot} \mid \underset{a_j}{\cdot}\ \theta_n \underset{a_n}{\cdot}\ \}$$

As produções correspondentes são da forma

$$(a_i, \epsilon) \rightarrow a_j \qquad \text{para } i = 1, 2, \ldots, n$$

(*f*) Transições em vazio adicionais devem ser incluídas para os agrupamentos entre colchetes ou entre chaves. Estas transições devem levar o autômato do estado correspondente ao número que precede imediatamente o agrupamento para o estado correspondente ao número que o sucede, e devem-se ao fato de que tais construções são opcionais:

$$\underset{a_i}{\cdot}\ [\theta_1 \mid \theta_2 \mid \ldots \mid \theta_n]\ \underset{a_j}{\cdot}$$

$$\underset{a_i}{\cdot}\ \{\theta_1 \mid \theta_2 \mid \ldots \mid \theta_n\}\ \underset{a_j}{\cdot}$$

A transição em vazio em questão é representada pela produção

$$(a_i, \epsilon) \rightarrow a_j$$

Exemplo:

Em FORTRAN, as declarações explícitas de variáveis/matrizes com tipo explícito (inteiro, real, complexo, etc.) assumem a forma definida através da seguinte expressão, em notação de Wirth:

$$D = T\ I\ [\ <N\ \{,\ N\ \}\ >\ ]\ \{,\ I\ [\ <N\ \{,\ N\ \}\ >\ ]\ \}\ \cdot$$

onde T indica o tipo, I simboliza um identificador e N um número inteiro. < e > representam os parênteses (e) respectivamente.

Para este caso, a numeração dos estados é a seguinte:

$$D = \underset{0}{\cdot}\ T\ \underset{1}{\cdot}\ I\ \underset{2}{\cdot}\ [\ \underset{2}{\cdot}\ <\ \underset{4}{\cdot}\ N\ \underset{5}{\cdot}\ \{\ \underset{6}{\cdot}\ ,\ \underset{7}{\cdot}\ N\ \underset{8}{\cdot}\ \}\ \underset{6}{\cdot}\ >\ \underset{9}{\cdot}\ ]\ \underset{3}{\cdot}$$

$$\underset{3}{\cdot}\ \{\ \underset{10}{\cdot}\ ,\ \underset{11}{\cdot}\ I\ \underset{12}{\cdot}\ [\ \underset{12}{\cdot}\ <\ \underset{14}{\cdot}\ N\ \underset{15}{\cdot}\ \{\ \underset{16}{\cdot}\ ,\ \underset{17}{\cdot}\ N\ \underset{18}{\cdot}\ \}\ \underset{16}{\cdot}\ >\ \underset{19}{\cdot}\ ]\ \underset{13}{\cdot}\ \}\ \underset{10}{\cdot}$$

Aplicando as regras de montagem do autômato, tem-se:

a) Estado inicial: 0
b) Estado final: 10

c) *Transições com consumo de átomo:*

(0, T) → 1	(10, ,) → 11
(1, I) → 2	(11, I) → 12
(2, <) → 4	(12, <) → 14
(4, N) → 5	(14, N) → 15
(6, ,) → 7	(16, ,) → 17
(7, N) → 8	(17, N) → 18
(6, >) → 9	(16, >) → 19

d) *transições em vazio de saída de agrupamentos entre parênteses ou colchetes:*

(9, ϵ) → 3 (19, ϵ) → 13

e) *transições em vazio responsáveis pelas iterações (agrupamentos entre chaves):*

(8, ϵ) → 6 (18, ϵ) → 16 (13, ϵ) → 10

f) *transições em vazio responsáveis pelo caráter opcional dos agrupamentos entre chaves ou colchetes:*

(2, ϵ) → 3 (5, ϵ) → 6 (3, ϵ) → 10 (12, ϵ) → 13 (15, ϵ) → 16

Aplicadas as regras acima, o autômato finito está especificado formalmente através do seu conjunto de estados, do conjunto dos terminais, do conjunto das produções construídas, do estado inicial do autômato e do conjunto de seus estados finais. Infelizmente um autômato construído mecanicamente desta maneira não é o melhor nem o mais compacto possível. A teoria dos autômatos finitos, por outro lado, indica que a partir de qualquer autômato finito é possível obter outro que lhe seja equivalente, porém mínimo. Indica também como isto pode ser feito. A seguir será mostrada passo a passo uma técnica de obtenção de tais autômatos ótimos a partir dos autômatos construídos.

Inicialmente, o autômato deve, por razão de simplicidade de manipulação, ser convertido para a notação tabular, ou seja, representado através de tabelas de transições. Cada célula da tabela conterá um conjunto de estados para onde o autômato pode evoluir a partir do estado corrente, através do consumo de um átomo, ou então em vazio (ver exemplo na próxima página).

O objetivo da transformação seguinte é o da obtenção de um autômato determinístico equivalente ao autômato dado, eventualmente não-determinístico. Para isto devem ser eliminadas as transições em vazio e as transições para estados diferentes a partir do mesmo estado e do mesmo átomo consumido (ou em vazio).

A eliminação, em um estado q_i, das transições em vazio para um ou mais estados q_j, se faz incorporando ao estado q_i todas as transições que partem dos estados q_j. Se algum dos estados q_j for estado final, q_i passará a ser estado final. Na tabela de transições, se na linha q_i estiver especificada uma transição em vazio para a linha q_j, então cada célula da linha q_i deverá absorver as transições especificadas na coluna correspondente da linha q_j (é feito um "merge" da linha q_j na linha q_i). Após esta operação, eventuais novas transições em vazio podem aparecer neste estado, devendo ser tratadas sucessivamente até que sejam eliminadas totalmente. Naturalmente, se reaparecer uma transição em vazio para um estado já considerado, não se deverá repetir o procedimento, caso contrário o processo pode não terminar.

Este procedimento deve ser aplicado a cada uma das linhas que apresentem transições em vazio. Uma vez tratados todos os estados, a coluna que especifica transições em vazio deixa de ter sentido, e deve ser eliminada (ver exemplo duas páginas adiante).

Exemplo:

Para o autômato finito construído acima, a tabela de transições correspondente é a seguinte:

estado \ *entrada*	*T*	*I*	<	*N*	>	,	ϵ
0	1						
1		2					
2			4				3
3							10
4				5			
5							6
6					9	7	
7				8			
8							6
9							3
(10)						11	
11		12					
12			14				13
13							10
14				15			
15							16
16					19	17	
17				18			
18							16
19							13

Exemplo:

A aplicação das regras mencionadas para a eliminação das transições em vazio do autômato finito do exemplo leva à obtenção da seguinte tabela:

estado \ entrada	*T*	*I*	<	*N*	>	,
0	1					
1		2				
②			4			11
③						11
4				5		
5					9	7
6					9	7
7				8		
8					9	7
⑨						11
⑩						11
11		12				
⑫			14			11
⑬						11
14				15		
15					19	17
16					19	17
17				18		
18					19	17
⑲						11

O passo seguinte da transformação refere-se à eliminação das transições não-determinísticas especificadas por células da tabela que indiquem mais de um estado destino para transições com o mesmo átomo a partir de um dado estado.

A eliminação destes não-determinismos se faz mediante a criação de estados adicionais para o autômato, e da reinterpretação do conteúdo destas células como sendo o nome do novo estado ao qual dá origem.

(*a*) criar um estado adicional associado a cada um dos conjuntos de transições não-determinísticas especificadas nas células da tabela original.

(*b*) preencher as células das linhas dos estados recém-criados, com o conjunto dos estados destino indicados nas células da mesma coluna de cada um dos estados que compõem o conjunto que deu origem ao estado em questão (ou seja, a linha da tabela de transições correspondente a um novo estado, originado por um conjunto $\{q_1, q_2, \ldots, q_n\}$ de estados-destino, é preenchida pelo "merge" das linhas $q_1, q_2, \ldots, q_n$). Marcar o novo estado como estado final se algum dos q_i for um estado final.

(*c*) iterar os passos anteriores enquanto novos estados forem surgindo.

Exemplo:

Seja a linguagem regular seguinte, definida em notação de Wirth:

L = < { *N*, } *N* > ·

Numerando-se os estados, obtém-se:

L = · < · { · *N* · , · } · *N* · > ·
0 1 2 3 4 2 5 6

Construindo-se a tabela de transições para o autômato finito derivado desta definição segundo as regras estudadas tem-se:

	<	*N*	,	>	ε
0	1				
1					2
2		3,5			
3			4		
4					2
5				6	
⑥					

Eliminando-se as transições em vazio, obtém-se:

	<	*N*	,	>
0	1			
1		3,5		
2		3,5		
3			4	
4		3,5		
5				6
⑥				

Note-se a persistência de transições não-determinísticas com o átomo N nos estados 1, 2 e 4.

Aplicando-se a regra de eliminação deste tipo de não-determinismo, cria-se o estado {3,5}, obtendo-se:

	<	N	,	>
0	1			
1		{3,5}		
2		{3,5}		
3			4	
4		{3,5}		
5				6
⑥				
{3,5}			4	6

A linha correspondente ao estado {3,5} foi obtida pela fusão das linhas correspondentes aos estados 3 e 5.

Como não surgiu nenhum novo não-determinismo no estado criado, então todos os não-determinismos deste autômato foram eliminados.

Após esta operação, nenhuma célula especificará transições não-determinísticas. Porém, com a aplicação deste processo, e com a eliminação de transições em vazio, alguns estados poderão perder sua razão de ser pelo fato de deixarem de ser atingíveis a partir do estado inicial do autômato. Tais estados devem ser eliminados, pois não contribuem para o funcionamento do autômato.

Para a eliminação dos estados inacessíveis, pode-se proceder à marcação sistemática dos estados acessíveis do autômato, descartando-se ao final do processo os estados não marcados:

(*a*) criar duas colunas vazias a mais na tabela, uma para indicar se os estados são acessíveis e outra para indicar se os estados já foram considerados.

(*b*) marcar como considerada e acessível a linha correspondente ao estado inicial do autômato. Marcar como acessíveis todos os estados referenciados nesta linha (estados para onde o autômato pode evoluir a partir do estado inicial).

(*c*) selecionar algum estado marcado como acessível mas não considerado, marcá-lo como considerado, e marcar como acessíveis todos os estados referenciados na linha.

(*d*) repetir o item (*c*) até que não haja estados acessíveis que não tenham sido considerados pelo algoritmo.

(*e*) descartar todos os estados não marcados na tabela (estados inacessíveis).

Exemplo:

Voltando ao caso desenvolvido anteriormente, das declarações do FORTRAN, nota-se que não ocorreram não-determinismos devidos a transições para estados diferentes a partir do mesmo estado e devidas ao mesmo átomo. Por esta razão não há necessidade de se eliminarem tais não-determinismos, podendo-se partir diretamente da tabela resultante após a eliminação das transições em vazio para a eliminação dos estados inacessíveis.

	T	*I*	<	*N*	>	,	*acessível*	*considerada*
0	1						1	2
1		2					3	4
(2)			4			11	5	6
(3)						11		
4				5			7	8
5					9	7	9	10
6					9	7		
7				8			11	12
8					9	7	13	14
(9)						11	11	15
(10)						11		
11		12					7	16
(12)			14			11	17	18
(13)						11		
14				15			19	20
15					19	17	21	22
16					19	17		
17				18			23	24
18					19	17	25	26
(19)						11	23	17

As colunas "acessível" e "considerada" foram preenchidas com números que indicam a ordem em que o preenchimento ocorreu, para facilitar o acompanhamento. Assim, por exemplo, ao ser considerado o estado 2 (indicado como sendo a 6ª operação realizada, na coluna "considerada" do estado 2), os estados 4 e 11, referenciados na linha em questão, foram marcados como acessíveis (indicado como sendo a 7ª operação realizada, na coluna "acessível" dos estados 4 e 11).

Notar que, em alguns casos, como nos estados 5, 8, 12, etc., alguns dos estados referenciados já haviam sido anteriormente marcados como acessíveis, em operações prévias, tendo sido, neste caso, desconsiderados.

Como resultado deste estudo, pode-se concluir que os estados 3, 6, 10, 13 e 16 são inacessíveis, podendo ser eliminados imediatamente.

Eliminados os estados inacessíveis, o autômato obtido será um autômato determinístico equivalente ao autômato original. Este autômato não é obrigatoriamente o mais compacto, uma vez que pode conter estados equivalentes. O passo final na obtenção do autômato ótimo será então o de reduzir o autômato através da minimização de seus estados.

Para isto pode ser aplicado o seguinte algoritmo clássico de minimização:

(*a*) como nenhum estado final pode ser equivalente a um estado não-final, separam-se os estados em dois grupos de acordo com este critério.

(*b*) estados que transitam consumindo átomos diferentes também não podem ser equivalentes. Assim, separam-se em grupos distintos os estados que indiquem transições com átomos diferentes, de tal modo que em cada grupo figurem apenas estados de onde partam transições com o mesmo conjunto de átomos.

(*c*) para cada conjunto assim construído deve-se procurar efetuar novos refinamentos até que sejam obtidos estados isolados no conjunto (neste caso este estado não é equivalente a nenhum outro) ou então até que nenhum dos estados deste conjunto possa ser distinguido de nenhum dos outros estados. Diz-se que um estado q_1 não é distinguível do estado q_2 quando qualquer cadeia de entrada, estando o autômato no estado q_1 ou q_2 respectivamente, levar o autômato sempre para estados indistinguíveis ou iguais. Se alguma cadeia, submetida ao autômato no estado q_1 levar o autômato ao estado q'_1, e submetida ao autômato no estado q_2, o conduzir ao estado q'_2, com q'_1 e q'_2 distinguíveis, então q_1 e q_2 são considerados distinguíveis, e portanto deverão ser separados em conjuntos diferentes.

(*d*) o procedimento deve ser iterado até que não haja novos possíveis refinamentos. Nesta ocasião, cada conjunto de estados conterá uma classe de equivalência, ou seja, todos os estados do conjunto são indistinguíveis para quaisquer cadeias de entrada, e portanto são equivalentes. Rebatizam-se todos os estados de cada conjunto com uma identificação única, e substituem-se todas as referências aos estados rebatizados, pela sua nova identificação. Isto feito, o autômato resultante conterá um estado para cada uma das classes de equivalência montadas neste algoritmo. O autômato resultante é o autômato mínimo equivalente ao autômato inicialmente construído.

Exemplo:

a) Observando-se a tabela resultante da manipulação do exemplo anterior, verifica-se que, a priori, distinguem-se duas classes de estados:

Estados finais: {2, 9, 12, 19}

Estados não-finais: {0, 1, 4, 5, 7, 8, 11, 14, 15, 17, 18}

b) Só estados que transitam com o mesmo conjunto de átomos podem ser eventualmente equivalentes:

Transitam com T apenas: {0}

Transitam com I apenas: {1, 11} .

Transitam com N apenas: {4, 7, 14, 17}

Transitam com , apenas: {9, 19}

Transitam com $<$ e ,: {2, 12}

Transitam com $>$ e ,: {5, 8, 15, 18}

Um primeiro particionamento dos conjuntos dos estados finais e não finais, com base nesta observação, pode ser feito:

Estados finais: {2, 12}, {9,19}

Estados não-finais: {0}, {1, 11}, {5, 8, 15, 18}, {4, 7, 14, 17}

c) Teste de distinguibilidade dos pares de estados de cada conjunto:

{2, 12} serão distinguíveis se {4, 14} o forem, já que ambos transitam para o mesmo estado 11 com o átomo ..

{4, 14} serão distinguíveis se {5, 15} o forem

{5, 15} serão distinguíveis se {9, 19} ou {7, 17} o forem

{9, 19} são indistinguíveis *pois ambos transitam para o mesmo estado 11*

{7, 17} serão distinguíveis se {8, 18} o forem

{8, 18} serão distinguíveis se {9, 19} ou {7, 17} o forem

Mas {9, 19} são indistinguíveis, e sobre {7, 17} nada foi encontrado que permitisse distingui-los. Logo, pode-se considerar que {7, 17} são indistinguíveis *e* {8, 18} são indistinguíveis *também.*

Isto leva a concluir que {5, 15} são indistinguíveis, *o que acarreta que* {4, 14} são indistinguíveis, *e finalmente que* {2, 12} são indistinguíveis.

É necessário verificar ainda se {1, 11}, {4, 7} ou {5, 8} podem ser distinguidos entre si:

{1, 11} serão distinguíveis se {2, 12} o forem, logo {1, 11} são indistinguíveis.

{4, 7} serão distinguíveis se {5, 8} o forem e vice-versa, logo não há como distingui-los. Daí resulta que: {4, 7} são indistinguíveis *e* {5, 8} também são indistinguíveis.

Logo, não havendo mais pares não redundantes a serem testados, as classes de equivalência de estados deste autômato são:

A – {0}

B – {1, 11}

C – {4, 7, 14, 17}

D – {9, 19}

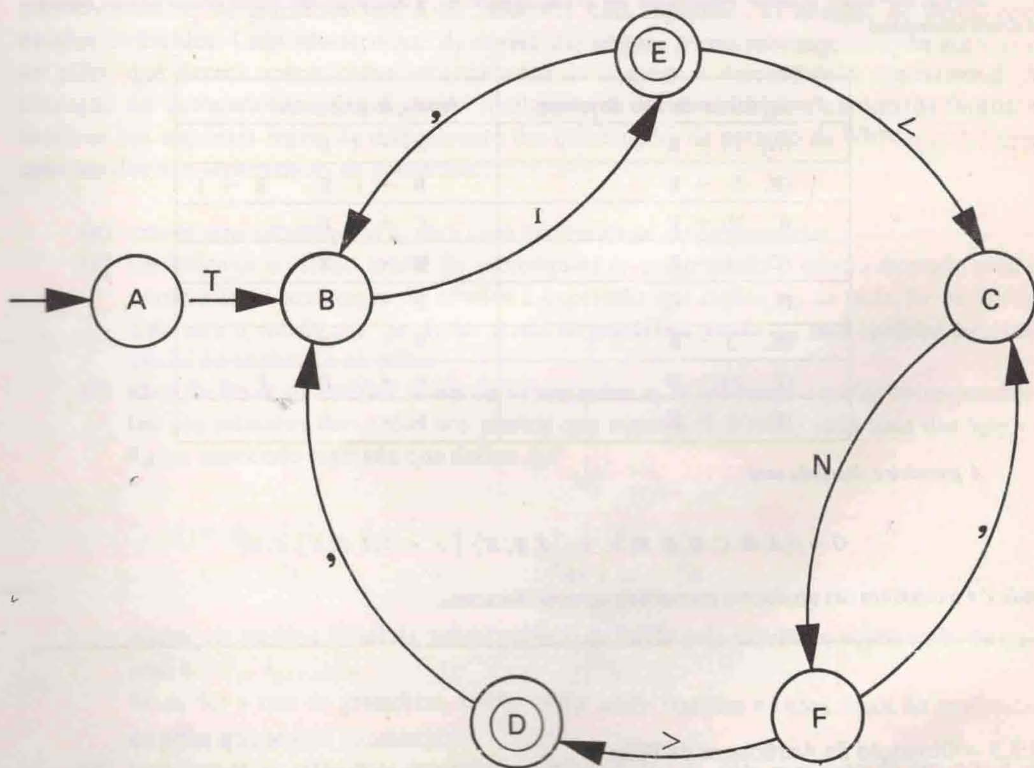
E – {2, 12}

F – {5, 8, 15, 18}

O autômato mínimo poderá, assim, ser representado pela tabela seguinte:

	T	*I*	<	*N*	>	
A	B					
B		E				
C				F		
D						B
E			C			B
F					D	C

O diagrama de estados do autômato mínimo é o seguinte (observe-se que a linguagem reconhecida é a inicialmente proposta):



3.3.2 – O Mapeamento de Autômatos Finitos em Gramáticas Regulares

Para completar este estudo sobre autômatos finitos, cumpre mencionar que, às vezes, dado um autômato finito determinístico já montado, deseja-se construir uma gramática que gere a mesma linguagem que o autômato é capaz de reconhecer. Esta gramática pode ser construída trivialmente aplicando-se as seguintes regras de mapeamento ($\sigma \in \Sigma \; U \{\epsilon\}$):

(*a*) para cada produção do autômato da forma $(q_1, \sigma) \rightarrow q_2$ construir uma produção gramatical de forma $Q_1 \rightarrow \sigma \; Q_2$ onde Q_1 e Q_2 são não-terminais correspondentes aos estados q_1 e q_2 respectivamente, e σ é um terminal.

(*b*) para cada produção da forma $(q_1, \sigma) \rightarrow q_2$, onde q_2 é um estado final do autômato, acrescentar uma produção gramatical da forma $Q_1 \rightarrow \sigma$ à gramática que está sendo construída.

Com a aplicação destas duas regras, obtém-se uma gramática regular que gera a mesma linguagem que o autômato finito original é capaz de reconhecer.

Exemplo:

Para o autômato mínimo construído no exemplo anterior, a aplicação das regras acima leva ao seguinte desenvolvimento:

Produções (extraídas da tabela)	*Produção gramatical*	
(A, T) → B	A → T B	
(B, I) → E	B → I E	B → I
(C, N) → F	C → N F	
(D, ,) → B	D → , B	
(E, <) → C	E → < C	
(E, ,) → B	E → , B	
(F, >) → D	F → > D	F → >
(F, ,) → C	F → , C	

A gramática desejada será:

$$G = (\{A, B, C, D, E, F, >, <, ,, I, T, N\}, \{>, <, ,, I, T, N\}, P, A)$$

onde P é o conjunto das produções gramaticais construídas acima.

3.3.3 – Obtenção de Autômatos de Pilha

O núcleo da maioria dos compiladores, para as linguagens de programação usuais, baseia-se em autômatos de pilha, que reconhecem linguagens livres de contexto que constituem aproximações razoáveis das linguagens de programação a que se destinam os compiladores.

Linguagens livres de contexto caracterizam-se pelas construções aninhadas que apresentam. As gramáticas que geram linguagens livres de contexto devem ser capazes de gerar tais aninhamentos, o que é efetuado através do uso de não-terminais auto-recursivos centrais, os quais não são elimináveis através de transformações gramaticais. Assim sendo, as expressões da notação de Wirth que definem tais não-terminais deverão obrigatoriamente exibir referências a não-terminais da gramática.

A partir de gramáticas expressas na notação de Wirth, envolvendo terminais e não-terminais, não é possível a construção de autômatos finitos para gramáticas que não sejam regulares, tornando-se necessário, sempre que não-terminais figurarem nas expressões que definem a sintaxe de algum não-terminal da gramática, utilizar autômatos de pilha para a construção dos correspondentes reconhecedores, no caso de linguagens não regulares.

Observe-se também que é possível construir autômatos de pilha para linguagens que são, na realidade, regulares, embora isto não seja necessário, uma vez que para tais casos é mais eficiente o uso de autômatos finitos. No entanto, conveniências de implementação podem conduzir o projetista a optar pela solução mais complexa, com base em critérios, por exemplo, de uniformidade de tratamento.

De qualquer forma, tendo-se decidido construir um autômato de pilha, pode-se partir da gramática, expressa na notação de Wirth, devidamente preparada, e na qual tenha sido efetuada uma atribuição de estados de acordo com as regras descritas anteriormente. Cada um dos não-terminais A_i da gramática terá a ele associada uma expressão, na notação de Wirth, com estados atribuídos. Cada não-terminal A_i deverá dar origem a uma submáquina a_i do autômato de pilha, que deverá operar como reconhecedor da linguagem descrita pelo não-terminal. A obtenção do autômato de pilha é muito similar à que foi descrita para autômatos finitos, e baseia-se nas seguintes regras de mapeamento das construções da notação de Wirth que definem cada um dos não-terminais A_i da gramática:

(*a*) cria-se uma submáquina a_i para cada não-terminal A_i da gramática.

(*b*) identifica-se o estado inicial da submáquina a_i como sendo o estado marcado com o número 0 na atribuição de estados à expressão que define A_i na notação de Wirth. Este será o estado q_{i0}. Se A_i for a raiz de gramática, então q_{i0} será também o estado inicial do autômato de pilha.

(*c*) identificam-se os estados finais da submáquina a_i como sendo aqueles correspondentes aos números associados aos pontos que seguem (à direita) cada uma das opções θ_{ik} na expressão marcada que define A_i:

$$A_i = \theta_{i1} \underset{r_1}{\cdot} \mid \theta_{i2} \underset{r_2}{\cdot} \mid \ldots \underset{r_{n-1}}{\cdot} \mid \theta_{in} \underset{r_n}{\cdot}$$

Assim, os estados finais da submáquina a_i definida pela expressão acima serão os q_{ik}, com $k = r_1, r_2, \ldots, r_n$.

Se A_i for a raiz da gramática, então os q_{ik} serão também estados finais do autômato de pilha que se está construindo.

(*d*) associam-se se transições internas, na submáquina a_i, com consumo de átomos σ, a cada ocorrência de terminais σ na expressão marcada que define A_i. O conjunto de todos os σ que forem encontrados na expressão que define A_i formará o conjunto Σ_i. A união de todos os conjuntos Σ_i, para todo A_i da gramática, corresponderá ao conjunto Σ, alfabeto de entrada do autômato de pilha.

$$\ldots \cdot \sigma \cdot \ldots$$
$$r \quad s$$

Estas transições partem dos estados q_{ir} marcados à esquerda do terminal σ na expressão que define A_i, e se destinam aos estados q_{is}, marcados à sua direita. Formalmente, para cada uma destas ocorrências, deve-se criar uma produção que designa um movimento interno em a_i, com consumo do átomo σ:

$$\gamma q_{ir} \sigma \alpha \to \gamma q_{is} \alpha$$

(*e*) associam-se transições internas em vazio, na submáquina a_i, partindo dos estados correspondentes aos pontos marcados à direita de cada uma das opções de um agrupamento entre parênteses, colchetes ou chaves, e com destino ao estado correspondente ao ponto marcado imediatamente à direita do agrupamento:

$$(\theta_{i1} \underset{r_1}{\cdot} \mid \theta_{i2} \underset{r_2}{\cdot} \mid \ldots \underset{r_{n-1}}{\cdot} \mid \theta_{in} \underset{r_n}{\cdot}) \underset{s}{\cdot}$$

$$[\theta_{i1} \underset{r_1}{\cdot} \mid \theta_{i2} \underset{r_2}{\cdot} \mid \ldots \underset{r_{n-1}}{\cdot} \mid \theta_{in} \underset{r_n}{\cdot}] \underset{s}{\cdot}$$

$$\{\theta_{i1} \underset{r_1}{\cdot} \mid \theta_{i2} \underset{r_2}{\cdot} \mid \ldots \underset{r_{n-1}}{\cdot} \mid \theta_{in} \underset{r_n}{\cdot}\} \underset{s}{\cdot}$$

As produções em questão assumem a seguinte forma:

$$\gamma q_{ik} \alpha \to \gamma q_{is} \alpha \qquad \text{com } k = r_1, r_2, \ldots, r_n$$

(*f*) transições internas em vazio devem ser incluídas para os agrupamentos entre colchetes ou entre chaves, especificando movimentos que levam o autômato do estado marcado à esquerda do agrupamento para o estado marcado à sua direita, caracterizando o fato de serem opcionais tais construções.

$$\underset{r}{\cdot} [\theta_{i1} \mid \theta_{i2} \mid \ldots \mid \theta_{in}] \underset{s}{\cdot}$$

$$\underset{r}{\cdot} \{\theta_{i1} \mid \theta_{i2} \mid \ldots \mid \theta_{in}\} \underset{s}{\cdot}$$

As produções que implementam tal movimento assumem a forma:

$$\gamma q_{ir} \alpha \to \gamma q_{is} \alpha$$

Até este ponto, todas as transições internas das submáquinas foram montadas. Para completar a construção do autômato de pilha, restam as transições que especificam movimentos entre duas submáquinas:

(*g*) para cada ocorrência de um não-terminal A_k nas expressões que definem os A_l na notação de Wirth, associar uma transição em vazio de chamada da submáquina a_k.

$$\ldots \underset{r}{\cdot} A_k \underset{s}{\cdot} \ldots$$

A produção que define esta chamada assume a forma:

$$\gamma\, q_{ir}\, \alpha \rightarrow \gamma\, (i, s)\, q_{kO}\, \alpha$$

O conjunto de todos os pares (i, s) construídos nestas produções compõe o alfabeto de pilha Γ do autômato.

(*h*) para cada um dos estados finais q_{ik} da submáquina a_i, identificados no item (*c*) acima, exceto para os estados finais do autômato de pilha, criar produções que especificam transições em vazio de retorno de submáquina:

$$\gamma\, (m, n)\, q_{ik}\, \alpha \rightarrow \gamma\, q_{mn}\, \alpha$$

Os elementos (m, n) do alfabeto de pilha Γ representam estados de retorno para a submáquina a_m.

Para todos os os não-terminais que não sejam a raiz da gramática, γ é irrelevante, e simboliza o conteúdo da pilha do autômato. Para as produções referentes à raiz S da gramática, porém, γ deverá ser substituído por Z_O, a marca de pilha vazia, em todas as suas ocorrências. Isto é válido particularmente porque S não é auto-recursivo central, por construção, e portanto enquanto transitar internamente à submáquina referente a S, a pilha estará certamente vazia.

Exemplo:

Voltando ao exemplo das expressões simplificadas, repete-se a seguir, por comodidade, as expressões numeradas de Wirth, que foram previamente obtidas:

```
S = · [ · T · ] ·
    0   0   2   1

T = · ( · n · | · < · T · > · ) · { · * · ( · n · | · < · T · > · ) · } ·
    0   0   2   0   3   4   5   1   6   7   7   9   7  10  11  12   8   6
    · { · — · ( · n · | · < · T · > · ) ·
    6  13  14  14  16  14  17  18  19  15
    · { · * · ( · n · | · < · T · > · ) · } · } ·
   15  20  21  21  23  21  24  25  26  22  20  13
```

O autômato a ser construído consta de duas submáquinas, correspondentes aos não-terminais S e T, sendo S a raiz da gramática, e portanto a submáquina correspondente será a submáquina principal do autômato.

Submáquina S: *Estado inicial:* q_{SO}
Estado final: q_{S1}
Transições com consumo de átomo: *não há*
Transições em vazio:
saída de agrupamento: $Z_0 q_{S2}\, \alpha \rightarrow Z_0 q_{S1}\, \alpha$
saída sobre agrupamento: $Z_0 q_{SO}\, \alpha \rightarrow Z_0 q_{S1}\, \alpha$
chamada de submáquina: $Z_0 q_{SO}\, \alpha \rightarrow Z_0 (S,2)\, q_{TO}\, \alpha$

Submáquina T. *Estado incial:* q_{TO}
Estado final: q_{T13}
Transições com consumo de átomo: $\gamma q_{Tr}\ \sigma\ \alpha \rightarrow \gamma\ q_{Ts}\ \alpha$

σ =	n	$<$	$>$	*	n	$<$	$>$	−	n	$<$	$>$	*	n	$<$	$>$
r =	0	0	4	6	7	7	11	13	14	14	18	20	21	21	25
s =	2	3	5	7	9	10	12	14	16	17	19	21	23	24	26

Transições em vazio: $\gamma q_{Tr}\ \alpha \rightarrow \gamma q_{Ts}\ \alpha$

saídas de agrupamentos entre parênteses ou colchetes:

r =	2	5	9	12	16	19	23	26
s =	1	1	8	8	15	15	22	22

saídas de agrupamentos entre chaves:

r =	8	22	20
s =	6	20	13

salto sobre agrupamentos entre chaves ou colchetes:

r =	1	6	15
s =	6	13	20

chamadas de submáquina: $\gamma q_{Tr}\ \alpha \rightarrow \gamma (T, s) q_{TO}\ \alpha$

r =	3	10	17	24
s =	4	11	18	25

retorno de submáquina: $\gamma (m, n) q_{Tr}\ \alpha \rightarrow \gamma q_{mn}\ \alpha$

$r = 13 \quad (m, n) \in \{(T, 4),\ (T, 11),\ (T, 18),\ (T, 25),\ (S, 2)\}$

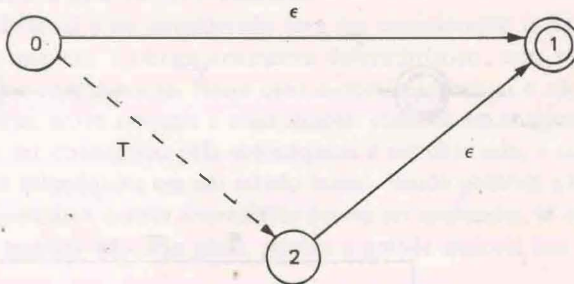
Aplicadas todas as regras acima, o reconhecedor estará totalmente construído, através das produções assim obtidas. Todavia, como no caso dos autômatos finitos, cabe ainda uma simplificação do autômato construído. Em geral não é conveniente procurar efetuar, na prática, minimizações globais destes autômatos. Em lugar disto, a estratégia mais adequada parece ser a de manter a estrutura do autômato construído e minimizar individualmente as submáquinas que o compõem, consideradas como autômatos finitos. Para tanto, transformam-se inicialmente as submáquinas em autômatos finitos de acordo com as seguintes regras de mapeamento:

produções originais de autômatos de pilha	*produções do autômato finito*
$\gamma q_{ir}\ \sigma\ \alpha \rightarrow \gamma q_{is}\ \alpha$	$(q_{ir}, \sigma) \rightarrow q_{is}$
$\gamma q_{ir}\ \alpha \rightarrow \gamma q_{is}\ \alpha$	$(q_{ir}, \epsilon) \rightarrow q_{is}$
$\gamma q_{ir}\ \alpha \rightarrow \gamma (i, s) q_{kO}\ \alpha$	$(q_{ir}, A_k) \rightarrow q_{is}$
$\gamma (m, n) q_{ir}\ \alpha \rightarrow \gamma q_{mn}\ \alpha$	marcar q_{ir} como estado final

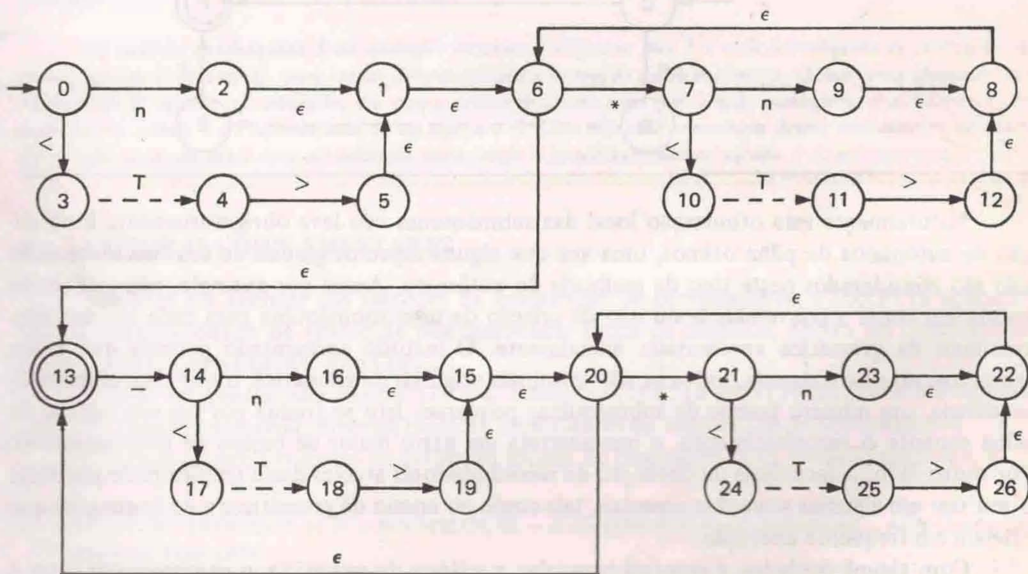
Exemplo:

Aplicando esta transformação para o exemplo que está sendo desenvolvido, e convertendo os autômatos finitos obtidos à forma de diagramas de estados, tem-se:

Submáquina S:



Submáquina T:



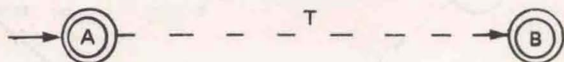
Aplicando-se as mesmas regras que foram estudadas para os autômatos finitos, minimiza-se o autômato finito assim construído, considerando A_k como se fosse um terminal. Após a minimização, as produções do autômato finito mínimo são novamente mapeadas para produções do autômato de pilha, agora livre de não-determinismos, transições em vazio e de estados equivalentes:

produções do autômato finito minimizado	*novas produções do autômato de pilha*
$(q_{ir}, \sigma) \rightarrow q_{is}$	$\gamma q_{ir}\, \sigma\, \alpha \rightarrow \gamma q_{is}\, \alpha$
$(q_{ir}, A_k) \rightarrow q_{is}$	$\gamma q_{ir}\, \alpha \rightarrow \gamma (i, s) q_{ko}\, \alpha$
estado q_{ir} final	$\gamma (m, n) q_{ir}\, \alpha \rightarrow \gamma q_{mn}\, \alpha$

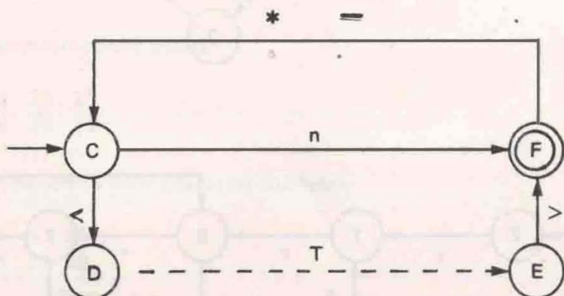
Exemplo:

Analogamente ao que foi efetuado no exemplo de minimização de autômatos finitos, é possível reduzir as duas submáquinas às formas mínimas apresentadas a seguir. Fica para o leitor o exercício de desenvolver em detalhes esta operação:

Submáquina S:



Submáquina T:



Naturalmente esta otimização local das submáquinas não leva obrigatoriamente à obtenção de autômatos de pilha ótimos, uma vez que alguns aspectos globais do seu funcionamento não são considerados neste tipo de melhoria do autômato. Assim por exemplo, não está sendo levada em conta a conveniência ou não da criação de uma submáquina para cada um dos não-terminais da gramática apresentada inicialmente. O método apresentado permite que sejam mantidos, no caso extremo, todos os não-terminais originais da gramática, o que cria, como conseqüência, um número grande de submáquinas pequenas. Isto se traduz por um uso intenso da pilha durante o reconhecimento, o que acarreta um gasto maior de tempo de processamento. Por outro lado, a facilidade de obtenção de reconhecedores através desta técnica pode justificar o seu uso em algumas situações especiais, tais como no ensaio de gramáticas e de linguagens que estejam em freqüente alteração.

Com alguns cuidados, é possível controlar, a critério do projetista, o compromisso entre o número de submáquinas e o tamanho das mesmas, ou seja, o compromisso entre a eficiência do autômato final e a facilidade de sua obtenção. Para tanto, deve-se inicialmente identificar os não-terminais que podem ser eliminados, e aqueles que devem ser obrigatoriamente preservados.

Obviamente o não-terminal que representa a raiz da gramática é um não-terminal essencial, que dará origem à submáquina que contém o estado inicial do autômato de pilha a ser construído.

Os não-terminais auto-recursivos centrais sugerem submáquinas a eles associadas. Entretanto, nem todos são essenciais, uma vez que podem existir conjuntos de tais não-terminais, cujos elementos sejam mutuamente recursivos. Para cada um destes conjuntos, um dos não-terminais apenas é suficiente para originar uma submáquina. Os demais podem ser expressos em função deste não-terminal escolhido. Em geral, o critério mais conveniente para a escolha do não-terminal que deve dar origem a uma submáquina consiste em eleger entre eles o não-terminal

que seja hierarquicamente mais importante na árvore da gramática, ou seja, aquele que, entre os elementos do conjunto, é referenciado em primeiro lugar por iniciativa das submáquinas que utilizam tais não-terminais.

Selecionados todos os não-terminais essenciais, outros podem ainda ser escolhidos, a critério do projetista, para serem preservados. Qualquer acréscimo, porém, certamente implicará em uma redução da eficiência do autômato resultante.

