CAPÍTULO 1

INTRODUÇÃO À COMPILAÇÃO

Os princípios e técnicas da construção de compiladores são tão penetrantes, que as idéias encontradas neste livro serão usadas muitas vezes na carreira de um cientista da computação. A construção de compiladores se estende através dos temas de linguagens de programação, arquitetura de máquina, teoria das linguagens, algoritmos e engenharia de *software*. Afortunadamente, umas poucas técnicas básicas de construção de compiladores podem ser usadas para construir tradutores para uma ampla variedade de linguagens e máquinas. Neste capítulo, introduzimos o tema da compilação através da descrição dos componentes de um compilador, do ambiente no qual realiza seu trabalho e de algumas ferramentas de *software* que facilitam a sua construção.

1.1 COMPILADORES

Posto de forma simples, um compilador é um programa que lê um programa escrito numa linguagem — a linguagem *fonte* — e o traduz num programa equivalente numa outra linguagem — a linguagem *alvo* (ver a Fig. 1.1). Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa fonte.

À primeira vista, a variedade de compiladores pode parecer assustadora. Existem milhares de linguagens fonte, que vão das linguagens de programação tradicionais, tais como Fortran e Pascal, às linguagens especializadas que emergiram virtualmente em quase todas as áreas de aplicação de computadores. As linguagens alvo são igualmente variadas; uma linguagem alvo pode ser uma outra linguagem de programação ou a linguagem de máquina de qualquer coisa entre um microprocessador e um supercomputador. Os compiladores são algumas vezes classificados como de uma passagem, de passagens múltiplas, de carregar e executar, depuradores ou otimizantes, dependendo de como tenham sido construídos ou que função se suponha que devam realizar. A despeito dessa aparente complexidade, as tarefas básicas que qualquer compilador precisa realizar são essencialmente as mesmas. Pela compreensão delas, podemos construir compiladores para uma ampla variedade de linguagens fonte e máquinas alvo, usando as mesmas técnicas básicas.

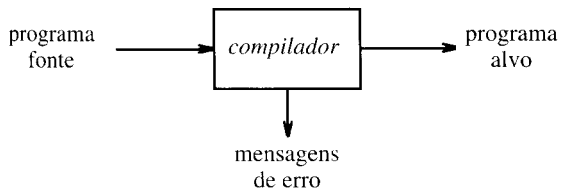

Fig. 1.1 Um compilador.

Nosso conhecimento sobre como organizar e escrever compiladores aumentou bastante desde que os primeiros começaram a surgir ao início dos anos 50. É difícil fornecer uma data exata para o primeiro compilador, porque inicialmente uma grande quantidade de experimentos e implementações foi realizada independentemente por diversos grupos. Muitos dos trabalhos iniciais em compilação lidavam com a tradução de fórmulas aritméticas em código de máquina.

Ao longo dos anos 50, os compiladores foram considerados programas notoriamente difíceis de se escrever. O primeiro compilador Fortran, por exemplo, consumiu 18 homens-ano para implementar (Backus *et al.* [1957]). Descobrimos, desde então, técnicas sistemáticas para o tratamento de muitas das mais importantes tarefas que ocorrem durante a compilação. Igualmente, foram desenvolvidas boas linguagens de implementação, ambientes de programação e ferramentas de *software*. Com esses avanços, até mesmo um compilador substancial pode ser escrito num projeto de estudantes, num curso de construção de compiladores com duração de um semestre.

O Modelo de Compilação de Análise e Síntese

Existem duas partes na compilação: a análise e a síntese. A parte de análise divide o programa fonte nas partes constituintes e cria uma representação intermediária do mesmo. A de síntese constrói o programa alvo desejado, a partir da representação intermediária. Das duas, a síntese requer as técnicas mais especializadas. Iremos considerar informalmente a análise na Secção 1.2 e esboçar, na Secção 1.3, a forma na qual o código alvo é sintetizado por um compilador padrão.

Durante a análise, as operações implicadas pelo programa fonte são determinadas e registradas numa estrutura hierárquica, chamada de árvore. Freqüentemente, é utilizado um tipo especial de árvore, chamado árvore sintática, na qual cada nó representa uma operação e o filho de um nó representa o argumento da operação. Por exemplo, a árvore sintática para um enunciado de atribuição é mostrada na Fig. 1.2.