Um aspecto adicional a ser considerado leva em consideração o fato de que o autômato constituído desta forma não é obrigatoriamente determinístico, uma vez que podem ocorrer chamadas opcionais de submáquinas. Neste caso, é possível eliminar o não-terminal do autômato de diversas maneiras, entre as quais a mais simples consiste em analisar o próximo átomo da cadeia de entrada a ser consumido pela submáquina a ser chamada, e compará-lo aos átomos com que transita esta submáquina em seu estado inicial. Sendo possível a transição, a submáquina é chamada, caso contrário, outras alternativas devem ser analisadas, se existirem.

Esta solução, embora não seja geral, resolve a grande maioria dos casos encontrados na prática.

Exemplo:

No caso da submáquina S do exemplo anterior, verifica-se que T é uma submáquina de utilização opcional. Assim, é necessário, para tornar determinística a operação deste autômato, efetuar uma chamada condicional de T, apenas nas situações em que o próximo átomo a ser consumido corresponder aos átomos consumidos no estado 0 da submáquina T, ou seja, n e <. Caso não ocorra nenhum destes dois átomos na cadeia de entrada, a opção vazia deve ser adotada, encerrando o processamento no estado 0 da submáquina S.

3.4 – LEITURAS COMPLEMENTARES

Também no assunto das técnicas de construção de compiladores, a literatura apresenta uma grande variedade de textos de boa qualidade. Entre os clássicos, devem ser considerados Gries (1971), Lewis (1976), Aho (1972).

Ótimas referências são Barrett (1979), Tremblay (1985) e Aho (1986), onde são apresentadas, de maneira acessível, diversas técnicas importantes de construção de compiladores.

Um bom artigo, a respeito da definição formal de linguagens, e da notação correlata, é:

MARCOTTY, M., LEDGARD, H. F. e BOCHMAN, G. – *A Sampler of Formal Definitions* – ACM Computing Surveys, June 1976.

Notações adicionais são descritas em artigos específicos, e geralmente apresentadas como instrumento de definição de uma particular linguagem:

NAUR, P. (editor) – *Revised Report on the Algorithmic Language Algol 60.* Communications of the ACM, Jan. 1963.

van WIJNGAARDEN, A. *et al.* – *Revised Report on the Algorithmic Language Algol 68.* Acta Informatica vol. 5, 1975.

ANSI – American National Standard Programming Language Cobol ANSI X3.23 – 1974, American National Standards Institute, New York, 1974.

ANSI – ANSI Standard Fortran. American National Standards Institute, New York, 1966.

WIRTH, N. – *The Programming Language Pascal.* Acta Informatica, vol. 1, 1971.

ANSI – American National Standard Programming Language PL/I. ANSI X3.53 – 1976, American National Standards Institute, New York, 1976.

A respeito do mapeamento de gramáticas em reconhecedores, descrito neste capítulo, as principais fontes são os artigos já referenciados no Cap. 2:

JOSÉ NETO, J. e MAGALHÃES, M. E. S. – *Reconhecedores Sintáticos – Uma alternativa didática para o uso em cursos de Engenharia.* Anais do XIV Congresso Nacional de Processamento de Dados, S. Paulo, 1981.

JOSÉ NETO, J. e MAGALHÃES, M. E. S. – *Um gerador automático de reconhecedores sintáticos para o SPD*. Anais do VIII SEMISH – Florianópolis, 1981.

Técnicas similares podem ser encontradas nas seguintes referências: Lewis (1979-1982), Lewis (1976) e no artigo:

CONWAY, M. E. – *Design of a Separable Transition-Diagram Compiler.* Communications of lhe ACM, July, 1963.

3.5 – EXERCÍCIOS

As questões 11 a 30 referem-se às gramáticas seguintes:

Gramática G1: (produções)

S → *b D C e*
D → *d; D* | *d;*
C → *C; c* | *C; S* | ε

raiz: *S*

Gramática G2: (BNF)

<S> : : = *i* <C> *e* <S> | *i* <C>
<C> : : = *c* | *i* <C> *e* <C> | ε

raiz: <S>

Gramática G3: (Wirth)

E = *E* + *T* | *T* ·
T = *T* * *F* | *F* ·
F = *a* | (*E*) ·

raiz: *E*

Gramática G4: (Wirth)

I = *aA*.
A = {*a* | *n*}.

raiz: *I*

Gramática G5: (notação do COBOL)

b = {*db*} . . . *B*

o = {*do*} . . . *K*

d = {*dd*} . . . [*D*]

h = *dd* [*dh*] . . . *H*

$$n = \begin{Bmatrix} b \\ o \\ d \\ h \end{Bmatrix}$$

$$db = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \qquad do = \begin{Bmatrix} db \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{Bmatrix} \qquad dd = \begin{Bmatrix} do \\ 8 \\ 9 \end{Bmatrix} \qquad dh = \begin{Bmatrix} dd \\ A \\ B \\ C \\ D \\ E \\ F \end{Bmatrix}$$

raiz: *n*

Gramática G6:

(produções)

$S' \rightarrow S$ — raiz: S'

$S \rightarrow iSeS$

$S \rightarrow iS$

$S \rightarrow a$

Gramática G7:

(produções)

$E \rightarrow E$ **or** E — raiz: E

$E \rightarrow E$ **and** E

$E \rightarrow$ **not** E

$E \rightarrow (E)$

$E \rightarrow a$

$E \rightarrow a\ R\ a$

$R \rightarrow < \mid > \mid = \mid \neq \mid \leq \mid \geq$

1 – Construa uma gramática para representar números reais em notação decimal ou exponencial, com ou sem sinal.

2 – Construa uma gramática para representar cadeias de caracteres entre aspas, nas quais as aspas possam aparecer apenas como primeiro elemento da cadeia. Esta notação deve permitir a concatenação de cadeias, e não deve permitir a expressão de cadeias vazias.

3 – Construa uma gramática para representar identificadores.

4 – Construa uma gramática para representar comentários de linguagens do tipo PL/I ou Pascal: o comentário é uma cadeia de caracteres que não inclua a seqüência */, e que é delimitada à esquerda pela seqüência /*, e à direita por */.

5 – Construa uma gramática para representar a linguagem das expressões regulares.

6 – Represente a metalinguagem BNF em BNF.

7 – Crie uma gramática para representar números de 1 a 999 em algarismos romanos.

8 – Crie uma gramática para representar a estrutura de blocos de uma linguagem como o Pascal ou o Algol.

9 – Crie uma gramática para representar números entre 0 e 1000, escritos por extenso.

10 – Exercite cada uma das notações estudadas, representando as gramáticas dos exercícios 1 a 9 em cada uma delas.

11 – Converta cada uma das gramáticas G1, G2, . . . acima à notação de Wirth, à notação BNF e à forma de expressão regulares (estendidas, se necessário).

12 – Efetue a conversão oposta sobre as gramáticas obtidas no exercício 1. Compare com as gramáticas fornecidas.

13 – Crie regras gerais para a conversão direita entre as notações de Wirth, BNF e expressões regulares, nos dois sentidos.

14 – Converta as gramáticas fornecidas para a notação de diagramas sintáticos.

15 – A notação dos diagramas sintáticos para as gramáticas e a dos diagramas de estados para os reconhecedores guardam entre si inúmeros paralelos. Estude detalhadamente cada aspecto da correspondência entre as duas notações e deduza uma regra para converter uma notação na outra, em ambos os sentidos.

16 – No exercício 15 você criou um meio de obtenção de reconhecedores a partir de gramáticas. Aplique-o às linguagens definidas nas gramáticas fornecidas.

17 – Construa um esboço de um reconhecedor exaustivo para a linguagem definida pela gramática G2. Teste as seguintes cadeias, simulando sua execução.

i c e c
i i c e i c e i c e c e i c
i
i c
i c e i c
i i e i
i i e
i i c e i c
i c e c
c
ϵ

18 – Construa reconhecedores descendentes determinísticos para as linguagens definidas pelas gramáticas G3 e G4. Efetue as transformações que forem necessárias sobre as gramáticas para tornar possível a realização de tais reconhecedores. Teste-os.

19 – Teste as gramáticas G1, G2 e G3 quanto às condições LL(k) e LR(k).

20 – Elimine das gramáticas G1 e G3 as recursões à esquerda através de manipulação gramatical.

21 – Manipule a gramática G2 para eliminar a presença de produções diversas com prefixo comum.

22 – Represente um analisador descendente recursivo para as gramáticas obtidas nos exercícios 20 e 21, através de tabelas de análise.

23 – Construa um reconhecedor ascendente para as linguagens representadas pelas gramáticas G1, G2, G3 e G4. Teste-os com cadeias previamente geradas a partir da gramática correspondente.

24 – Monte as árvores de derivação para as sentenças testadas nos exercícios 18 e 23. Compare-as.

25 – Mapeie as gramáticas G1, G2, G3, G4 e G5 em reconhecedores segundo o método apresentado no Cap. 3.3. Teste os reconhecedores construídos.

26 – Minimize as submáquinas do exercício 25.

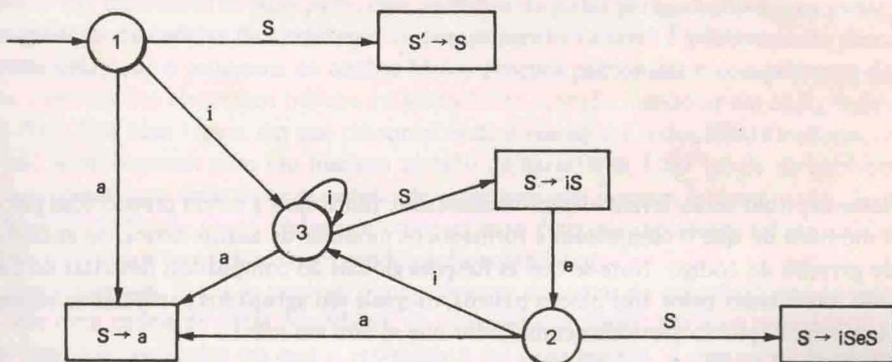
27 – Mostre que para a gramática seguinte:

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow a$

pode-se construir um reconhecedor ascendente LR(0) definido pela seguinte tabela:

entrada / *estado*	*a*	+	*	(	)	⊥	*E*
0	r, 3			r, 2			r, 1
1		r, 4	r, 5			accept	
2	r, 3			r, 2			r, 6
3		a, E → a	a, E → a		a, E → a	a, E → a	
4	r, 3			r, 2			r, 7
5	r, 3			r, 2			r, 8
6		r, 4	r, 5		r, 9		
7		a, E → E + E	r, 5		a, E → E + E	a, E → E + E	
8		a, E → E * E	a, E → E * E		a, E → E * E	a, E → E * E	
9		a, E → (E)	a, E → (E)		a, E → (E)	a, E → (E)	

28 – Construa um reconhecedor LR(0) para a gramática G6. Mostre que este reconhecedor pode ser expresso pelo autômato abaixo:



29 – Manipule a gramática G7, no sentido de eliminar suas ambigüidades. Construa, para a gramática obtida, um reconhecedor descendente recursivo.

4

Especificação das Funções Internas do Compilador

Neste capítulo serão levantadas as necessidades funcionais a serem preenchidas pelos três grandes módulos de que o compilador é formado: os módulos de análise léxica, de análise sintática e de geração de código. Note-se que as funções globais do compilador, descritas no Capítulo 1.5, são executadas pelos três blocos básicos, os quais são agrupados conforme as conveniências da implementação do particular compilador que se tem em mãos.

Não são apresentadas ainda soluções, sendo apenas identificadas as características a que deve satisfazer uma boa solução de cada um dos problemas mais importantes envolvidos no projeto e implementação dos diversos módulos do compilador.

Desta maneira, pretende-se delinear, dentre as ferramentas técnicas apresentadas no Capítulo 2, quais as mais adequadas à construção de cada um dos módulos do compilador, tendo-se como objetivo a obtenção de soluções de alta eficiência e baixo custo. Como esta combinação nem sempre é possível, soluções de compromisso aparecem com freqüência, dando origem à adoção de métodos em que o critério de eficiência predomina nos casos mais críticos, enquanto o critério de simplicidade é mais utilizado na resolução dos problemas ligados às partes do compilador onde não seja essencial que a implementação seja a mais compacta ou rápida possível.

Deve-se entender claramente que as argumentações empregadas e as escolhas efetuadas ao longo do texto não refletem obrigatoriamente soluções ótimas ou opções adotadas universalmente, mas são utilizados como exemplos de como se pode conduzir o processo de projeto técnico de um compilador, através da apresentação de algumas das principais opções que o projetista deve efetuar durante a confecção de um compilador típico.

O texto subseqüente procura apresentar, para cada módulo do compilador, um conjunto significativo de questões a serem consideradas, bem como indicações das opções de solução geralmente encontradas em implementações práticas.

4.1 – ANÁLISE LÉXICA

A análise léxica implementa, como foi mencionado, uma das três grandes atividades desempenhadas pelos compiladores, das quais constitui aquela que faz a interface entre o texto-fonte e os programas encarregados de sua análise e tradução.

Sua missão fundamental é a de, a partir do texto-fonte de entrada, fragmentá-lo em seus componentes básicos, identificando trechos elementares completos e com identidade própria, porém individuais para efeito de análise por parte dos demais programas do compilador.

Uma vez identificadas estas partículas do texto-fonte, estas devem ser classificadas segundo o tipo a que pertencem, uma vez que para o módulo da análise sintática, que deverá utilizá-las em seguida, a informação mais importante acerca destas partículas é a classe à qual pertencem, e não propriamente o seu valor. Do ponto de vista do módulo de geração do código, no entanto, o valor assumido pelos elementos básicos da linguagem é que fornece a informação mais importante para a obtenção do código-objeto. Assim sendo, tanto a classe como o valor assumido pelos diversos componentes básicos da linguagem devem ser preservados pela análise léxica.

Observando uma linguagem qualquer de programação, nota-se imediatamente a não uniformidade das dimensões de suas partículas, inclusive daquelas pertencentes a uma mesma classe. A manipulação de cadeias de caracteres de comprimento variável é relativamente desconfortável, razão pela qual o programa de análise léxica procura padronizar o comprimento das informações contidas nos elementos básicos do texto-fonte, transformando-as em códigos de comprimento fixo. Um caso típico em que tal compressão é essencial é o dos identificadores, os quais, em geral, se apresentam com um número variado de caracteres. Uma tabela de símbolos, neste caso, é utilizada para armazenar as cadeias de caracteres que formam o identificador, e um índice passa a ser utilizado como código de comprimento fixo que represente tal identificador para efeito de análise por parte dos outros módulos do compilador.

Desta maneira, torna-se possível o mapeamento da cadeia de entrada, formada pelo texto-fonte, em uma cadeia de pares de códigos, de comprimento fixo, cada qual responsável por simbolizar uma das partículas em que o texto-fonte foi decomposto. A tais pares de códigos, formados pelas informações de classe e de valor, associados à particula que representam, dá-se o nome de *átomos.*

Em geral, os átomos extraídos do texto-fonte pelo analisador léxico são representados, na metalinguagem que descreve a linguagem-fonte, na mesma forma em que aparecem no texto. Os átomos representam, portanto, os terminais da gramática. Correspondem, ainda, aos elementos do alfabeto de entrada do reconhecedor, através de cujo consumo as transições de estado são promovidas. Do ponto de vista de implementação do programa compilador, o analisador léxico atua, portanto, como uma interface entre o reconhecedor sintático, que forma, em geral, o núcleo do compilador, e o texto de entrada, convertendo a seqüência de caracteres de que este se constitui na seqüência de átomos de que aquele necessita para efetuar suas transições.

4.1.1 – Funções do Analisador Léxico

Para a consecução de seus objetivos, o analisador léxico executa usualmente uma série de funções, não obrigatoriamente ligadas à análise léxica propriamente dita, porém todas de grande importância como infraestrutura para a operação das partes do compilador mais ligadas à tradução propriamente dita do texto-fonte. São discutidas a seguir algumas entre as mais usuais ope-

rações efetuadas pelo analisador léxico, ou por ele comandadas, conjuntamente com a manipulação propriamente dita do texto de entrada.

Extração e Classificação de Átomos – Esta é propriamente a razão de ser do analisador léxico, encarregada de mapear o texto-fonte em outro texto formado pelos átomos que os símbolos componentes do texto-fonte representam. Entre as classes de átomos mais encontradas em analisadores léxicos usuais, destacam-se as seguintes: identificadores, palavras reservadas, números inteiros sem sinal, números reais, cadeias de caracteres ("strings"), sinais de pontuação e de operação, caracteres especiais, símbolos compostos de dois ou mais caracteres especiais, comentários, etc. Na maioria dos analisadores léxicos mais simples, as palavras reservadas são tratadas a parte por rotinas auxiliares do analisador, e os números que não sejam inteiros sem sinal são manipulados ao nível da análise sintática, como seqüências formadas por inteiros sem sinal, sinais de pontuação e sinais de operação. Símbolos compostos também sofrem análise a nível sintático, como se fossem seqüências de sinais isolados.

Eliminação de Delimitadores e Comentários – Importantíssimo no texto-fonte por razões de legibilidade para o programador, os delimitadores, tais como espaços em branco ou símbolos separadores, e os comentários, são totalmente irrelevantes do ponto de vista de geração de código, razão pela qual podem ser eliminados pelo analisador léxico.

Conversão numérica – Cadeias de caracteres que representam números aprentam-se, no texto-fonte, denotando valores numéricos segundo diversas notações possíveis, tais como: inteiros decimais, binários, octais e hexadecimais, inteiros de precisão múltipla, números reais em ponto fixo, em ponto flutuante, e notação científica. Muitos analisadores léxicos encarregam-se de tratar exclusivamente números inteiros, decompondo os demais nas seqüências de números inteiros e sinais de pontuação que os compõem. Outros tratam todos eles, encarregando-se de interpretá-los adequadamente. Em qualquer caso, é encontrada no analisador léxico uma atividade de conversão numérica, através da qual os números, representados externamente em notações diversas, são todos mapeados para a forma interna de representação na qual deverão ser manipulados pelos demais módulos do compilador. Neste mapeamento é resolvido o problema decorrente da existência de cadeias numéricas de comprimentos diversos, escolhendo-se o formato interno de tal modo que o valor correspondente a qualquer número seja representado em um número constante de bits. Uma outra solução, muito utilizada quando isto não for possível, correspondente à criação de *tabelas de constantes numéricas,* onde são organizados. Neste caso, ao invés de mapear os números em seus valores, é possível mapeá-los em códigos que representam índices para esta tabela de constantes numéricas. Na tabela, na posição indicada pelos índices encontram-se os valores verdadeiros, armazenados com o comprimento correto para cada caso, e os índices representam neste caso códigos de comprimento fixo com os quais tais valores passam a ser representados nos átomos pelo analisador léxico.

Tratamento de Identificadores – Outro caso semelhante ao dos números ocorre quando são estudados os identificadores. Identificadores também são cadeias de comprimento variável, não podendo portanto ser utilizados diretamente como parte dos átomos pela dificuldade de manipulação que isto pode acarretar. Assim sendo, surge a necessidade de se efetuar uma compressão na sua representação, o que é feito em geral com o auxílio de uma *tabela de símbolos.* Nesta tabela são armazenadas as cadeias, tais como se apresentam no texto-fonte, em geral com comprimentos não padronizados. Conforme a conveniência, ditada pela linguagem que se está manipulando, a tabela de símbolos reserva para cada símbolo uma área de comprimento cons-

tante, igual ao comprimento máximo dos identificadores aceitos pela linguagem. Isto simplifica muito o tratamento da tabela nos casos em que tal técnica possa ser aplicada. Caso isto não seja viável, devido à não imposição de restrições no comprimento dos identificadores, estes podem ser armazenados por extenso em listas, ficando numa tabela apenas apontadores para os identificadores armazenados nestas listas. Os átomos produzidos pelo analisador léxico para representar os identificadores codificam-nos como um índice para a tabela de símbolos. Estes índices, códigos de comprimento constante, representam, portanto, nos átomos que designam identificadores, as cadeias de comprimento variável de que são constituídas as representações dos identificadores no texto-fonte.

Identificação de Palavras Reservadas – Encontrado um identificador no texto-fonte, é preciso verificar se tal identificador pertence a um conjunto de identificadores especiais, com significado pré-determinado pela linguagem, denominados *palavras-chave* ou *palavras reservadas*. Em contraste com os identificadores usuais, estes não se referem a objetos da linguagem, definidos pelo programador, mas têm seu sentido estabelecido a priori pelo compilador. Em geral, não há nenhuma distinção entre identificadores e palavras reservadas, do ponto de vista formal. Por esta razão, a maioria dos analisadores léxicos efetuam a separação entre estas duas classes de símbolos da linguagem em duas etapas: reconhecem-nos inicialmente como identificadores para, em seguida, verificar se o identificador em questão pertence ao conjunto de palavras reservadas.

Se isto ocorrer, a classe do átomo é alterada para designar a palavra reservada particular em questão.

Em geral, a técnica utilizada para a identificação de palavras reservadas consiste em percorrer uma tabela, uma lista ou uma árvore que represente o conjunto das cadeias de caracteres que formam as palavras reservadas da linguagem. Em alguns compiladores, as palavras reservadas e os identificadores coexistem em uma mesma estrutura de dados, para permitir o uso das mesmas rotinas de pesquisa, sendo que as palavras reservadas são encontradas na estrutura desde o início de compilação, enquanto os demais identificadores são inseridos na mesma à medida que vão sendo definidos ou utilizados.

Por questão de conveniência para o restante do compilador, os analisadores léxicos geralmente associam a cada palavra reservada uma classe separada, que contém apenas aquela palavra reservada particular. Desta maneira, apenas a classe bastaria para fornecer todas as informações que o compilador necessita sobre aquele átomo. Por questões de uniformidade, estes átomos apresentam-se, como todos os outros, como um par de códigos indicando respectivamente a classe e o valor, sendo que ambos carregam o mesmo código em algumas implementações, enquanto em outras o código que representa o valor é irrelevante.

Outras implementações ainda, classificam-nas em uma classe única de átomos, distinguindo-se entre si através do código de valor.

Recuperação de Erros – A ocorrência de caracteres não identificáveis no texto-fonte, bem como a identificação de cadeias de caracteres que não obedecem à lei de formação de nenhuma das classes de átomos que o analisador léxico tem condição de reconhecer, determina a constatação da existência de erros léxicos no texto-fonte.

Para que o analisador léxico possa prosseguir a análise do programa-fonte apesar da perda de sincronismo entre o autômato que implementa o seu reconhecedor e a cadeia de entrada, torna-se necessário criar mecanismos de ressincronização, denominados *mecanismos de recuperação de erros*.

Para erros estruturais, ou seja, aqueles que correspondem ao não cumprimento das regras de formação dos símbolos da linguagem, o analisador léxico procura levar o reconhecedor a algum estado final atingível a partir do ponto em que o erro foi detectado. Isto corresponde a

"inserir" caracteres, como se tais caracteres tivessem sido omitidos do texto-fonte, tendo tal omissão caracterizado a ocorrência do erro. Caso o ponto de detecção de erro seja tal que nenhum caractere havia sido utilizado até o momento como parte de algum símbolo básico, então o caractere em mãos, que não é capaz de promover o início do reconhecimento de nenhum átomo, é, em geral, descartado. Outra opção de recuperação de erros léxicos consiste em, detectado um erro na seqüência de caracteres de entrada, a cadeia de entrada é percorrida à procura de algum delimitador (como, por exemplo, um espaço em branco, um sinal de pontuação, um final de linha, etc.), descartando-se os caracteres intermediários, e levando-se o autômato reconhecedor ao estado de início do reconhecimento de símbolos. Caracteres não identificáveis são, em geral, descartados do texto-fonte.

Em alguns compiladores, as palavras reservadas são marcadas de alguma forma no texto-fonte, de modo que possam ser identificadas. Neste caso, como seu número é reduzido, eventuais erros detectados em sua grafia podem ser, na grande maioria dos casos, corrigidos com uma razoável segurança pelo analisador léxico.

Listagens — Embora não se trate propriamente de uma tarefa de análise, a maioria dos analisadores léxicos, construídos manualmente para linguagens específicas, incorpora entre suas funções auxiliares as atividades ligadas à geração de listagens do texto-fonte. Isto se deve ao fato de ser o analisador léxico a interface entre o compilador e o texto a traduzir, controlando usualmente a leitura física deste texto. Assim sendo, torna-se o módulo do compilador no qual as atividades de listagem podem ser incorporadas de forma natural, evitando a transferência dos fragmentos do texto-fonte para outros módulos.

Geração de Tabelas de Referências Cruzadas — Outra listagem de fácil obtenção quando gerenciada pelas rotinas de análise léxica é uma tabela de referências cruzadas do texto-fonte. O analisador léxico, tendo os átomos e as linhas do texto-fonte em mãos, pode armazenar informações acerca da ocorrência dos símbolos, coletando-as em uma tabela, para que, terminada a análise léxica do texto, possa ser gerada uma listagem indicativa dos símbolos encontrados, com menção à localização de todas as suas ocorrências no texto do programa-fonte.

Definição e Expansão de Macros — Algumas linguagens incorporam, como recurso-padrão, mecanismos através dos quais possam ser criadas abreviaturas, eventualmente paramétricas, na forma de definições de macros. Tais abreviaturas são mencionadas no texto, na forma de chamadas de macros, exigindo um trabalho de substituição, por parte do compilador, de modo que seja obtido um novo texto expandido, isento de definições e de chamadas de macros. Para a execução desta tarefa, muitos compiladores optam pelo uso de um passo inicial, encarregado de efetuar o pré-processamento do texto original, convertendo-o para uma forma, tratável pelo compilador, que não apresenta definições nem chamadas de macros. Outros compiladores executam esta operação por meio de rotinas comandadas pelo analisador léxico. Neste caso, entretanto, há em geral uma comunição entre os analisadores léxico e sintático, no sentido de obter-se, do analisador sintático, dados adicionais sobre a expansão a ser realizada, tais como informações acerca do escopo das macros no texto-fonte.

Interação com o sistema de arquivos — Na qualidade de interface entre o compilador e o programa-fonte, ao analisador léxico cabe, em diversos casos, a tarefa de coordenar a leitura física dos dados no meio externo em que são fornecidos ao compilador. Desta forma, como, em geral, os texto são apresentados para serem traduzidos sob a forma de arquivos, gerenciados pelo sistema operacional, torna-se necessário que o analisador léxico promova, sempre que necessário, o acesso ao arquivo, para que seja efetuada a leitura de um novo registro para análise, sempre que necessário. Isto é uma tarefa relativamente simples, uma vez que o sistema operacio-

nal, no qual o compilador está integrado, via de regra oferece ao usuário do seu sistema de arquivos uma série de operadores de acesso, adequados às diversas atividades a serem solicitadas ao sistema para a manipulação do arquivo em questão.

No caso geral, entretanto, o problema é mais complexo. Caso haja a possibilidade de o programador especificar, através de comandos de controle do compilador, o chaveamento entre diversos arquivos, permitindo deste modo que um texto-fonte seja composto pela justaposição de arquivos, pela inclusão de fragmentos de arquivos em um texto-base, pela omissão de partes de um arquivo, ou por outras operações congêneres, torna-se necessário criar uma interface mais elaborada com o sistema operacional. Através desta interface, são compatibilizados e seqüencializados os acessos aos diversos arquivos de onde devem ser extraídos os registros que compõem o programa fonte, tornando transparente, às rotinas que promovem operações de leitura do texto-fonte, a identidade dos arquivos de onde tais informações foram extraídas.

Compilação Condicional – Ainda através de comandos de controle do compilador, é possível ao programador, em alguns compiladores, parametrizar seus textos-fonte, tornando-os configuráveis às suas necessidades, de modo tal que, fornecidos os parâmetros adequados, através de comandos de controle, o compilador possa compor a versão adequada a partir de arquivos que contenham um texto-matriz único. Isto pode ser efetuado com o auxílio de comandos que permitam a compilação condicional de trechos do texto contido nos arquivos. Mediante o teste do valor assumido pelos parâmetros de compilação, trechos do texto-fonte são compilados, enquanto outros são omitidos. O analisador léxico pode exercer um papel importante neste mecanismo de composição do texto final a ser compilado, através da chamada de rotinas que coordenam o chaveamento de arquivos e a seleção dos trechos de texto a serem utilizados na compilação em cada ocasião. Em geral, as rotinas encarregadas de selecionar as partes do texto-fonte que devem ser compiladas, e descartar as demais, são localizadas entre o analisador léxico básico e as rotinas de acesso aos arquivos que contêm o programa-fonte, operando como um filtro de seleção, controlado por parâmetros definidos pelo programador. Durante o processo de leitura dos comandos de controle em questão, são coletados os parâmetros de interesse, a partir dos quais as rotinas de seleção se encarregam de escolher a fonte adequada para a extração dos átomos pelo analisador léxico.

Controles de Listagens – Entre os comandos de controle do compilador figuram, em geral, aqueles através dos quais o programador tem acesso à seleção dos trechos do texto sobre os quais deseja que as rotinas encarregadas da geração de listagens atuem.

Entre os comandos incluídos nesta classe, figuram em geral aqueles responsáveis por permitir ao programador que ligue e desligue opções de listagem, de coleta de símbolos em tabelas de referência cruzadas, de geração e impressão de tais tabelas, de impressão de tabelas de símbolos do programa compilador, de tabulação e formatação das saídas impressas do programa-fonte ("pretty-printing" ou indentação), etc.

Em geral, o tratamento desta classe de comandos se resume à coleta dos parâmetros que determinam tais atuações, cabendo às rotinas de execução das diversas atividades consultar os parâmetros correspondentes, condicionando-se a efetuar somente as ações relativas àqueles parâmetros indicativos de que o programador tenha feito solicitação neste sentido.

4.1.2 – Considerações sobre a Implementação de Analisadores Léxicos

Os analisadores léxicos são módulos funcionais do compilador, cujas funções são ativadas inúmeras vezes durante o processo de compilação de um texto-fonte. Assim sendo, são tipica-

mente ativados milhares de vezes durante a tradução de um programa: basta notar que cada símbolo, cada número, cada palavra reservada, cada identificador, correspondem ao resultado da execução de uma chamada deste módulo. Assim sendo, e levando-se em conta que o tratamento envolvido na manipulação dos diversos átomos nem sempre é trivial, o processamento da análise léxica apresenta todos os requisitos para tornar-se um gargalo do compilador, no que diz respeito à parcela absoluta do tempo de execução que representam, bem como à porcentagem do tempo total de compilação pela qual são responsáveis. Desta maneira, convém que os analisadores léxicos sejam programas construídos com extremo cuidado, através da aplicação de técnicas que sejam capazes de permitir a obtenção de um programa de alta eficiência. A não observância destes cuidados pode prejudicar seriamente a eficiência do compilador, comprometendo gravemente seu desempenho global.

Por esta razão, convém analisar criteriosamente as operações que o analisador léxico executa com mais freqüência, e buscar técnicas que as implementem com a maior eficiência possível, dentro dos limites de custo toleráveis. Tal análise permite constatar facilmente que todos os tipos de átomos extraídos pelo analisador léxico são representáveis por meio de expressões regulares, ou seja, formam uma linguagem do tipo 3, para a qual, como foi visto, estão disponíveis mecanismos de reconhecimento para os quais a teoria garante uma facilidade de obtenção de formas minimizadas, de desempenho máximo. Desta maneira, a implementação de analisadores léxicos pode ser efetuada com segurança, se fundamentada em autômatos finitos.

Para melhorar o desempenho do analisador léxico, muitas implementações evitam sobrecarregá-lo com tarefas não essenciais. Assim, funções importantes, tais como a recuperação de erros, são eliminadas, sem prejuízo do programa, simplesmente optando-se pelo tratamento sintático, ao invés de léxico, de quaisquer construções que possam eventualmente permitir o aparecimento de erros a nível de análise léxica. Com isto, o analisador léxico nunca deixa de encontrar um átomo, em qualquer parte do texto-fonte, independentemente do histórico de análise. Outras funções, que possam criar sobrecargas desnecessárias na análise léxica, são muitas vezes confinadas em módulos isolados, que efetuam um pré-processamento do texto de entrada, livrando o analisador léxico de controlar as suas tarefas. É o caso do processamento dos comandos de controle do compilador, do tratamento de macros, da formatação das listagens, e da geração de listagens especiais.

Um outro aspecto relevante quando se tecem considerações acerca da implementação de um analisador léxico relaciona-se com o seu papel físico no âmbito do compilador.

Em princípio, o analisador léxico pode ser implementado como um programa independente, que opera como um passo inicial do compilador. Neste caso, o texto-fonte é o seu texto de entrada, e um arquivo intermediário, preenchido com a cadeia de átomos extraídos do texto-fonte, forma a saída deste programa, a qual desempenha o papel de fonte para o módulo de análise sintática, que deverá utilizá-lo como entrada.

Muitas implementações de compiladores optam por realizar o analisador léxico como subrotina ou como co-rotina do analisador sintático. Neste caso torna-se viável que estes dois módulos se comuniquem por meio de parâmetros e de chamadas: em cada ocasião em que o analisador sintático necessita de um átomo adicional, o analisador léxico é chamado ou reativado, ou, em sistemas distribuídos, estimulado através de uma solicitação de novo átomo (por exemplo, através de um envio de mensagem neste sentido).

Preenchem-se então os parâmetros de comunicação, a mensagem de resposta, ou uma área global convencionada, com os códigos que representam o átomo extraído pelo analisador léxico, colocando-se tais códigos à disposição do analisador sintático.

Note-se que, para esta última classe de implementação, não há necessidade de ser criado um arquivo de saída, já que o consumo dos átomos produzidos pela análise léxica é efetuado

concomitantemente com a sua produção. Isto viabiliza a adoção do esquema de compilação em um passo único, muito adequado a uma grande classe de linguagens de alto nível de interesse.

Um aspecto importantíssimo, na especificação de um analisador léxico, refere-se ao fato de que, sendo a lógica deste módulo relativamente trivial, muitas vezes os responsáveis pela construção do compilador tendem a considerá-lo um problema de menor importância, construindo-o sem os necessários cuidados que lhe confiram a eficiência necessária, e canalizando esforços para a elaboração dos outros módulos do compilador, em geral mais atraentes em seus aspectos teóricos ou de implementação. Isto tem, como conseqüência, o efeito de tornar mais lento o compilador, degradando o conjunto, em seu desempenho global, às vezes drasticamente.

Como caminho prático para evitar os problemas mencionados, o projetista, tendo em mente que os analisadores léxicos operam sobre textos, considerando-os, via de regra, como seqüências de cadeias que formam uma linguagem regular, pode optar pela implementação dos analisadores léxicos, com base em autômatos finitos que descrevem tais linguagens regulares. Assim sendo, todas as técnicas conhecidas de otimização dos autômatos finitos podem ser aplicadas, resultando assim um reconhecedor ótimo, ao menos do ponto de vista teórico.

A implementação do modelo formal do autômato finito assim projetado, por sua vez, dá margem à escolha de formas eficientes para a realização do reconhecedor na prática: pode-se, por exemplo, evitar o uso de tabelas de transições interpretadas durante a execução do analisador, uma vez que este tipo de simulação do autômato finito, pela sua natureza, introduz uma ineficiência no programa resultante. Em lugar desta implementação, pode-se, por outro lado, mapear diretamente as tabelas de transições do autômato em instruções de um programa, cujos estados se apresentam em estrita correspondência com os estados do autômato mínimo. Este tipo de implementação é muito mais adequada, e pode, através de uma codificação cuidadosa, conduzir a uma implementação extremamente eficiente.

Do ponto de vista prático, no entanto, a implementação de programas correspondentes aos autômatos ótimos pode acarretar o desenvolvimento de programas de grandes dimensões físicas, embora muito eficientes em matéria de velocidade. Isto é um efeito esperado, e o projetista deve manter-se atento à necessidade de eventuais soluções de compromisso entre o custo e o desempenho do programa. Sendo o analisador léxico indiscutivelmente o mais importante gargalo do compilador, o custo adicional em volume de código é em geral compensado amplamente pela melhora do desempenho global que pode daí ser obtido, sempre que tal custo estiver dentro das tolerâncias da realidade do projetista.

Outrossim, sendo as implementações usuais dos analisadores léxicos, bem como a dos demais módulos que compõem o compilador, baseadas em algoritmos seqüenciais, uma consideração adicional deve ser feita a respeito da simulação daqueles estados do autômato dos quais partem muitas transições: em geral, opta-se por implementar a escolha da transição a ser efetuada por intermédio de um conjunto de testes encadeados, ou então de uma pesquisa em tabela, ou de outra solução semelhante. Todas estas implementações levam a programas cujo tempo de resposta não é fixo, mas depende do conteúdo da cadeia de entrada a ser analisada. Torna-se, assim, impossível, a priori, maximizar a velocidade do programa construído, de forma absoluta. Para melhorar o comportamento integrado do analisador léxico, pode-se, nestes casos, adotar soluções baseadas em levantamentos estatísticos da freqüência com que cada um dos símbolos de entrada aparece no texto-fonte, e ordenar os testes realizados nestes pontos de tal modo de que os símbolos mais prováveis sejam testados em primeiro lugar, aumentando a probabilidade de localização rápida de tais símbolos, o que resulta em um aumento da eficiência do programa resultante. Todavia, disto decorre que soluções ótimas não são possíveis para a implementação dos analisadores léxicos.

Outro aspecto que deve ser considerado, em relação ao estudo da implementação de analisadores léxicos, é que em muitos casos práticos o analisador léxico não se comporta, devido a características da linguagem que o compilador se propõe a tratar, como um módulo estanque e autônomo, interagindo com, no mínimo, um módulo adicional do compilador, em geral o analisador sintático. Isto se deve ao fato de que as regras de extração dos átomos varia, em algumas linguagens, conforme o contexto em que as cadeias de entrada são encontradas no texto-fonte. O analisador léxico não tem, nestes casos, condições para discernir entre os diversos possíveis contextos que a linguagem analisada oferece, devendo então os outros módulos do compilador comunicar ao analisador léxico de que maneira a cadeia de entrada deve ser interpretada.

Um caso clássico em que esta dependência se manifesta ocorre na ocasião da análise de formatos, em linguagens como o FORTRAN, ou na análise de comandos de controle para o compilador. A lista de especificadores de formatos de entrada/saída é analisada segundo regras de separação de átomos totalmente diversas do que ocorre no restante do texto-fonte, exigindo soluções em que uma informação de contexto sintático permaneça à disposição da análise léxica, de modo que as regras adequadas de extração de átomos possam ser aplicadas em cada caso.

Uma das maneiras de resolver este problema consiste em subordinar a execução do analisador léxico a uma informação, proveniente do restante do compilador, que se incumbe de definir o modo de operação da análise léxica. O analisador sintático é o elemento usualmente encarregado de alterar esta informação, quando necessário, impondo, em cada situação, o modo de operação adequado ao analisador léxico, o qual, em função desta informação externa, seleciona as rotinas de análise adequadas ao caso particular correspondente a cada situação. No exemplo mencionado acima, o analisador pode operar em dois modos de funcionamento: o modo normal, válido para todo o programa-fonte, exceto as declarações de formatos, e o modo "formato", em que o analisador se configura para interpretar a linguagem peculiar interna às declarações de formatos.

Um método alternativo, também utilizável para a solução do problema das múltiplas interpretações da cadeia de entrada, utiliza diferentes analisadores léxicos, cada um dos quais destinado a ser executado em um dos modos de operação, correspondente ao contexto em que a parte em análise da cadeia de entrada está inserida.

Localizado, na cadeia de entrada, o contexto referente à ocorrência de cadeias da sublinguagem que exige uma interpretação diferente da seqüência de símbolos de entrada, a análise é chaveada totalmente do analisador correntemente em uso para outro, encarregado de interpretar o texto de entrada de acordo com novas regras de extração. O chaveamento é comandado, em geral, pelas rotinas de análise sintática, às quais é dado o poder de decisão quanto ao tipo de entrada adequado em cada situação, com base em informações estruturais sintáticas do texto-fonte que está sendo analisado.

Implementações diversas destas idéias podem ser efetuadas, através da escolha do modo de realização dos diversos analisadores. Assim, por exemplo, é possível criar co-rotinas de análise léxica, cada qual responsável pela extração dos átomos segundo uma política de interpretação diferente do texto-fonte. Outro modo de programar o analisador consiste em implementar as diferentes regras de extração através de um conjunto de subrotinas, cabendo ao analisador sintático chamar a subrotina adequada a cada caso. Em casos onde a velocidade não é um requisito muito forte, o analisador léxico pode ser implementado como um simulador de autômatos finitos, dirigido pelas tabelas de transição dos autômatos. O chaveamento pode ser realizado, neste caso, mediante uma simples troca da tabela de transições em uso.

Conforme a linguagem a cujo compilador o analisador se destina, nota-se uma variação na política de extração de átomos, anteriormente mencionada. Para algumas linguagens, delimi-

tadores obrigatórios no texto-fonte permitem que a extração dos átomos seja feita mediante uma simples pesquisa de uma cadeia cujo comprimento é determinado de antemão pela presença do delimitador, marcando o final da cadeia a ser pesquisada. Tabelas ou mesmo autômatos podem ser utilizados para a execução da busca em questão.

Em outras situações, os delimitadores não estão disponíveis no texto-fonte, sendo necessário ao autômato analisar além da cadeia propriamente dita, com o objetivo de determinar o final da cadeia que implementa o átomo. Para tais casos, mecanismos de reanálise de partes do texto já lidas se fazem necessárias, para que símbolos, já lidos uma vez para determinar o final de uma cadeia, possam ser lidos novamente, agora como parte integrante do próximo átomo a ser extraído.

Em linguagens como o FORTRAN, por exemplo, ocorrem casos mais significativos de necessidades de reanálise. Como todas as palavras-chave, que introduzem comandos e declarações da linguagem, podem ser utilizadas também como identificadores de variáveis, matrizes e outros objetos da linguagem, torna-se necessário que o analisador léxico efetue uma análise prévia do texto que compõe uma parte substancial da construção que está sendo analisada, para só então decidir se a cadeia em análise se refere a um comando de atribuição iniciado por um identificador soletrado igualmente a alguma das palavras-chave da linguagem, ou se tal cadeia é uma declaração ou comando iniciado por uma destas palavras-chave. A dificuldade de se efetuar tal decisão é agravada pela ausência de delimitadores na linguagem.

Um aspecto do projeto de analisadores léxicos relaciona-se com a sua dependência em relação à linguagem a que se destina. Sendo as linguagens de programação usuais relativamente semelhante, no que se referem à sua linguagem léxica, torna-se atraente construir um analisador léxico que seja o mais possível independente da linguagem a que se destina. Para que o analisador assim construído não interfira nos mecanismos de análise sintática dos compiladores em que for implantado, uma diretriz importante em sua concepção pode estabelecer que a extração dos átomos seja realizada através da técnica da determinação do final da cadeia com base na leitura do símbolo seguinte à cadeia que está sendo extraída.

Outra consideração, acarretada pela adoção da técnica mencionada, é a de que não convém que, em um analisador léxico destinado a mais de um compilador, sejam extraídos átomos cuja sintaxe dê margem à detecção de erros ao nível da análise léxica. Desta maneira, o analisador léxico resultante reconhecerá sempre construções muito simples e extrairá sempre algum átomo do texto de entrada, cabendo ao analisador sintático a verificação de construções mais elaboradas, montadas a partir deste átomos elementares. Isto exige, às vezes, que a análise sintática da linguagem seja adaptada às condições impostas pelo analisador léxico em questão, de modo que todos os seus componentes básicos sejam definidos em função das classes de átomos gerados pelo analisador léxico adotado, ao invés de serem mantidos aqueles determinados pela gramática original da linguagem.

Uma observação complementar acerca da implementação de analisadores léxicos na prática refere-se à viabilidade que existe de sua construção automática. Levando-se em consideração que é perfeitamente possível, e relativamente simples. definir formalmente, de maneira concisa e rigorosa, a transdução que um analisador léxico deve implementar, torna-se claro que a obtenção automática de um programa que implementa o analisador léxico desejado é factível.

Um programa gerador de analisadores léxicos, com base nesta definição formal, pode construir automaticamente tabelas de transição que representam o autômato finito em que o analisador se baseia. Manipulações, também automáticas, deste autômato permitem a obtenção de versões minimizadas, de onde possa ser construído um programa sem nenhuma programação manual, com rapidez e segurança.

Por se tratar de uma tarefa de porte relativamente modesto, a construção de analisadores léxicos em geral não é feita por este caminho, embora ferramentas voltadas para tal elaboração existam e possam ser utilizadas para a obtenção de bons analisadores léxicos sem esforço de programação que não seja o de definir formalmente o analisador desejado.

4.2 – ANÁLISE SINTÁTICA

O segundo grande bloco componente dos compiladores, e que se pode caracterizar como o mais importante, na maioria dos compiladores, por sua característica de controlador das atividades do compilador, é o analisador sintático. A função principal deste módulo é a de promover a análise da seqüência com que os átomos componentes do texto-fonte se apresentam, a partir da qual efetua a síntese da árvore da sintaxe do mesmo, com base na gramática da linguagem-fonte.

A análise sintática cuida exclusivamente da forma das sentenças da linguagem, e procura, com base na gramática, levantar a estrutura das mesmas. Como centralizador das atividades da compilação, o analisador sintático opera, em compiladores dirigidos por sintaxe, como elemento de comando da ativação dos demais módulos do compilador, efetuando decisões acerca de qual módulo deve ser ativado em cada situação da análise do texto-fonte.

O analisador sintático implementa a atividade dos compiladores responsável pela recepção de uma seqüência de átomos, provenientes do texto-fonte, do qual foram extraídos pelo analisador léxico. A partir desta seqüência, o analisador sintático efetua uma verificação acerca da ordem de apresentação dos átomos na seqüência, identificando, em cada situação, o tipo da construção sintática por eles formada, de acordo com a gramática na qual se baseia o reconhecedor.

4.2.1 – Funções da Análise Sintática

A análise sintática engloba, em geral, diversas funções de grande importância:

Identificação de sentenças – antes de tudo, o analisador sintático pode ser visto como um aceitador de cadeias, cujo conjunto forma a linguagem a que se refere o analisador.

Detecção de erros de sintaxe – fornecida ao analisador uma cadeia que não pertença à linguagem a que se refere, o analisador sintático deve identificar a ocorrência, acusando a presença de erros de sintaxe, de preferência através de uma indicação impressa na qual o programador seja informado sobre o ponto de detecção do erro, sobre o tipo de erro detectado, e, eventualmente, sobre a possível causa do erro.

Recuperação de erros – muitos compiladores incorporam, no mecanismo de análise sintática de que dispõem, meios de, uma vez identificadas construções não pertinentes à linguagem, ressincronizar o reconhecedor, de modo que o restante do texto-fonte possa continuar sendo analisado, a despeito da ocorrência do erro em questão. Isto se justifica pela utilidade que traz ao programador, ao permitir que o restante do texto de entrada seja criticado, em busca de todos os erros que o compilador puder detectar.

Correção de erros – alguns compiladores mais sofisticados incluem, no seu mecanismo de recuperação de erros, recursos através dos quais o próprio analisador sintático introduz alterações no texto-fonte incorreto, de tal modo que o texto resultante esteja, ao menos, sin-

taticamente correto. Os mecanismos de correção de erros são geralmente muito empíricos, pela própria natureza dos erros comumente encontrados, o que faz com que, na grande maioria dos casos, a correção efetuada pelo reconhecedor não passe de um artifício de recuperação do erro, já que sua verdadeira correção é geralmente inviável.

Montagem da árvore abstrata da sentença – ao menos conceitualmente, o analisador sintático deveria, com base na gramática, levantar, para a cadeia de entrada, a seqüência de derivação da mesma. Entretanto, em geral a construção desta seqüência, ou seja, da árvore abstrata da sintaxe da sentença, é totalmente dispensável, podendo ser substituída por outras atividades equivalentes, porém menos onerosas.

Comando da ativação do analisador léxico – em muitas implementações, em função do progresso do reconhecimento do texto-fonte, o analisador sintático detecta a necessidade de novos átomos e ativa para tanto o analisador léxico, comandando desta maneira, em função de sintaxe, a operação da análise léxica do texto de entrada.

Comando do modo de operação do analisador léxico – em compiladores de linguagens onde os átomos não se apresentam com formato uniforme em todo o texto do programa, exigindo regras diferentes para sua identificação, conforme o contexto em que são encontrados, cabe em geral ao analisador sintático decidir sobre eventuais chaveamentos do modo de operação da análise léxica, de modo que, conforme o contexto, a rotina adequada de extração de átomos seja utilizada.

Ativação de rotinas da análise referente às dependências de contexto da linguagem – como complemento da análise livre de contexto, que aproxima a linguagem de programação a que se refere o analisador sintático em questão, em geral são necessárias rotinas externas encarregadas da análise sintática dependente de contexto, muitas vezes denominada análise semântica estática (impropriamente, visto serem tais atividades puramente sintáticas). Estas rotinas cuidam da verificação do escopo das variáveis, da coerência de tipos de dados em expressões, do relacionamento entre as declarações e os comandos executáveis, e outras verificações semelhantes.

Ativação de rotinas de análise semântica – estas rotinas são responsáveis pelo gerenciamento dos objetos da linguagem, alocação de áreas para os mesmos, ligação entre os objetos e os comandos que os manipulam. Este gerenciamento é efetuado em tempo de compilação.

Ativação de rotinas de síntese do código objeto – estas rotinas encarregam-se de produzir o código-objeto correspondente ao texto-fonte cuja tradução se deseja, e são o núcleo das atividades semânticas do compilador. Observe-se que as rotinas mencionadas anteriormente são, muitas vezes, fundidas em uma única, que as engloba, e que são conhecidas informalmente como rotinas semânticas, mas que geralmente acumulam atividades não propriamente semânticas do processo de compilação.

O analisador sintático, que executa as tarefas enumeradas acima, implementa, como foi mencionado, o núcleo dos compiladores dirigidos por sintaxe, comandando, desta maneira, a compilação do programa-fonte. Em compiladores organizados de outra forma, as funções selecionadas com o acionamento de rotinas pertencentes a outros módulos comparecem como partes integrantes do analisador sintático, sendo executadas internamente ao mesmo.

Em qualquer caso, entretanto, a tônica da missão dos analisadores sintáticos, em relação ao programa-fonte, nos compiladores de linguagens de alto nível, consiste em efetuar a análise e o reconhecimento das construções sintáticas mais complexas dessas linguagens. Na grande maioria dos casos, trata-se de construções não regulares, definidas através de gramáticas livres de contexto.

Tratando-se, em geral, de linguagens não regulares, a implementação de reconhecedores para as linguagens-fonte dos compiladores via de regra não pode ser efetuada através da utilização exclusiva de autômatos finitos, sendo necessária, na quase totalidade dos casos, a utilização de autômatos de pilha para a realização do reconhecedor básico que implementa o núcleo do analisador sintático.

Ao mesmo tempo que os reconhecedores sintáticos se encarregam de verificar se uma cadeia de entrada pertence ou não ao conjunto de sentenças da linguagem para a qual foram projetados, efetuam também, em muitos casos, a seleção de construções que são relevantes ao restante do processo de compilação. O resultado fundamental desta atividade, sem dúvida, refere-se à obtenção da árvore de derivação do texto-fonte. Em muitas implementações, a árvore não é construída fisicamente, manifestando-se conceitualmente apenas, através da seqüência de derivações ou reduções efetuadas durante o processo de análise.

4.2.2 – Considerações sobre a Implementação de Analisadores Sintáticos

O projeto de um reconhecedor, que deverá servir como núcleo de um analisador sintático, exige que seja efetuado um estudo prévio da linguagem de alto nível a que será destinado, e a conseqüente preparação da gramática que a descreve. Muitas técnicas de construção de reconhecedores exigem que a gramática utilizada exiba algumas propriedades que viabilizem o uso da técnica, o que nem sempre ocorre com a gramática inicialmente fornecida como descrição formal da linguagem.

Em alguns casos, são necessárias alterações na gramática, que permitam a obtenção da mesma linguagem através de produções diferentes das originais. Em outros, a definição original pode mostrar-se inadequada ao uso de determinadas técnicas de obtenção de reconhecedores, pelo simples fato de ser inviável uma eventual transformação gramatical, necessária para converter a gramática original em outra que seja adequada à aplicação da técnica em questão.

De qualquer modo, raramente a obtenção do analisador sintático pode ser feita sem nenhuma alteração da definição formal disponível da linguagem. Para algumas técnicas, a gramática deve ser às vezes alterada de modo que passe a incluir os chamados *pontos de conexão*, ou seja, pontos em que os módulos responsáveis por atividades não sintáticas do compilador devem comunicar-se com o analisador sintático para extrair informações relevantes à compilação do texto de entrada.

Uma vez preparada a gramática, quer por introdução de novas produções, quer por manipulação formal das definições ou por simples mudanças de notação, o método de análise poderá ser determinado, e a construção propriamente dita do reconhecedor pode ser efetuada.

Esta visão do processo de construção de reconhecedores sintáticos caracteriza tal atividade como um trabalho básico de construção de gramáticas adequadas à obtenção de uma determinada classe de reconhecedores. Esta aderência se manifesta fundamentalmente pelo fato de que a gramática construída deve satisfazer às restrições necessárias para a viabilização do reconhecedor sintático em questão.

A etapa de preparação da gramática para a obtenção do reconhecedor é muito importante, pois, durante esta atividade, defeitos e incoerências da gramática e da própria linguagem podem ser detectados, e eliminados, na maior parte da vezes, por meras manipulações das produções mais problemáticas, ou pela restrição do seu uso a casos em que tais efeitos indesejáveis não se manifestem. Outros tipos de problemas da linguagem, tais como ambigüidades semânticas, que possam originar interpretações múltiplas do texto-fonte, entretanto, não podem ser eliminados por alterações simples da gramática, exigindo, muitas vezes, o reprojeto da linguagem.

O mais importante a observar em relação à preparação da gramática é que as linguagens descritas pela gramática original e pela gramática preparada sejam rigorosamente iguais sintática e semanticamente.

Outra importante decisão de projeto deve ser feita quando da construção do reconhecedor sintático: o número de passos de compilação, que deverá reger todo o comportamento do compilador a ser construído, com base no analisador sintático em questão. Algumas linguagens permitem a utilização dos identificadores antes de sua declaração, exigindo, desta maneira, que o texto-fonte seja pesquisado em busca das declarações, para só então tornar-se viável o reconhecimento sintático, nos moldes habituais.

Para viabilizar semelhantes recursos lingüísticos, torna-se conveniente implementar a análise em mais de um passo, de modo que, após a coleta de informações acerca de todos os identificadores do texto, um novo passo de análise passe a receber, do analisador léxico, átomos que incorporam informações obtidas nos passos prévios de análise, a partir das quais as decisões adequadas de análise possam ser efetuadas como nos casos tradicionais de linguagens convencionais.

Uma característica dos métodos de análise ascendente e descendente estudados é que os símbolos da cadeia de entrada, que tenham sido consumidos na análise, podem ser considerados como sendo partes integrantes de construções corretas do texto-fonte, que não necessitarão de nova análise, podendo pois ser descartados. Havendo necessidade de preservar informações sobre tais símbolos, para uso por parte de outros módulos do compilador, ações explícitas nesse sentido devem ser executadas por rotinas ativadas pelo analisador sintático na ocasião do consumo do átomo.

Outra consideração de implementação refere-se aos símbolos da cadeia de entrada utilizados apenas para consulta, com a finalidade de permitir que o analisador decida acerca da produção a ser aplicada, em caso de dúvida. Para implementar este aspecto da construção do reconhecedor, uma das possíveis soluções consiste em manter um grupo de átomos como que à disposição do analisador sintático. Cada vez que um deles (o primeiro) é consumido, o seguinte toma o seu lugar, e o analisador léxico é ativado para preencher a vaga resultante.

Outra técnica, também utilizada em algumas implementações, concentra no analisador léxico esta atividade de manter símbolos já lidos, mas ainda não consumidos. O analisador sintático pode dispor, neste caso, de interfaces com o texto de entrada: uma solicitação de átomo, e uma devolução de átomo. Com a primeira, um átomo é passado para o módulo de análise sintática. Com a segunda, eventuais átomos, que tenham servido apenas para atividades de decisão, retornam, ficando à disposição da interface de solicitação de átomo, para ser entregue ao analisador sintático em sua próxima ativação.

Mais um problema de custo que uma questão técnica é a escolha do tipo de reconhecedor a ser adotado. Isto ocorre devido à existência de meios através dos quais gramáticas podem ser transformadas de modo tal que satisfaçam as restrições impostas pelos vários métodos de análise. A disponibilidade de um programa gerador automático de reconhecedores sintáticos, segundo um determinado tipo de análise, pode ser, neste caso, a chave da decisão acerca do uso daquele tipo de análise. Se tal disponibilidade não ocorrer, então o reconhecedor deverá ser implementado manualmente, e neste caso, certamente os reconhecedores descendentes determinísticos terão preferência pela substancial diferença de complexidade que exibem, em relação aos que utilizam métodos ascendentes.

As implementações de reconhecedores LL(1) e LR(1) são, entre as realizações determinísticas dos autômatos de pilha, aquelas que mais adequadas se mostram em razão de sua viabilidade prática, uma vez que correspondem às formas mais realísticas de reconhecedores aplicáveis

às linguagens usuais, e que apresentam, além disso, uma abrangência aceitável em relação ao espectro de linguagens que são capazes de reconhecer.

Um aspecto fundamental, que deve ser considerado no projeto e implementação de analisadores sintáticos, refere-se à inclusão de mecanismos de recuperação de erros, e de emissão de mensagens de erro claras e adequadas. Muitos mecanismos existem, aplicáveis aos diversos tipos de reconhecedores. Entretanto, na prática, mecanismos de correção e recuperação realmente eficazes são excessivamente complexos, o que tira dos mesmos a aplicabilidade desejada. Em geral, apenas erros simples são tratáveis a baixo custo, envolvendo inserções, remoções e substituições de símbolos do texto de entrada, na tentativa de ressincronização do autômato. Falhando estes, um mecanismo mais agressivo procura descartar parte do texto-fonte, em busca de algum símbolo conveniente que sirva como ponto de referência para a ressincronização do reconhecedor.

4.3 – ANÁLISE SEMÂNTICA E GERAÇÃO DE CÓDIGO

A terceira grande tarefa do compilador refere-se à tradução propriamente dita do programa-fonte para a forma do código-objeto. Em geral, a geração de código vem acompanhada, em muitas implementações, das atividades de análise semântica, responsáveis pela captação do sentido do texto-fonte, operação essencial à realização da tradução do mesmo, por parte das rotinas de geração de código. Em muitos textos encontrados na literatura, é dado o nome de *ações semânticas* a essa classe de tarefas do compilador. O nome deve ser entendido com cautela, uma vez que estas atividades realizam, na prática, toda sorte de operações, necessárias à compilação, e que não estejam compreendidas nos mecanismos das análises léxica e sintática. Nos compiladores usuais, dirigidos por sintaxe, o analisador sintático ativa a execução das ações semânticas sempre que forem atingidos certos estados do reconhecimento, ou sempre que determinadas transições ocorrerem durante a análise do texto-fonte.

Denomina-se, genericamente, *semântica* de uma sentença ao exato significado por ela assumido dentro do texto em que tal sentença se encontra, no programa-fonte. Semântica de uma linguagem é a interpretação que se pode atribuir ao conjunto de todas as suas sentenças. Ao contrário da sintaxe, que é facilmente formalizável, com a ajuda de metalinguagens de baixa complexidade, a semântica, apesar de também poder ser expressa formalmente, exige para isto notações substancialmente mais complexas, de aprendizagem mais difícil e em geral apresentando simbologias bastante carregadas e pouco legíveis. Assim sendo, na grande maioria das linguagens de programação, a semântica tem sido especificada informalmente, via de regra através de textos em linguagem natural.

Não é uma tarefa trivial descrever completamente uma linguagem, ainda que simples, de tal modo que tanto sua sintaxe como sua semântica sejam contidas nesta descrição de maneira completa e precisa. As atividades de tradução, exercidas pelos compiladores, baseiam-se fundamentalmente em uma perfeita compreensão da semântica de linguagem a ser compilada, uma vez que é disto que depende a criação das rotinas de geração de código, responsáveis pela obtenção do código-objeto a partir do programa-fonte.

4.3.1 – Funções das Ações Semânticas do Compilador

Conforme foi mencionado, as ações semânticas encarregam-se, em geral, de todas as tarefas do compilador que não sejam as análises léxica e sintática, sendo sua execução, normalmen-

te, promovida por iniciativa do analisador sintático, em compiladores dirigidos por sintaxe. Várias das funções abaixo relacionadas, que são executadas por muitos compiladores, já foram incluídas entre as atividades secundárias exercidas pelos analisadores léxico e sintático. Neste caso, cabe ao projetista escolher, em função de suas conveniências no projeto, em qual dos grandes blocos do compilador tais funções deverão ser incluídas. Tipicamente, as ações semânticas englobam funções tais como as seguintes:

Criação e manutenção de tabelas de símbolos – esta tarefa é, em geral, executada pelo analisador léxico, porém, em muitos compiladores, por razão de reaproveitamento dos programas de análise léxica e mesmo sintática, os respectivos analisadores são construídos de modo tal que suas funções se limitam estritamente às operações mínimas a que se referem. Desta maneira, todas as atividades secundárias, usualmente executadas por estes módulos, devem ser transferidas para outra parte do compilador, no caso as ações semânticas, que passam então a incorporá-las.

Associar aos símbolos os correspondentes atributos – uma tabela de símbolos é apenas uma coleção dos identificadores encontrados ao longo do texto-fonte. Porém, os identificadores isolados não oferecem ao compilador informações acerca dos objetos que representam. Dessa maneira, torna-se necessário acrescentar, para cada um deles, um conjunto de informações que seja suficiente para caracterizá-lo como sendo correspondente a um determinado objeto, indicando, adicionalmente, todas as características que tal objeto exibe, e que sejam de interesse para o processo de geração de código.

Manter informações sobre o escopo dos identificadores – conforme a linguagem de programação a que se destina, o compilador se depara com o problema da localidade dos objetos, problema esse muito ligado, em geral, à dinâmica do comportamento do programa em tempo de execução. Para dar condições para que o compilador associe os identificadores aos escopos correspondentes, atribuindo-lhes o significado correto em todos os pontos do texto-fonte, torna-se necessário que a tabela de símbolos seja organizada segundo a hierarquia dos escopos a que pertencem os identificadores nela registrados. Em alguns compiladores, o analisador sintático incumbe-se de identificar as fronteiras entre os diversos escopos declarados no texto-fonte, e tal informação é passada ao analisador léxico para que este registre e utilize a informação. Em outros, as ações semânticas associadas às mudanças de escopo promovem a execução de tais operações.

Representar tipos de dados – linguagens modernas oferecem, via de regra, um grande repertório de tipos, em geral representados por agregados homogêneos (tabelas) ou heterogêneos (estruturas). Algumas permitem, inclusive, que o programador especifique seus próprios tipos de dados. Ao compilador cabe, neste caso, a tarefa de registrar as especificações dos diversos tipos de dados, utilizados no programa, bem como empregar tais informações para a elaboração de previsão da utilização de memória, a ser efetuada em época de execução, pelos objetos que forem declarados como sendo do tipo especificado.

Analisar restrições quanto à utilização dos identificadores – em função do contexto em que são empregados, os identificadores devem ou não devem exibir determinados atributos. Cabe ao compilador, através das ações semânticas, efetuar a verificação da coerência de utilização de cada identificador em cada uma das situações em que é encontrado, no texto-fonte.

Verificar o escopo dos identificadores – mediante consulta à informação do escopo em que um identificador está sendo referenciado, o compilador deve executar procedimentos capazes de garantir que todos os identificadores utilizados no texto-fonte correspondam a objetos definidos nos pontos dos programas em que seus identificadores ocorreram.

Identificar declarações contextuais – algumas linguagens permitem, para alguns tipos de objetos, que a sua declaração seja feita de modo implícito, e não através de construções sintáticas específicas. É outra função das ações semânticas do compilador localizar tais identificadores em seu contexto sintático, e associar-lhes atributos compatíveis com tal contexto. Um caso particular desta classe de declarações corresponde aos identificadores pré-declarados, cuja semântica é fixa e faz parte da definição da própria linguagem de programação. Outra situação semelhante ocorre em relação a objetos com valor inicial pré-estabelecido ("default"). Em todos estes casos, o compilador deve apresentar recursos para que a interpretação correta do identificador seja adequadamente efetuada.

Verificar a compatibilidade de tipos – cabe às ações semânticas, ainda, efetuar a verificação do uso coerente dos objetos, que representam os dados do programa, nos diversos comandos de que o programa é composto. Esta verificação ("type checking") é importante como recurso auxiliar para as atividades de geração de código, uma vez que, na maioria das linguagens, a utilização simultânea de dados cujos tipos sejam diferentes, embora coerentes, impõe, ao gerador de código, condições para que seja gerado um código contendo as conversões eventualmente necessárias para permitir que sejam realizadas adequadamente as operações especificadas pelos comandos em que os dados em questão são manipulados.

Efetuar o gerenciamento da memória – o compilador deve incorporar mecanismos através dos quais os requisitos de memória do programa, que está sendo traduzido, sejam calculados, controlados e previstos em tempo de compilação, sempre que possível. Pode-se dividir tais requisitos em estáticos e dinâmicos. Os requisitos estáticos de memória referem-se às necessidades que o programa tem de área de memória, que independam da execução do programa. Os requisitos dinâmicos relacionam-se às necessidades, em geral dependentes dos dados e da execução do programa, que este exibe durante sua operação.

Em geral, os requisitos dinâmicos de memória de um programa decorrem da execução da chamada de procedimentos recursivos, da declaração de matrizes com limites dinâmicos, do gerenciamento de áreas de trabalho ("heap"), e da implementação de ambientes para as variáveis declaradas em escopos diferentes, especialmente no caso de linguagens estruturadas em blocos.

Representar o ambiente de execução dos procedimentos – ao compilador cabe, através das ações semânticas, interpretar corretamente o contexto de execução das rotinas deciaradas no programa. Para tanto, é significativo o mecanismo de passagens de parâmetros (por nome, por valor, por referência, etc.), a maneira através da qual as transferências de parâmetros são realizadas, e os resultados de execução dos procedimentos são transferidos ao contexto chamador. Outro elemento importante a ser considerado neste item é a comunicação eventual com ambientes externos (outros programas, sistema operacional, módulos compilados separadamente, rotinas de pacotes de aplicação, etc.), em que os protocolos de comunicação porventura exigidos devem ser implementados pelos mecanismos do compilador executados pelas rotinas semânticas.

Efetuar a tradução do programa – a principal função das ações semânticas é exatamente a de criar, a partir do texto-fonte, com base nas informações tabeladas e nas saídas dos outros analisadores, uma interpretação deste texto-fonte, expresso em alguma notação adequada. Esta notação não se refere obrigatoriamente a alguma linguagem de máquina, sendo em geral representada por uma linguagem intermediária do compilador.

Implementação de um ambiente de execução – o programa traduzido em geral não é autônomo, exigindo um suporte para a sua execução. Este suporte corresponde, principalmente, a um conjunto de recursos de execução de operações não elementares, e é implementado, na

maioria dos casos, através de um conjunto de rotinas de biblioteca de execução e de uma interface com o sistema operacional. Algumas linguagens exigem, adicionalmente, alguns mecanismos permanentes de gerenciamento de memória, devido às características dinâmicas de seus requisitos de armazenamento. Outras, por seu caráter interativo, exigem um ambiente em que a comunicação com o operador é importante, e que por essa razão é implementada de maneira mais sofisiticada. Em alguns casos, são incorporadas, aos ambientes de execução, ferramentas através das quais podem ser efetuadas medidas de desempenho, ou acompanhada a execução do programa, para efeito de testes e de depuração.

Comunicação entre dois ambientes de execução – deve ser providenciado, adicionalmente, um mecanismo através do qual informações possam ser passadas entre dois ambientes de execução, com a finalidade de permitir que procedimentos se comuniquem entre si (através da passagem de parâmetros), que trechos de programa, pertencentes a escopos diferentes, porém que possam coexistir, tenham ao seu alcance objetos aos quais possuam direitos de acesso, e que processos paralelos (tarefas) possam ser implementados, comunicando-se através de mensagens, ou de objetos globais. Cabe aos procedimentos implementados pelas ações semânticas ter conhecimento exato de tais mecanismos e gerar um código que permita utilizá-los adequadamente em todos os casos.

Geração de código – com base na tradução prévia para uma linguagem intermediária (eventualmente esta tradução é omitida, como ocorre em muitos compiladores), o código-objeto do programa pode ser construído. Em geral, a construção deste código pode ser feita de duas maneiras: ou sem cuidados adicionais, gerando um código que é uma tradução literal da interpretação do programa-fonte (e, portanto, geralmente muito inferior em qualidade ao código equivalente produzido por um programador), ou então, preparado para ser otimizado, caso o compilador tenha uma seção de otimização incorporada. Via de regra, o código-objeto gerado é relocável, para que seja possível ligá-lo a outros módulos desenvolvidos em separado.

Otimização – a maioria dos bons compiladores modernos, mesmo desenvolvidos para máquinas de pequeno porte, estão, cada vez mais, explorando técnicas de otimização com a finalidade de construir automaticamente código de qualidade comparável com o gerado manualmente. Há diversos tipo de otimização, encontrados nos compiladores usuais. A primeira é a otimização independente de máquina, em que manipulações da árvore sintática são efetuadas com a finalidade de reduzi-la, compactá-la e eliminar redundâncias e processamento desnecessário. Nesta classe englobam-se as otimizações de expressões, a eliminação de subexpressões comuns, a "fatoração" do cálculo de subexpressões de uso freqüente, as otimizações das construções iterativas, etc. Outra classe de otimização é a das otimizações dependentes de máquina, em que o hardware em que vai ser executado o programa influi na otimização a ser feita. Nesta classe contam-se: a otimização do uso de registradores, a otimização do uso do conjunto de instruções da máquina, a otimização do tempo de execução do programa, etc. Muitos programas de otimização dependente de máquina operam por transformações diretas do código-objeto gerado previamente.

4.3.2 – Considerações sobre a Implementação das Ações Semânticas

As ações semânticas do compilador são projetadas de tal modo que, executadas por iniciativa do núcleo do compilador, que em geral é baseado em um reconhecedor sintático, executem uma cadeia de operações que promovam, como resultado global, a tradução do programa-fonte para a forma de programa-objeto, bem como a execução de toda a série de outras atividades complementares que integram o compilador. Executam, além, disso, uma importante tarefa no

contexto da estrutura do compilador: a integração entre os grandes módulos do programa, efetuada através da intercomunicação entre os mesmos, por meio da criação e manutenção de estruturas de dados responsáveis pelo armazenamento de todas as informações relevantes ao processo de compilação.

Descrevem-se, a seguir, os principais aspectos relacionados com a realização das diversas funções de compilação, implementadas pelas rotinas semânticas. Cabe frisar novamente que poucas dessas operações são, de fato, semânticas, sendo este termo utilizado neste texto por extensão de conceito, visto ser utilizado em muitas publicações sobre o assunto, embora de maneira não rigorosa.

Tabelas de Símbolos – Uma vez extraídos do texto-fonte pelo analisador léxico, os identificadores são coletados em uma tabela, e a cada um dos identificadores armazenados nesta tabela de símbolos é associado um código único. A cada ocorrência de um identificador no texto do programa-fonte, a tabela de símbolos é consulta ou alterada: toda vez que um objeto da linguagem é declarado, ou seja, toda vez que um identificador é encontrado em uma declaração da linguagem-fonte, ou então em algum contexto em que se configure sua declaração, tal identificador deve ser inserido e definido na tabela de símbolos, naturalmente se já não estiver presente, caso em que uma situação de erro pode ser identificada. Encontrados em outros contextos, os identificadores são interpretados como referências aos objetos previamente definidos através das declarações. Assim sendo, nestas situações a tabela de símbolos é consultada, em busca do símbolo em questão. Se o símbolo consta na tabela, como tendo sido previamente definido, nada é feito além de uma consulta aos seus atributos e, eventualmente, alguma ação acessória, como marcá-lo como referenciado, ou registrar a informação de onde o símbolo foi referenciado, com a finalidade de construir tabelas de referências cruzadas. Se o símbolo não constar na tabela, em geral se configura uma situação de erro, exceto nos casos de compiladores de linguagens que admitem referências a objetos declarados adiante.

A operação de criação e manutenção da tabela de símbolos do compilador é vital para o seu funcionamento, visto ser esta a estrutura de dados mais importante do compilador. Esta importância se faz presente especialmente devido ao fato de que é na tabela de símbolos que são guardados os nomes dos objetos definidos pelo programador, e que são referenciados simbolicamente, por nome, ao longo de todo o programa. As tabelas de símbolos são, em geral, organizadas de um modo tal que reflitam a estrutura do programa-fonte, guardando informações sobre o escopo em que os identificadores são definidos. Assim, nos casos mais simples, as tabelas de símbolos são meras tabelas construídas linearmente, a partir de elementos de comprimento fixo, com política de acesso aleatório para leitura, e crescimento pela extremidade mais recente (fila). A pesquisa em uma tabela desta natureza em geral é feita na forma de busca seqüencial. Em algumas implementações, a linguagem-fonte permite que os identificadores tenham um comprimento máximo muito grande, impedindo, por razões práticas, sua implementação como tabela de elementos homogêneos em comprimento. Para estes casos, uma estruturação na forma de listas de símbolos pode resolver este problema, dando maior flexibilidade à tabela.

A aplicação principal deste tipo de tabelas é feita no caso de linguagens com escopo único.

A coleta do identificador e seu armazenamento na tabela poderiam ser efetuados diretamente pelo analisador léxico. Entretanto, em algumas implementações, tal atividade é retirada do analisador léxico e introduzida nas rotinas semânticas, por questão de clareza do texto do compilador, uma vez que todas as atividades ligadas à tabela de símbolos podem, desta forma, ser agrupadas, ao invés de distribuídas pelos diversos módulos do compilador.

As informações, necessárias à compilação, referentes aos identificadores, não se restringem, no entanto, apenas aos nomes dos objetos. A cada nome estão associados, via de regra,

diversos parâmetros, que funcionam como atributos do objeto a que se refere o identificador. Para armazenar tais informações e associá-las aos identificadores, uma estrutura de dados adicional deve ser mantida: a tabela de atributos.

Uma organização trivial que permite a coexistência das tabelas de símbolos e de atributos na mesma estrutura de dados corresponde a armazenar os atributos em uma tabela (vetor) cujos elementos correspondem um a um aos elementos da tabela de símbolos, qualquer que seja a ordem destes. Esta organização peca, no entanto, pelo fato de as informações e os identificadores ficarem fisicamente separados.

Como alternativa imediata, as tabelas de símbolos podem ser organizadas como listas ligadas. Cada elemento da lista pode conter, desta maneira, não apenas símbolos de comprimento variável, mas também os atributos a eles correspondentes. Há implementações que utilizam uma pilha para armazenar as tabelas. A vantagem de utilizar-se a pilha é a de ser possível o uso da mesma para outras aplicações simultâneas, tais como a construção da árvore abstrata do programa. As diversas estruturas de dados ficam, neste caso, embaralhadas fisicamente, porém, como são listas ligadas, ficam logicamente separadas entre si pela limitação aos acessos, impostos pelos apontadores que as implementam.

O armazenamento, em pilha, das estruturas de dados que implementam a tabela de símbolos, dá margem ao uso deste tipo de organização em compiladores de linguagens que apresentam mais de um escopo. Para tais linguagens, diferentemente do que ocorre com linguagens de escopo único, um identificador pode ser declarado diversas vezes no programa, desde que em escopos diferentes.

Em compiladores de um único passo, o desenvolvimento da tabela de símbolos acompanha a estrutura estática do programa-fonte. Sendo de um único passo, tais compiladores não mais utilizam a parte já analisada do programa-fonte. Dessa forma, uma vez terminada a análise de um trecho de programa, correspondente a um dos escopos por ele definidos, todos os símbolos que nele tenham sido porventura declarados deixam de ser utilizados, podendo ser removidos da tabela de símbolos.

Adicionalmente, pelo fato de um ponto qualquer do programa poder pertencer a mais de um escopo, é possível que em alguma situação o mesmo identificador seja utilizado simultaneamente em mais de um escopo. Isto sugere a utilização múltipla dos identificadores registrados na tabela de símbolos, de modo que o nome dos objetos exiba, como um dos seus atributos, uma indicação do identificador a ele correspondente. Dessa maneira, uma única vez o identificador é armazenado na tabela, sendo daí para diante compartilhado pelos diversos elementos da tabela de atributos, se necessário.

O uso de uma pilha para implementar tal estrutura faz com que sejam, neste caso, coexistentes três estruturas: os identificadores, os atributos e a árvore abstrata. É necessário ainda que sejam marcados, na tabela ou externamente a ela, os limites correspondentes ao agrupamento dos identificadores, segundo o escopo a que pertencem, de modo tal que, terminada a análise do trecho do programa referente a um dado escopo, possam os identificadores correspondentes ser removidos da tabela com facilidade. Nestas circunstâncias, em cada instante da compilação, a tabela de símbolos e atributos contém estritamente as informações relativas aos objetos declarados no escopo a que se refere o elemento que estiver correntemente sendo tratado. Os demais foram oportunamente descartados, ou então ainda não foram inseridos na tabela.

Em compiladores de mais de um passo, a situação se modifica radicalmente, já que se torna necessário preservar toda a informação coletada, para ser utilizada nos passos seguintes de compilação. Por esta razão, a estrutura de dados que implementa a tabela de símbolos deve refletir a estrutura do programa que lhe dá origem. Assim sendo, no primeiro passo da compila-

ção é construída uma árvore estruturalmente correspondente à árvore de encadeamento estático dos escopos definidos pelo programa-fonte. Em geral, só seriam necessários os atributos para os passos seguintes de compilação. Entretanto, alguns compiladores preservam também os nomes dos identificadores, para uso no caso de emissão de mensagens, relativas aos objetos a que se referem os atributos, não em forma codificada, mas usando o identificador em sua forma simbólica.

Outra dificuldade adicional é encontrada no caso de linguagens que permitem que os objetos sejam utilizados antes de sua declaração. Para estas linguagens, as regras de construção da tabela de símbolos devem considerar que as declarações não são as únicas fontes de novos símbolos na tabela, devendo ser considerado o texto onde os símbolos podem ser referenciados como potenciais geradores de novos símbolos, que são inseridos na tabela como símbolos apenas referenciados. Estes símbolos serão posteriormente marcados com o atributo indicativo de símbolo definido, quando da sua ocorrência em um contexto de declaração para tal classe de identificadores. Este tipo de problema se faz sentir nos desvios incondicionais do Algol 60.

Linguagens como o Algol 68, em que identificadores podem ser declarados após a sua utilização, e que também oferecem ao programador recursos de definição de novos tipos de dados e operadores, tornam essencial o uso de mais de um passo de compilação, exatamente para evitar que um identificador seja interpretado incorretamente em algum ponto da compilação. Neste caso, o segundo passo de compilação deve receber, do primeiro passo, todas as informações acerca dos identificadores válidos em cada ponto do programa fonte.

Em qualquer destes dois casos, é imprescindível que, no primeiro passo, ao final do tratamento de um trecho de programa correspondente a um escopo completo, seja efetuado um teste de consistência das tabelas de atributos montadas, para aquele escopo, de tal forma que não sejam transmitidas incoerências para os passos seguintes de compilação.

Algumas implementações incluem, ao início da compilação, uma atividade ligada à inclusão de um conjunto de símbolos reservados, na tabela de símbolos. Em casos como esse, o tratamento dos identificadores, das palavras-chave e dos símbolos pré-declarados é uniforme, distinguindo-se os identificadores de objetos dos demais pelo escopo a que pertencem: os símbolos pré-declarados e as palavras-chave figuram na tabela como se pertencessem a um escopo artificial, que engloba o próprio escopo do programa principal.

Seja qual for a situação, as informações usualmente armazenadas nestas tabelas se resumem, em geral, ao seguinte conjunto: nome do identificador; tipo do objeto a que se refere; endereço de memória correspondente ao objeto em tempo de execução; apontador para um descritor do objeto, quando necessário; informações para a construção de tabelas de referências cruzadas – número da linha em que o objeto foi declarado e lista dos números das linhas onde foi referenciado no programa-fonte; apontador para o próximo identificador, destinado à ordenação alfabética; apontadores de limites de escopo, etc.