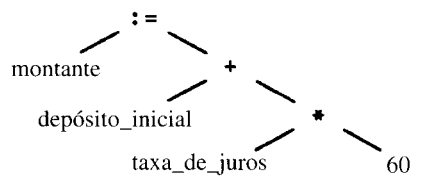

Fig. 1.2 Árvore sintática para `montante := depósito_inicial + taxa_de_juros * 60`.

Muitas ferramentas de *software* que manipulam programas fonte realizam primeiro algum tipo de análise. Alguns exemplos de tais ferramentas incluem:

1. *Editores de estruturas.* Um editor de estruturas toma como entrada um conjunto de comandos para construir um programa fonte. Realiza não só as funções de um editor de textos ordinário, tais como criação e a modificação dos mesmos, mas também analisa o conteúdo de um programa fonte, estabelecendo-lhe uma estrutura hierárquica apropriada. O editor de estruturas pode, também, realizar tarefas adicionais que são úteis ao preparo de programas. Por exemplo, pode verificar se a entrada está corretamente formada, fornecer palavras-chave automaticamente (por exemplo, quando o usuário digita `while`, o editor fornece o `do` correspondente e lembra-o que um enunciado condicional deve figurar entre ambos) e saltar de um `begin`, ou parênteses à esquerda, para o seu `end`, ou parênteses à direita, correspondente, respectivamente. Adicionalmente, a saída de um tal editor é freqüentemente similar àquela da fase de análise de um compilador.

2. *Pretty printers*.* Um *pretty printer* analisa um programa e o imprime numa forma em que a sua estrutura se torne claramente visível. Por exemplo, os comentários podem figurar numa fonte especial e os enunciados podem aparecer com uma indentação proporcional à profundidade do seu nível de aninhamento, na organização hierárquica dos comandos.

3. *Verificadores estáticos.* Um verificador estático lê um programa, analisa-o e tenta descobrir erros potenciais, sem executá-lo. A parte de análise é freqüentemente similar àquela encontrada nos compiladores otimizantes, do tipo discutido no Capítulo 10. Por exemplo, um verificador estático pode detectar quais as partes do programa fonte que não poderão nunca ser executadas, ou que uma certa variável poderia ser usada antes de ter sido definida. Adicionalmente, pode localizar erros lógicos, tais como usar uma variável real como um apontador ou empregar as técnicas de verificação de tipos discutidas no Capítulo 6.

4. *Interpretadores.* Em lugar de produzir um programa alvo como resultado da tradução, um interpretador realiza as operações especificadas pelo programa fonte. Para um enunciado de atribuição, por exemplo, poderia construir uma árvore, como a da Fig. 1.2, e subseqüentemente levar a termo as operações indicadas nos nós, à medida que a percorresse. Na raiz da árvore, descobriria a necessidade de realizar uma atribuição, chamaria uma rotina para avaliar a expressão à direita e armazenaria o valor resultante na localização associada ao identificador `montante`. No filho à esquerda da raiz, descobriria ter de computar a soma de duas expressões. Chamaria recursivamente a si mesma para computar o valor da expressão `taxa_de_juros * 60`. Adicionaria, então, aquele valor ao da variável `depósito_inicial`.

   Os interpretadores são freqüentemente usados para executar linguagens de comandos, dado que cada operador numa tal linguagem é usualmente uma invocação de uma rotina complexa, como um editor ou compilador. Similarmente, algumas linguagens de "nível muito alto", como APL, são normalmente interpretadas, pois existem muitos atributos de dados que não podem ser determinados em tempo de compilação.

Tradicionalmente, pensamos num compilador como um programa que transforma uma linguagem fonte, como Fortran, numa linguagem de montagem ou na linguagem de máquina de algum computador. No entanto, existem algumas áreas, visivelmente irrelacionadas, onde a tecnologia de compiladores é regularmente utilizada. A parte de análise, em cada um dos exemplos seguintes, é similar àquela de um compilador convencional.

1. *Formatadores de texto.* Um formatador de texto toma por entrada um fluxo de caracteres, a maior parte do mesmo como texto a ser composto tipograficamente, mas com alguma parte incluindo comandos, a fim de indicar parágrafos, figuras ou estruturas matemáticas, tais como subscritos e sobrescritos. Mencionamos algumas das análises realizadas por formatadores de texto na próxima seção.

2. *Compiladores de silício.* Um compilador de silício possui uma linguagem fonte que é similar ou idêntica à de uma linguagem de programação convencional. Entretanto, as variáveis da mesma não representam localizações de memória, mas sinais lógicos (0 ou 1) ou grupos de sinais de um circuito de chaveamento. A saída é um projeto de circuito, numa linguagem apropriada. Ver Johnson [1983], Ullman [1984] ou Trickey [1985] para uma discussão da compilação de silício.

3. *Interpretadores de queries**.* Um interpretador de *queries* traduz um predicado, contendo operadores booleanos ou relacionais, em comandos, para percorrer um banco de dados, de forma a satisfazer ao predicado.

O Contexto de um Compilador

Adicionalmente ao compilador, vários outros programas podem ser necessários para se criar um programa alvo executável. Um programa fonte pode ser dividido em módulos armazenados em arquivos separados. A tarefa de coletar esses módulos é, algumas vezes, confiada a um programa distinto, chamado de pré-processador. O pré-processador pode, também, expandir formas curtas, chamadas de macros, em enunciados da linguagem fonte.

A Fig. 1.3 mostra uma "compilação" típica. O programa alvo criado pelo compilador pode exigir processamento posterior antes que possa ser executado. O compilador da Fig. 1.3 cria um código de montagem que é traduzido no de máquina por um montador e, então, ligado a algumas rotinas de biblioteca, formando o código que é efetivamente executado em máquina.

Iremos considerar os componentes de um compilador nas próximas duas seções; os programas restantes na Fig. 1.3 são discutidos na Seção 1.4.

1.2 ANÁLISE DO PROGRAMA FONTE

Nesta seção, introduzimos a análise e ilustramos seu uso em algumas linguagens de formatação de texto. O assunto é tratado em mais detalhes nos Capítulos 2-4 e 6. Na compilação, a análise consiste em três fases:

1. *Análise linear,* na qual um fluxo de caracteres constituindo um programa é lido da esquerda para a direita e agrupado em *tokens,* que são seqüências de caracteres tendo um significado coletivo.

2. *Análise hierárquica,* na qual os caracteres ou *tokens* são agrupados hierarquicamente em coleções aninhadas com significado coletivo.

3. *Análise semântica,* na qual certas verificações são realizadas a fim de se assegurar que os componentes de um programa se combinam de forma significativa.

*N. do T. Manteremos o original em inglês por absoluta falta de correspondência na língua portuguesa de um termo que sequer se aproxime da idéia expressa na linguagem original.

*N. do T. A questão aqui é que não há termo, em português, que satisfaça o conceito de *query*. Optamos, então, por não traduzi-lo.

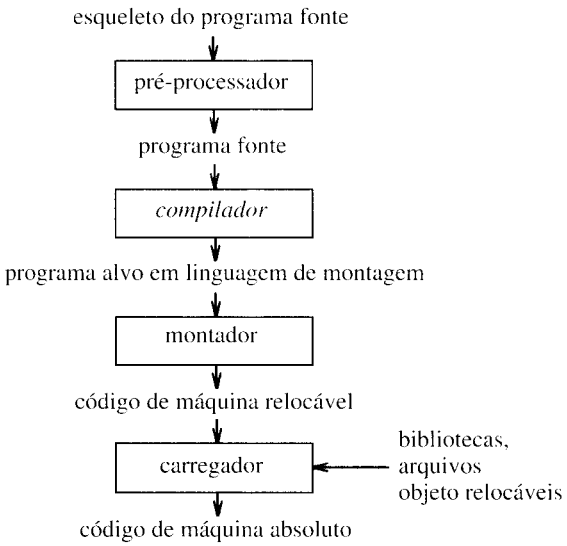

Fig. 1.3 Um sistema de processamento de linguagem.

Análise Léxica

Num compilador, a análise linear é chamada de *análise léxica* ou *esquadrinhamento (scanning)*. Por exemplo, na análise léxica, os caracteres no enunciado de atribuição

```
montante := depósito_inicial + taxa_de_juros * 60
```

poderiam ser agrupados nos seguintes *tokens:*

1. O identificador `montante`.
2. O símbolo de atribuição `:=`
3. O identificador `depósito_inicial`.
4. O sinal de adição.
5. O identificador `taxa_de_juros`.
6. O sinal de multiplicação.
7. O número `60`.

Os espaços que separam os caracteres desses *tokens* seriam normalmente eliminados durante a análise léxica.

Análise Sintática

A análise hierárquica é chamada de *análise gramatical* ou *análise sintática*. Envolve o agrupamento dos *tokens* do programa fonte em frases gramaticais, que são usadas pelo compilador, a fim de sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical, tal como a mostrada na Fig. 1.4.