Representação dos Objetos – A grande maioria das linguagens de programação oferece ao programador duas classes de tipos de dados: os tipos básicos, representáveis através de elementos abstratos simples (variáveis e constantes inteiras, reais, booleanas, caracteres, etc.) e os tipos derivados, que são agregados homogêneos de elementos de tipo simples (vetores, matrizes, cadeias) ou heterogêneos (estruturas). Os dados são elementos passivos, que armazenam informações de maneira controlada pelo programa. Adicionalmente, são definidos pelo programador objetos de natureza diferente, tais como os operadores, funções, procedimentos, blocos e processos, que são os elementos ativos que executam a lógica do programa e atuam sobre os dados.

Uma terceira classe de elementos componentes dos programas são os tipos. Em algumas linguagens são oferecidas declarações que permitem associar a um identificador uma estrutura

de dados. A estrutura não define um objeto e sim informações sobre a composição de uma classe de objetos. O identificador assim declarado passa a representar o nome desta classe, e pode ser utilizado como se fosse uma das palavras-chave que introduzem declarações de objetos da classe a que se referem.

Há dois aspectos a serem considerados quando se estuda a representação dos objetos da linguagem: a maneira como os objetos devem ser modelados para uso do compilador (representação estática) e a maneira como tais objetos devem ser implementados para que seja possível sua manipulação por parte do programa-objeto (representação dinâmica) durante a execução.

Alguns dos problemas, referentes à representação estática dos objetos, já foram considerados no estudo das tabelas de símbolos: seus nomes, atributos e escopo, que são os elementos associados à representação simbólica dos objetos no texto-fonte, e que são utilizados na comunicação entre o compilador e o programador para referenciar os objetos propriamente ditos.

Objetos primitivos, que são reconhecidos pela própria máquina através de instruções e operadores de hardware capazes de manipulá-los diretamente, exigem do compilador apenas as ações semânticas necessárias à conversão de seu formato externo, utilizado pelo programador, no texto-fonte, para formato interno, diretamente utilizável pela máquina. Parte desta conversão já é efetuada, em muitos compiladores, durante a análise léxica. Cabe mencionar que, embora o mapeamento dos formatos externos para os internos seja unívoca, o inverso não ocorre, e por esta razão, raramente são encontrados textos-objeto a partir dos quais seja possível determinar a forma fonte utilizada para a representação externa dos seus dados. Adicionalmente, máquinas diferentes exibem em geral codificações diferentes para as mesmas informações, bem como uma variação grande na precisão (número de bits) dos dados por elas manipuláveis.

Objetos primitivos de tipo booleano exigem apenas um bit para sua representação. Entretanto, na maioria das implementações, representá-los desta maneira acarreta problemas referentes à alocação de memória, uma vez que as instruções de máquina em geral referenciam palavras completas, exigindo que, para a manipulação do particular bit, sejam efetuadas, na maioria das máquinas, operações adicionais de extração da informação desejada. Por esta razão, e pelo fato de a eficiência do código ficar comprometida, utiliza-se, para a representação das variáveis booleanas, em quase todas as implementações, palavras completas de memória, apesar do gasto adicional que esta prática pode acarretar.

Definidas as representações dos objetos primitivos, e levando em conta que, em alguns casos, há mais de uma representação para o mesmo tipo de objeto (por exemplo, os resultados de comparações entre expressões aritméticas geralmente não são códigos expressos como objetos booleanos, e sim valores aritméticos ou bits de condição, em uma palavra de estado de máquina), torna-se necessário criar operadores de conversão entre tais formatos, bem como operadores responsáveis pela conversão de formato dos dados, capazes de converter tais dados de um para outro tipo primitivo.

Algumas máquinas não possuem, em hardware, instruções capazes de manipular certos tipos de dados menos triviais. É o caso típico dos valores reais, que as máquinas mais simples não manipulam diretamente. Para tais classes de objetos, em geral são desenvolvidos pacotes de rotinas de biblioteca, que integrarão o ambiente de execução do programa objeto, e que são chamadas pelo código objeto sempre que alguma operação sobre tais objetos se fizer necessária. Cabe às ações semânticas identificar tais necessidades e gerar chamadas destas rotinas como se fossem instruções de máquina.

Os objetos que representam agregados de dados homogêneos são os vetores, cadeias e matrizes. As cadeias podem ser consideradas, estrutural e logicamente, semelhantes aos vetores, e são casos particulares de matrizes, do ponto de vista de organização e de acesso.

Uma matriz multidimensional é geralmente implementada através de um vetor, em que os dados pertencentes às diversas colunas que compõem a matriz são armazenados seqüencialmente. A obtenção do índice deste vetor, correspondente a uma ênupla de índices da matriz original, é, via de regra, efetuada através do cálculo de uma *fórmula de linearização.*

Seja uma matriz $M\,[i_1 : s_1, i_2 : s_2, \ldots, i_n : s_n]$ de n dimensões, cujos limites inferiores dos índices são $i_1, \ldots, i_n$ e os correspondentes limites superiores $s_1, \ldots, s_n$.

Seja uma política de alocação de memória, para os elementos que compõem esta matriz, tal que os índices mais à esquerda são fixados, variando-se primeiramente os índices mais à direita. Assim, os elementos de M ocuparão posições sucessivas de memória, na seguinte ordem:

$$M\,[i_1, i_2, \ldots, i_n];\; M\,[i_1, i_2, \ldots, i_n + 1];\; \ldots;\; M\,[i_1, i_2, \ldots, s_n];$$

$$M\,[i_1, i_2, \ldots, i_{n-1} + 1, i_n];\; \ldots;\; M[i_1, i_2, \ldots, i_{n-1} + 1, s_n];$$

$$\vdots$$

$$M\,[i_1, i_2, \ldots, s_{n-1}, i_n];\; \ldots;\; M\,[i_1, i_2, \ldots, s_{n-1}, s_n];$$

$$\vdots$$

$$M\,[s_1, s_2, \ldots, s_{n-1}, i_n];\; \ldots;\; M\,[s_1, s_2, \ldots, s_{n-1}, s_n].$$

O endereço E correspondente a $M\,[j_1, j_2, \ldots, j_n]$ pode ser calculado, supondo-se que o endereço de $M\,[i_1, i_2, \ldots, i_n]$ seja E_0 e que cada elemento da matriz ocupe p posições de memória:

(*a*) calcula-se a amplitude de cada uma das dimensões:

$$a_k = s_k - i_k + 1 \qquad k = 1, \ldots, n$$

(*b*) calculam-se os fatores que representam o número total de posições de memória ocupados pelos elementos de matriz escolhidos variando-se todos os índices à direita daquele que representa cada uma das dimensões:

$$f_k = p * \prod_{m = k+1}^{n} a_m \quad k = 1, \ldots, n-1$$

$$f_n = p$$

(*c*) calcula-se a *origem virtual* E_v da matriz:

$$E_v = E_0 - \sum_{k=1}^{n} f_k * i_k$$

(*d*) calcula-se finalmente o endereço E desejado:

$$E = E_v + \sum_{k=1}^{n} f_k * j_k$$

A representação das matrizes pode ser feita, portanto, através de vetores que as linearizam, e o acesso a seus elementos, executado através do cálculo das fórmulas acima apresentadas. Se as matrizes forem estáticas, ou seja, se os seus índices e amplitudes não variarem durante a execução do programa-objeto, E_v, bem como f_k, podem ser calculados durante a compilação, bastando que seja gerado código para o cálculo de E sempre que houver necessidade de um acesso à matriz.

Durante a execução, porém, muitos compiladores oferecem ao programador recursos automáticos de detecção do uso de índices fora dos limites declarados. Neste caso, assim como nos casos em que a linguagem permite a declaração de matrizes dinâmicas, com dimensões variáveis durante a execução, torna-se necessária a presença de *descritores* para as matrizes.

Descritores são estruturas de dados que contêm informações acerca do objeto a que se referem, informações essas que permitem ao programa-objeto calcular todas as fórmulas acima, com parâmetros não fixados a priori.

Para uma matriz, no caso geral, um possível descritor pode conter: a origem virtual da matriz; os limites superior e inferior de cada índice; a indicação do número de dimensões; os fatores f_k para cada índice. Tal descritor pode, em situações em que a matriz é estática, ser utilizado apenas durante a compilação, como parte dos atributos dos objetos a que se referem, sendo estendidos para a época de execução, como parte do código gerado pelo compilador, no caso de matrizes dinâmicas e no caso em que é necessária a verificação dinâmica dos índices a serem utilizados nas indexações.

Para a representação de vetores e cadeias em linguagens em que agregados mais complexos de natureza matricial não são permitidos, é possível efetuar uma simplificação muito grande nas estruturas de dados que correspondem a seus descritores, bastando que figure sua dimensão (comprimento), seus pares de limites dos índices (se for o caso) e o endereço de sua origem virtual. Para o caso particular das cadeias, em que geralmente não é utilizado o conceito de indexação, apenas a dimensão basta, em geral, para fornecer toda a informação necessária ao acesso desejado.

Um aspecto interessante acerca da implementação de objetos indexados é o da utilização do conceito de *origem virtual*. Os vetores, as colunas ou linhas de uma matriz, e as cadeias podem, geralmente, ser indexados, nas linguagens de alto nível, através de índices cujos limites são números inteiros quaisquer. Em algumas linguagens, mesmo números negativos são permitidos, e, em outras, os limites podem ser declarados em ordem crescente ou descrecente. Para a implementação, por razões de eficiência na indexação, o mais conveniente, em geral, é a utilização de limites crescentes, iniciados em zero. É possível preservar a generalidade imposta pelas linguagens, e ao mesmo tempo obter uma implementação eficiente, explorando-se o conceito de origem virtual, fazendo-se com que os descritores apontem para a posição da memória onde está, ou estaria, localizado o elemento de índice zero da matriz, vetor ou cadeia. Com esta "origem virtual", a indexação é feita diretamente a partir do índice utilizado pelo programador, sem a necessidade de efetuar nenhum cálculo. Naturalmente devem permanecer ativos os mecanismos de proteção contra possíveis erros de indexação, para que as áreas de memória correspondentes a índices ilegais (fora do intervalo declarado) sejam feitas inacessíveis ao programa.

A representação de agregados heterogêneos, tais como as estruturas, envolve dois problemas, a representação dos seus dados na memória e a seleção dos campos de que a estrutura se compõe. Analogamente ao que ocorre no caso das matrizes e vetores, os campos de uma estrutura podem eventualmente ser, eles próprios, objetos agregados, e, além disto, podem apresentar-se ainda como objetos de tamanho não determinável em época de compilação. Desta manei-

ra as estruturas podem ser consideradas como objetos representáveis através de duas partes: uma estática, cuja alocação pode se efetuada durante a compilação, e outra dinâmica, alocável apenas em tempo de execução.

Os agregados heterogêneos são usualmente representados, em tempo de compilação, através de descritores, que são construídos como estruturas de dados que representam grafos de relacionamento entre os elementos que compõem a estrutura. Cada um dos elementos é representado, nesta estrutura de dados, através de um conjunto de atributos, que indicam essencialmente o tipo do elemento, sua posição relativa dentro da estrutura e endereços de alocação na memória, para campos estáticos.

Para campos dinâmicos o compilador não tem condição de obter, em tempo de compilação, informações acerca de suas dimensões, sendo, portanto, impossível nesta ocasião preencher tais campos com os dados adequados. O compilador limita-se, em tais casos, a separar os campos dinâmicos dos estáticos, efetuar a alocação prévia destes e deixar para a época da execução a alocação de memória para os demais, em função das dimensões dinâmicas que tais campos venham a exibir durante a execução.

Cabe às ações semânticas do compilador a tarefa de criar e manter as estruturas de dados que implementam os descritores, bem como efetuar a geração de código, correspondente ao endereçamento de cada um de seus campos, para que seja possível ao programa-objeto efetuar o acesso correto à estrutura. Para tanto, caso alocações dinâmicas se façam necessárias, torna-se imprescindível que o código-objeto esteja preparado para a ativação dos procedimentos adequados, pertecentes ao ambiente de execução onde o programa-objeto irá operar. Além de rotinas próprias para efetuar a alocação de área de memória, o ambiente de execução deverá estar capacitado a preencher os campos do descritor da estrutura que, durante a fase de compilação, não puderam ser preenchidos por falta de informações acerca do tamanho físico dos campos dinâmicos em questão.

Como diretriz geral, portanto, o tratamento das estruturas, em tempo de compilação, resume-se em criar um descritor, preechendo-o com informações acerca da parte estática da estrutura, e reservando áreas para o preenchimento posterior das informações dinâmicas em tempo de execução. O compilador deve também criar os descritores a serem utilizados em tempo de execução pelo programa-objeto através das rotinas, por este chamadas, pertencentes ao ambiente de execução do programa. Com base no descritor construído em tempo de compilação, as ações semânticas têm condições de efetuar a geração do código que seja necessário aos acessos à estrutura, efetuados pelos programa. Testes para a verificação de limites e de proteção contra endereçamentos incorretos são introduzidos no programa-objeto, onde for necessário.

Para a obtenção de um código eficiente, muitos compiladores alteram a ordem em que os campos foram declarados, com a intenção de manter fisicamente próximos os campos da estrutura que tiverem requisitos de memória semelhantes. A finalidade desta alteração na ordem dos campos é a de permitir o melhor aproveitamento de espaço na memória, especialmente nos casos em que o número de campos da estrutura é grande, e a máquina permite diversos tipos de acesso à memória: endereçamento de palavras, bytes, bits, caracteres, etc.

Para o caso da declaração de tipos, são criados descritores semelhantes aos mencionados para as estruturas, em tempo de compilação. Declarações de tipos, no entanto, apenas informam ao compilador acerca das características do novo tipo de objeto que está sendo criado, e não gera alocações de espaço em memória. Declarações de variáveis cujo tipo é um destes novos tipos, declarados pelo programador, provocam a geração de uma área de memória que irá alojar objetos cuja estrutura corresponde àquela declarada no tipo a que se refere.

Compatibilidade de Tipos ("type checking") – Na ocasião da compilação das declarações, os objetos são identificados pelo compilador através dos seus nomes, e registrados nas tabelas de símbolos e atributos. Para objetos estruturalmente complexos, descritores são ainda criados pelo compilador, sendo que alguns são mantidos também durante a execução, caso seja necessário ou conveniente.

Ao serem compilados os demais comandos do programa, são encontradas e identificadas referências aos objetos. Nesta oportunidade, torna-se necessário que seja efetuada uma verificação acerca dos atributos correspondentes aos objetos referenciados, em relação às necessidades locais do particular comando onde tais objetos são utilizados, com base nas declarações em que os mesmos foram definidos. Erros de utilização de tipos podem ser detectados e informados ao programador sempre que houver algum uso indevido dos mesmos.

Nem sempre uma discrepância entre o esperado e o encontrado corresponde, entretanto, a um erro. O caso mais comum em que isto ocorre é, sem dúvida, a mistura de tipos numéricos em expressões nos comandos de atribuição de valor. Para semelhantes situações, muitas vezes os compiladores adotam uma hierarquia entre os tipos, de modo que, se em uma operação binária for encontrado um objeto de hierarquia maior que a do outro, em relação ao seu tipo, uma conversão de tipo deve ser efetuada sobre o objeto de menor hierarquia, de modo que a operação binária possa ser realizada no tipo de maior hierarquia.

Estas operações de conversão automática, às vezes denominadas *coerções*, implicam em uma geração de chamadas de rotinas especiais de conversão de formato, pertecentes ao ambiente de execução do programa-objeto. Através destas chamadas, o programa-objeto tem a possibilidade de executar as coerções à medida que elas se fizerem necessárias.

Algumas linguagens especificam coerções para cada operação binária realizada. É o caso do FORTRAN, em que, à medida que as operações vão sendo realizadas, as coerções necessárias são executadas correspondentemente. Algumas implementações do FORTRAN, entretanto, adotam uma outra convenção, também usada em outras linguagens: as coerções são efetuadas a nível de sub-expressão entre parênteses, onde são impostas coerções que unifiquem os tipos envolvidos.

Geração de Código – A tradução do programa-fonte pelo compilador implica na geração de um código que preserve a semântica do programa original. Para o caso de compiladores de um passo único, a geração de código consiste na tradução do texto-fonte para uma forma equivalente na linguagem-objeto, geralmente uma linguagem de máquina, para execução direta em alguma máquina física. Formas intermediárias são geradas, entretanto, quando se constrói um compilador de múltiplos passos, ou então um compilador cuja saída seja um programa-objeto executável em uma máquina abstrata.

Um dos grandes objetivos do uso de linguagens intermediárias é o de permitir uma facilidade maior de manipulação com finalidades ligadas à otimização do código gerado. O uso de tais linguagens exibe uma característica também bastante importante: a de aumentar a portabilidade do programa-objeto, tornando o compilador menos dependente de máquina. Há diversas formas intermediárias, usualmente encontradas, para o código-objeto. A primeira corresponde ao uso da chamada *notação polonesa* para o código-objeto. Originalmente utilizada na compilação de expressões, foi estendida posteriormente para abranger construções não aritméticas. Utiliza o conceito de pilha, onde os operandos são armazenados até que um operador force sua manipulação, operação e desempilhamento. O compilador que gera código em notação polonesa necessita de um bom suporte do ambiente de execução para a realização das operações correspondentes aos operadores da notação polonesa. Há duas variantes para a notação polonesa: a variante pré-fixa, correspondente àquela em que o operador aparece antes dos operandos, e a

variante pós-fixa, onde a lista de operandos é finalizada pelo operador. Ambas eliminam os parênteses, e outros delimitadores semelhantes, do texto-fonte.

Uma segunda forma intermediária corresponde à expressão do programa-objeto como uma seqüência de *ênuplas*, cada qual representando uma operação completa: operandos e operador. Se o número de operandos for variável, a geração do código-objeto executável se complica, devido à incerteza acerca do comprimento de cada ênupla. Assim, em geral fixa-se o número de elementos de ênupla em 3 ou 4, obtendo-se as triplas e as quádruplas, respectivamente.

Nas *triplas,* cada operação é numerada, e consiste de um operador e dois operandos, sendo que o resultado de uma tripla é referenciado através do número da tripla que o gerou. Operandos inexistentes aparecem no caso de operações unárias tais como desvios incondicionais ou troca do sinal de uma variável.

Os endereços numéricos utilizados nas triplas dificultam operações de otimização sobre as mesmas, sugerindo o uso de nomes para os resultados das operações que elas representam. Incluindo-se esta informação, obtêm-se as *quádruplas,* em que o resultado da operação em geral designa o nome de uma posição de memória temporariamente utilizada como rascunho durante a avaliação de algum trecho mais complexo do programa. O uso de quádruplas é adequado quando se pensa em utilizar otimizadores automáticos, porém exige que os símbolos utilizados para representar os temporários sejam cuidadosamente gerenciados para evitar o consumo indevido de memória auxiliar.

Uma outra forma intermediária, encontrada em muitos compiladores, representa o programa estruturalmente através de uma *árvore,* constituindo uma boa alternativa para uso em compiladores otimizadores, visto que a manipulação de árvores é relativamente simples, permitindo aos algoritmos de otimização efetuarem simplificações substanciais no programa através da eliminação de redundâncias e do rearranjo das operações indicadas. As folhas da árvore representam os objetos que formam os operandos, e os nós internos da árvore correspondem aos operadores a serem aplicados às subárvores a eles subordinadas. Existe uma forte correspondência entre as árvores e as duas outras notações anteriores (notação polonesa e ênuplas), que são formas alternativas de apresentação da árvore.

Para uso em máquina abstratas, ou seja, para serem interpretadas ao invés de executadas diretamente por uma máquina, destacam-se as seguintes notações para o programa-objeto intermediário: a primeira é a do *código alinhavado* ("threaded code"), muito encontrado em implementações portáteis. Códigos alinhavados são, na realidade, seqüências de chamadas de rotinas da biblioteca de rotinas usadas pelo compilador para implementar a execução de operações não padronizadas. A segunda efetua a geração de *código para a máquina abstrata* a que se destina o programa-objeto. Um interpretador encarrega-se da execução deste código.

A máquina abstrata deve ser adequada à linguagem-fonte que o compilador traduz. Esta técnica permite um fácil transporte do compilador para outras máquinas, bem como facilita a alteração do compilador para aplicações específicas. O interpretador mencionado implementa o ambiente de execução da linguagem. Um exemplo bem sucedido do uso desta técnica é o "P-code", linguagem intermediária de largo uso como saída de compiladores Pascal.

Otimização – Com a finalidade de produzir um código-objeto eficiente, os compiladores incorporam, muitas vezes, estratégias de otimização, que são aplicadas durante ou após a geração do código-objeto. Muitas operações de otimização podem ser aplicadas ao programa sem a necessidade de levar em consideração a máquina para a qual o código está sendo gerado. Estas otimizações são ditas *otimizações independentes de máquina,* e se voltam à melhoria da qualidade do programa através de transformações que levam em conta basicamente a estrutura do mesmo. Conforme o tipo de otimização realizada, estas operações são extensivas ao programa como

um todo (otimização *global*) ou então restringem-se a partes específicas do programa (otimização *local*). Algumas implementações estabelecem regiões em que as otimizações locais podem ser realizadas sem perigo, regiões essas usualmente delimitadas por rótulos ou por comandos de desvio (*região de otimização local*). Em otimizações mais simples, restringe-se a ação do otimizador a um comando por vez, o que não dá oportunidade ao otimizador de efetuar uma melhoria do código com a qualidade obtenível da outra maneira.

A técnica de otimização local mais simples é a da *resolução prévia de subexpressões* que possam ser analisadas em tempo de compilação, evitando que o código-objeto seja obrigado a efetuar tais cálculos em época de execução. Desta maneira, as operações envolvendo constantes, bem como variáveis cujo valor pode ser calculado em tempo de compilação, são realizadas a priori, simplificando as expressões resultantes, e conseqüentemente, o código-objeto.

Esta operação pode ser realizada diretamente sobre a árvore abstrata que representa o programa, através da substituição das subárvores, que correspondem a operações sobre constantes, por nós terminais correspondentes ao resultado das operações em questão. Os resultados obtidos podem ser propagados para os comandos subseqüentes, desde que estes não estejam além do primeiro delimitador da região de otimização corrente.

Outra técnica de otimização local refere-se à *eliminação de subexpressões equivalentes* calculadas em diversos pontos de uma região de otimização. Para a detecção da presença de tais subexpressões, a árvore dos comandos deve ser pesquisada, em busca de subárvores equivalentes. A otimização, neste caso, consiste em manter apenas uma das subárvores equivalentes, descartando as redundantes, e substituindo-as por apontadores para a única subárvore mantida. A árvore abstrata original deixa, desta maneira, de ser uma árvore, passando a ser um grafo orientado acíclico equivalente.

Outra otimização local consiste na detecção e eliminação de *regiões inacessíveis de código*. Para isto, o compilador deve verificar cada uma das regiões de otimização, com a finalidade de assegurar que os rótulos que as introduzem sejam sempre referenciados em algum ponto do programa através de um comando de desvio. Regiões que não exibem esta propriedade podem ser descartadas completamente, sem prejuízo do código-objeto.

Um caso particular do cálculo prévio de subexpressão corresponde à detecção de *operações com elemento neutro* (adições de zero ou multiplicações por um), bem como algumas operações cujo resultado se conhece a priori (divisão de duas expressões equivalentes, ou produto por zero).

Uma otimização suplementar de expressões pode ser obtida levando-se em conta que determinadas operações, por gozarem da propriedade comutativa, associativa ou distributiva, podem ser remanejadas de modo tal que produzam um código equivalente, porém mais eficiente.

Através da uma análise mais global da árvore abstrata do programa, o otimizador tem condições de verificar se, no interior de uma iteração, aparecem comandos que não contribuem para a lógica da mesma, podendo, portanto, ser movidos para fora da iteração. Para realizar esta operação, as ações encarregadas de implementar o otimizador deverão montar um grafo do fluxo do programa, na qual cada nó represente uma das regiões de otimização local, e cada arco, uma possibilidade de desvio de uma região para outra. A análise das seqüências possíveis de execução das regiões permite determinar os pontos do programa em que ocorrem iterações, e portanto dá condições para que seja feita a busca dos comandos que podem ser postos em evidência.

Outra utilidade do grafo do fluxo do programa é a de permitir a aplicação das outras técnicas de otimização, mencionadas anteriormente, fora das regiões de otimização local. Isto pode ser feito promovendo-se a verificação da abrangência da ação dos comandos, dentro do programa. Desta maneira, é possível permitir a propagação da ação dos mecanismos de otimização, dis-

cutidos anteriormente, para além da região original de otimização a que se referem, desde que nas demais regiões em que se fará também a otimização, as condições de execução dos comandos seja favorável, ou seja, se forem semelhantes àquelas encontradas na região de otimização original.

Entre as otimizações *dependentes de máquina*, as que mais atenção têm tido na literatura são as seguintes: a *otimização da alocação de registradores*, em que os registradores da máquina (acumuladores) são alocados para a execução das operações do programa de tal maneira que reduzam o número de variáveis temporárias, bem como o de acessos à memória em época de execução. Os algoritmos de alocação de registradores variam conforme o número dos registradores disponíveis no hardware da máquina onde irá ser executado o código-objeto, embora sua finalidade seja sempre a de procurar alocar da melhor maneira possível os registradores, os quais, entre os meios de armazenamento disponíveis na máquina, permitem efetuar o armazenamento e o acesso aos dados da maneira mais econômica possível, tanto em tempo como em espaço de memória.

Outra classe de otimizações dependentes de máquina consiste na *otimização da escolha de instruções* de máquina particulares que sejam as melhores possíveis em cada caso. Isto é possível na grande maioria das máquinas, nas quais existe uma variedade de instruções que executam as mesmas operações. Cabe ao compilador, no caso, decidir qual a mais adequada em cada uma das situações apresentadas no código gerado.

Em alguns compiladores, é efetuada uma análise suplementar do código gerado, em busca de seqüências de instruções que possam ser compactadas na forma de uma seqüência mais eficiente. É esta também uma classe de otimizações dependentes de máquina, responsável, segundo a literatura, por uma redução às vezes sensível no volume de código-objeto gerado. Um caso particular destas otimizações ocorre no caso de compiladores que permitem a mistura de áreas de programa e de dados, exigindo assim desvios forçados para que os dados não sejam executados. Esta situação permite uma otimização através do rearranjo do código na memória.

Ambientes de Execução – Uma atividade essencial à execução do programa-objeto consiste em criar um ambiente no qual as diversas funções, não implementadas explicitamente pelo código-objeto, possam ser executadas. Durante a compilação, as ações semânticas devem encarregar-se da geração de um código que leve em consideração a existência de tal ambiente, na época da execução do programa-objeto.

De linguagem para linguagem e de compilador para compilador, as diversas implementações de linguagens de programação exigem ambientes de execução de requisitos diversos.

Essencialmente, as funções principais que são implementadas pelos ambientes de execução referem-se ao gerenciamento de memória, à comunicação com o sistema operacional e com outros programas, e à execução de funções especiais, usualmente representadas por pacotes de rotinas, da biblioteca do sistema ou de aplicação.

Quanto às funções de *gerenciamento de memória*, o ambiente de execução deve ocupar-se do gerenciamento da área disponível de memória, alocada pelo compilador para o armazenamento dos objetos (dados e programas).

Em alguns casos, a alocação de memória pode ser efetuada totalmente em época de compilação. Linguagens como o FORTRAN e o BASIC apresentam apenas objetos cujas dimensões são calculáveis em época de compilação: seus dados são todos de dimensões fixas e seus procedimentos e funções exigem uma área constante de memória, visto que não são recursivos. Em um caso como este, a maneira mais econômica de implementar o gerenciamento da memória é através do uso de alocação contígua dos objetos, e do dimensionamento prévio das áreas necessárias aos endereços de retorno, passagens de parâmetros e variáveis temporárias. Este tipo de

gerenciamento tem o nome de *alocação estática* de memória, e não exige nenhum suporte em tempo de execução.

Linguagens como o Algol, por sua vez, oferecem ao programador uma série de recursos que só são viáveis com o apoio de um ambiente de execução apropriado: nem sempre os objetos têm dimensões conhecidas em tempo de compilação (por exemplo, matrizes de dimensões variáveis, e procedimentos recursivos), e a estratégia imposta pelas estruturas de blocos exige um gerenciamento de memória, implementado em tempo de execução, através de uma estrutura de pilha, onde a cada bloco é reservada uma área para os objetos correspondentes.

Para cada entrada em um bloco, ou chamada de procedimento, a pilha é alterada, sendo nela inseridos os chamados *registros de ativação*, que são estruturas de dados em que consta uma área para os objetos declarados no procedimento, outra para seus parâmetros, e outra referente a informações de controle: endereço de retorno, apontadores para outros registros de ativação acessíveis no ponto de ativação do bloco e valores de retorno do procedimento.

Com este tipo de organização, obtém-se a chamada *alocação automática*, em que os movimentos da pilha ocorrem na ocasião da entrada e da saída dos blocos ou procedimentos. Procedimentos recursivos podem apresentar mais de um registro de ativação simultaneamente, cada um correspondente a uma das chamadas do procedimento.

Um terceiro tipo de gerenciamento de memória refere-se à chamada *alocação dinâmica*, exigida por algumas linguagens em que as necessidades de memória não são calculáveis em tempo de compilação nem podem ser resolvidas por meio de uma pilha. Cria-se neste caso, uma área de rascunho ("heap") onde são alocadas áreas aos objetos, conforme a demanda.

Um exemplo de tais classes de objetos são as cadeias de caracteres de comprimento não fixado, ou objetos que possam mudar de dimensão dinamicamente. Outra situação que exige tais recursos é encontrada em linguagens como o PL/I, em que durante a execução do programa, este pode requisitar ou liberar áreas de memória do "heap". O gerenciamento de memória dinâmica não é trivial, devido à diferença da dimensão entre os diversos objetos, e ao fato de ser aleatório o instante em que é liberada uma das áreas ocupadas pelos diversos objetos. Assim, ocorre o fenômeno da *fragmentação*, em que muitos espaços vazios são encontrados dispersos pela memória, exigindo estratégias de recuperação, tais como a *compactação* de memória através da movimentação física dos blocos de dados, ou o chamado "*garbage collection*", que compacta a memória quando não for possível alocar áreas para novas requisições, devido ao excesso de fragmentação, e à não disponibilidade de áreas vazias contíguas para atender a solicitação.

Quanto à *comunicação com o sistema operacional*, o ambiente de execução deve incorporar uma interface entre o programa-objeto e o sistema, através de chamadas de supervisor ou rotinas que as executam, de mecanismos especiais para a comunicação entre programas e eventualmente até com outros processadores, e de mecanismos de acesso a bancos de dados. Observe-se que, em alguns casos, o gerenciamento de memória é deixado a cargo do sistema operacional, que se encarregará, nestas situações, de executar as tarefas citadas anteriormente.

Em relação às *funções especiais de biblioteca* implementadas pelo ambiente de execução, podem-se destacar dois grandes grupos: as funções que são chamadas implicitamente pelo código-objeto, representadas por rotinas que executam operações aritméticas em precisão estendida e em ponto flutuante, operações matemáticas e trigonométricas, logaritmos, passagens de parâmetros, rotinas de conversão numérica, rotinas de formatação de entrada/saída, etc.; e as funções explicitamente utilizadas pelo programador, em geral pertencentes a pacotes do próprio usuário, ou a pacotes disponíveis no sistema: pacotes gráficos, pacotes algébricos, pacotes para a manipulação de matrizes, e pacotes de aplicação.

Note-se que muitas linguagens fazem uso de tais pacotes da maneira automática, ao serem utilizados alguns dos comandos especiais de que dispõem. Neste caso, as rotinas em questão são fornecidas juntamente com o compilador, fazendo parte integrante do ambiente de execução básico associado ao compilador.

Entre as inúmeras rotinas comumente encontradas nestas bibliotecas, as mais importantes são, sem dúvida, aquelas responsáveis pelo acesso aos objetos agregados e pela implementação da comunicação entre ambientes de execução (passagens de parâmetros de vários tipos, comunicação entre programas paralelos, etc.). Cumpre ainda mencionar algumas rotinas adicionais, nem sempre presentes em bibliotecas usuais, responsáveis pela implementação de funções ligadas à depuração do programa-objeto: são rotinas ditas de *instrumentação* do programa, que são chamadas pelo programa-objeto, mediante pedido do programador, efetuado através de comandos ao compilador, e que basicamente efetuam "trace", medidas de tempo de execução, cálculo de estatísticas de utilização de trechos selecionados do código, etc.

4.4 – LEITURAS COMPLEMENTARES

Em relação à descrição das funções executadas pelos diversos módulos de um compilador, a literatura é relativamente rica, havendo excelentes livros que apresentam tais informações. Destacam-se especialmente Bauer (1976), Aho (1979), Aho (1986), Barrett (1979), Gries (1971), Llorca (1986), Tremblay (1985) e Waite (1984), e suas referências bibliográficas.

Em relação a assuntos específicos, destacam-se:

Análise Léxica: Bauer (1976), Waite (1984) e Aho (1979)
Análise Sintática: todos os mencionados acima, Gries (1971), Lewis (1976) e Backhouse (1979).
Recuperação de Erros: Backhouse (1979), Tremblay (1985), Aho (1972).
Semântica e Geração de Código: Tremblay (1985), Barrett (1979), Waite (1984), Cunin (1980).
Representação de Objetos: Cunin (1980), Tremblay (1985), Barrett (1979), Lee (1967), Hopgood (1969).

4.5 – EXERCÍCIOS

1 – Quais são os três grandes módulos funcionais que compõem um compilador?

2 – Descreva o papel do analisador léxico no processo de compilação.

3 – Quais são os componentes de um átomo extraído pelo analisador léxico, e em que aplicação são utilizados no compilador?

4 – Como é resolvido o problema da não homogeneidade das dimensões assumidas no texto-fonte pelas cadeias que representam os átomos?

5 – Qual é o papel dos átomos de uma linguagem nas gramáticas que a descrevem?

6 – Liste um conjunto de funções desempenhadas usualmente pelo analisador léxico, descrevendo-as brevemente.

7 – Quais são as classes de átomos mais comumente utilizados como saídas da análise léxica?

8 – Por que os analisadores léxicos são às vezes denominados filtros léxicos?

9 – Quais os problemas acarretados ao analisador léxico por linguagens que permitem uma grande variedade de representações numéricas para as constantes?

10 – Que são tabelas de símbolos? Como é que o analisador léxico pode participar de sua criação e manutenção?

11 – Como é que são tratadas as tabelas de símbolos compostos de identificadores de comprimento não uniforme?

12 – Que são palavras reservadas? Qual a sua função nas linguagens de alto nível?

13 – Como é que o analisador léxico efetua o tratamento de palavras reservadas?

14 – Que vem a ser recuperação de erros? Como se aplica tal conceito na análise léxica?

15 – O que o analisador léxico tem em comum com os mecanismos de geração e controle de listagens e outros relatórios durante a compilação? Como isto é efetuado?

16 – Que são tabelas de referências cruzadas? Descreva como o analisador léxico pode ser empregado na sua geração.

17 – Que são macros? Como são utilizadas em linguagens de alto nível? Quais são as opções que um projetista pode explorar para processá-las? Como o analisador léxico pode ser utilizado para esta finalidade?

18 – Como o analisador léxico interage com o sistema operacional?

19 – Que é compilação condicional? Como pode ser implantada no compilador com o auxílio do analisador léxico?

20 – Por que os analisadores léxicos são considerados gargalos no processamento de uma compilação?

21 – Como é que os analisadores léxicos podem ser construídos de modo tal que a sua eficiência seja alta? Por que esta meta é essencial?

22 – Por que é conveniente que as gramáticas léxicas sejam regulares? Qual a influência deste fato na eficiência do compilador? Por quê?

23 – Como é que o analisador léxico se relaciona estrutural e funcionalmente com as outras partes do compilador?

24 – Por que é impossível a obtenção de um analisador léxico ótimo? Como pode ser feita uma solução de compromisso de qualidade quase ótima?

25 – Descreva os problemas enfrentados na confecção de um analisador léxico para uma linguagem como o FORTRAN, decorrentes da interpretação não única das seqüências de caracteres do texto-fonte. Como podem ser resolvidos?

26 – Ainda no FORTRAN, o analisador léxico se complica pela inexistência de palavras reservadas. Como é que deve ser feita a extração de átomos neste caso, uma vez que a divisão da cadeia de entrada não é trivial?

27 – Qual é o papel do analisador sintático no processo de compilação?

28 – Liste as funções principais do analisador sintático, descrevendo-as brevemente.

29 – Detalhe o papel do analisador sintático no tratamento de erros do texto-fonte.

30 – Comente a interação entre os analisadores sintático e léxico em um compilador.

31 – Comente a interação entre o analisador sintático e as ações semânticas em um compilador dirigido por sintaxe.

32 – Justifique a necessidade de preparação da gramática para a obtenção de um analisador sintático. Quais as metas principais desta atividade?

33 – Como o número de passos do compilador influi no programa de análise sintática?

34 – Qual a influência da linguagem sobre o número de passos do compilador?

35 – Como pode ser implementada, na prática, a consulta a átomos da cadeia de entrada que não devem ser consumidos pelo analisador?

36 – Quais são as opções de técnicas que você conhece, utilizadas para a construção de reconhecedores sintáticos?

37 – Quais as vantagens e desvantagens dos reconhecedores determinísticos em relação aos não-determinísticos?

38 – Comente acerca do tratamento de erros pelo reconhecedor sintático: conceito, função, utilidade, aplicação.

39 – Que são ações semânticas? Qual é o seu papel no processo de compilação?

40 – Como é que as ações semânticas se relacionam com os analisadores léxico e sintático em um compilador dirigido por sintaxe?

41 – Que é semântica de uma linguagem? Como é que as ações semânticas se encarregam de interpretá-la?

42 – Por que a semântica é descrita informalmente nas especificações das linguagens de programação usuais? Que problemas isto causa na implementação dos compiladores?

43 – Liste as principais funções executadas pelas ações semânticas em um compilador usual.

44 – Qual a relação entre as ações semânticas e as tabelas de símbolos do compilador? Explique a interação entre as ações semânticas e o analisador léxico.

45 – Que são tabelas de atributos? Qual a sua relação com as tabelas de símbolos? Como as ações semânticas se encarregam de criá-las e mantê-las?

46 – Como é que a estrutura de blocos da linguagem-fonte se reflete sobre as tabelas de símbolos e de atributos? Qual o papel das ações semânticas neste caso?

47 – Explique a atuação das ações semânticas em relação ao tratamento de identificadores: restrições de utilização, escopo, declarações explícitas e implícitas.

48 – Como são tratados os tipos de dados, originais da linguagem ou criados pelo programador, por parte das ações semânticas?

49 – Que significa o "teste de compatibilidade de tipos", executado pelas ações semânticas do compilador? Exemplifique.

50 – Qual é o papel das ações semânticas no gerenciamento da memória do programa que está sendo compilado? Classifique e conceitue cada caso.

51 – Que vem a ser ambiente de execução de um programa? Qual é a relação entre as ações semânticas e este ambiente, referente à geração de código?

52 – Como são criados e mantidos os ambientes de execução para o código-objeto? De que modo isto é efetuado pelas ações semânticas?

53 – Que significa "comunicação entre ambientes de execução"? Como isto é efetuado?

54 – Como é que o código-objeto é gerado pelas ações semânticas?

55 – Que são otimizações e como podem ser realizadas pelo compilador?

56 – Quais as classes mais importantes de otimizações que você conhece para o código-objeto? Exemplifique.

57 – Apresente as ações semânticas como elo de conexão entre as diversas partes do compilador, integrando-as e dando unidade ao mesmo.

58 – Como são tratados os identificadores encontrados durante a análise de uma declaração e durante a de um comando executável?

59 – Como você organizaria uma tabela de símbolos para uma linguagem que permite identificadores de uma única letra, seguida ou não por um único dígito (como as versões antigas do BASIC)? Justifique.

60 – Compare a organização de tabelas de símbolos para linguagens com escopo único e com escopos aninhados.

61 – Compare o comportamento de tabelas de símbolos para identificadores de comprimento máximo limitado e pequeno em relação às voltadas para identificadores arbitrariamente longos.

62 – Qual a influência do número de passos de compilação na organização e manutenção de uma tabela de símbolos?

63 – Como são tratadas as ocorrências múltiplas de um mesmo identificador no caso de tabelas de símbolos para linguagens de escopos aninhados?

64 – Como é feito o tratamento de identificadores utilizando-se tabelas de símbolos para linguagens em que os identificadores podem ser utilizados antes de sua declaração? Quais os problemas adicionais acarretados pela possibilidade de declaração de tipos neste caso? Como isto pode ser resolvido?

65 – De que modo pode ser unificado no compilador o tratamento de identificadores normais e o de palavras reservadas?

66 – Esboce uma tabela de símbolos para uso em um compilador de alguma linguagem de seu interesse, como Pascal, FORTRAN ou Algol, por exemplo.

67 – Conceitue objetos de tipos básicos, agregados e tipos. Como é que cada um destes elementos pode ser representado pelo compilador?

68 – Comente a respeito do problema da multiplicidade de representações internas de valores associados a certas classes de objetos. Exemplifique.

69 – Como são contornadas as deficiências do hardware, relativas ao reconhecedor e à manipulação de objetos estruturalmente complexos?

70 – Como podem ser representados os agregados homogêneos no programa-objeto? Qual é a infra-estrutura que o ambiente de execução deve oferecer para permitir sua manipulação?

71 – Quais são as diferenças entre as implementações de matrizes estáticas e dinâmicas no desempenho do programa objeto? Qual a influência deste fator nas ações semânticas do compilador?

72 – Como podem ser efetuados os testes de índices ilegais de matrizes em tempo de execução?

73 – Descreva a problemática da representação de agregados heterogêneos.

74 – Qual o papel dos descritores na compilação de agregados heterogêneos? Como são utilizados tais descritores em tempo de execução? Poderiam ser eliminados?

75 – Faça um paralelo entre a compilação de agregados heterogêneos e a de tipos definidos pelo programador.

76 – Que são coerções? Como é que as ações semânticas promovem as coerções no programa-objeto?

77 – Quais os tipos de códigos-objeto comumente gerados pelo compiladores e em quais casos cada um deles é o mais indicado?

78 – Faça um paralelo entre a geração de quádruplas e a de árvores como texto-objeto.

79 – Que é código alinhavado? Em que casos se aplica?

80 – Que são máquinas abstratas? Cite ao menos um caso importante da aplicação da técnica de geração de código para máquinas abstratas.

81 – Liste alguns tipos de otimização independente de máquina. Podem elas ser executadas fora do compilador? Como, ou por que não?

82 – Comente acerca da atividade de otimização dependente de máquina. Exemplifique.

83 – Descreva alguns esquemas de gerenciamento de memória ligados aos ambientes de execução dos programas. Em quais casos cada um deles se aplica?

84 – Que é fragmentação? Quando ocorre? Como é evitada?

85 – Discorra sobre a interação entre o programa-objeto e o sistema operacional.

86 – Descreva sucintamente o conteúdo de uma biblioteca de subrotinas do ambiente de execução.

87 – Que é instrumentação de um programa? Como pode ser implementada pelo compilador?

5

A Construção de um Compilador

Uma vez vistos diversos aspectos dos fundamentos pertinentes ao estudo dos compiladores, e levantadas as especificações relacionadas à sua implementação, resta pôr em prática os métodos e técnicas estudados, com a finalidade de obter um compilador para uma linguagem desejada.

Para tanto, é necessário que sejam efetuadas inúmeras decisões de projeto, relativas especialmente aos itens para os quais foram apresentadas múltiplas opções. Uma vez escolhido o roteiro de implementação, a aplicação cuidadosa dos métodos escolhidos para cada parte do compilador, e a integração das mesmas, para compor o programa final, podem ser realizadas com relativa facilidade, já que, em um grande número de casos, correspondem à escolha e aplicação de algoritmos suficientemente conhecidos e divulgados na literatura.

Neste capítulo, um exemplo completo é desenvolvido, procurando-se, através dele, fornecer ao leitor um roteiro que possa servir como base para a elaboração de projetos semelhantes.

5.1 – ESPECIFICAÇÕES GERAIS DO COMPILADOR

A primeira importante tarefa do projetista consiste em determinar alguns vínculos de projeto que deverão nortear todo o desenvolvimento a ser efetuado. Entre os principais pontos a considerar destacam-se:

(*a*) finalidade a que se destina o compilador.
(*b*) requisitos de velocidade do compilador.
(*c*) requisitos de velocidade do código-objeto.
(*d*) máquina hospedeira do compilador.
(*e*) máquina-objeto a ser utilizada.
(*f*) sistemas operacionais disponíveis em ambas as máquinas.
(*g*) ambiente de execução oferecido pela máquina-objeto.

(*h*) grau de complexidade da linguagem-fonte.
(*i*) número de passos de compilação desejado.
(*j*) requisitos de compatibilidade a nível do programa-objeto.
(*k*) recursos computacionais disponíveis.
(*l*) técnicas de implementação do compilador a serem adotadas.

No exemplo a ser desenvolvido a seguir, a finalidade prevista para o compilador é a de servir como instrumento didático. Dispensam-se as características de eficiência, estabelecendo-se a necessidade de um compilador simples e compacto, cujo tempo de desenvolvimento seja tão pequeno quanto possível. O compilador deverá ser transportável, razão pela qual seu desenvolvimento deverá ser feito em uma linguagem de alto nível. Neste texto, não será utilizada nenhuma linguagem de programação específica, optando-se pela expressão dos algoritmos em linguagem natural, estruturando-se idéias para que possam ser implementadas em qualquer linguagem com pouco esforço. Como se trata de um compilador com finalidade didática, é possível escolher, para a construção do código-objeto, um subconjunto muito restrito de instruções de máquina, efetuando-se a geração do código em uma linguagem simbólica (tipo "assembler"). Com esta escolha, evita-se uma série de problemas para o compilador, no que tange ao gerenciamento de endereços de memória para as instruções e para os dados. Mantidos na forma simbólica, repassa-se a sua manipulação para o montador, que deverá encarregar-se de efetuar alocações de memória, resolução de referências à frente, preenchimento de endereços dos operandos em instruções de referência à memória, etc.

Neste exemplo, não serão exigidos grandes recursos do sistema operacional hospedeiro, bastando que estejam disponíveis funções ou sub-rotinas de biblioteca do sistema que se encarregam de efetuar operações de entrada e saída. Para maior simplicidade de geração de código, pode-se criar a exigência de que estes recursos ofereçam entrada e saída formatados, exigência esta que na maioria dos sistemas usuais pode ser facilmente atendida. Quanto ao ambiente de execução, procurar-se-á explorar apenas os recursos mencionados da linguagem de máquina e do sistema operacional, de modo que o ambiente de execução seja o mais simples possível. A viabilidade desta simplificação prende-se exclusivamente à simplicidade da linguagem que será compilada, não devendo impor-se como restrição à confecção de compiladores para linguagens mais complexas. Para estas, recursos adicionais devem ser incorporados ao ambiente de execução de maneira totalmente natural, à medida que forem surgindo as necessidades correspondentes.

Quanto ao esquema utilizado para a organização do compilador, pode-se optar, pela sua simplicidade, por um compilador dirigido por sintaxe, onde o programa principal é o analisador sintático, e as ações semânticas se encarregam de traduzir o programa-fonte para um texto-objeto em linguagem simbólica, a ser posteriormente convertido para linguagem de máquina por um montador disponível. Desta forma, ficam automaticamente preenchidos os requisitos de compatibilidade do programa-objeto com o sistema operacional e com as bibliotecas do ambiente de execução utilizadas para outras finalidades. Com as especificações descritas, não são exigidos quaisquer recursos computacionais especiais, ficando o projeto totalmente compatível para que seja implementado em qualquer sistema computacional disponível.

Em relação à maneira de implementar o compilador, pode-se optar pelo uso de um simulador dos autômatos a serem utilizados, dirigido por tabelas que especificam as transições a serem executadas em cada caso. O mesmo interpretador de tabelas pode ser utilizado para simular autômatos finitos e as submáquinas dos autômatos de pilha. Para simplificar as tabelas, opta-se pelo uso de ações semânticas especiais para promover "look-ahead", chamadas e retornos de submáquinas.

5.2 – ESPECIFICAÇÃO DA LINGUAGEM-FONTE

Para tornar possível a implementação dos reconhecedores necessários à construção do esqueleto do compilador dirigido por sintaxe, é indispensável que a linguagem a compilar seja formalmente descrita através de uma gramática livre de contexto ou regular.

Caso a linguagem em questão seja alguma linguagem já formalizada, convém obter um relatório de especificação da mesma, e eventuais publicações nas quais estejam descritas suas normas de padronização. A importância desta atitude é muito grande, uma vez que deste cuidado depende a futura compatibilidade do compilador a ser construído com todos os já existentes.

Caso se trate de uma linguagem nova, deve existir um trabalho prévio de projeto e especificação formal, que levem à construção da gramática em que o projetista irá se basear para a obtenção do compilador.

Neste texto não serão cobertos os aspectos do projeto de linguagens, uma vez que este assunto não pertence ao seu escopo, podendo-se encontrar informações valiosas neste sentido na literatura específica sobre características de linguagens de programação e seu projeto.

O uso da notação de Wirth ou de diagramas de sintaxe na definição da linguagem é muito desejável, visto que tornam mais direta a construção do reconhecedor, embora obviamente este requisito não seja necessário.

De qualquer forma, através da aplicação das regras de conversão de notação das gramáticas (Seção 3.1), torna-se sempre possível reduzi-la à notação de Wirth, para a aplicação do método de construção de reconhecedores descrito em 3.3.

Desejando-se aplicar as soluções apresentadas em 3.2, pode-se obter com uma certa facilidade bons reconhecedores descendentes recursivos, *LL* (1) ou *LR* (1), através de manipulações convenientes da gramática original.

Para o exemplo a ser desenvolvido neste capítulo, foi escolhida uma linguagem muito simples, definida a seguir em BNF.

O formato em que a linguagem deve ser escrita é livre: espaços entre átomos são ignorados, servindo apenas como sepradores. Entre quaisquer dois átomos pode ser inserido um comentário, iniciado com o símbolo %, que se estende até o final da linha em que ocorre. O final de linha também funciona como separador entre dois átomos. Separadores só são obrigatórios em caso de ambigüidade.

Descrição BNF da Linguagem-Exemplo

```
(1) <programa>   ::= <seqüência de comandos> END
(2) <seqüência de comandos> ::= <comando>
                              | <seqüência de comandos>;<comando>
(3) <comando> ::= <atribuição>
                | <desvio>
                | <leitura>
                | <impressão>
                | <decisão>
                | ε
                | <rótulo> : <comando>
(4) <atribuição> ::= LET <identificador> := <expressão>
(5) <expressão> ::= <expressão> + <termo>
                  | <expressão> − <termo>
                  | <termo>
```

```
(6)  <termo> ::= <termo> * <fator>
              | <termo> / <fator>
              | <fator>
(7)  <fator> ::= <identificador>
              | <número>
              | (<expressão>)
(8)  <desvio> ::= GO TO <rótulo>
              | GO TO <identificador> OF <lista de rótulos>
(9)  <lista de rótulos> ::= <rótulo>
                          | <lista de rótulos> , <rótulo>
(10) <rótulo> ::= <identificador>
(11) <leitura> ::= READ <lista de identificadores>
(12) <lista de identificadores>
          ::= ϵ
            | <identificador> , <lista de identificadores>
(13) <impressão> ::= PRINT <lista de expressões>
(14) <lista de expressões> ::= ϵ
                             | <expressão> , <lista de expressões>
(15) <decisão> ::= IF <comparação> THEN <comando> ELSE <comando>
(16) <comparação> ::= <expressão> <operador de comparação> <expressão>
(17) <operador de comparação> ::= >
                                | =
                                | <
(18) <identificador> ::= <letra>
                       | <identificador> <letra>
                       | <identificador> <dígito>
(19) <número> ::= <dígito>
               | <número> <dígito>
(20) <letra> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
(21) <dígito> ::= Ø|1|2|3|4|5|6|7|8|9
```

A descrição formal livre de contexto apresentada mostra que um programa qualquer deve ser formado através da construção de uma seqüência de comandos, separados por ponto e vírgula. O comando vazio é uma das opções válidas.

Os rótulos que precedem um comando dão nome ao mesmo, e estes nomes não precisam ser únicos, sendo refenciáveis em comandos de desvio.

No programa, um nome pode aparecer uma só vez como rótulo de algum comando. Não há restrições quanto aos desvios permitidos na linguagem.

Além do comando vazio, cinco outros são disponíveis: atribuições de valor a uma variável, desvios incondicionais, comandos de leitura de valores, comandos de impressão de valores, e comandos de decisão.

O comando vazio não tem nenhuma função no programa, podendo ser utilizado livremente como recurso lingüístico de programação, apenas.

O comando de atribuição envolve o cálculo do valor associado a uma expressão, valor este que é depositado em uma variável, explicitada à esquerda da expressão através do seu identificador. A linguagem não possui declarações explícitas para as variáveis, sendo estas consideradas como declaradas no instante em que são referenciadas. Cabe portanto ao programador cuidar para que suas variáveis sejam utilizadas adequadamente ao longo do programa. A avalia-

ção das expressões segue as convenções usuais, sendo efetuada da esquerda para a direita, tendo os produtos e divisões precedência sobre as somas e subtrações, precedência esta alterável através da utilização de parênteses. Apenas números e identificadores de variáveis são aceitos nas expressões. Sobre os valores que representam, incidem as quatro operações aritméticas fundamentais. A aritmética utilizada nesta linguagem é inteira, com a precisão oferecida pela máquina em que o programa-objeto deve ser executado.

Os comandos de desvio apresentam-se em duas formas: na primeira, um desvio incondicional para um rótulo explicitamente indicado é especificado; na segunda, um identificador, que representa uma variável inteira, é utilizado como índice para a seleção de um dos rótulos de uma lista. O primeiro rótulo corresponde ao valor zero associado à variável indexadora, o segundo, ao valor um, e assim por diante. Valores indevidos para a variável indexadora tornam inativo o comando.

Os comandos de leitura promovem a entrada de dados de um meio externo (por exemplo, o teclado) e, com os valores lidos, preenchem uma lista de variáveis especificada pelo comando. Cada um dos dados deve ser fornecido em uma leitura separada. Assim, para um comando de leitura que especifica *n* variáveis, deverá ser promovida a leitura de *n* linhas de dados, cada qual referente a uma das variáveis, na ordem especificada pela lista de variáveis expressa no comando. Caso a lista seja vazia, o comando descarta a primeira linha de dados fornecida.

O comando de impressão é análogo, porém opera em sentido oposto: cada uma das expressões da lista que consta no comando é avaliada, e o valor obtido é impresso, um por linha, na ordem em que as expressões correspondentes comparecem na lista. Caso a lista seja vazia, uma linha em branco é saltada.

O comando de decisão envolve uma comparação entre expressões, efetuada através dos operadores > (maior que), = (igual a) e < (menor que). As expressões são avaliadas, e comparadas de acordo com o operador escolhido, da maneira convencional (os valores são considerados como números relativos). Caso a comparação seja verdadeira, o comando que figura entre as palavras THEN e ELSE será executado. Caso contrário, o outro comando o será. Note-se que existe a obrigatoriedade de fornecimento dos dois comandos. Para simular operadores não disponíveis, ou comparação do tipo IF-THEN apenas, pode-se lançar mão do artifício do uso do comando vazio.

Este sistema deverá aceitar todos os programas sintaticamente corretos, e executá-los. Nenhum erro de execução deve ser detectado. Casos anormais deverão sofrer um tratamento pré-determinado:

(*a*) em expressões que causem "overflow", esta indicação é ignorada.
(*b*) variáveis não preenchidas pelo programa são consideradas com o valor contido na posição de memória em que foram alocadas.
(*c*) divisão por zero não é executada. O resultado é o próprio dividendo.
(*d*) divisão com resultado não inteiro: é considerada apenas a parte inteira do quociente.
(*e*) nos resultados de multiplicações, é considerada apenas a palavra menos significativa.
(*f*) desvios indexados com índice inválido são desconsiderados.
(*g*) na leitura, são considerados apenas os bits correspondentes à palavra menos significativa do dado fornecido, caso este tenha amplitude muito grande.

5.3 – ESPECIFICAÇÃO DA LINGUAGEM DE SAÍDA

Para determinar o conjunto de instruções da máquina-objeto, optou-se por reduzir ao mínimo o número de instruções utilizadas, de tal modo que o subconjunto assim escolhido possa

ser facilmente adaptado para qualquer arquitetura existente a ser utilizada em uma implementação real. Escolhe-se ainda um conjunto de pseudoinstruções muito reduzido, composto de pseudoinstruções encontradas na grande maioria dos montadores disponíveis.

Para uma implementação real, eventuais discrepâncias entre o conjunto de códigos aqui proposto e o disponível na máquina podem ser contornados com facilidade através de adaptações triviais.

As instruções de máquina a serem utilizadas são todas instruções de referência à memória, com endereçamento direto ou imediato:

Instrução:		*Interpretação:*
LDA	X	Obtém o conteúdo da posição X, e deposita no acumulador.
STA	X	Armazena o conteúdo do acumulador na posição X.
ADA	X	Soma ao acumulador o conteúdo da posição X.
SUB	X	Subtrai, no acumulador, o conteúdo da posição X.
MUL	X	Multiplica, no acumulador, o conteúdo da posição X.
DIV	X	Divide o acumulador pelo conteúdo da posição X.
CALL	X	Chama a subrotina X (da biblioteca de execução).
BRI	X	Desvia incondicionalmente para a posição X.
BRN	X	Desvia para a posição X se o conteúdo do acumulador for negativo.
BRZ	X	Desvia para a posição X se o conteúdo do acumulador for zero.
BRP	X	Desvia para a posição X se o conteúdo do acumulador for positivo.

As pseudoinstruções adotadas são as seguintes:

Pseudoinstrução:		*Interpretação:*
LBL	X	Atribui ao endereço da próxima instrução o rótulo X.
DS	X	Define uma posição de memória com conteúdo inicial igual ao número X.
END		Define o final físico do programa-fonte.
EXT	X	Define o rótulo X como nome de rotina externa, chamada pelo código-objeto (rotina de biblioteca).

Pode-se admitir, sem perda de generalidade, que os códigos simbólicos sejam separados entre si por um sinal de pontuação, por exemplo, ponto e vírgula.

Comentários podem ser inseridos, como no programa-fonte, na forma de um texto qualquer, compreendido entre o sinal % e o final da linha.

Cada linha do código simbólico pode conter uma ou mais instruções e/ou pseudoinstruções.

Os operadores das instruções e das pseudoinstruções, quando se referirem a posições de memória, são representados por identificadores, cuja sintaxe é a mesma dos identificadores do programa-fonte. Adicionalmente, são aceitos identificadores especiais, representados por # seguido de um número inteiro, os quais simbolizam posições de memória conhecidas apenas pelo compilador, às quais o programa-fonte não tem acesso explícito.

Operandos imediatos referem-se a valores e não a endereços, e são representados pelo símbolo = seguido de um número inteiro, que representa o valor em questão.

A sintaxe dos números inteiros é a mesma utilizada para os números inteiros do programa-fonte.

Para a geração do código-objeto adotam-se os seguintes critérios:

(*a*) o texto-fonte é incluído na forma de comentários, intercalados com o texto-objeto gerado pelo compilador. Uma variável de controle (MIX) pode ser utilizada para controlar a geração destes comentários:

    MIX=TRUE: intercala comentários
    MIX=FALSE: suprime os comentários

(*b*) para compactar o texto-objeto, a linha de código-objeto é preenchida seqüencialmente pelos códigos da linguagem de saída, mudando-se de linha apenas quando não houver, na linha correntemente em uso, espaço para comportar mais nenhum código. Uma variável de controle (PACK) pode ser utilizada para controlar esta compactação:

    PACK=TRUE: compacta
    PACK=FALSE: gera uma instrução por linha

A variável de controle MIX, quando TRUE, faz com que, sempre que um comentário for intercalado, uma nova linha seja utilizada para isto, o que produz uma separação lógica entre os códigos gerados para cada comando-fonte.

(*c*) ao final da geração do código, imediatamente antes de ser gerada a pseudoinstrução END, uma seqüência de pseudoinstruções EXT deverá ser emitida, indicando quais rotinas do ambiente de execução serão necessárias ao funcionamento do programa.

(*d*) uma variável de controle (CODE) pode ser utilizada para controlar a geração de código-objeto:

    CODE=TRUE: permite a geração de código
    CODE=FALSE: inibe a geração de código

Quando CODE=FALSE, são desativadas as gerações de comentários, e portanto a variável MIX perde sua função. Entretanto, ao ser alterado o valor de CODE, as demais funções são restauradas.

(*e*) outra variável de controle (LIST) controla a listagem do programa-fonte:

    LIST=TRUE: permite a listagem do texto-fonte
    LIST=FALSE: inibe a listagem do texto-fonte

Na listagem, é impresso o número da linha do texto-fonte, seguido de uma cópia da linha em questão. Eventuais erros de compilação provenientes da análise daquela linha são impressos logo a seguir.

A supressão da listagem não deve inibir a impressão dos erros.

Em uma implementação, as variáveis LIST e MIX seriam controladas pela rotina de leitura de programa-fonte: a cada linha lida, consulta-se a variável LIST, imprimindo-se a imagem da linha no dispositivo de saída de listagens caso LIST=TRUE. Consulta-se a variável MIX, e, caso MIX=TRUE, emite-se uma cópia da linha, na forma de um comentário, no dispositivo de saída do código objeto.

As variáveis PACK e CODE são controladas pela rotina de geração de código: a cada vez que a rotina é chamada, consulta-se inicialmente a variável CODE. Caso CODE=FALSE, nada é gerado. Caso CODE=TRUE, efetua-se a emissão do código solicitado no dispositivo de saída do código-objeto.. Consulta-se então a variável PACK. Caso PACK=TRUE, aguarda-se que a linha de código-objeto seja totalmente preenchida antes que seja emitida uma mudança de linha. Caso PACK=TRUE, muda-se de linha a cada código gerado.

No exemplo a ser detalhado a seguir estas variáveis não serão utilizadas, uma vez que correspondem a um nível de detalhamento do compilador maior que o adotado para o restante do exemplo.

Convém, entretanto, mencionar que os valores destas quatro variáveis podem ser introduzidos no compilador através da leitura de uma linha de controle correspondente, por exemplo, à linha que precede a primeira das linhas que compõem o programa a ser compilado.

5.4 – PROJETO DO ANALISADOR LÉXICO

Através da inspeção da gramática BNF da linguagem a ser compilada, pode-se levantar os terminais correspondentes essencialmente às palavras reservadas, sinais de pontuação, letras e dígitos. Levando-se em conta o grande número de vezes que o analisador léxico é chamado durante a compilação, e a necessidade de utilização de "look-ahead" para a distinção entre identificadores e palavras reservadas, bem como a grande utilização de identificadores e números na gramática, opta-se por considerar como terminais tanto os números como os identificadores, distinguindo, a posteriori, por consulta a tabela, as palavras reservadas dos identificadores. Tratamento semelhante é dado aos símbolos compostos. Assim sendo, o conjunto de átomos, correspondente aos terminais adotados, é o seguinte:

(*a*) Palavras Reservadas: END
LET
GO
TO
OF
READ
PRINT
IF
THEN
ELSE

(*b*) Átomos Especiais: identificadores
números inteiros

(*c*) Sinais de Pontuação, Operação e Separação:

;
:
+
–
*
/
(

)
.
>
=
<

(*d*) Sinais Compostos: :=
GO TO

Brancos, sinais de mudança de linha e comentários inseridos no texto-fonte serão eliminados, e desta maneira o analisador léxico terá o comportamento de um filtro. A divisão do texto em seus componentes básicos utilizará o critério do comprimento máximo, ou seja, só será considerado identificado um átomo no instante em que o próximo caractere de entrada, ainda não analisado, não puder ser integrado ao mesmo. Assim sendo, não haverá detecção de erros léxicos, já que todo símbolo que não puder ser acoplado a um átomo será considerado como símbolo inicial do próximo átomo a ser extraído, ou de algum separador.

Para efetuar a análise léxica, será desenvolvido um transdutor finito, cujo alfabeto de entrada é o conjunto de todos os caracteres da linguagem, bem como todos os caracteres que poderão figurar nos comentários, e o seu alfabeto de saída, o conjunto dos átomos relacionados anteriormente.

Cada átomo será representado por um par de números inteiros, o primeiro correspondente ao tipo de átomo, e o segundo, a uma informação complementar, de acordo com as convenções mostradas na seguinte tabela:

Terminal	*Tipo do Átomo*	*Informação Complementar*
palavra reservada	a própria palavra reservada	irrelevante
identificador	<<identificador>>	índice do identificador na tabela de símbolos
número inteiro	<<número>>	o valor numérico associado ao número
sinal	o próprio sinal	irrelevante
qualquer outro símbolo	o próprio símbolo	irrelevante

Uma tabela de símbolos é utilizada para coletar todos os identificadores declarados no programa, e possui uma correspondência direta com uma tabela de atributos, utilizada pelas ações semânticas.

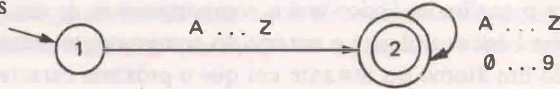
Pode-se admitir a disponibilidade de rotinas de acesso às tabelas de símbolos e de palavras reservadas. Uma das rotinas se encarrega de efetuar buscas de identificadores nestas tabelas, retornando a informação do local em que o identificador foi encontrado, ou então a informação de que o identificador não consta na tabela. Neste caso, outra rotina deve ser utilizada para inserir um identificador na tabela de símbolos.

Para uma primeira implementação, não há necessidade de implementar-se algoritmos rebuscados, de modo que, para as especificações do projeto em questão, pode-se adotar um método de busca seqüencial, de fácil implementação. Otimizações do compilador podem alterar, posteriormente, a organização de tais estruturas de dados e os métodos de busca a serem aplicados.

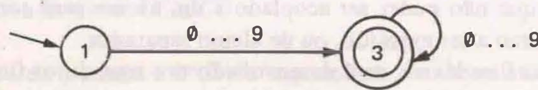
Com as considerações até aqui efetuadas, pode-se conceber o autômato finito que servirá de base para o transdutor que implementa o analisador léxico.

Cada um dos átomos formados por mais de um símbolo pode ser reconhecido através dos seguintes autômatos particulares:

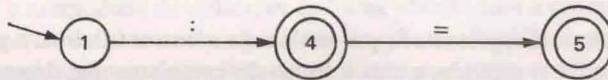
(*a*) identificadores



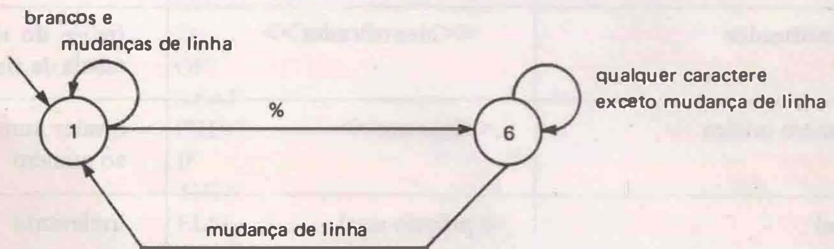
(*b*) números



(*c*) sinal composto de atribuição

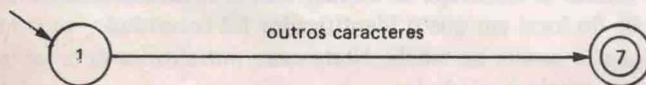


Os separadores (comentários, brancos e mudanças de linha) podem ser descartados através de um filtro instalado no estado inicial do autômato que implementará o substrato do analisador léxico:

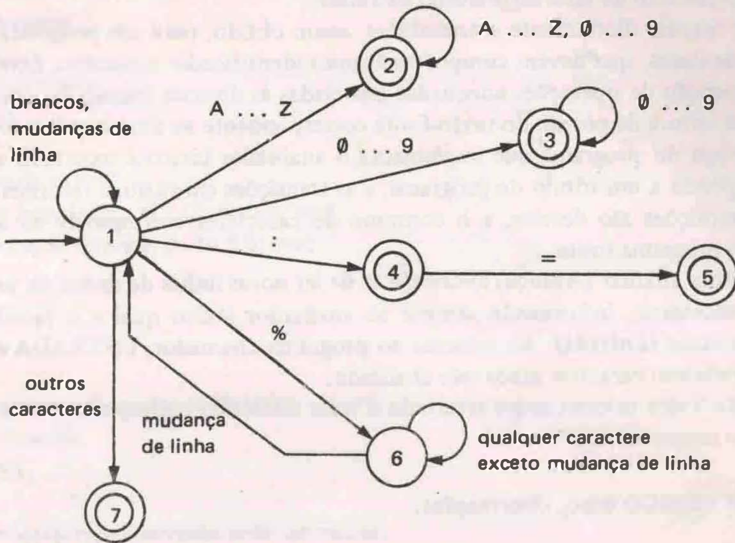


Note-se que este filtro não apresenta estados finais, uma vez que nenhum átomo é reconhecido por ele, ao contrário do que ocorre nos demais casos.

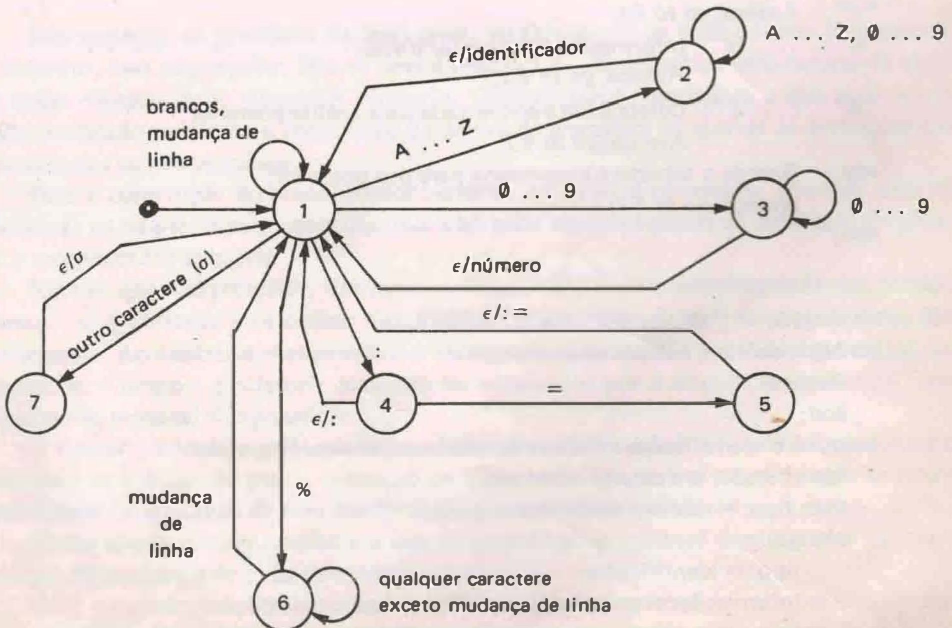
Quaisquer caracteres adicionais encontrados serão reconhecidos como átomos isolados:



Compondo estes autômatos, tem-se o reconhecedor-base do transdutor finito desejado:



Para obter, finalmente, o transdutor, eliminam-se os estados finais, acrescentando-se uma transição em vazio partindo de cada um deles para o estado inicial. Associa-se a tal transição a emissão do átomo que acaba de ser reconhecido e extraído:



Para a operação deste transdutor, convenciona-se que, ao ser executada uma transição em vazio, é efetuado um retorno da subrotina que o implementa para o programa chamador, passando como parâmetro de retorno o átomo extraído.

Pode-se mapear diretamente o transdutor, assim obtido, para um programa. As informações complementares, que devem compor os átomos identificador e número, devem ser obtidas através da execução de operações adequadas associadas às diversas transições, para que seja evitada uma nova leitura da porção do texto-fonte correspondente ao átomo extraído.

Um esboço do programa que implementa o analisador léxico é mostrado a seguir. Cada estado corresponde a um rótulo do programa, e as transições em vazio, a retornos ao programa chamador. Transições são desvios, e o consumo de caracteres corresponde ao avanço de um cursor sobre o programa-fonte.

Uma rotina auxiliar (Avança) encarrega-se de ler novas linhas de dados do programa-fonte sempre que necessário, informando sempre ao analisador léxico qual é o próximo caractere ainda não analisado (Entrada). Ao retornar ao programa chamador, ENTRADA está sempre se referindo ao próximo caractere ainda não analisado.

A função Valor retorna como resultado o valor numérico correspondente ao dígito fornecido como seu argumento.

```
PROCEDURE LÉXICO (tipo, informação);

  begin
  E1: while entrada = branco or entrada = mudança de linha
        do Avança;
        Case entrada of
           begin
           "%":  Avança; go to E6;
           ":" : Avança; go to E4;
           "0" : ... "9" : Informação: = Valor (entrada);
                           Avança; go to E3;
           "A" : ... "Z" : Coleta a letra encontrada para análise posterior;
                           Avança; go to E2;
           else : Guarda o caractere encontrado para uso posterior;
                  Avança; go to E7;
           end;

  E2: while entrada in {"0", ..., "9", "A", ..., "Z"}
           do begin coleta a entrada encontrada;
              Avança;
              end;
           pesquisa o identificador coletado na tabela de palavras reservadas;
           if identificador era palavra reservada
              then tipo: = número da palavra reservada
              else begin
                    tipo: = identificador
                    informação: = posição do identificador na tabela de símbolos
                    end;
           return;
```

```
E3: while entrada in {"0", ..., "9"}
        do begin
            Informação: = Informação * 10 + Valor (entrada);
            Avança;
        end;
    tipo: = número;
    return;

E4: if  entrada ≠ "="
        then begin tipo: = " :"; return; end
        else begin Avança; go to E5; end;

E5: tipo: = ": =";
    return;

E6: while entrada ≠ mudança de linha
        do Avança;
    go to E1;

E7: tipo: = caractere guardado anteriormente;
    return;

end léxico;
```

5.5 – PROJETO DO ANALISADOR SINTÁTICO

Pela inspeção da gramática da linguagem, verifica-se que se trata de uma linguagem livre de contexto, mas não regular. Isto se deve à presença de não-terminais auto-recursivos centrais, tais como <expressão>, <termo>, <fator>, <comando> e <decisão>, o que pode ser facilmente verificado mediante a construção da árvore da gramática ou através de derivações exaustivas de todos os não-terminais.

Para a construção do reconhecedor sintático desejado, é necessário, antes de mais nada, determinar os não-terminais essenciais, que irão gerar as submáquinas do autômato de pilha em que o reconhecedor se baseia.

Note-se que <expressão>, <termo> e <fator> são mutuamente dependentes, sendo que <termo> só é utilizado para definir <expressão>, e que <fator> só é empregado na definição de <termo>. Ao contrário, <expressão> é vastamente referenciado por toda a gramática. Desta maneira, se <termo> e <fator> puderem ser eliminados por manipulação gramatical, restará apenas o não-terminal <expressão>.

O mesmo raciocínio pode ser feito em relação a <comando> e <decisão>: o conceito de <decisão> só é utilizado para a definição de <comando>, e poderá ser eliminado se houver a possibilidade de aplicação de uma manipulação gramatical apropriada.

Outro não-terminal essencial é a raiz de gramática, que deverá dar origem à submáquina principal do autômato de pilha desejado.

Com estas observações em mente, pode-se iniciar o processo de manipulação da gramática, no sentido de se obter um reconhecedor sintático que servirá como núcleo do compilador dirigido por sintaxe.

A primeira providência é a de exprimir a gramática através da notação de Wirth, eliminando recursões à direita ou à esquerda, e representar todos os não-terminais através de iterações.

O resultado desta primeira manipulação é o seguinte:

Descrição da Linguagem-exemplo em Notação de Wirth Iterativa

(1) programa = seqüência-de-comandos “END”.
(2) seqüência-de-comandos = comando {“;” comando}.
(3) comando = {rótulo “:”} [atribuição|desvio|leitura|impressão|decisão].
(4) atribuição = “LET” identificador “: =” expressão.
(5) expressão = termo {(“+” | “–”) termo}.
(6) termo = fator {(“ *” | “/”) fator}.
(7) fator = identificador | número | “(”expressão“)”.
(8) desvio = “GO” “TO” (rótulo | identificador “OF” lista-de-rótulos).
(9) lista-de-rótulos = rótulo {“,” rótulo}.
(10) rótulo = identificador.
(11) leitura = “READ” lista-de-identificadores.
(12) lista-de-identificadores = [identificador {“,” identificador}].
(13) impressão = “PRINT” lista-de-expressões.
(14) lista-de-expressões = [expressão {“,” lista-de-expressões}].
(15) decisão = “IF” comparação “THEN” comando “ELSE” comando.
(16) comparação = expressão operador-de-comparação expressão.
(17) operador-de-comparação = “>” | “=” | “<”.
(18) identificador = letra {letra | dígito}.
(19) número = dígito {dígito}.
(20) letra = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z.
(21) dígito = Ø|1|2|3|4|5|6|7|8|9.

Sendo essenciais os não-terminais programa, comando e expressão, serão necessárias ao menos três submáquinas para a construção do autômato que reconhece a linguagem-exemplo. Números e identificadores serão considerados terminais.

A construção destas três submáquinas depende inicialmente da eliminação de todos os demais não-terminais, o que é feito através de substituições sucessivas dos não-terminais pelas suas definições, acompanhadas de eventuais simplificações das expressões que compõem a notação de Wirth. Aplicando-se este procedimento obtém-se:

(*a*) Para o não-terminal *programa:*
Substituindo-se (2) em (1) obtém-se

programa = comando {";" comando} "END"

Esta expressão já expressa o não-terminal programa exclusivamente em função de terminais e do não-terminal essencial *comando*, dispensando novas manipulações.

(*b*) Para o não-terminal *comando:*
Substituindo-se em (3) as definições (4), (8), (10), (11), (13) e (15), tem-se:

```
comando =  { identificador ":" }
           ["LET" identificador ":=" expressão
           |"GO" "TO" (identificador | identificador "OF" lista-de-rótulos)
           |"READ" lista-de-identificadores
           |"PRINT" lista-de-expressões
           |"IF" comparação "THEN" comando "ELSE" comando].
```

Restaram os não-terminais não-essenciais: rótulo, lista-de-rótulos, lista-de-identificadores, lista-de-expressões e comparação.

Através da substituição das definições (10), (12), (14), (16) e (17), seguida da fatoração da definição do comando de desvio, tem-se, finalmente, a expressão de *comando* em função apenas de terminais e de não-terminais essenciais:

```
comando = { identificador ":" }
          ["LET" identificador ":=" expressão
          |"GO" "TO" (identificador ["OF" identificador { "," identificador }])
          |"READ" [identificador { "," identificador }]
          |"PRINT" [expressão { "," expressão }]
          |"IF" expressão ("<" | "=" | ">") expressão "THEN" comando
               "ELSE" comando].
```

(*c*) Resta o não-terminal *expressão*, que pode ser obtido pela substituição de (6) em (5), seguida da substituição de (7) na expressão resultante:

```
expressão = (identificador | número | "("expressão")")
            {("*" | "/") (identificador | número | "("expressão")")}
            {("+" | "-")
             (identificador | número | "("expressão")")
            {("*" | "/") (identificador | número | "("expressão")")} }.
```

Obtidas as três expressões de Wirth, para os não-terminais essenciais, é possível iniciar a construção das submáquinas que a eles correspondem. Para isso, o primeiro passo consiste em numerar os estados correspondentes aos diversos pontos das definições. Para tornar mais concisas as expressões de Wirth, serão efetuadas as seguintes alterações de notação:

(*a*) Supressão da aspas que envolvem os terminais.
(*b*) Palavras reservadas serão sublinhadas.
(*c*) Identificador e número serão denotados I e N, respectivamente.
(*d*) Comando e expressão serão denotados C e E, respectivamente.
(*e*) Os parênteses "("e")" serão denotados ∢ e ≯, respectivamente, para evitar confusão com os da metalinguagem.