Na expressão `depósito_inicial + taxa_de_juros * 60`, a frase `taxa_de_juros * 60` é uma unidade lógica, porque as convenções usuais das expressões aritméticas nos dizem que a multiplicação é realizada antes da adição. Como a expressão `depósito_inicial + taxa_de_juros` é seguida por um *, não é agrupada numa única frase, na Fig. 1.4.

A estrutura hierárquica de um programa é usualmente expressa por regras recursivas. Por exemplo, poderíamos ter as seguintes regras como parte da definição de expressões:

1. Qualquer *identificador* é uma expressão.
2. Qualquer *número* é uma expressão.
3. Se $expressão_1$ e $expressão_2$ são expressões, então também o são

$$expressão_1 + expressão_2$$
$$expressão_1 * expressão_2$$
$$(expressão_1)$$

As regras (1) e (2) são as regras base (não recursivas), enquanto que (3) define expressões em termos dos operadores aplicados às demais expressões. Então, pela regra (1) `depósito_inicial` e `taxa_de_juros` são expressões. Pela regra (2), `60` é uma expressão, enquanto que, pela regra (3), podemos primeiro inferir que `taxa_de_juros * 60` é uma expressão e, finalmente, que `depósito_inicial + taxa_de_juros * 60` também o é.

Similarmente, muitas linguagens definem recursivamente enunciados tais como:

1. Se $identificador_1$ é um identificador e $expressão_2$ uma expressão, então

   $$identificador_1 := expressão_2$$

   é um enunciado.

2. Se $expressão_1$ é uma expressão e $comando_2$ é um enunciado, então

   while ($expressão_1$) **do** $comando_2$
   if ($expressão_1$) **then** $comando_2$

   são enunciados.

A divisão entre a análise léxica e a sintática é um tanto arbitrária. Usualmente, escolhemos uma que simplifique a tarefa global de

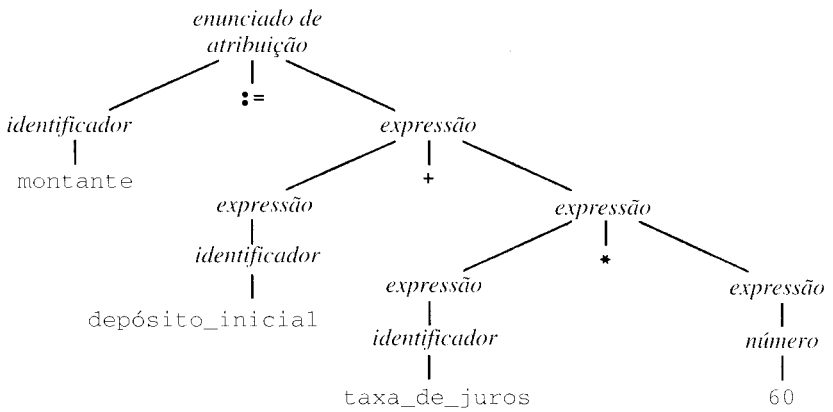

Fig. 1.4 Árvore gramatical para `montante := depósito_inicial + taxa_de_juros * 60`.

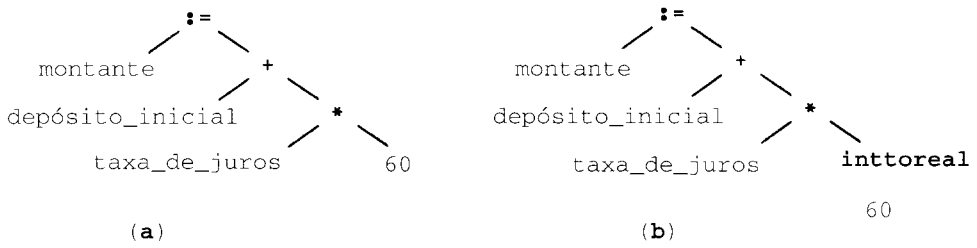

Fig. 1.5 A análise semântica insere uma conversão de inteiro para real.

análise. Um fator determinante na divisão é o de uma construção da linguagem fonte ser inerentemente recursiva ou não. As construções léxicas não requerem recursão, enquanto que as sintáticas freqüentemente a exigem. As gramáticas livres de contexto são uma formalização das regras recursivas que podem ser usadas para guiar a análise sintática. São introduzidas no Capítulo 2 e extensivamente estudadas no Capítulo 4.

Por exemplo, a recursão não é requerida para reconhecer identificadores, que são tipicamente cadeias de letras e dígitos, começando por uma letra. Reconheceríamos normalmente os identificadores por um simples esquadrinhamento do fluxo de entrada, aguardando até que um caractere, que não uma letra ou um dígito, fosse encontrado, e agrupando, num *token* tipo identificador, todas as letras e dígitos coletados até aquele ponto. Os caracteres, dessa forma agrupados, seriam registrados numa tabela, chamada tabela de símbolos, e removidos da entrada, de tal forma que o processamento do próximo *token* pudesse se iniciar.

Por outro lado, esse tipo de esquadrinhamento linear não é poderoso o suficiente para analisar expressões ou enunciados. Por exemplo, não podemos fazer corresponder apropriadamente os parênteses nas expressões, ou o `begin` e o `end` nos enunciados, sem criar algum tipo de estrutura hierárquica ou aninhamento na entrada.

A árvore gramatical da Fig. 1.4 descreve a estrutura sintática da entrada. Uma representação interna mais comum dessa estrutura sintática é dada pela árvore sintática na Fig. 1.5(a). Uma árvore sintática é uma representação condensada da árvore gramatical, na qual os operadores figuram como nós interiores e os operandos de um operador são os filhos do nó daquele operador. A construção de árvores, tais como aquela da Fig. 1.5(a), é discutida na Seção 5.2. Iremos examinar no Capítulo 2, e em mais detalhes no Capítulo 5, o tema da *tradução dirigida pela sintaxe,* na qual o compilador usa a estrutura hierárquica da entrada a fim de auxiliar a geração da saída.

Análise Semântica

A fase de análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subseqüente de geração de código. Utiliza a estrutura hierárquica determinada pela fase de análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados.

Um importante componente da análise semântica é a verificação de tipos. Nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, muitas definições nas linguagens de programação requerem que o compilador relate um erro a cada vez que um número real seja usado para indexar um *array.* No entanto, a especificação da linguagem pode permitir algumas coerções de operandos, como, por exemplo, quando um operando aritmético binário é aplicado a um inteiro e a um real. Nesse caso, o compilador pode precisar converter o inteiro para real. A verificação de tipos e a análise semântica são discutidas no Capítulo 6.

Exemplo 1.1. Dentro da máquina, um padrão de *bits* representando um inteiro é geralmente diferente do padrão de *bits* para um real, mesmo que o número inteiro e o número real tenham o mesmo valor. Suponha, por exemplo, que todos os identificadores na Fig. 1.5 tenham sido declarados como reais e assuma que 60, por si só, seja um inteiro. A verificação de tipos da Fig. 1.5(a) revela que o `*` está aplicado a um real, `taxa_ de_ juros`, e a um inteiro, 60. O enfoque geral é o de converter o inteiro em real. Isso foi conseguido na Fig. 1.5(b) pela criação de um nó extra para o operador **inttoreal**, que converte explicitamente um inteiro num real. Alternativamente, como o operando de **inttoreal** é uma constante, o compilador pode, em lugar, substituir a constante inteira por uma constante real equivalente. □

A Análise nos Formatadores de Texto

É útil considerar a entrada para um formatador de texto como especificando uma hierarquia de *compartimentos,* que são regiões retangulares a serem preenchidas por algum padrão de *bits,* representando pontos claros e escuros a serem impressos no dispositivo de saída.

Por exemplo, o sistema TEX (Knuth [1984a]) vê sua entrada dessa forma. Cada caractere, que não seja parte de um comando, representa um compartimento contendo o padrão de *bits* para aquele caractere, na fonte e tamanho apropriados. Os caracteres consecutivos não separados por "espaços em branco" (espaços ou caracteres de avanço de linha) são agrupados em palavras, consistindo em uma seqüência de compartimentos horizontalmente arranjados, mostrados esquematicamente na Fig. 1.6. O agrupamento de caracteres em palavras (ou comandos) é o aspecto linear ou léxico da análise do formatador de texto.

Os compartimentos, em TEX, podem ser construídos a partir de outros menores, através de combinações arbitrárias, horizontais e verticais. Por exemplo,

```
\hbox{ <lista de boxes> }
```

agrupa a lista de compartimentos pela justaposição dos mesmos horizontalmente, enquanto que operador `\vbox` pode agrupar uma lista de compartimentos por justaposição vertical. Dessa forma, se escrevermos em TEX

```
\hbox {\vbox{! 1} \vbox{@ 2}}
```

obteremos o arranjo de compartimentos mostrado na Fig. 1.7. A determinação do arranjo hierárquico de compartimentos estabelecido pela entrada é parte da análise sintática em TEX.