A atribuição de estados pode então ser efetuada.

(*a*) *programa:*

```
. C . { . ;   C . } . END .
0   1   2     3   4   2     5
```

(*b*) *comando:*

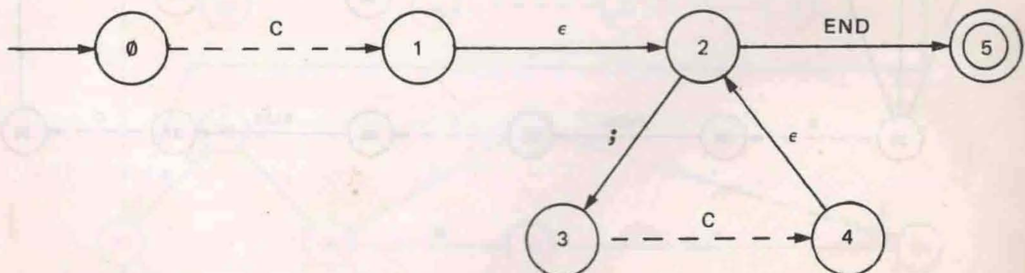
```
. { . I . : . } .
0   1   39  40  1

. [ . LET . I . := . E .
1   1       3   4    5   6

  | . GO TO . ( . I . [ . OF . I .  { . , . I . } . ] . ) .
    1         7   7   9   9    11  12   13  14   15 13  10  8

  | . READ . [ . I . { . , . I . } . ] .
    1        16  16  18  19  20  21  19  17

  | . PRINT . [ . E . { . , . E . } . ] .
    1         22  22  24  25  26  27  25 23

  | . IF . E . ( . < . | . = . | . > . ) . E .
    1    28  29  29  31  29  32  29  33  30  34

. THEN . C . ELSE . C . ] .
34      35  36     37  38  2
```

(*c*) *expressão*:

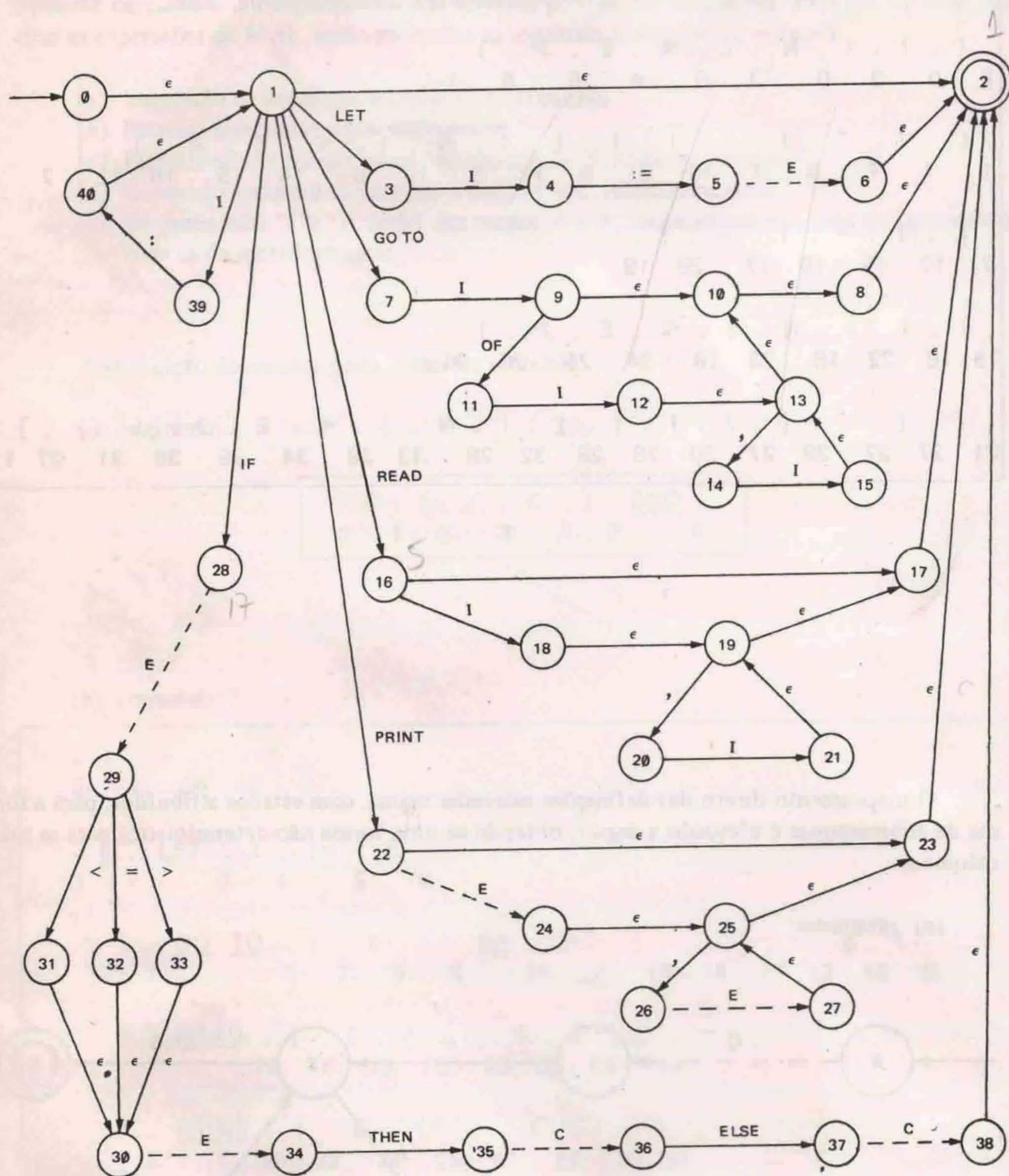
```
. ( . I . | . N . | . ∢ . E . ᐅ . ) .
0  0   2   0    3   0    4    5    6   1

. { . ( . * . | . / . ) . ( . I . | . N . | . ∢ . E . ᐅ . ) . } .
1   7   7    9   7   10   8    8   12   8    13   8    14   15   16  11    7

. { . ( . + . | . — . ) .
7   17   17   19   17   20   18

. ( . I . | . N . | . ∢ . E . ᐅ . ) .
18  18   22   18   23   18   24   25   26   21

. { . ( . * . | . / . ) . ( . I . | . N . | . ∢ . E . ᐅ . ) . } . } .
21   27   27   29   27   30   28   28   32   28   33   28   34   35   36   31   27  17
```

O mapeamento direto das definições marcadas acima, com estados atribuídos, para a forma de submáquinas é efetuado a seguir, obtendo-se uma forma não-determinística para as submáquinas.

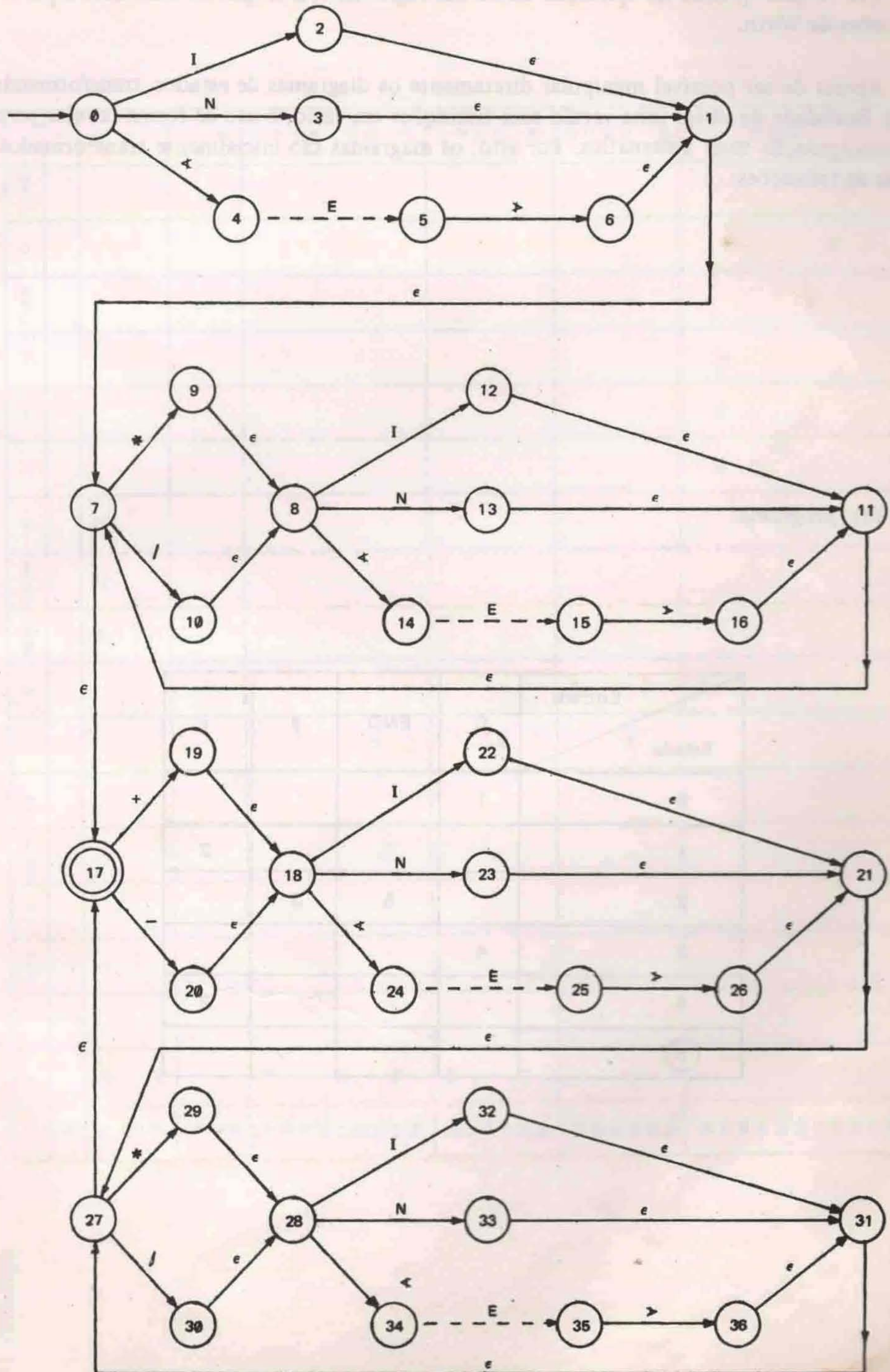
(*a*) *programa:*



(*b*) *comando:*



c) *expressão*:



A seqüência do projeto do reconhecedor consiste em eliminar os não-determinismos das submáquinas. Essencialmente, isto pode ser efetuado através da eliminação das transições em vazio que surgem quando da aplicação direta das regras de construção do autômato a partir das expressões de Wirth.

Apesar de ser possível manipular diretamente os diagramas de estados, transformando-os com a finalidade de obter uma versão sem transições em vazio, o uso da forma tabular permite uma manipulação mais sistemática. Por isto, os diagramas são inicialmente transformados em tabelas de transições:

(*a*) *programa:*

Estado \ Entrada	C	END	;	ϵ
0	1			
1				2
2		5	3	
3	4			
4				2
(5)				

(*b*) *comando:*

	I	:	LET	:=	GOTO	OF	,	E	READ	PRINT	IF	>	=	<	THEN	C	ELSE	ε
0																		1
1	39		3		7				16	22	28							2
2																		
3	4																	
4				5														
5								6										
6																		2
7	9																	
8																		2
9						11												10
10																		8
11	12																	
12																		13
13							14											10
14	15																	
15																		13
16	18																	17
17																		2
18																		19
19							20											17
20	21																	
21																		19
22								24										23
23																		2
24																		25
25							26											23
26								27										
27																		25
28								29										
29												33	32	31				
30								34										
31																		30
32																		30
33																		30
34															35			
35																36		
36																	37	
37																38		
38																		2
39		40																
40																		1

(*c*) *expressão:*

	I	N	⋖	E	⋗	+	−	*	/	ϵ
0	2	3	4							
1										7
2										1
3										1
4				5						
5					6					
6										1
7								9	10	17
8	12	13	14							
9										8
10										8
11										7
12										11
13										11
14				15						
15					16					
16										11
⑰						19	20			
18	22	23	24							
19										18
20										18
21										27
22										21
23										21
24				25						
25					26					
26										21
27								29	30	17
28	32	33	34							
29										28
30										28
31										27
32										31
33										31
34				35						
35					36					
36										31

Para a eliminação das transições em vazio, parte-se das tabelas assim construídas, e, para as linhas em que alguma transição estiver programada na coluna ϵ (das transições em vazio), incorpora-se a esta linha todas as transições indicadas no estado indicado na transição em vazio em questão. Isto pode causar a incorporação de nova transição em vazio neste estado. Se isto ocorrer, repete-se o procedimento até que não mais seja incorporada nenhuma transição em vazio que ainda não tenha sido considerada. Se algum dos estados para os quais havia uma transição em vazio for estado final, a linha que estiver sendo considerada receberá este atributo.

A manipulação das tabelas, de acordo com estas regras, produz os seguintes resultados:

(*a*) *programa:*

	C	END	;	ϵ
0	1			
1		5	3	~~2~~
2		5	3	
3	4			
4		5	3	~~2~~
(5)				

Observações:

As linhas 1 e 4 sofreram a inclusão das transições provenientes do estado 2. Notar que o estado 2 tornou-se, e portanto, desnecessário, podendo, desta maneira, ser eliminado.

Apesar de não ser referenciado, o estado 0 deve ser mantido por ser o estado inicial desta submáquina.

(*b*) *comando*:

	I	:	LET	:=	E	GO TO	OF	,	READ	PRINT	IF	>	=	<	THEN	C	ELSE	ε
⓪	39		3			7			16	22	28							~~1~~
①	39		3			7			16	22	28							~~2~~
②																		
3	4																	
4				5														
5					6													
⑥																		~~2~~
7	9																	
⑧																		~~2~~
⑨							11											~~10~~
⑩																		~~8~~
11	12																	
⑫								14										~~13~~
⑬								14										~~10~~
14	15																	
⑮								14										~~13~~
⑯	18																	~~17~~
⑰																		~~2~~
⑱								20										~~19~~
⑲								20										~~17~~
20	21																	
㉑								20										~~19~~
㉒					24													~~23~~
㉓																		~~2~~
㉔								26										~~25~~
㉕								26										~~23~~
26					27													
㉗								26										~~25~~
28					29													
29												33	32	31				
30					34													
31					34													~~30~~
32					34													~~30~~
33					34													~~30~~
34															35			
35																36		
36																	37	
37																38		
㊳																		~~2~~
39		40																
㊵	39		3			7			16	22	28							~~1~~

Observações:

A linha 0 recebe a inclusão das transições correspondentes aos estados 1 (originalmente referenciado) e 2 (referenciado no estado 1), e o atributo de estado final, decorrente do fato de o estado 2 ser um estado final.

A linha 6 recebe o atributo de estado final, pois transita em vazio para o estado 2.

A linha 8 analogamente torna-se um estado final. A linha 10, incorporando a linha 8, também sofre o mesmo tratamento. O mesmo sucede com a linha 9, devido à incorporação da linha 10.

A linha 12, incorporando as transições do estado 13, torna-se um estado final devido à incorporação da linha 10. O mesmo ocorre com as linhas 13 e 15.

A linha 16 referencia a linha 17, que por sua vez referencia em vazio a linha 2. Isto transmite às linhas 16 e 17 a característica de estados finais.

As linhas 18 e 21, incorporando as transições da linha 19, referenciam também em vazio a linha 17, e tornam-se também estados finais pela mesma razão.

A linha 22 transita em vazio para a linha 23 e esta para o estado final 2, o que faz com que ambas (linhas 22 e 23) se transformem em estados finais.

A linha 25, transitando em vazio para a linha 23, também assimila este atributo.

As linhas 24 e 27, incorporando as transições da linha 25, também absorvem dela a característica de estados finais.

As linhas 31, 32, 33, que transitam em vazio para a linha 30, assimilam as transições que a linha 30 indica.

A linha 38, transitando em vazio para o estado final 2, torna-se estado final.

A linha 40, transitando para o estado 1 em vazio, assimila deste estado todas as transições que dele emanam, bem como o atributo de estado final.

Note-se que os seguintes estados tornam-se inacessíveis, e portanto dispensáveis: 1, 2, 8, 10, 13, 17, 19, 23, 25 e 30.

O estado 0 não é referenciado, porém não pode ser eliminado por tratar-se do estado inicial da sub-máquina.

(*c*) *expressão*:

	I	N	⋖	E	⋗	+	−	*	/	ε
0	2	3	4							
①						19	20	9	10	~~7~~
②						19	20	9	10	~~1~~
③						19	20	9	10	~~1~~
4				5						
5					6					
⑥						19	20	9	10	~~1~~
⑦						19	20	9	10	~~17~~
8	12	13	14							
9	12	13	14							~~8~~
10	12	13	14							~~8~~
⑪						19	20	9	10	~~7~~
⑫						19	20	9	10	~~11~~
⑬						19	20	9	10	~~11~~
14				15						
15					16					
⑯						19	20	9	10	~~11~~
⑰						19	20			
18	22	23	24							
19	22	23	24							~~18~~
20	22	23	24							~~18~~
㉑						19	20	29	30	~~27~~
㉒						19	20	29	30	~~21~~
㉓						19	20	29	30	~~21~~
24				25						
25					26					
㉖						19	20	29	30	~~21~~
㉗						19	20	29	30	~~17~~
28	32	33	34							
29	32	33	34							~~28~~
30	32	33	34							~~28~~
㉛						19	20	29	30	~~27~~
㉜						19	20	29	30	~~31~~
㉝						19	20	29	30	~~31~~
34				35						
35					36					
㊱						19	20	29	30	~~31~~

Observações:

Os estados 7 e 27 assimilam as transições correspondentes ao estado 17, para o qual transitam em vazio, e recebem dele a característica de estados finais.

Os estados 1, 11, 21 e 31, que transitam em vazio para os estados 7 ou 27, também deles incorporam as transições e o atributo de estados finais.

Os estados 2, 3, 6, 12, 13, 16, 22, 23, 26, 32, 33 e 36 analogamente recebem dos estados 1, 11, 21 ou 31 as características mencionadas, tornando-se estados finais também.

Os estados 9 e 10, 19 e 20, 29 e 30 absorvem, respectivamente, as transições indicadas nos estados 8, 18 e 28.

Tornam-se inacessíveis, após estas modificações, os estados seguintes: 1, 7, 8, 11, 17, 18, 21, 27, 28 e 31, podendo ser eliminados do autômato.

Note-se que, apesar de não referenciado, o estado 0 não pode ser eliminado por tratar-se do estado inicial da submáquina.

Obteve-se, desta maneira, um conjunto de três submáquinas determinísticas, que podem ser representadas conforme as tabelas a seguir. Note-se que nenhuma das células das tabelas apresenta transições não-determinísticas. Caso algum não-determinismo ocorresse, deveria ser eliminado através da criação de estados adicionais compostos pela incorporação das transições que emanam de cada um dos estados referenciados na célula que o descreve.

Não sendo este o caso, pode-se prosseguir, partindo-se das formas determinísticas obtidas:

(*a*) *programa:*

	C	END	;
0	1		
1		5	3
3	4		
4		5	3
(5)			

(*b*) *comando:*

	I	:	LET	:=	E	GO TO	OF	,	READ	PRINT	IF	>	=	<	THEN	C	ELSE
(0)	39		3			7			16	22	28						
3	4																
4				5													
5					6												
(6)																	
7	9																
(9)							11										
11	12																
(12)								14									
14	15																
(15)								14									
(16)	18																
(18)								20									
20	21																
(21)								20									
(22)					24												
(24)								26									
(26)					27												
(27)								26									
28					29												
29												33	32	31			
31					34												
32					34												
33					34												
34															35		
35																36	
36																	37
37																38	
(38)																	
39		40															
(40)	39		3			7			16	22	28						

(*c*) *expressão:*

	I	N	<	E	>	+	—	*	/
0	2	3	4						
(2)						19	20	9	10
(3)						19	20	9	10
4				5					
5					6				
(6)						19	20	9	10
9	12	13	14						
10	12	13	14						
(12)						19	20	9	10
(13)						19	20	9	10
14				15					
15					16				
(16)						19	20	9	10
19	22	23	24						
20	22	23	24						
(22)						19	20	29	30
(23)						19	20	29	30
24				25					
25					26				
(26)						19	20	29	30
29	32	33	34						
30	32	33	34						
(32)						19	20	29	30
(33)						19	20	29	30
34				35					
35					36				
(36)						19	20	29	30

No processo de otimização do autômato, o passo seguinte é o de manipular as submáquinas determinísticas obtidas, visando à eliminação de redundâncias causadas pela presença de estados equivalentes. Como se sabe, dois estados são considerados equivalentes quando for impossível distingui-los aplicando-se uma mesma seqüência finita de entradas, ou seja, quando, para qualquer seqüência de entrada, a seqüência de estados pelos quais o autômato evolui, partindo-se dos dois estados suspeitos de equivalência, for composta de estados correspondentes iguais ou indistinguíveis (equivalentes). Assim sendo, descarta-se de partida a possibilidade de equivalência entre dois estados que transitem com conjuntos diferentes de átomos, bem como entre um estado final e um estado não final. Por outro lado, são equivalentes estados que, sendo ambos finais ou não finais, evolvam para os mesmos estados para cada átomo de entrada considerado. Os outros casos devem ser analisados com mais cuidado, pois os estados não podem ser diretamente classificados em equivalentes ou não por simples inspeção.

A seguir, as três submáquinas serão minimizadas:

(*a*) *programa:*

Estados não finais: {0, 1, 3, 4}
Estados finais: {5}

Desta primeira classificação, chega-se à conclusão que o *estado 5 não tem equivalentes.* Entre os demais, transitam com os mesmos átomos os seguintes grupos:

transitam com C: {0, 3}
transitam com END e ;: {1, 4}

Assim, os estados do primeiro grupo não podem ser equivalentes a nenhum estado do segundo, ou vice-versa, uma vez que são distinguíveis por uma entrada de comprimento 1.

Resta verificar se os estados 0 e 3 são equivalentes, bem como 1 e 4.

Observando-se a tabela, verifica-se que 0 será equivalente a 3 apenas se os estados para onde ambos transitam forem iguais ou equivalentes, ou seja, 0 é equivalente a 3 apenas se 1 for equivalente a 4, pois do estado 0 parte uma transição para o estado 1, enquanto do estado 3, para a mesma entrada, parte uma transição para o estado 4.

Observando-se os estados 1 e 4, percebe-se que não são distinguíveis, já que ambos transitam para o estado 5 como resposta ao átomo END, e para estado 3 como resposta ao átomo ;, sendo estas as únicas transições possíveis nestes dois estados. Logo, os *estados 1 e 4 são equivalentes.*

Com esta conclusão, a pendência anterior a respeito dos estados 0 e 3 se desfaz, podendo-se, portanto, afirmar que os *estados 0 e 3 são equivalentes.*

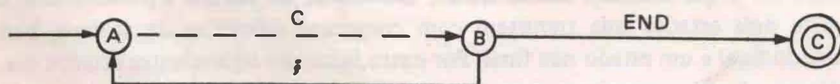
Nada mais havendo para minimizar, pode-se considerar como mínima a submáquina obtida através da identificação entre os estados equivalentes obtidos.

Estado A: correspondente aos estados 0 e 3 (não final)
Estado B: correspondente aos estados 1 e 4 (não final)
Estado C: correspondente ao estado 5 (final)

A tabela de transições obtida rebatizando-se os grupos de estados equivalentes com nomes comuns torna-se:

	C	END	;
A	B		
B		C	A
(C)			

O diagrama de estados correspondente a esta tabela é o seguinte:



Este autômato implementa a forma mínima da submáquina responsável pelo reconhecimento do não-terminal programa.

(*b*) *comando:*

Estados não finais: {3, 4, 5, 7, 11, 14, 20, 28, 29, 31, 32, 33, 34, 35, 36, 37, 39}.
Estados finais: {0, 6, 9, 12, 15, 16, 18, 21, 22, 24, 26, 27, 38, 40}.

Desta classificação nada se pode concluir a não ser que os estados de cada conjunto não podem ser equivalentes a nenhum dos estados do outro conjunto.

Pela observação da tabela, pode-se criar novos grupos, menores, correspondentes aos estados que têm alguma possibilidade de serem equivalentes por transitarem com as mesmas entradas.

Estados não finais que transitam apenas com I: {3, 7, 11, 14, 20}.
Estados não finais que transitam apenas com :=: {4}.
Estados não finais que transitam apenas com E: {5, 28, 31, 32, 33}.
Estados não finais que transitam com >, = e <: {29}.
Estados não finais que transitam com THEN: {34}.
Estados não finais que transitam apenas com C: {35, 37}.
Estados não finais que transitam apenas com ELSE: {36}.
Estados não finais que transitam apenas com :: {39}.

Estados finais que transitam com I, LET, GOTO, READ, PRINT e IF: {0, 40}.
Estados finais que não transitam com nenhum átomo: {6, 38}.
Estados finais que transitam apenas com OF: {9}.
Estados finais que transitam apenas com ,: {12, 15, 18, 21, 24, 27}.
Estados finais que transitam apenas com E: {22, 26}.
Estados finais que transitam apenas com I: {16}.

Desta classificação, conclui-se diretamente que os *seguintes estados não têm nenhum equivalente: 4, 29, 34, 36, 39, 9, 16.*

Cada um dos demais grupos apresentam estados que podem ser equivalentes entre si, mas que nunca podem ser equivalentes a qualquer dos estados que não pertencem ao mesmo grupo.

Pode-se analisar em seguida os grupos que contêm apenas dois estados, em busca de equivalência triviais:

Os estados 35 e 37 serão equivalentes apenas se os estados 36 e 38, para onde transitam com C, também o forem. Mas, como os estados 36 e 38 pertencem a grupos diferentes, pode-se afirmar que tanto o *estado 35 como o estado 37 não têm equivalentes.*

Os estados 0 e 40 são indistinguíveis por inspeção direta, já que as mesmas entradas os levam aos mesmos estados. Assim, pode-se concluir que os *estados 0 e 40 são equivalentes.*

Os estados 6 e 38 são também equivalentes, uma vez que, não tendo estados sucessores, são também indistinguíveis.

Os estados 22 e 26 também são equivalentes, já que os estados 24 e 27, para os quais transitam, evoluem para o mesmo estado (26).

Passa-se, a seguir, à análise dos estados que compõem os outros grupos.

Para o grupo {3, 7, 11, 14, 20}, verifica-se que estados que o compõem transitam respectivamente para os estados 4, 9, 12, 15 e 21.

Como o estado 4 é não equivalente a qualquer outro, então o estado 3, a ele correspondente, é distinto dos demais. O mesmo ocorre com o estado 9, o que leva à conclusão que o estado 7 também é distinto dos demais.

Já os estados 11, 14 e 20 não podem ser classificados como distinguíveis apenas com esta observação, já que os estados 12, 15 e 21, para onde transitam, pertencem a um mesmo grupo, que ainda não foi analisado. Fica assim pendente a conclusão a respeito de sua possível equivalência.

Assim, da análise deste grupo, pode-se concluir, até o momento, apenas que *os estados 3 e 7 não são equivalentes a qualquer outro estado.*

Procurando resolver a pendência acima, pode-se passar à análise do grupo {12, 15, 18, 21, 24, 27}. Pode-se, por inspeção, reconhecer a indistinguibilidade dos estados 12 e 15, bem como dos estados 18 e 21, e também dos estados 24 e 27, uma vez que cada um destes pares transita para o mesmo estado (14, 20 e 26, respectivamente).

É necessário, agora, verificar se há equivalência entre alguns dos estados de pares diferentes.

O estado 12 será equivalente ao estado 18 se os estado 14 e 20 forem indistinguíveis.

A equivalência entre os estados 14 e 20 está, por sua vez, condicionada à indistinguibilidade entre os estados 15 e 21, os quais, por sua vez, dependem da indistinguibilidade entre os estados 14 e 20.

Como se pode notar, não existe nenhum tipo de transição que seja capaz de distinguir entre estes estados, o que leva a concluir que são indistinguíveis os estados 12 e 18 originalmente considerados. Resta verificar se entre os estados 12 e 24 existe algum tipo de distinção. Os estados 12 e 24 serão distinguíveis se os estados 14 e 26 o forem. Como os estados 14 e 26 pertencem a grupos diferentes, conclui-se que não pode haver equivalência entre os estados 12 e 24 e, portanto, entre os estados a eles equivalentes.

Desta longa análise, pode-se concluir que: *os estados 12, 15, 18 e 21 são equivalentes entre si,* bem como *os estados 24 e 27 também são equivalentes entre si.*

Voltando agora à pendência relativa à possível equivalência entre estados do grupo {11, 14, 20}: As transições que emanam destes três estados desviam para os estados 12, 15 e 21 respectivamente. Como estes estados são equivalentes entre si, então pode-se concluir que os *estados 11, 14 e 20 também são equivalentes entre si.*

Resta analisar o grupo {5, 28, 31, 32, 33}. Uma equivalência óbvia entre os estados 31, 32 e 33 pode ser detectada por inspeção, já que os três estados transitam para o estado 34, um estado que não tem equivalentes.

Para os demais estados, verifica-se que transitam para os estados 6, 27 e 29 respectivamente. Cada um destes estados pertence a um grupo diferente, não podendo ser equivalente a nenhum dos outros.

Desta análise, pode-se concluir que: *os estados 31, 32 e 33 são equivalentes entre si,* e *os estados 5 e 28 não possuem estados equivalentes.*

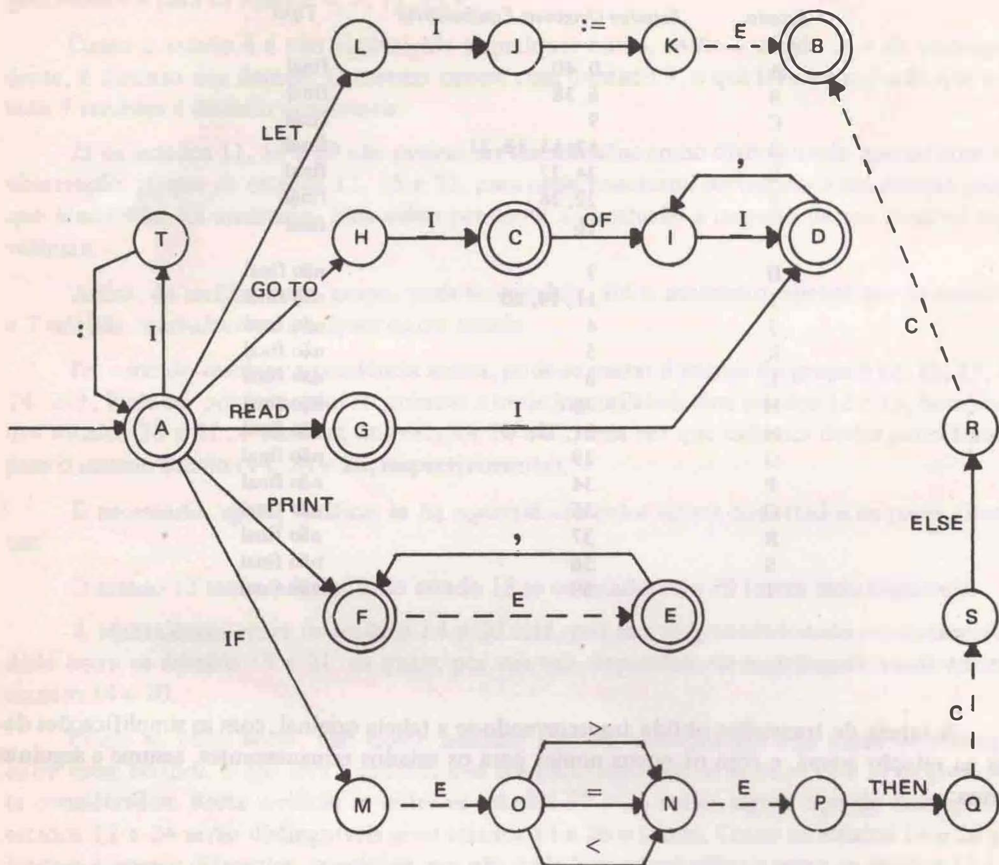
A submáquina mínima correspondente ao não-terminal comando deverá ter os estados correspondentes às classes de equivalência seguintes:

Estado	*Estados Originais Equivalentes*	*Tipo*
A	0, 40	final
B	6, 38	final
C	9	final
D	12, 15, 18, 21	final
E	24, 27	final
F	22, 26	final
G	16	final
H	7	não final
I	11, 14, 20	não final
J	4	não final
K	5	não final
L	3	não final
M	28	não final
N	31, 32, 33	não final
O	29	não final
P	34	não final
Q	35	não final
R	37	não final
S	36	não final
T	39	não final

A tabela de transições obtida transcrevendo-se a tabela original, com as simplificações dadas na relação acima, e com os novos nomes para os estados remanescentes, assume a seguinte forma:

	I	:	LET	:=	E	GO TO	OF	,	READ	PRINT	IF	<	=	>	THEN	C	ELSE
Ⓐ	T		L			H			G	F	M						
Ⓑ																	
Ⓒ							I										
Ⓓ								I									
Ⓔ								F									
Ⓕ					E												
Ⓖ	D																
H	C																
I	D																
J				K													
K					B												
L	J																
M					O												
N					P												
O												N	N	N			
P															Q		
Q																S	
R																B	
S																	R
T		A															

O diagrama de estados da submáquina mínima assim projetada torna-se:



(*c*) *expressão:*

Estados não finais: {0, 4, 5, 9, 10, 14, 15, 19, 20, 24, 25, 29, 30, 34, 35}.
Estados finais: {2, 3, 6, 12, 13, 16, 22, 23, 26, 32, 33, 36}.

Com esta classificação inicial, os estados estão separados em grupos mutuamente exclusivos, do ponto de vista de equivalência. O refinamento seguinte é feito conforme as entradas em cada estado:

Estados não finais que transitam com I, N, ≮: {0, 9, 10, 19, 20, 29, 30}.
Estados não finais que transitam só com E: {4, 14, 24, 34}.
Estados não finais que transitam só com ≯: {5, 15, 25, 35}.
Estados finais que transitam com +, –, *, /: {2, 3, 6, 12, 13, 16, 22, 23, 26, 32, 33, 36}.

Note-se que este critério não foi suficiente para refinar o conjunto dos estados finais. Os estados não finais, apesar de terem sido subdivididos em três conjuntos, não permitem ainda nenhuma conclusão imediata acerca da sua equivalência mútua.

Por observação da tabela, pode-se concluir diretamente acerca da equivalência entre os pares 9-10, 19-20 e 29-30, assim como daquela que existe entre os estados dos grupos {2, 3, 6, 12, 13, 16} e {22, 23, 26, 32, 33, 36}, uma vez que seus elementos transitam para os mesmos estados. É preciso verificar se entre os três pares acima, ou entre os elementos dos dois grupos listados, existe alguma equivalência:

Inicialmente, pode-se verificar se entre 0 e 9 há possibilidade de equivalência. Para 0 e 9 serem indistinguíveis, deve haver equivalência entre 2 e 12, 3 e 13, 4 e 14, o que pode ser verdade, uma vez que pertencem aos mesmos grupos. As duas primeiras afirmações são verdadeiras, devido ao que foi exposto acima. Para que 4 e 14 sejam distinguíveis, é necessário que 5 e 15 o sejam, o que exige que 6 e 16 também o sejam. Mas 6 e 16 são estados equivalentes, conforme foi estudado acima. Assim sendo, os três requisitos necessários para se afirmar a não distinguibilidade entre os estados 0 e 9 são satisfeitas.

Raciocínio análogo leva à conclusão que 9 e 19 são indistinguíveis, bem como 19 e 29. Assim sendo, todos os estados do grupo são indistinguíveis, e, portanto, *0, 9, 10, 19, 20, 29 e 30 são estados equivalentes.*

Durante o estudo que leva a esta conclusão, chega-se a verificar que 5 e 15, 15 e 25, e 25 e 35 são pares de estados indistinguíveis, e o mesmo ocorre em relação aos pares 4 e 14, 14 e 24, e 24 e 34. Isto leva a concluir que os estados em questão formam duas classes de equivalência. A conclusão final é *de que os estados 4, 14, 24 e 34 são equivalentes, o mesmo ocorrendo com os estados 5, 15, 25 e 35.*

Em relação aos dois grupos de estados finais, basta verificar se o estado 2 é equivalente ou não ao estado 22. Para isto, é necessário que haja a equivalência entre os pares 9-29 e 10-30, o que já foi demonstrado. Logo, conclui-se que *são todos equivalentes entre si os estados 2, 3, 6, 12, 13, 16, 22, 23, 26, 32, 33 e 36.*

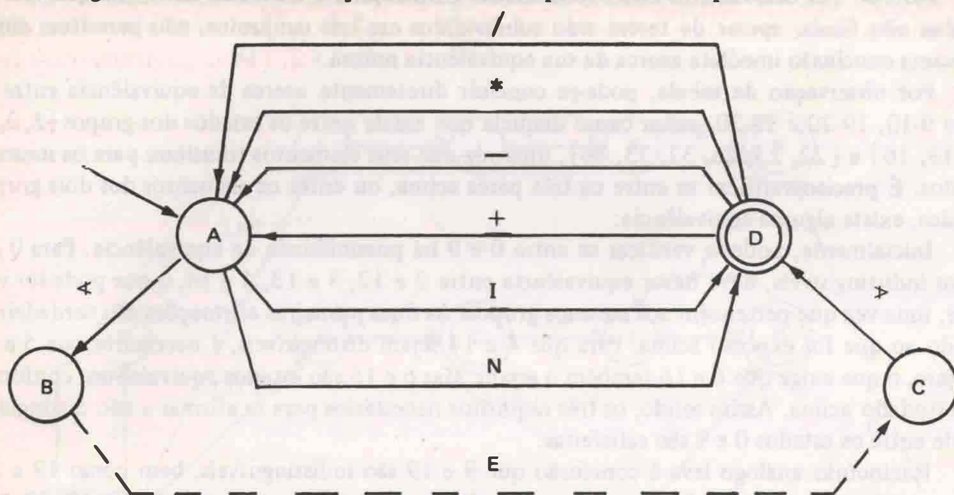
Nenhuma outra simplificação é viável, logo estes estados formam a submáquina mínima para o não-terminal expressão, de acordo com as equivalências seguintes:

Estado	*Estados Originais Equivalentes*	*Tipo*
A	0, 9, 10, 19, 20, 29, 30	não final
B	4, 14, 24, 34	não final
C	5, 15, 25, 35	não final
D	2, 3, 6, 12, 13, 16, 22, 23, 26, 32, 33, 36	final

A tabela de transições da submáquina minimizada é a seguinte:

	I	*N*	<	*E*	>	+	−	*	/
A	D	D	B						
B				C					
C					D				
ⓓ						A	A	A	A

O diagrama de estados correspondente à submáquina mínima é portanto:



Completa-se, desta maneira, o projeto do analisador sintático da linguagem proposta. O analisador construído possui submáquinas mínimas, que estão prontas para receber as ações semânticas, formando, desta maneira, o compilador desejado.

5.6 – PROJETO DAS AÇÕES SEMÂNTICAS

Uma vez montado o autômato que deverá servir como base para o compilador, torna-se necessário projetar as rotinas que devem ser associadas a cada uma das suas transições, de modo tal que o conjunto se comporte como um transdutor que traduza o texto-fonte para a forma do texto-objeto especificado anteriormente.

Para que isto seja possível, torna-se importante efetuar um estudo da maneira como cada construção da linguagem de alto nível deve corresponder à linguagem-objeto, e quais são as alterações necessárias no estado das estruturas de dados do compilador, para que o referido mapeamento seja possível.

À medida que este levantamento for realizado, procurar-se-á definir as necessidades correspondentes, relativas às diversas estruturas de dados que forem necessárias à realização da tradução do programa-fonte.

Segue inicialmente um estudo individual de cada uma das construções do programa-fonte. Através deste estudo pretende-se levantar as necessidades básicas do compilador. Em seguida, são detalhadas as estruturas de dados a serem utilizadas, e especificadas as rotinas de acesso às mesmas. Finalmente, aos autômatos serão anexadas as rotinas semânticas, que implementam as funções necessárias ao processamento em questão.

(*a*) *Comando vazio*

Tratando-se apenas de um recurso sintático, o comando vazio não deve gerar código, nem alterar o estado das estruturas de dados do compilador.

(*b*) *Comando de atribuição*

Das partes que compõem o comando de atribuição, são significativas, para a compilação, o identificador à esquerda do sinal :=, e a expressão que figura à sua direita. O identificador

corresponde à variável que representa a posição de memória onde deve ser depositado o valor associado à expressão. Assim sendo, é necessário que tal identificador não tenha sido previamente definido ou utilizado como rótulo. Se for sua primeira aparição no programa, deve caracterizar-se como um identificador de variável.

Quanto à expressão, deve ser calculada de maneira convencional, como será estudado adiante. O resultado do cálculo deve ser depositado no acumulador da máquina ao final da avaliação da expressão.

Desta maneira, um comando de atribuição, cuja forma geral é do tipo:

```
LET identificador := expressão
```

pode ser traduzido para linguagem-objeto de saída como:

```
        .
        .
        .
código objeto correspondente à expressão
(resultado no acumulador)
STA identificador
        .
        .
        .
```

Note-se a inversão na ordem entre o identificador e a expressão no código-objeto, em relação ao texto-fonte. Isto exigirá que o identificador seja memorizado antes da compilação da expressão, para que possa ser utilizado para a geração da instrução STA que finaliza o código-objeto do comando de atribuição.

(*c*) *Comandos de desvio*

Estes comandos apresentam-se, na linguagem deste exemplo, em duas formas básicas. A primeira, mais simples, referencia diretamente o rótulo para onde um desvio incondicional deve ser realizado:

```
GO TO identificador
```

Neste caso, o identificador em questão deve referir-se a um rótulo, que deve ter aparecido em algum ponto do programa já tratado, ou deverá ocorrer em algum ponto do texto-fonte ainda não compilado. Em alguma ocasião deve ser efetuada uma verificação de que todos os identificadores referenciados em comandos de desvio devem ocorrer como rótulos em alguma parte do texto-fonte. Neste exemplo, esta verificação é efetuada no final físico do programa.

A tradução do desvio incondicional simples é trivial:

```
BRI identificador
```

A segunda forma do comando de desvio é a seguinte:

```
GO TO identificador, OF identificador0, identificador1, ..., identificadorn
```

Para este caso, a interpretação é a seguinte: o identificador, que consta entre as palavras GO TO e OF, representa uma variável que deve conter um número a ser usado como índice. Se o seu conteúdo for k, então deve ser feito um desvio para o rótulo representado por identificador_k. Se seu conteúdo não estiver contido no intervalo de 0 a n, o comando não deve ter nenhum efeito. Para gerar código para este comando, pode-se optar por carregar inicialmente identificador no acumulador, e decrementá-lo de uma em uma unidade, desviando condicionalmente para os identificadores à direita da palavra OF, sucessivamente, a cada decremento:

```
LDA   identificador

BRZ   identificador0
SUB   = 1
BRZ   identificador1
  .
  .
  .
SUB   = 1
BRZ   identificadorn
```

Note-se que se identificador contiver números fora do intervalo esperado, o código-objeto não executará o desvio especificado em nenhuma das instruções BRZ geradas.

(*d*) *Comando de leitura*

O comando de leitura assume duas formas, uma das quais especifica uma lista de variáveis a serem lidas, enquanto a outra não apresenta parâmetros, indicando que apenas deve ser saltada a próxima linha de dados de entrada.

Na primeira forma, cuja aparência é

```
READ
```

o código-objeto a ser gerado pode ser simplesmente a chamada de uma subrotina do ambiente de execução, encarregada de ler uma linha. O produto desta leitura, neste caso, é descartado pelo programa-objeto.

```
CALL # READ
```

Para a segunda forma, uma lista de variáveis é apresentada, referindo-se aos endereços de memória onde devem ser depositados os resultados da leitura.

```
READ identificador₁, identificador₂, ..., identificadorₙ
```

Esta forma do comando de leitura pode ser traduzido como uma seqüência de chamadas de # READ, seguidas do armazenamento dos valores lidos na variável correspondente. # READ deverá, neste caso, depositar o produto da leitura no acumulador para que possa ser utilizado desta forma.

```
CALL # READ
STA    identificador₁
CALL # READ
STA    identificador₂
  .
  .
  .
CALL # READ
STA    identificadorₙ
```

(*e*) *Comando de impressão*

Os formatos sintáticos do comando de impressão são semelhantes aos do comando de leitura. Ao invés de identificadores de variáveis, porém, o comando de impressão explicita uma lista de expressões cujos valores devem ser impressos. Na sua forma mais simples, a aparência do comando de impressão é a seguinte:

```
PRINT
```

O código a ser gerado para esta forma do comando de impressão limita-se à chamada de uma subrotina do ambiente de execução, encarregada de mudar de linha no dispositivo de saída:

```
CALL # NEWLINE
```

A outra forma do comando de impressão exige a avaliação de uma lista de expressões, e, para cada um dos valores calculados, a chamada de uma rotina de conversão para imprimir o valor numérico calculado, em formato decimal, no dispositivo de saída. Como a linguagem especifica que as diversas expressões são impressas uma por linha, a rotina de mudança de linha deve também ser chamada pelo código-objeto. Assim, para a forma seguinte:

```
PRINT expressão₁, expressão₂, ..., expressãoₙ
```

o código a ser gerado pelo compilador é o seguinte:

```
código relativo à expressão₁
CALL # CONVERT
CALL # NEWLINE

código relativo à expressão₂
CALL # CONVERT
CALL # NEWLINE
      .
      .
      .
código relativo à expressãoₙ
CALL # CONVERT
CALL # NEWLINE
```

onde o código relativo à expressão$_k$ refere-se ao programa-objeto gerado pelas ações semânticas executadas na compilação da expressão$_k$, dirigida pela submáquina correspondente ao não-terminal expressão.

(*f*) *Comando de decisão*

Há uma só forma para este comando, a saber:

IF expressão$_1$ operador de comparação expressão$_2$ THEN comando$_1$ ELSE comando$_2$

Para gerar código para este comando, torna-se necessário inicialmente que o compilador avalie a comparação, verificando se a relação existente entre as duas expressões é verdadeira ou falsa:

```
código relativo à expressão₁
STA # TEMP
código relativo à expressão₂
SUB # TEMP
```

A variável auxiliar # TEMP é utilizada para guardar o valor da primeira expressão, enquanto a segunda expressão é avaliada. Ao final da execução deste código, o resultado obtido no acumulador será o valor de expressão$_2$ – expressão$_1$, o que permite o uso deste resultado em instruções de desvio condicional para implementar a tradução do comando. Para cada operador de comparação, os desvios condicionais a serem utilizados devem ser escolhidos conforme os três casos seguintes:

– operador > :

```
BRP # THENᵢ
BRI # ELSEᵢ
```

– operador = :

```
BRZ # THENi
BRI # ELSEi
```

– operador < :

```
BRN # THENi
BRI # ELSEi
```

Os rótulos # $THEN_i$ e # $ELSE_i$ referem-se às posições do código-objeto onde se iniciam respectivamente os códigos referentes a $comando_1$ e $comando_2$. Assim, para prosseguir a geração do código, é necessário que, ao início da geração do código relativo ao $comando_1$, seja gerada uma pseudoinstrução de definição de rótulo:

```
LBL # THENi
código relativo ao comando1
```

Ao ser compilado o $comando_2$, em seqüência no código-objeto, deve-se tomar o cuidado de gerar um comando de desvio incondicional antes do código correspondente, para que, terminada a execução do $comando_1$ no caso de a comparação ter sido verdadeira, não prossiga executando o $comando_2$, mas desvie para o final do comando de decisão. Os rótulos correspondentes ao início do $comando_2$ e ao final global do comando de desvio devem ser também gerados para completar o código objeto do comando de decisão.

```
BRI # FIMi
LBL # ELSEi
código relativo ao comando2
LBL # FIMi
```

Cabem aqui importantes observações acerca da utilização destes rótulos (# TEMP, # $THEN_i$, # $ELSE_i$, # FIM_i):

– o rótulo # TEMP refere-se a uma posição de memória que deve ser utilizada como variável temporária pelo comando de decisão. É utilizada apenas durante a execução do cálculo da comparação. Desta maneira, uma única posição de memória pode ser utilizada por todos os comandos de decisão para esta finalidade, uma vez que sua utilização é efêmera. Torna-se assim necessário reservar uma posição # TEMP, única para todo o código-objeto. Opta-se por efetuar esta geração no final da compilação.

– os rótulos # $THEN_i$, # $ELSE_i$ e # FIM_i, por outro lado, referem-se a posições de memória correspondentes a pontos de desvio no código-objeto, sendo, desta maneira, característicos do comando de decisão *i* ao qual se referem.

Para implementá-los, torna-se conveniente utilizar um contador, em tempo de compilação, cujo valor é zero no início da compilação e deve ser incrementado a cada início de um novo comando de decisão.

A aparência destes rótulos, personalizados em relação aos comandos de decisão correspondentes, torna-se: # $THEN_1$, # $ELSE_1$, # FIM_1, . . ., # $THEN_{35}$, # $ELSE_{35}$, . . .

— a geração das instruções de máquina que utilizam estes rótulos depende da informação exata de qual rótulo deve ser gerado em cada ocasião. Considerando que a estrutura de encadeamento dos comandos de decisão corresponde a uma árvore, pode-se manter sempre a informação, necessária para a correta geração das instruções do código-objeto, através do uso de uma pilha. Esta pilha deve receber um número correspondente ao comando de decisão sempre que se iniciar sua compilação, sendo descartado o conteúdo do seu topo sempre que terminar a compilação de um comando de decisão. Toda vez que for necessário gerar algum rótulo no código, o número associado ao comando de decisão corrente estará, desta maneira, disponível no topo da pilha (um esquema semelhante pode ser utilizado para a compilação de todas as estruturas de controle convencionais, das linguagens de alto nível usuais).

(*g*) *Declarações de rótulos*

A última construção a ser analisada na submáquina correspondente ao não-terminal comando, quanto à geração do código-objeto, é a declaração de rótulo. Rótulos são definidos através da colocação dos identificadores a eles correspondentes diante do comando a que se referem.

No caso geral, mais de um rótulo pode estar associado ao mesmo comando:

```
identificador₁ : identificador₂ : . . .  identificadorₙ : comando
```

Cabe ao compilador verificar incoerências quanto ao uso dos diversos identificadores: não podem ter sido declarados anteriormente, nem poderão ser redeclarados posteriormente. Não podem, ainda, ter sido utilizados como variáveis. Podem, no entanto, ter sido referenciados anteriormente através de comandos de desvio. O código gerado é trivial:

```
LBL   identificador₁
LBL   identificador₂
  .
  .
  .
LBL   identificadorₙ
```

Para completar o estudo da forma a ser utilizada na geração do código-objeto, restam ainda duas construções: a da seqüência de comandos que compõem o programa, e a das expressões.

(*h*) *expressões*

As expressões são compiladas com o auxílio de uma análise da seqüência das operações a serem realizadas. Como existe precedência do produto e da divisão sobre a soma e a subtração, e como os parênteses são utilizados para alterar a ordem da execução das operações, torna-se necessário, ao gerar o código, promover uma verificação da influência desta ordem sobre o cálculo do valor associado à expressão. Novamente, o uso de pilhas auxilia na resolução dos problemas suscitados na compilação destas construções.

Para a geração do código das expressões da linguagem-exemplo, far-se-á uso de duas pilhas. A primeira é utilizada para guardar informações sobre operadores que já foram lidos no texto-

fonte, mas ainda não foram tratados para efeito de geração de código, devido a problemas de precedência, parênteses, ou porque os operandos a eles relativos ainda não foram obtidos. A segunda pilha é utilizada analogamente para o armazenamento de operandos, relativamente aos quais não foi ainda obtido o código-objeto.

Na pilha de operadores são armazenados os sinais de operação e delimitadores (parênteses e indicadores de pilha vazia). Na pilha de operandos, são guardados identificadores e números inteiros ainda não tratados.

Pode-se com estas estruturas utilizar, para a geração de código, um algoritmo simples em que são efetuados empilhamentos a cada aparecimento de operandos e de parênteses, enquanto, ao ser encontrado um operador, deve ser efetuada uma análise do conteúdo do topo da pilha de operadores. Caso seja encontrado um operador de precedência menor ou igual na pilha de operadores, gera-se o código a ele correspondente, desempilhando-se o operador e os dois operandos associados. O processo deve prosseguir até que seja encontrado um operador de precedência maior, ou então um delimitador. Nesta ocasião, o novo operador deve ser empilhado para o uso futuro.

A geração de código escolhida para este exemplo é a mais trivial possível, e portanto, o código gerado não é muito eficiente. Este ponto, porém, não prejudica o compilador, visto que na sua especificação não são feitas exigências de que apresente alto desempenho.

Assim, para cada operação a ser gerada como código-objeto, será efetuada a geração de uma seqüência de três instruções, segundo os esquemas abaixo:

Topo da Pilha de Operadores	*Topo da Pilha de Operandos*	*2ª Posição da Pilha de Operandos*	*Código Objeto a Ser Gerado:*
+	A	B	LDA B ADA A STA # $TEMP_i$
–	A	B	LDA B SUB A STA # $TEMP_i$
*	A	B	LDA B MUL A STA # $TEMP_i$
/	A	B	LDA B DIV A STA # $TEMP_i$

A cada geração de uma seqüência de três instruções, deve-se empilhar, na pilha de operandos, a variável temporária # $TEMP_i$, e incrementar a variável contadora i.

A ineficiência causada pelo uso de temporários desnecessários pode ser eliminada através da inclusão de algoritmos de otimização, ou da alteração do algoritmo de geração para evitar a criação de temporários onde não for estritamente necessário. Isto não será incluído no exemplo, embora seja relativamente trivial inserir testes para a detecção e eliminação da criação de tem-

porários novos que não sejam essenciais, bem como a eliminação da geração de código redundante, como por exemplo as seqüências do tipo STA X; LDA X ou semelhantes, que podem deixar de ser geradas, uma vez que neste caso X já se encontra no acumulador. Sendo X um temporário, torna-se possível evitar sua criação.

(*i*) *programa*

A forma geral de um programa é

```
comando₁; comando₂; . . .  comandoₙ END
```

onde $n \geqslant 1$. O código referente a um programa consiste em uma seqüência de códigos, cada qual correspondendo a um dos comandos, gerados como foi estudado anteriormente.

Ao seu final, um código suplementar deve ser gerado, após o último comando, com a função de terminar o processamento. Adicionalmente, este código deve incluir o conjunto de rotinas de biblioteca do ambiente de execução utilizadas no programa, bem como pseudoinstruções encarregadas de definir áreas de memória para as variáveis do programa e para uso do próprio código-objeto como regiões de rascunho no cálculo de expressões e em outras situações congêneres. Ao final, uma pseudoinstrução de final de programa deve ser gerada, para uso do montador. O código gerado tem a seguinte organização: inicialmente, comparece todo o código correspondente às instruções compiladas:

```
código referente ao comando₁
código referente ao comando₂
            .
            .
            .
código referente ao comandoₙ
      CALL # STOP
```

A subrotina # STOP pertence ao ambiente de execução, e sua função é a de retornar o controle da execução ao sistema operacional hospedeiro, se houver, ou então, simplesmente parar a execução do programa.

A seguir, devem ser geradas áreas para as variáveis do programa.

Supondo que os identificadores utilizados para as variáveis sejam $identificador_1$, $identificador_2$, ..., $identificador_n$, o código-objeto deve apresentar-se como:

```
LBL  identificador₁
DS   Ø
LBL  identificador₂
DS   Ø
.
.
.
LBL  identificadorₙ
DS   Ø
```

Pode-se em seguida gerar área para as variáveis auxiliares utilizadas pelo compilador: #TEMP, utilizada como área temporária na avaliação das comparações, e $\#TEMP_1$, ..., $\#TEMP_n$, utilizadas como áreas temporárias na avaliação das expressões aritméticas:

```
LBL # TEMP
DS Ø
LBL # TEMP1
DS Ø
LBL # TEMP2
DS Ø
  .
  .
  .
LBL # TEMPn
DS Ø
```

Finalmente, pode ser explicitado o código-fonte, na linguagem simbólica, correspondente às rotinas do ambiente de execução, ou então, caso estejam tais rotinas disponíveis em uma biblioteca do sistema (como é o caso deste projeto e da grande maioria dos compiladores reais), podem ser geradas pseudoinstruções para informar ao sistema quais rotinas do ambiente de execução devem ser incluídas no código-objeto:

```
EXT # NEWLINE
EXT # CONVERT
EXT # READ
EXT # STOP
END
```

A pseudoinstrução END, ao final desta seqüência, completa a geração do texto-objeto simbólico correspondente à tradução do programa, indicando ao montador o final físico do mesmo.

Não há necessidade de gerar código referente aos valores das constantes numéricas, utilizadas no programa, uma vez que foi suposta a existência de instruções simbólicas com operando imediato para representá-las. Caso isto não fosse verdade, deveria ser gerado um conjunto de declarações DS, cada qual referente a uma constante utilizada no programa. Neste caso, deveriam tais constantes ser coletadas ao longo da compilação para serem geradas nesta ocasião na forma de definições de posições preenchidas de memória.

Como resultados do estudo realizado acima, acerca da geração de código-objeto a partir do programa-fonte, os principais resultados foram:

- o levantamento das regras de correspondência entre o texto-fonte e o programa-objeto relativo a este texto.
- o levantamento das necessidades do programa-objeto em tempo de execução, e portanto dos requisitos do ambiente de execução.

– o levantamento das necessidades das ações semânticas do compilador, no tocante às estruturas de dados e funções a serem executadas.
– a escolha dos principais métodos e técnicas a utilizar na resolução dos diversos problemas levantados.

Para completar o projeto, é necessário detalhar, ainda, as estruturas de dados a serem empregadas, a lógica das ações semânticas, e a relação entre as ações semânticas e o reconhecedor sintático.

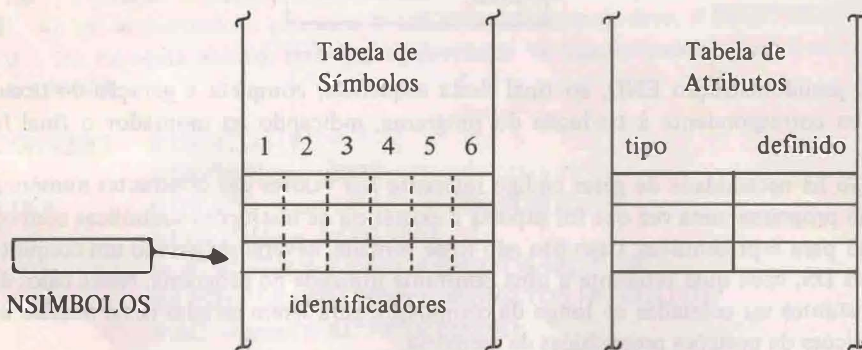
5.6.1 – Estruturas de Dados do Compilador

Para a implementação das ações semântidas, são necessárias algumas estruturas de dados, entre as quais destacam-se:

– *Tabelas de Símbolos e Atributos* – Esta estrutura de dados, de uso comum ao analisador léxico e às ações semânticas, destina-se ao armazenamento dos nomes das variáveis e dos rótulos utilizados pelo programa fonte, com as indicações da aplicação a que se destinam no mesmo, e da utilização que deles faz o programa. Assim sendo, um modo de organizar logicamente esta tabela é através de três campos, relativos às seguintes informações:

– nome do identificador;
– tipo do identificador (rótulo ou variável);
– definido ou indefinido (para rótulos).

O nome do identificador, por simplicidade, pode ser limitado em cinco ou seis caracteres, obtidos por truncamento do identificador utilizado pelo programador, que, em princípio, não tem limitações em seu comprimento.
A organização física desta estrutura de dados pode ser a seguinte:



Uma variável adicional (NSÍMBOLOS) indica o número de símbolos existentes. Na tabela de símbolos, uma linha k contém seis colunas, cada qual relativa a um dos seis primeiros caracteres do nome do identificador utilizado.
Na posição correspondente (k) da tabela de atributos, podem ser encontradas as informações acerca do tipo do identificador e do fato de ter sido previamente definido ou não o símbolo em questão, no caso de tratar-se de um rótulo.

Para empregar esta estrutura de dados, podem ser utilizados os seguintes procedimentos:

- Iniciatabela – faz NSÍMBOLOS = ∅ (esvazia a tabela).
- Pesquisa (símbolo, posição) – efetua uma busca do símbolo passado como parâmetro. O resultado da busca é a posição em que o símbolo foi encontrado (∅ se não o for).
- Insere (símbolo) – acrescenta, na posição NSÍMBOLOS + 1 da tabela, o novo símbolo fornecido, e incrementa NSÍMBOLOS.
- Atributos (posição, tipo, definido) – retorna o conteúdo da tabela de atributos, relativo ao símbolo da posição fornecida.
- Alteratipo (posição, tipo) – preenche o atributo tipo com a informação fornecida.
- Alteradefinido (posição, definido) – preenche o atributo de definição com a informação fornecida.

Com estes procedimentos, não há necessidade de serem efetuados acessos explícitos às tabelas, o que, de um certo modo, protege seus dados contra acessos indevidos, facilitando a depuração do compilador.

- *Tabela de Palavras Reservadas* – Esta tabela é semelhante à tabela de símbolos, porém seu conteúdo é constante, bem como seu comprimento. Destina-se a guardar o conjunto das palavras reservadas da linguagem, para uso por parte do analisador léxico.

  Seu conteúdo e organização são esboçados abaixo:

	1	2	3	4	5
1	G	O			
2	I	F			
3	O	F			
4	T	O			
5	E	N	D		
6	L	E	T		
7	E	L	S	E	
8	R	E	A	D	
9	T	H	E	N	
10	P	R	I	N	T

Uma constante (NRESERVADAS) indica o número total de linhas desta tabela.

Para o uso da tabela de palavras reservadas, é necessário que seja disponível uma rotina de busca:

- Buscareservada (identificador, posição) – compara o identificador fornecido com cada uma das palavras da tabela. Se o identificador for encontrado, o parâmetro posição fornecerá o número da linha da tabela, em que isto ocorreu. Caso contrário, retornará ∅, indicando que o identificador não foi encontrado.

– *Pilhas de Operadores e de Operandos* – Ambas as estruturas, utilizadas na compilação de expressões aritméticas, constam de vetores, destinados a guardar os valores a serem empilhados, e dos correspondentes apontadores para o topo da pilha, cujo valor indica o número de elementos contidos na pilha. Nesta implementação, o contador é o primeiro elemento do vetor que realiza a pilha.

Ambas as pilhas têm implementação como mostra o esboço acima. Para que sejam satisfeitos os acessos necessários, devem estar disponíveis as seguintes rotinas:

– Iniciapilha (Pilha) – faz Pilha [∅] = ∅ ou seja, indica que a pilha em questão está vazia.
– Consulta (Pilha, Dado) – obtém Pilha [Pilha [∅]], que devolve no parâmetro Dado. Se não houver dados, devolve Dado = "⊥"; não altera a pilha.
– Empilha (Pilha, Dado) – Incrementa Pilha [∅]; armazena Dado em Pilha [Pilha [∅]].
– Desempilha (Pilha, Dado) – deposita em Dado o conteúdo de Pilha [Pilha [∅]]; decrementa Pilha [∅].

– *Variáveis complementares* – Foi levantada ainda a necessidade das seguintes variáveis:

CONTAIF – contador de comandos de decisão. Utilizada pelas rotinas semânticas que geram rótulos para as diversas condições dos comandos de decisão.
CONTATEMP – contador de temporários utilizados pelas rotinas semânticas de geração de código para expressões aritméticas.

Para a utilização destas variáveis, devem ser empregadas as seguintes rotinas:

– Iniciacontador (contador) – zera o contador indicado.
– Incrementacontador (contador) – incrementa o contador indicado.

Naturalmente, estando disponíveis os próprios contadores, torna-se mais eficiente efetuar acessos diretos aos mesmos para executar estas operações. O uso de rotinas, porém, permite que sejam auditadas as operações em questão, através da inclusão de comandos de impressão adequados, o que facilita a depuração do compilador, podendo esta técnica ser também aplicada aos demais procedimentos de acesso aos dados.

5.6.2 – Ações Semânticas

Estudadas as estruturas de dados e as funções a serem executadas pelas ações semânticas, resta esboçar sua lógica, bem como sua relação com o reconhecedor sintático. Para tanto, serão aqui repetidas as submáquinas que o implementam, associando-se, às transições convenientes, as ações semânticas a elas referentes. Por comodidade, serão utilizadas as formas tabulares de representação. Nas células serão representadas duas informações: o próximo estado e a ação semântica a executar.

A ordem de apresentação das submáquinas será alterada por conveniência didática.

(*a*) *expressão*

A tabela de transições, com a inclusão das ações semânticas para expressão, é a seguinte:

	I	N	≮	E	≯	+	–	*	/	Ação de Saída
A	D/1	D/2	B/3							11
B				C/4						12
C					D/5					13
(D)						A/6	A/7	A/8	A/9	10

As ações semânticas podem ser construídas de acordo com o seguinte roteiro:

1 – empilha (pilha de operandos, identificador encontrado)
2 – empilha (pilha de operandos, número encontrado)
3 – empilha (pilha de operadores, "(")
4 – nada executa
5 – X5: consulta (pilha de operadores, Y);
Se Y ≠ "(": executa GERACÓDIGO, detalhada adiante; GO TO X5.
Se Y = "(": desempilha (pilha de operadores, Y);
6 – X6: consulta (pilha de operadores, Y);
Se Y for "+", "–", "*" ou "/":
executa GERACÓDIGO, detalhada adiante; GO TO X6;
Caso contrário: empilha (pilha de operadores, "+");
7 – X7: consulta (pilha de operadores, Y);
Se Y for "+", "–", "*", ou "/":
executa GERACÓDIGO, detalhada adiante; GO TO X7;
Caso contrário: empilha (pilha de operadores, "–");
8 – X8: consulta (pilha de operadores, Y);
Se Y for "*" ou "/": executa GERACÓDIGO, detalhada adiante; GO TO X8;
Caso contrário: empilha (pilha de operadores, "*");
9 – X9: consulta (pilha de operadores, Y);
Se Y for "*" ou "/": executa GERACÓDIGO, detalhada adiante; GO TO X9;
Caso contrário: empilha (pilha de operadores, "/");
10 – (*Esta rotina é associada ao final do reconhecimento da expressão*)
X1Ø: consulta (pilha de operadores, Y);
Se Y não for "⊥": executa GERACÓDIGO, detalhada adiante, GO TO X1Ø;
11 – ERRO ("esperava-se identificador, número ou '(' neste ponto").
12 – ERRO ("esperava-se uma expressão correta neste ponto").
13 – ERRO ("esperava-se ')' neste ponto").

As rotinas auxiliares utilizadas pelas ações semânticas acima são as seguintes:

GERACÓDIGO: Desempilha (pilha de operadores, Y);
Desempilha (pilha de operandos, B);
Desempilha (pilha de operandos, A)
Gera ("LDA", A);
Se Y = "+": gera ("ADA", B);
Se Y = "–": gera ("SUB", B);
Se Y = "*": gera ("MUL", B);
Se Y = "/": gera ("DIV", B);
Incrementacontador (CONTATEMP);
Gera ("STA", "# TEMP". CONTATEMP);
empilha (pilha de operandos, "# TEMP". CONTATEMP);

GERA (X, Y) – Esta rotina deve limitar-se a emitir, em um dispositivo de saída, a lista de variáveis fornecidas.
O primeiro parâmetro é o mnemônico da instrução gerada, e o segundo é o operando. Note-se que, em alguns casos, o operando aparece na forma P.Q, designando a concatenação das cadeias de caracteres P e Q, nesta ordem.

ERRO (X) – Esta rotina deve imprimir a mensagem X passada como parâmetro, e contabilizar os erros ocorridos.

(*b*) *comando*

Com a inclusão de ações semânticas, a tabela de transições para comando torna-se:

	I	:	LET	:=	E	GO TO	OF	,	READ	PRINT	IF	>	=	<	THEN	C	ELSE	AÇÃO DE SAÍDA
Ⓐ	T/1		L/2			H/3			G/4	F/5	M/6							27
Ⓑ																		28
Ⓒ							I/7											29
Ⓓ								I/8										30
Ⓔ								F/9										31
Ⓕ					E/10													32
Ⓖ	D/11																	33
H	C/12																	34
I	D/13																	35
J				K/14														36
K					B/15													37
L	J/16																	38
M					O/17													39
N					P/18													40
O												N/19	N/20	N/21				41
P															Q/22			42
Q																S/23		43
R																B/24		44
S																	R/25	45
T		A/26																46

Para simplificar a apresentação das ações semânticas relativas a esta tabela, ao invés de esboçar a codificação da lógica, já exemplificada anteriormente, serão apresentados, para cada uma das construções, a seqüência de ações semânticas associadas, bem como as informações acerca do papel das mesmas na lógica do compilador.

Definição de rótulos

Nesta construção estão envolvidas as rotinas seguintes:

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
1	identificador	LBL identificador	–
26	:	–	–

Comando vazio

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
27	(nenhum)	–	–

Atribuição

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
2	LET	–	Comando:= "LET"
16	identificador	–	(detalhada adiante)
14	:=	–	–
15	expressão	STA identificador	(identificador=destino)
28	(nenhum)	–	(detalhada adiante)

Desvio

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
3	GO TO	–	Comando:="GO TO"
12	identificador	–	(detalhada adiante)
29	(nenhum)	BRI identificador	Assegurar que o identificador é rótulo
7	OF	LDA identificador	Assegurar que o identificador é variável
13	identificador	BRZ identificador	(detalhada adiante)
30	(nenhum)	–	–
8	,	ADA=–1	(detalhada adiante)

Leitura

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
4	READ	CALL # READ	COMANDO:="READ"
11	identificador	STA identificador	–
33	(nenhum)	–	–
8	,	CALL # READ	(detalhada adiante)
30	(nenhum)	–	–
13	identificador	STA identificador	(detalhada adiante)

Escrita

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
5	PRINT	–	–
32	(nenhum)	CALL # NEWLINE	–
10	expressão	CALL # CONVERT CALL # NEWLINE	–
9	,	–	–
31	(nenhum)	–	–

Decisão

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
6	IF	–	COMANDO:= "IF"; CONTAIF:= CONTAIF+1
17	expressão	STA # TEMP	–
19	>	–	OPERADOR:= "BRN"
20	=	–	OPERADOR:= "BRZ"
21	<	–	OPERADOR:= "BRP"
18	expressão	SUB # TEMP operador # THEN contaif BRI # ELSE. contaif	–
22	THEN	LBL # THEN. contaif	–
23	comando	–	–
25	ELSE	BRI # FIM contaif LBL # ELSE contaif	–
24	comando	–	–
28	(nenhum)	LBL # FIM. contaif	(detalhada adiante)

Relativamente às ações semânticas 16, 28, 12, 13 e 8, devido a algumas peculiaridades, são apresentados a seguir esboços contendo um pouco mais de detalhes sobre sua lógica, a título de esclarecimento do seu funcionamento.

```
Ação 16 – (* trata o aparecimento de um identificador do lado esquerdo de um comando de
           atribuição *).
        Pesquisa (identificador, posição);
        Se posição = Ø (*não encontrou o identificador na tabela*)
           então insere (identificador);
                 alteratipo (Nsímbolos, "variável");
                 alteradefinido (Nsímbolos, "definido");
           caso contrário Atributos (posição, tipo, definido);
                 Se tipo ≠ "variável"
                    então erro ("atribuição deve ser feita a variável")
                    caso contrário (*nada faz*);
        Destino:=identificador; (*salva o identificador para uso futuro*)
```

Ação 28 – (*final de comandos LET e IF*)
Se comando = "if"
então gera ("LBL", "#FIM". contaif)
caso contrário (*comando="LET". Nada faz*)

Ação 12 – (*trata o aparecimento do identificador após a palavra GO TO*)
ID:=identificador; (*slva o identificador para uso futuro*)

Ação 13 – (*trata identificadores que são rótulos do desvio indexado, ou da lista de variáveis do comando de leitura*)
Se comando ="READ"
então gera("STA", identificador);
(*deve também assegurar que o identificador seja referente a uma variável*)
caso contrário (*comando="GOTO"*)
gera ("BRZ", identificador);
(*aqui deve-se garantir que o identificador seja referente a um rótulo*).

Ação 8 – (*trata o aparecimento da vírgula separadora dos identificadores nos comandos de desvio indexado e de leitura*)
Se comando ="READ"
então gera ("CALL", #READ");
caso contrário (*comando="GOTO"*)
gera ("ADA", "= –1");

As ações semânticas de números 34 a 36 simplesmente enviam uma mensagem de erro adequada.

Cabe observar a complexidade adicional introduzida nas ações semânticas, devido ao compartilhamento de partes das submáquinas por construções independentes, exigindo testes adicionais para a execução das ações corretas.

Isto pode ser evitado ou contornado identificando estes casos e refazendo a minimização da submáquina de forma parcial, de tal modo que os estados envolvidos com ações semânticas incompatíveis não sejam fundidos uma vez que não são semanticamente equivalentes, apesar de o serem sintaticamente.

Isto, porém, não será efetivado neste texto.

Para completar o desenvolvimento das ações semânticas do compilador, resta apenas a submáquina de *programa*.

(*c*) *programa*

	C	END	;	Ação de Finalização
A	B/1			4
B		C/2	A/3	5
(C)				6

Ações Semânticas	*Átomo Associado*	*Código Gerado*	*Outras Ações*
1	comando	–	–
2	END	(ver detalhes adiante)	(ver detalhes adiante)
3	;	–	–
4	(nenhum)	–	mensagem de erro
5	(nenhum)	–	mensagem de erro
6	(nenhum)	–	finalização da compilação

Ação 2 – (* tratamento do aparecimento de "END" *)
Gera ("CALL", "# STOP");
(* para cada um dos identificadores, declarados como variáveis, aloca neste ponto as áreas de memória necessárias *);
(* faz o mesmo para as variáveis temporárias # TEMP, # TEMP1, ..., TEMPN onde N é o número de temporários utilizados pelo programa, dado por CONTATEMP *);
Gera ("EXT", "# NEWLINE");
Gera ("EXT", "# CONVERT");
Gera ("EXT", "# READ");
Gera ("EXT", "# STOP");
Gera ("END", " ");
(* varre a tabela de símbolos à procura de rótulos indefinidos, imprimindo para cada um que aí figura uma mensagem de erro *);
(* finaliza todas as atividades pendentes e encerra o processamento *);

5.7 – O AMBIENTE DE EXECUÇÃO

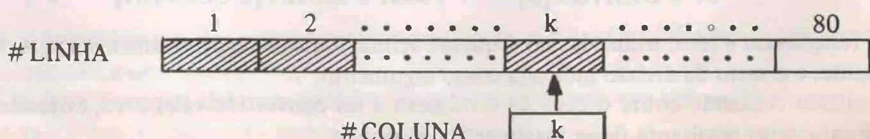
Do projeto das ações semânticas pode-se efetuar um levantamento das necessidades do compilador em matéria de suporte lógico em tempo de execução. Pela observação das exigências do código-objeto projetado, percebe-se facilmente que para este exemplo o ambiente de execução é muito simples, consistindo basicamente de algumas poucas rotinas: # NEWLINE, # CONVERT, # READ e # STOP.

Para este exemplo, as rotinas do ambiente de execução, além de poucas, são muito simples:

(*a*) # NEWLINE

Esta rotina, assim como # READ e # CONVERT, utiliza uma área de memória que consta de um vetor (# LINHA) de comprimento igual ao de uma linha do dispositivo de entrada/saída utilizado pelos comandos de entrada e de saída da linguagem.

A este vetor está associado um contador (# COLUNA), que indica a última posição preenchida:



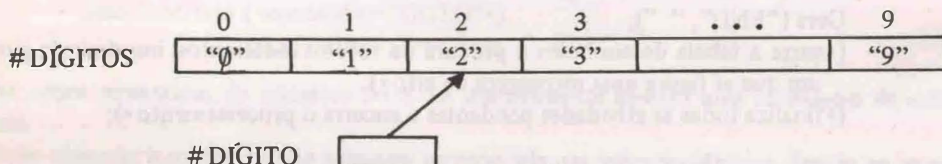
A ação da rotina # NEWLINE consiste em apenas promover a impressão das k primeiras posições de # LINHA, sendo k o valor contido em # COLUNA. Em seguida, # COLUNA pode ser preenchido com o valor zero, indicando que # LINHA está vazia:

```
#NEWLINE: k: = #COLUNA;
          Enquanto k não for nulo
              promove a saída do caractere # LINHA [k – # COLUNA + 1];
              k: = k – 1;
          promove a saída dos caracteres "carriage return" e "line feed";
          #COLUNA: = Ø;
```

(*b*) # CONVERT

A função desta rotina é a de converter para a notação decimal, preparando para ser impresso, o conteúdo do acumulador (# AC). Para isto, executa um algoritmo em que o número contido no acumulador é dividido sucessivamente por potências decrescentes de dez. Os quocientes serão números entre Ø e 9, utilizados para a construção de cadeia de caracteres a ser impressa. Os restos destas divisões são utilizados para prosseguir o algoritmo.

Note-se que os quocientes são também números binários, e não caracteres que podem ser diretamente impressos. Para convertê-los para uma forma que possa ser utilizada diretamente na impressão, é possível utilizá-los como índices de uma tabela de dígitos (em código ASCII, por exemplo), previamente preenchida com os dígitos decimais.



Zeros à esquerda em geral são inconvenientes, razão pela qual, no início da execução de # CONVERT, a posição # DÍGITOS [Ø] contém " ", ou seja um espaço em branco ao invés do dígito "Ø". Ao ser encontrado o primeiro # DÍGITO diferente de zero, # DÍGITOS [Ø] é feito igual a "Ø". No esquema abaixo, feito para palavras de 16 bits, o maior inteiro é certamente menor que 1ØØØØØ.

```
#CONVERT:  #DÍGITOS [Ø]: = " ";
           #DIVISOR: = 1ØØØØ;
#LOOP:     #DÍGITO: = int (#AC / #DIVISOR);
           #COLUNA: = #COLUNA + 1;
           Se #DÍGITO ≠ Ø, fazer #DÍGITOS [Ø]: = "Ø";
           #LINHA [#COLUNA]: = #DÍGITOS [#DÍGITO];
           #AC: = rem (#AC / #DIVISOR);
           #DIVISOR: = #DIVISOR /1Ø;
           Se #DIVISOR ≥ 1, voltar para #LOOP;
           Se #DÍGITOS [Ø] = " ", fazer #LINHA [#COLUNA]: = "Ø";
```

As funções int e rem, utilizadas no esquema acima, indicam respectivamente a parte inteira do quociente, e o resto da divisão indicada como argumento.

O último comando cobre o caso de o número a ser convertido valer zero, evitando que a cadeia de caracteres resultante fique totalmente em branco.

(*c*) # READ

Esta é a rotina de leitura de dados. Inicialmente, transfere para # LINHA, a partir da posição 1, até a posição # COLUNA, uma cadeia de caracteres que devem ser interpretados como um número decimal. A seguir, executa uma conversão oposta à detalhada anteriormente em # CONVERT. O resultado desta conversão é produzido no acumulador # AC. Um possível esquema da lógica desta rotina é o seguinte:

```
# READ:      # COLUNA: = Ø;
# LOOP:      lê um caractere;
             Se caractere não for dígito, desvia para # BINÁRIO
             # COLUNA: = # COLUNA + 1;
             # LINHA [# COLUNA]: = dígito lido;
             volta para # LOOP;
# BINÁRIO:   # I: = Ø;
             # AC: = Ø;
# TESTE:     Se # COLUNA = # I, terminou a conversão: retorna;
             # COLUNA: = # COLUNA + 1;
             pesquisa # LINHA [# COLUNA] na tabela # DÍGITOS. Como todos os ca-
             racteres de # LINHA são dígitos, será certamente encontrado (na posição
             # DÍGITO);
             fazer # AC: = # AC * 1Ø + # DÍGITO;
             # I: = # I + 1;
             voltar para # TESTE;
```

(*d*) # STOP

Esta rotina basicamente interfaceia com o sistema operacional, acionando uma chamada de supervisor para parar o processamento e retornar o controle ao sistema. Cada sistema tem seu próprio modo de efetuar esta tarefa, devendo ser consultado o manual do sistema operacional hospedeiro para que possa ser implementado este comando. Na ausência de sistema operacional, uma simples instrução do tipo HALT ou equivalente implementa esta função.