Como outro exemplo, o pré-processador EQN para a matemática (Kernighan e Cherry [1975]) ou o processador matemático em TEX constroem expressões matemáticas a partir de operadores como `sub` e

Fig. 1.6 Agrupamento de caracteres e palavras em *compartimentos.*

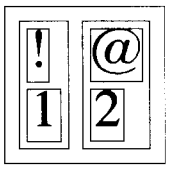

Fig. 1.7 Hierarquia de compartimentos em TEX.

sup, significando subscritos e sobrescritos. Se EQN encontra um texto de entrada da forma

BOX `sub` *box*

comprime o tamanho de *box* e o atrela a *BOX*, próximo ao canto inferior direito, como ilustrado na Fig. 1.8. O operador *sup*, similarmente, atrela *box* ao canto superior direito.

Esses operadores podem ser aplicados recursivamente, de tal forma que o texto EQN

```
a sub {i sup 2}
```

resulta em a_{i^2}.* O agrupamento dos operadores `sub` e `sup` em tokens é parte da análise léxica do texto EQN. No entanto, é necessária a sua estrutura sintática para determinar o tamanho e a localização de um compartimento.

1.3 AS FASES DE UM COMPILADOR

Conceitualmente, um compilador opera em *fases*, cada uma das quais transforma o programa fonte de uma representação para outra. Uma decomposição típica de um compilador é mostrada na Fig. 1.9. Na prática, algumas das fases podem ser agrupadas e a representação intermediária entre as mesmas não precisa ser explicitamente construída.

As três primeiras fases, formando o núcleo da parte de análise do compilador, foram introduzidas na última secção. Duas outras atividades, o gerenciamento da tabela de símbolos e a manipulação de erros, são mostradas interagindo com as seis fases de análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização e geração de código. Informalmente, também chamaremos de "fases" o gerenciador da tabela de símbolos e o manipulador de erros.

Gerenciamento da Tabela de Símbolos

Uma função essencial do compilador é registrar os identificadores usados no programa fonte e coletar as informações sobre os seus diversos atributos. Esses atributos podem providenciar informações sobre a memória reservada para o identificador, seu tipo, escopo (onde é válido no programa) e, no caso de nomes de procedimentos, coisas tais como o número e os tipos de seus argumentos, o método de transmissão de cada um (por exemplo, por referência) e o tipo retornado, se algum.

Uma *tabela de símbolos* é uma estrutura de dados contendo um registro para cada identificador, com os campos contendo os atributos do identificador. A estrutura de dados nos permite encontrar rapidamente cada registro e, igualmente, armazenar ou recuperar dados do mesmo. As tabelas de símbolos são discutidas nos Capítulos 2 e 7.

Quando, no programa fonte, o analisador léxico detecta um identificador, instala-o na tabela de símbolos. No entanto, os atributos

*N. do T. operador *sub* se aplica a toda a expressão entre chaves; o operador *sup* faz com que o 2 figure como o sobrescrito de *i* e não de *a*.

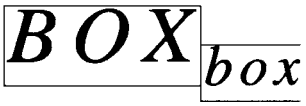

Fig. 1.8 Construindo uma estrutura de subscritos num texto matemático.

do identificador não podem ser normalmente determinados durante a análise léxica. Por exemplo, em Pascal, numa declaração como

```
var montante, depósito_inicial,
taxa_de_juros: real;
```

o tipo real ainda não será conhecido quando `montante`, `depósito_inicial` e `taxa_de_juros` forem enxergados pelo analisador léxico.

As fases remanescentes colocam informações sobre os identificadores na tabela de símbolos e em seguida as usam de várias maneiras. Por exemplo, ao realizar a análise semântica e a geração de código intermediário, precisamos conhecer de que tipos os identificadores são, a fim de verificar se o programa fonte os usa de forma válida e, por conseguinte, gerar as operações apropriadas sobre os mesmos. O gerador de código tipicamente instala e usa as informações detalhadas a respeito da memória atribuída aos identificadores.

Detecção de Erros e Geração de Relatórios

Cada fase pode encontrar erros. Entretanto, após encontrá-los, precisa lidar de alguma forma com os mesmos, de tal forma que a compilação possa continuar, permitindo que sejam detectados erros posteriores no programa fonte. Um compilador que pare ao encontrar o primeiro erro não é tão prestativo quanto poderia sê-lo.