Uma observação importante sobre o sistema que implementa o ambiente de execução: o exemplo que está sendo aqui desenvolvido é muito simples, uma vez que a linguagem-fonte não dispõe de recursos usualmente encontrados nas linguagens comercialmente disponíveis: subrotinas, funções, passagens de parâmetros, expressões aritméticas com funções de biblioteca (aritméticas, de ponto flutuante, trigonométricas, logaritmos, etc.).

Caso tais funções estivessem disponíveis ao programador, certamente o ambiente de execução deveria ser enriquecido com muitas rotinas, responsáveis pela execução de tais operações. O mesmo ocorre em relação às bibliotecas de subrotinas responsáveis pela leitura e impressão formatada de textos de entrada/saída, pela comunicação com o sistema operacional e pelo acionamento de pacotes especiais de rotinas utilitárias (pacotes gráficos, pacotes de rotinas estatísticas, pacotes de rotinas matriciais, etc.).

Em determinadas linguagens, a estrutura de blocos, a existência de variáveis locais e globais, o armazenamento dinâmico de dados na memória, o caráter interativo ou conversacional da linguagem e outras características peculiares exigem um suporte adequado do ambiente de execução. Entre as principais funções exigidas, destacam-se as de gerenciamento de memória: alocação de áreas, recuperação, compactação, etc. Para o presente exemplo, mais uma vez, tais rotinas são desnecessárias, já que todas as variáveis têm alocação estática, e portanto resolvida em tempo de compilação.

5.8 – ASPECTOS DE IMPLEMENTAÇÃO E INTEGRAÇÃO DO COMPILADOR

Um último detalhe importante, na construção de um compilador, do ponto de vista prático, refere-se à maneira através da qual as diversas partes do mesmo devem ser conjuntamente utilizadas para formar o compilador completo. Nesta seção, são discutidos alguns aspectos relevantes relativos à implementação do compilador e à integração de suas partes, e levantadas algumas considerações adicionais relativas ao compromisso entre a minimização dos autômatos e a complexidade associada das ações semânticas do compilador.

Três elementos que não foram detalhados até o momento, e que são essenciais para a realização do compilador, são: o programa principal, o procedimento de iniciação do compilador e o programa que implementa os autômatos.

Tratando-se de um compilador dirigido por sintaxe, optou-se por organizá-lo de tal modo que o programa principal se comporte como o analisador sintático do compilador, sendo os demais elementos, responsáveis pelo processo de compilação, ativados por este analisador sintático.

Por questões de ordem prática, convém que o procedimento de análise sintática seja recursivo, ou então, sendo interativo, simule recursividade, já que a compilação de linguagens livres de contexto não regulares assim o exige. Optou-se pela implementação recursiva por ser mais intuitiva, podendo ser alterada com certa facilidade para eliminar as recursões através da inclusão de pilhas explícitas, caso a linguagem de implementação do compilador não permita a criação de procedimentos recursivos.

Assim sendo, o *programa principal* se limitará a executar três procedimentos, em seqüência:

Programa principal: iniciação;
análise sintática (PROGRAMA);
finalização;

O parâmetro do procedimento análise sintática corresponde à tabela de transições da primeira submáquina a ser chamada.

Os procedimentos de *iniciação* e *finalização* deverão conter o código necessário para permitir, respectivamente, que a execução do compilador possa ter início (por exemplo, atribuindo valores iniciais às variáveis de cada analisador, abrindo arquivos, etc.), e que as atividades efetuadas pelas rotinas semânticas sejam complementadas de modo que, por exemplo, nenhum arquivo fique aberto, ou que nenhuma operação iniciada deixe de ser completada. No nível de detalhes relativamente superficial em que este exemplo está sendo apresentado, não será possível detalhar de modo suficientemente completo estas duas funções.

O procedimento de *análise sintática,* por outro lado, pode ser representado, agora com base nas tabelas projetadas anteriormente, e com o enfoque de implementação escolhido. Inicialmente, é necessário estabelecer as bases de dados em que este procedimento deve apoiar-se:

SUBMÁQUINA — variável que indica a submáquina correntemente em uso.
ESTADO — variável que indica o estado corrente da submáquina em uso.
PILHA SINTÁTICA — estrutura organizada em pilha, cujos elementos são pares (submáquina, estado) e registram os estados de retorno durante o reconhecimento sintático.

TOPO SINTÁTICA	– apontador para o elemento de PILHA SINTÁTICA mais recentemente colocado nesta estrutura (topo da pilha).
TRANSIÇÕES PROGRAMA	– tabela de transições da submáquina correspondente ao não-terminal programa.
TRANSIÇÕES EXPRESSÃO	– idem, expressão.
TRANSIÇÕES COMANDO	– idem, comando.
AÇÕES PROGRAMA	– tabela de ações semânticas correspondentes a TRANSIÇÕES PROGRAMA.
AÇÕES EXPRESSÃO	– idem, TRANSIÇÕES EXPRESSÃO.
AÇÕES COMANDO	– idem, TRANSIÇÕES COMANDO.
ÁTOMOS PROGRAMA	– vetor de átomos e submáquinas com que transita TRANSIÇÕES PROGRAMA.
ÁTOMOS EXPRESSÃO	– idem, TRANSIÇÕES EXPRESSÃO.
ÁTOMOS COMANDO	– idem, TRANSIÇÕES COMANDO.

As tabelas de transições e de ações semânticas são organizadas na forma de matrizes bidimensionais, por facilidade de construção. Suas dimensões devem ser as adequadas para implementar a tabelas projetadas anteriormente: o número de linhas é igual ao número de estados, e o de colunas, igual ao número de átomos e submáquinas utilizadas nas transições. Colunas adicionais são incluídas nas tabelas para finalidades complementares: na tabela de transições, uma coluna extra é utilizada para a indicação de quais são os estados finais, e na tabela de ações semânticas, uma coluna adicional é utilizada para conter as ações de saída da submáquina.

Os estados são numerados a partir de 1, e o estado 1 é sempre o estado inicial da submáquina.

A cada coluna da tabela de transições está associado um elemento do vetor de átomos/submáquinas correspondente, da seguinte forma: o primeiro elemento do vetor de átomos/submáquinas indica o número de elementos de que é formado. Associados a este elemento são armazenadas, na primeira coluna da tabela de transições, as indicações de estado final/não final (estado final = $\emptyset$, estado não final $\neq \emptyset$).

Os demais elementos do vetor contêm números associados aos diversos terminais utilizados (códigos com números positivos) e não-terminais, ou seja, submáquinas (códigos com números negativos).

No caso deste exemplo, as submáquinas serão codificadas como:

– 1 = PROGRAMA
– 2 = EXPRESSÃO
– 3 = COMANDO

Todas as outras colunas da tabela apresentam, em cada linha, um conteúdo que representa a transição programada para o estado correspondente, quando for encontrado o átomo associado àquela coluna do vetor de átomos/submáquinas.

Quanto ao conteúdo das células destas colunas, foi adotada uma codificação, cuja interpretação é a seguinte:

ZERO – indica que não há transição prevista nesta célula. Se o estado correspondente for estado final, indica que deve ser efetuado um retorno para a submáquina chamadora, ou então que terminou o processamento. Para estados não finais, isto corresponde a uma situação de erro de sintaxe.

POSITIVO – o número contido na célula é interpretado como o número do estado para onde a submáquina deve evoluir nesta transição.

Quanto às transições com átomo, o programa, após obter o átomo a ser utilizado na transição, deve pesquisá-lo no vetor de átomos da submáquina corrente, identificando, deste modo, a coluna da tabela de transição a ser utilizada.

Caso o átomo não seja encontrado neste vetor, há os seguintes casos a considerar:

(*a*) O estado é final
Neste caso, o átomo não é consumido, e deve ser feito um retorno à submáquina chamadora, para o estado registrado no topo da pilha sintática.

(*b*) O estado é não-final
Neste caso, houve uma detecção de erro de sintaxe. Por questão de simplicidade, o processamento pode ser encerrado com um mensagem de erro ou, então, caso haja interesse, pode ser ativada uma rotina de recuperação de erro. Neste exemplo, isto não será feito.

As transições com submáquinas poderiam ser implementadas através da inclusão de informações adicionais nas tabelas de transições. Entretanto, isto aumentaria ligeiramente as dimensões das tabelas e a complexidade de sua interpretação. Por esta razão optou-se por uma solução não tão elegante, porém de implementação mais simples: a utilização de ações adicionais (*"ações sintáticas"*) para promover a chamada e o retorno de submáquinas, complementando assim a lógica da análise sintática. Nos estados que precedem imediatamente transições com submáquina, programam-se, em adição às ações semânticas já estudadas, ações sintáticas encarregadas de:

– obter o próximo átomo, sem consumi-lo.
– decidir, caso seja necessário, qual submáquina S deve ser chamada, em função do átomo extraído.
– chamar recursivamente o analisador sintático: ANÁLISE SINTÁTICA (S).

Analogamente, ao ser atingida uma situação em que um retorno de submáquina se faça necessário, uma ação sintática, associada às ações semânticas de retorno correspondentes aos estados finais de cada submáquina, deve ser executada, com as funções seguintes:

– desempilhar o conteúdo do topo da pilha sintática.
– com as informações desempilhadas, atualizar as informações sobre a submáquina corrente e sobre o estado corrente.
– fornecer ao analisador léxico o código correspondente à submáquina corrente, para que seja utilizado como átomo na próxima transição.

Resta comentar acerca do conteúdo das células das tabelas de ações semânticas.
Na primeira coluna destas tabelas, estão registradas as informações necessárias à execução das ações de retorno. Nas células correspondentes a estados finais, um único número identifica as ações (sintática e semântica) a serem executadas quando do retorno normal à submáquina chamadora. Para estados não finais, o número identifica a mensagem de erro a ser emitida devido à localização do erro de sintaxe que provocou a execução desta ação de retorno.

Em todas as demais células, os números correspondem a identificações das ações semânticas e/ou sintáticas envolvidas com a transição em questão.

Note-se que só devem existir células preenchidas nos casos em que houver, na tabela de transições associada, uma célula correspondente indicando uma transição.

Convenciona-se que um zero contido em uma célula da tabela de ações semânticas indica que nada há para ser executado.

Com estas convenções, é possível finalmente esboçar a lógica do reconhecedor sintático, de acordo com as condições impostas anteriormente:

```
Análise Sintática (S):
         Salvar inicialmente ESTADO e SUBMÁOUINA na PILHA SINTÁTICA:
         Fazer ESTADO: = 1; SUBMÁQUINA: = S;
LOOP:    Chamar o Analisador LÉXICO (TIPO, INFORMAÇÃO);
         Buscar TIPO no vetor de átomos da submáquina corrente obtendo a coluna da tabela
         de transições associada;
         (* se COLUNA = Ø, indica que não encontrou o átomo *)
         Obter CÉLULA: = tabela de transições corrente [ESTADO, COLUNA]
         Se COLUNA ≠ Ø
            então se CÉLULA ≠ Ø,
                  então – executar a rotina indicada em
                          tabela de ações semânticas [ESTADO, COLUNA];
                        – fazer ESTADO: = CÉLULA;
                        – executar a ação sintática associada à transição realizada, se existir
                  caso contrário,  Se tabela de ações semânticas [ESTADO, Ø] indicar estado
                                      final, retornar (*à submáquina chamadora*)
                                   caso contrário, emitir mensagem de erro e terminar o proces-
                                      samento
            caso contrário (*COLUNA = Ø*)
             – não consumir o átomo;
             – se o estado for final, retornar à submáquina chamadora.
               caso contrário, emitir mensagem de erro e terminar o processamento.
```

As rotinas semânticas, já esquematizadas anteriormente, podem ser organizadas fisicamente de modo tal que seja facilitada a sua escolha. Para isto, um dos possíveis arranjos corresponde a um encadeamento de comandos de decisão múltipla, conforme o esboço a seguir:

```
Conforme a submáquina corrente:
– 1: (*Programa*)
      conforme o número da rotina, extraído da tabela de ações semânticas:
         Ø: nada faz;
         1: (*ações semântica e sintática A correspondentes a Programa*)
            .
            .
            .
      outros: erro;
```

– 2: (* Expressão *)
conforme o número da rotina, extraído da tabela de ações semânticas:
∅: nada faz;
1: (* ações semântica e sintática A correspondente a Expressão *)
.
.
.
outros: erro;
– 3: (* Comando *)
conforme o número da rotina, extraído da tabela de ações semânticas:
∅: nada faz
1: (* ações semântica e sintática A correspondente a Comando *)
.
.
.
outros: erro;
outros: erro;

São indicadas a seguir as transições em que são necessárias ações sintáticas de chamada e de retorno de submáquina:

(*a*) *Programa:*
– ao ser chamada pela primeira vez (na iniciação): CHAMAR C.
– na transição B → A com o átomo ;: CHAMAR C.
– não há transições de retorno nesta submáquina principal.

(*b*) *Comando:*
– na transição J → K com átomo := : CHAMAR E.
– na transição A → F com átomo PRINT: CHAMAR E se o próximo átomo não for vírgula.
– na transição E → F com átomo ,: CHAMAR E.
– transição A → M com átomo IF: CHAMAR E.
– nas transições O → N com >, = ou <: CHAMAR E.
– na transição P → Q com átomo THEN: CHAMAR C.
– na transição S → R com átomo ELSE: CHAMAR C.
– nos estados A, B, C, D, E, F, G, se não for possível transitar com nenhum átomo: RETORNAR com o código de comando (– 3).

(*c*) *Expressão:*
– na transição A → B com ∢: CHAMAR E.
– no estado D, se não for possível nenhuma transição com átomo: RETORNAR com o código de expressão (– 2).

Para concluir, resta fazer uma consideração acerca do compromisso entre minimizar os autômatos e simplificar as rotinas semânticas.

No exemplo desenvolvido neste capítulo, optou-se por efetuar a minimização do autômato sem levar em consideração as rotinas semânticas que o autômato ativa durante o reconhecimento sintático. Em outras palavras, a minimização foi efetuada com base em considerações puramente sintáticas, envolvendo especialmente a eliminação de transições em vazio e a aglutinação de estados sintaticamente equivalentes.

Um estudo um pouco mais criterioso pode levar à conclusão que, se dois estados forem sintaticamente equivalentes, porém envolvendo transições correspondentes, às quais são associadas ações semânticas diferentes, tais estados não devem ser aglutinados, sob pena de aumentar a complexidade das rotinas semânticas associadas.

O mesmo raciocínio pode ser feito em relação a transições em vazio. Caso esteja associada uma ação semântica a alguma transição em vazio do autômato original, pode-se optar pela preservação desta transição em vazio.

Para que seja possível estudar estes aspectos, torna-se importante projetar as ações semânticas associadas às diversas transições do autômato original, efetuando-se as minimizações subseqüentemente, com base nas ações semânticas projetadas.

Para que sejam fixadas adequadamente as técnicas estudadas neste texto, é muito importante que um projeto completo seja elaborado, envolvendo inclusive a implementação em uma máquina real. Com este complemento prático, é possível ao leitor enfrentar projetos de porte maior sem nenhum receio e sem a mística que usualmente envolve os assuntos ligados a esta matéria.

5.9 – LEITURAS COMPLEMENTARES

Apesar de, na maioria dos livros-texto clássicos, serem apresentados exemplos da construção de partes de um compilador, alguns se destacam por oferecerem um detalhamento mais completo, servindo como guia de implementação para compiladores semelhantes. Entre estes, podem ser mencionados: Abramson (1973), Kowaltowski (1983), Payne (1982), Pemberton (1982), Setzer (1983) e Tremblay (1982). Nesta bibliografia, são encontrados estudos de casos, com aplicações de diferentes técnicas.

Como texto didático, cumpre destacar Halstead (1974), que é um excelente roteiro para um laboratório de construção de compiladores. Adicionalmente, Lewi (1979-1982) é outra ótima publicação deste gênero, voltado aos aspectos da construção de reconhecedores e do estudo de notações e representações formais.

5.10 – EXERCÍCIOS DE PROJETO

1 – Implemente um analisador léxico para alguma linguagem de seu interesse.

2 – Construa uma rotina eficiente de identificação de palavras reservadas, com a ajuda de livros ou artigos sobre estruturas de dados aplicados à compilação.

3 – Construa uma rotina eficiente de manipulação de tabelas de símbolos para linguagens que permitem símbolos de comprimento qualquer.

4 – Implemente uma rotina de ordenação alfabética para tabelas de símbolos. Não se esqueça de levar em consideração o caso de linguagens com estrutura de blocos, em que o mesmo símbolo pode estar repetido na tabela, referindo-se a diferentes objetos do programa.

5 – Implemente um analisador léxico acoplado a rotinas de listagem do texto-fonte, capaz de reconhecer comandos de controle da listagem ao longo do texto-fonte e executá-los.

6 – Incorpore ao seu analisador léxico uma rotina para a geração de tabelas de referências cruzadas. Inclua comandos de controle para ativá-la ou desativá-la.

7 – Incorpore ao programa construído anteriormente uma rotina para a impressão da tabela de símbolos em ordem alfabética, por bloco (no caso de linguagem estruturada).

8 – Escolha um subconjunto de alguma linguagem de seu interesse, e detalhe sua gramática, testando cuidadosamente sua consistência.

9 – Implemente um analisador léxico específico para esta linguagem, através do uso da técnica ascendente (LR(∅)).

10 – Levante os não-terminais essenciais da gramática do exercício 8.

11 – Projete um reconhecedor sintático para a linguagem do exercício 8 através do uso da técnica das submáquinas. Minimize-as.

12 – Alternativamente ao exercício 11, construa um reconhecedor descendente para esta linguagem, implementando-o como um reconhecedor LL(1).

13 – Defina uma máquina onde deverá ser executado o programa-objeto gerado pelo compilador que você vai construir. Mapeie para a linguagem desta máquina cada construção da linguagem-fonte.

14 – Defina as rotinas semânticas e o ambiente de execução utilizado como suporte pelo compilador.

15 – Integre o compilador na forma de um programa escrito em linguagem de alto nível à sua escolha.

16 – Alternativamente ao exercício 15, codifique o compilador na própria linguagem que ele deverá compilar. Traduza manualmente o programa assim escrito para uma linguagem disponível e teste-o. Faça sempre as alterações no texto-fonte original e não na linguagem de implementação. Procure com este exercício aplicar a técnica de "bootstrapping", obtendo um compilador autocompilável.

17 – Instrumente o seu compilador para que seja fornecido, sempre que for solicitado, um "trace" para acompanhamento passo a passo das transições efetuadas e das rotinas semânticas executadas.

18 – Implemente as rotinas do ambiente de execução, e teste-as individualmente de modo satisfatório.

19 – Procure executar os programas-objeto, gerados automaticamente pelo compilador, juntamente com seu ambiente de execução. Construa para este teste programas que exercitem todas as funções importantes.

20 – Instrumente também o ambiente de execução e o programa-objeto, de modo que seja possível obter um "trace" de sua execução passo a passo.

21 – Incorpore ao compilador as variáveis de controle mencionadas em 5.3.

Bibliografia

A bibliografia a seguir consta apenas de livros, todos relacionados ao estudo da teoria e técnicas de compilação.

Destacam-se, entre os textos teóricos, ligados à linguagens formais e assuntos correlatos, Aho (1972), Backhouse (1979), Beckman (1980), Harrison (1978), Hopcroft (1979 a, 1979 b), Lewis (1976), Salomaa (1978), Wulf (1981).

Sobre a especificação da forma de linguagens de programação, são relevantes os trabalhos de McGettrick (1980) e Pagan (1981).

Um valioso material que cobre de modo abrangente todos os aspectos do estudo de compiladores pode ser encontrado em Aho (1972), Aho (1979), Aho (1986), Barrett (1979), Bauer (1976), Gries (1971), Hunter (1981), Lewis (1976), Llorca (1986), Pyster (1980), Tremblay (1982), Tremblay (1985), White (1984) e Wulf (1975).

Textos mais compactos sobre a construção de compiladores, voltados mais aos aspectos práticos ou à visão macroscópica do processo de compilação são representados por Calingaert (1979), Cunin (1980), Donovan (1972), Ullman (1976), Welsh (1980), Wirth (1976) e Zelkowitz (1979).

Textos mais específicos, que detalham técnicas particulares, enfoques dirigidos ou estudos de casos, podem ser encontrados em Abramson (1973), Barron (1981), Brown (1979), Hopgood (1969), Kowaltowski (1983), Lee (1967), Loeliger (1981), Payne (1982), Peck (1971), Pemberton (1982), Schreiner (1985) e Setzer (1983).

Textos ligados ao ensino e à geração automática de compiladores podem ser encontrados em Halstead (1974) e Lewi (1979, 1982). Todos estes textos são ricos em referências bibliográficas, que complementam o respectivo material e dão ao leitor uma vasta gama de escolha para estudo e pesquisa na área.

ABRAMSON, H. – *Theory and Applications of a Bottom-up Syntax-directed Translator.* Academic Press, 1973.

AHO, A. V. e ULLMAN, J. D. – *The Theory of Parsing, Translation and Compiling, vol. 1 e 2.* Prentice Hall, 1972.

AHO, A. V. e ULLMAN, J. D. – *Principles of Compiler Design.* Addison-Wesley, 1979.

AHO, A. V.; SEHTI, R. e ULLMAN, J. D. – *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

BACKHOUSE, R. C. – *Syntax of Programming Languages – Theory and Practice.* Prentice-Hall, 1979.

BARRET, W. A. e COUCH, J. D. – *Compiler Construction – Theory and Practice.* SRA-1979 (Revised Printing).

BARRON, D. W. (ed) – *Pascal – The Language and its Implementation.* John Wiley & Sons, 1981.

BAUER, F. L. e EICKEL, J. (editores) – *Compiler Construction – An advanced Course.* Springer-Verlag, 1976 (2nd Edition).

BECKMAN, F. S. – *Mathematical Foundations of Programming (Cap. 7 a 10).* Addison-Wesley, 1980.

BROWN, P. J. – *Writing Interactive Compilers and Interpreters.* John Wiley & Sons, 1979.

CALINGAERT, P. – *Assemblers, Compilers and Program Translation* – (Cap. 5 a 8) Computer Science Press, 1979.

CUNIN, P. Y.; GRIFFITHS, M. e VOIRON, J. – *Comprendre la Compilation.* Springer-Verlag, 1980.

DONOVAN, J. J. – *Systems Programming (Cap. 8).* McGraw-Hill, 1972.

GRIES, D. – *Compiler Construction for Digital Computers.* John Wiley and Sons, 1971.

HALSTEAD, M. H. – *A Laboratory Manual for Compiler and Operating System Implementation.* Elsevier North Holland, 1974.

HARRISON, M. A. – *Introduction to Formal Language Theory.* Addison-Wesley, 1978.

HOPCROFT, J. E. e ULLMAN, J. D. – *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979 (a).

HOPCROFT, J. E. e ULLMAN, J. D. – *Formal Languages and their Relation to Automata.* Addison-Wesley, 1979 (b).

HOPGOOD, F. R. A. – *Compiling Techniques.* MacDonald/Elsevier, 1969.

HUNTER, R. – *The Design and Construction of Compilers.* John Wiley & Sons, 1981.

KASTENS, U.; HUTT, B. e ZIMMERMANN, E. – *G A G: A Practical Compiler Generator – Lecture Notes in Computer Science nº 141.* Springer-Verlag, 1982.

KOWALTOWSKI, J. – *Implementação de Linguagens de Programação.* Editora Guanabara Dois S.A., 1983.

LEE, J. A. N. – *The Anatomy of a Compiler.* Van Nostrand Reinhold, 1967.

LEWI, J. *et al.* – *A Programming Methodology in Compiler Construction. Part 1: Concepts; Part 2: Implementation.* North Holland, 1979-1982.

LEWIS, P. M. II; ROSENKRANTZ, D. J. e STEARNS, R. E. – *Compiler Design Theory.* Addison-Wesley, 1976.

LLORCA, S. e PASCUAL, G. *Compiladores – Teoría y Construcción.* Paraninfo, 1986.

LOELIGER. R. G. – *Threaded Interpretive Languages.* Byte Publications, Inc. 1981.

McGETTRICK, A. D. – *The Definition of Programming Languages.* Cambridge University Press, 1980.

PAGAN, F. G. – *Formal Specification of Programming Languages.* Prentice-Hall, 1981.

PAYNE, W. e PAYNE, P. – *Implementing Basics – How Basics Work.* Prentice-Hall, 1982.

PECK, J. E. L. (editor) – *Algol 68 Implementation.* North Holland, 1971.

PEMBERTON, S. e DANIELS, M. – *Pascal Implementation: The P4 Compiler and Assembler/Interpreter.* Ellis Horwood Ltd. 1982.

PYSTER, A. B. – *Compiler Design and Construction.* Van Nostrand Reinhold, 1980.

RÉVÉSZ, G. E. – *Introduction to Formal Languages.* McGraw-Hill, 1983.

SALOMAA, A. – *Formal Languages.* Academic Press, 1973.

SCHREINER, A. T. e FRIEDMAN, H. G. Jr. – *Introduction to Compiler Construction With UNIX.* Prentice Hall, 1985.

SETZER, V. W. e MELO, I. S. H. de – *A Construção de um Compilador.* Editora Campus, 1983.

TREMBLAY, J. P. e SORENSON, P. G. – *An Implementation Guide to Compiler Writing.* McGraw-Hill, 1982.

THEMBLAY, J. P. e SORENSON, P. G. – *The Theory and Practice of Compiler Writing.* McGraw-Hill, 1985.

ULLMAN, J. D. – *Fundamental Concepts of Programming System (Caps. 8, 9).* Addison-Wesley, 1976.

WEINGARTEN, F. W. – *Translation of Computer Languages.* Molden-Day, 1973.

WELSH, J. e McKEAG, M. – *Structured System Programming (section 2).* Prentice-Hall, 1980.

WIRTH, N. – *Algorithms + Data Structures = Programs (Cap. 5).* Prentice-Hall, 1976.

WAITE, W. M. e GOOS, G. – *Compiler Construction.* Springer-Verlag, 1984.

WULF, W. A. *et al.* – *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

WULF, W. *et al.* – *The Design of an Optimizing Compiler.* North-Holland, 1975.

ZELKOWITZ, M. V.; SHAW, A. C. e GANNON, J. D. – *Principles of Software Engineering and Design (Cap. 5).* Prentice-Hall, 1979.

Índice Remissivo

Neste índice estão assinalados com asterisco (*) as páginas correspondentes ao aparecimento do assunto no exemplo de projeto de compilador.

Abstrato
  código de máquina, 142
Aceitação, 34, 84
Aceitador, 12
  semântico, 13
Ações
  semânticas, 130
    implementação, 133, 198*, 200*, 202*, 203*, 204*, 205*
    funções, 135
    projeto 186*, 187*, 188*. 189*, 190*, 192*
  sintáticas, 212*
Agregados
  homogêneos (matrizes), 137
  heterogêneos (estruturas), 139
Agrupamento, 49
Alfabeto, 27
  de entrada, 36, 37, 43
  de memória, 34
  de pilha, 37, 43
  de saída, 39
Algol 60, 48, 136, 145
Algol 68, 52, 136
Alinhavado (código), 142
Alocação de memória
  automática, 145
  dinâmica, 145
  estática, 145
Alternância, 47
Ambientes de execução, 10, 13, 132, 144
  projeto, 205*
Ambigüidade, 57, 74, 81
Analisador, análise – ver também *reconhecedor*
  ascendente, 57
    determinístico, 82
  de contexto, 127
  descendente, 57
    recursivo, 76
  léxico, 10, 15, 117, 121
    ativação, 127
    comando do modo de operação, 127
    eficiência, 123
    filtro, 10
    funções, 117
    implementação (esboço), 162*
    projeto, 158*
  sintático, 12, 15, 126
    funções, 126
    implementação, 128, 163*
    projeto, 163
  semântico, 12, 13, 15, 127, 130
Aninhamentos, 105
"Apply", 84
Árvore
  abstrata, 135
  como forma intermediária, 142
  de derivação, 57
  reduzida, 12

sintática, 12, 126
abstrata, 127
Ascendente – ver *reconhecedor; análise*
Átomos, 26, 36, 37, 117
extração e classificação, 118
políticas, 124
Atributos, 131
gramáticas de, 53
tabela de, 134
Autocompilável, 7
Auto-recursão central, 46, 90, 110
Auto-residente, 6
Autômatos
de pilha, 35, 37
determinísticos, 38
não-determinísticos, 38
notação alternativa, 41
obtenção, 104
finitos, 35-6
obtenção, 92

BASIC, 144
Básica (gramática), 39
"Backtracking", 74
Biblioteca, 137
funções de, 145
Blocagem, 10
BNF, 6, 48
estendido, 49
Booleano, 137
"Bootstrapping", 7, 9
"Bottom-up", 82

Cadeia, 26
comprimento de, 27
concatenação, 27-8
de caracteres, 137
elementar, 27
vazia, 47-8
Canônico
seqüências de produções, 57
Chamada de submáquina, 42, 46, 107
Chomsky
hierarquia de, 31
Classes
de átomos, 118
de gramáticas/linguagens – ver *hierarquia de Chomsky*
Cobol
notação, 50
Código
alinhavado, 142
de máquina, 142
intermediário, 15
geração de, 128
Coerção, 141
Comentários, 5
eliminação de, 118
Compactação (de memória), 145
Compilação
comandos de controle de, 5
condicional, 121
Compilador, 2
autocompilável, 7
auto-resistente, 6
cruzado, 6
depuradores, 16
de um só passo, 16
de vários passos, 17, 135
definição BNF da linguagem fonte, 153*
didáticos, 15, 152*
dirigidos por sintaxe, 19, 127
especificação
da linguagem-fonte, 153*
da linguagem de saída, 155*
instruções de máquina, 156*
pseudoinstruções, 156*
dos comandos da linguagem-fonte, 155*, 156*
dos controles de opções de compilação, 156*
dos erros de execução, 155*
geral, 151*
esqueleto, 62
estratégias, 6
estruturação
(esquema) de, 152*
lógica, 10
(exemplo completo) de construção, 151*
funções internas, 116
implementação, 151
integração, 208
organização física, 15
para máquinas limitadas, 16
para produção, 16
para tempo real, 16
requisitos, 16
de projeto, 151
Comunicação
com o sistema operacional, 145
entre ambientes de execução, 133
Concatenação, 47-9
de cadeias, 27
de linguagens, 29
fechamentos da – ver *fechamentos*
Configuração de reconhecedor, 34
inicial, 34
final, 34
Conjuntos
recursivamente enumeráveis, 35
regulares, 35
Consumo de átomos (movimento de), 79, 84, 93
Constantes (tabelas de), 118
Conversão
de notações gramaticais, 63

BNF e notação de Wirth, 63
diagramas de sintaxe e notação de Wirth, 69
expressões regulares estendidas e notação de Wirth, 68
notação do Cobol e notação de Wirth, 67
produções gramaticais e BNF, 63
entre formatos de reconhecedores, 70
diagramas de estados e tabelas de transições, 70
produções de autômatos e diagramas de estados, 71
numérica, 15, 118
Co-rotinas, 21, 122, 124
"Cross-compiler" – ver *cruzado* (compilador)
"Cross-reference", 119
Cruzado (compilador), 6
Cursor, 33

Dados – ver *estrutura de dados*
De-blocagem, 10
Declaração
contextual, 132
explícita, 141
implícita, 132
Definição formal – ver também *metalinguagem; notações*
manipulação, 62
Delimitadores (eliminação), 118
Derivação, 27, 30
direta, 30
mais à direita, 82
Descendente – ver *reconhecedor; análise*
Descritores, 139
Determinístico (reconhecedor), 75
ascendente, 87
descendente, 75
Diagramas
de estados, 53
de sintaxe, 51
ferroviários, 51
T, 8
Dinâmicos (campos: alocação), 140

Eficiência, 23
da análise léxica, 123
Empilhamento, 83
Enumeração, 26
Ênupla, 142
Erros
Correção, 126
Detecção, 4, 12, 84, 125, 126
Recuperação, 12, 119, 126
Escopo, 131, 135
Essencial (não-terminal), 90
Estados, 36, 37, 42
atribuição de 91, 166*, 167*
corrente, 34
de entrada, 12
de saída, 12
diagramas de, 55
finais (de aceitação), 34, 36, 37, 43, 53, 93, 105
iniciais, 36, 37, 53, 83, 93, 105
indistinguíveis, 101
mudanças de, 34
Estruturas, 140
de dados, 136, 196*, 197*, 198*
Exaustivo (método), 74
Execução de linguagem de alto nível, 14
Expressão
regular, 47
estendida, 51

Fase (de compilação), 18
Fatoração
à esquerda, 77
Fechamentos (da concatenação)
recursivo e transitivo, 28, 47
transitivo, 28, 48
Ferroviária (notação), 51
Fila, 134
Filtro, 3
léxico, 10
Finalização (movimento), 79
Finito (autômato), 35
Formalização de linguagens, 5
FORTRAN, 124, 125, 141, 144
análise léxica, 124
Fragmentação, 145
Função de transição, 36-7

"Garbage collection", 145
Geração
automática
de analisadores léxicos, 125
de reconhecedores, 6
de código, 15, 133, 141
canônico, 13
dispositivos de, 5, 50
Gerenciamento de memória, 132, 144
Grafo, 51
de fluxo do programa, 143
Gramática, 5, 26, 29
básica, 39
de atributos, 53
de dois níveis, 52
de transdução, 38
irrestritas, 31
lineares (à direita/esquerda), 32, 47
livres de contexto, 32
regulares, 47
sensíveis ao contexto, 31
tipo (Chomsky)
0, 31

1, 31, 53
2, 32, 49, 50
3, 33, 49, 51
W, 52-3

"Hardware", 2
"Heap", 132, 145
Hierarquia de Chomsky, 31

Identificador, 118
coerência de uso, 131
escopo, 131
Inacessíveis
estados – eliminação de, 99
regiões de código – eliminação de, 143
Indentação, 121
Indexação, 139
Indistinguíveis (estados), 101
Instruções (otimização da escolha), 144
Instrumentação do programa, 146
Intermediário
código, 15
forma, 17, 141
linguagem, 13, 141
Interpretador, 2, 14, 142

Léxico – ver *analisador léxico*
LIFO, 35
Linearizam as matrizes, 139
Linguagem – ver também *metalinguagem*, 5, 26, 30
básica, 40
de alto/baixo nível, 2, 22
de desenvolvimento, 8
de saída, 40
execução, 14
fonte, 2
intermediária, 13, 141
livre de contexto, 38, 104
determinística, 38
modelos de, 27
natural, 5
objeto, 2
regular, 36, 47, 92
tipo (Chomsky)
0, 35
1, 35
2, 35, 50, 54, 56
3, 35, 47, 54
Lista ligada, 135
Listagem, 120
controle de, 121
Livre de contexto (propriedades das gramáticas), 56
LL(1), 81, 129
LL(k)
gramática, 75, 83
produção, 75
"Look-ahead", 84

LR(1), 85, 129
LR(k), 35, 83
gramática, 82
item, 86
obtenção de, 87

Macro, 120
Manipulador de caracteres, 10
Mapeamento de gramáticas em reconhecedores, 62, 88
Máquina
abstrata, 142
de estados, 33, 53
de Turing, 35, 38
hospedeira, 6
Marcador de pilha vazia, 37
Matriz, 137
multidimensional, 138
Memória
auxiliar, 34
gerenciamento, 132
Metalinguagem – ver também *definição formal; notações*
Minimização
de estados, 101
de submáquinas, 108
Modelos de linguagens, 27
Montador, 2, 3
Movimento do reconhecedor, 34, 79
Mudança de estado, 34
Multidimensional (matriz), 138
Multipasso (compilador), 17

Não-determinístico
reconhecedor, 73
eliminação de transições, 97
Não-terminal, 48, 49
essencial, 110
Notações
para a (definição de linguagens), 47
ferroviária, 51
Núcleo
de compilador dirigido por sintaxe, 127
de analisador sintático, 127

Objetos
primitivos, 137
representação de, 136
Opcional (construção), 49
Organização de um passo de compilação, 18
có-rotinas, 21
processos paralelos, 21
programa principal:
analisador léxico, 19
analisador sintático, 18
gerador de código, 19
Origem virtual, 139

Otimização, 141-2
de autômatos, 179
de código-objeto, 13, 133
de reconhecedores, 62
dependente de máquina, 13, 144
global, 143
independente de máquina, 12, 142
local, 143
Otimizador, 12
"Overlay" – ver *fase de compilação*
"P-code", 142
Pacote, 137
Palavra
chave, 119
reservada, 119
Pascal, 51, 142
Passo
de compilação, 16, 129
único, 16, 141
Pilha, 141
alteração da, 42
conteúdo antigo, 42
explícita, 79
símbolo inicial, 43
uso da, 45
tabela de símbolos estruturada em, 134
PL/I, 50
Pré-processador, 2, 4
Polonesa (notação), 141
"Pretty-printing", 121
Produções, 29, 55
do reconhecedor, 37
gramaticais, 52
marcadas, 86
seqüências canônicas de, 57

Quádruplas, 142

Real, 137
Reanálise, 125
Reconhecedor – ver também *alfabeto; configuração; analisador*, 5, 6, 26, 33
básico, 40
construção, 62
configuração, 34
inicial, 34
final, 34
descendente recursivo, 76
determinístico, 34, 75, 87
não-determinístico, 34
sintático, 12
Recuperação de erros, 4, 119, 130
Recursão
à esquerda, 76, 81
eliminação de, 76
ciclo de, 76
indireta, 76
eliminação de, 76
Redução, 82, 84
Referências cruzadas (tabela), 120
Região de otimização local, 143
Registrador
otimização da 144
Registro
de ativação, 145
físico, 10
lógico, 10
Regra de aceitação, 26
Regulares
conjuntos, 35
expressões, 47
linguagens, 47
Representação de objetos, 137
Ressincronização, 130
Resolução prévia de subexpressões, 143
Retorno de submáquina, 42, 107

Semântica – ver também *ações; análise*, 130
ação, 130
aceitação, 13
estática, 53

Sentença, 26, 30
identificação de, 126
Sentencial (forma), 30
Sintática – ver também *analisador; árvore*
ação, 212
Símbolo, 26
de entrada, 36, 37, 42
inicial, 29
Sistema
de arquivos, 120
de programação, 2
de substituição, 29
operacional, 2, 13, 121
comunicação com, 145
Situação, 44
inicial, 44, 83
Subexpressões comuns (eliminação), 143
Submáquina, 42, 167*, 168*, 169*
chamada, 46
inicial, 43
Substituição
movimento de, 79
sistema de, 29
"Syntax-driven", 19

T (diagrama), 8
Tabela
de análise, 79
de atributos, 15, 135
de símbolos, 15, 118, 131, 134
de transições, 54
canônicas, 170*, 171*

Terminal, 47, 48, 49
Texto de entrada, 33
"Threaded code" – ver *código alinhavado*
Tipo
  0, 1, 2, 3 (da hierarquia de Chomsky), 35
  compatibilidade de, 132, 141
  de dados, 136
"Top-down", 75
"Trace", 146
Tradução, 132, 141
Transdução (gramáticas de), 39
Transdutor, 40
  finito, 40
  seqüencial (teorema), 41
Transição
  com consumo de átomo, 53, 105
  com não-terminal, 54
  com submáquina, 46
  do autômato, 44
  de chamada, 107
  de retorno, 42, 107
  em vazio, 53, 93, 94, 106
    eliminação, 95, 173 *
  função de, 36-7
  interna, 42, 105
  não-determinísticas (eliminação), 97, 172 *, 173 *, 174 *, 176 *, 177 *
  tabela de, 95
Tripla, 142
Turing (máquina de), 35, 38

Van Wijngaarden, 52
Vetor, 137
Virtual (origem), 139
Vocabulário, 29

W (gramáticas), 52
Wirth (notação de), 47, 49, 89, 164 *