As fases de análise sintática e semântica tratam usualmente de uma ampla fatia dos erros detectáveis pelo compilador. A fase de análise léxica pode detectá-los quando os caracteres remanescentes na entrada não formem qualquer *token* da linguagem. Os erros, onde o fluxo de *tokens* viole as regras estruturais (sintaxe) da linguagem, são determinados pela fase de análise sintática. Durante a análise semântica, o

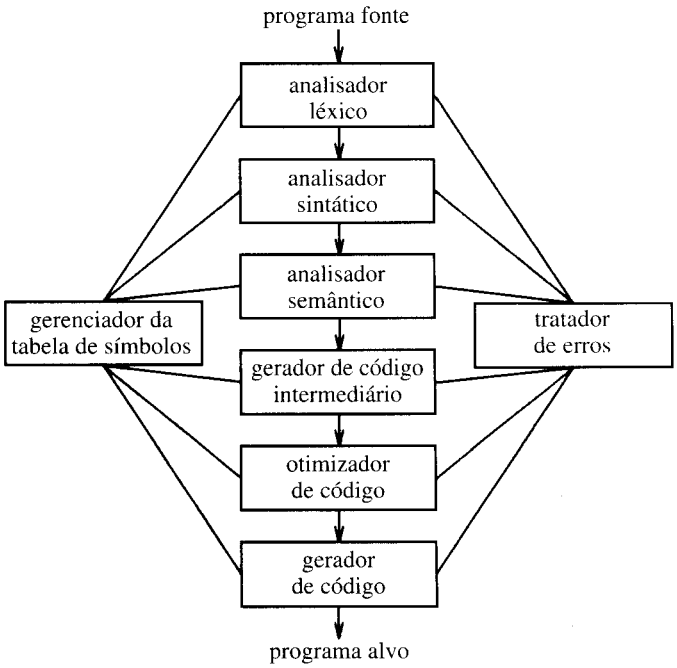

Fig. 1.9 Fases de um compilador.

compilador tenta detectar as construções que possuam a estrutura sintática correta, sem nenhuma preocupação com o significado da operação envolvida, como, por exemplo, ao tentarmos adicionar dois identificadores, um dos quais seja o nome de uma *array* e o outro o nome de um procedimento. Neste livro discutiremos o tratamento dos erros de fase ao examinarmos especificamente cada fase.

As Fases de Análise

À medida que a tradução progride, a representação interna do compilador para o programa fonte muda. Ilustramos essas representações considerando a tradução do enunciado

```
montante := depósito_inicial + taxa_de_juros
* 60                                          (1.1)
```

A Fig. 1.10 mostra a representação desse enunciado após cada fase.

A fase de análise léxica lê os caracteres de um programa fonte e os agrupa num fluxo de *tokens*, no qual cada *token* representa uma seqüência de caracteres logicamente coesiva, como, por exemplo, um identificador, uma palavra-chave (`if`, `while` etc.), um caractere de pontuação ou um operador composto por vários caracteres, como `:=`. A seqüência dos caracteres que formam um *token* é chamada o *lexema* para aquele *token*.

Certos *tokens* serão enriquecidos por um "valor léxico". Por exemplo, quando um identificador, como `taxa_de_juros`, é encontrado, o analisador léxico não somente gera um *token*, digamos **id**, mas, também, instala o lexema `taxa_de_juros` na tabela de símbolos, se já não estiver lá. O valor léxico associado a essa ocorrência de **id** aponta para a entrada de taxa de juros na tabela de símbolos.

Nesta secção, usaremos $\mathbf{id}_1$, $\mathbf{id}_2$, e $\mathbf{id}_3$ para `montante`, `depósito_inicial`, e `taxa_de_juros`, a fim de enfatizar que a

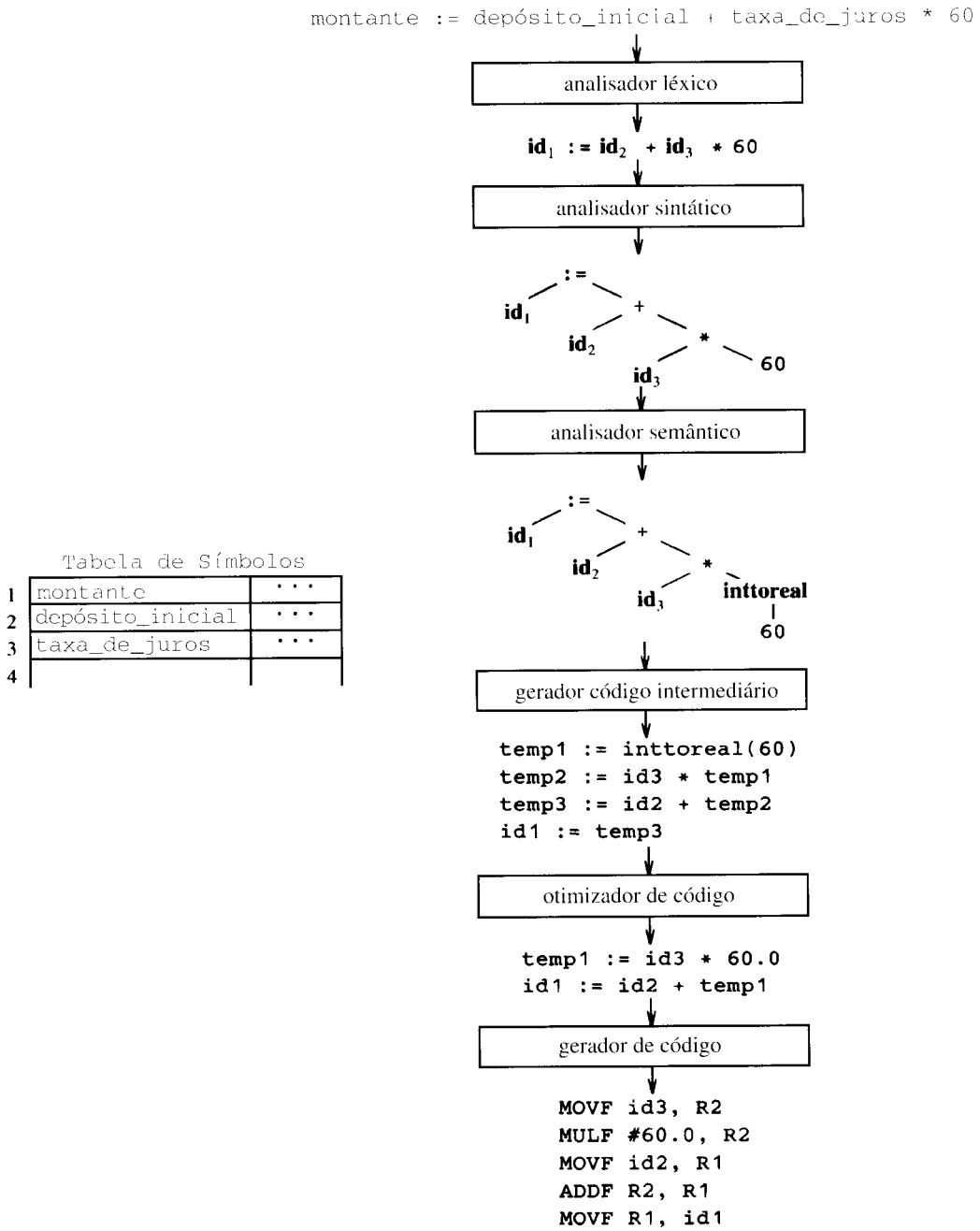

Fig. 1.10 Tradução de um enunciado.

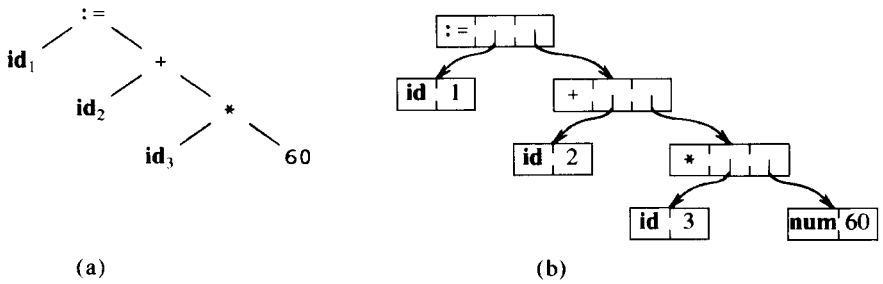

Fig. 1.11 A estrutura de dados em (b) para a árvore em (a).

representação interna de um identificador é diferente da seqüência de caracteres que formam o identificador. A representação de (1.1) após a análise léxica é, por conseguinte, sugerida por:

$$\mathbf{id}_1 := \mathbf{id}_2 * \mathbf{id}_3 * 60 \tag{1.2}$$

Deveríamos criar, também, *tokens* para o operador multicaractere := e para o número `60`, a fim de refletir suas representações internas, mas tal será postergado até o Capítulo 2. A análise léxica é coberta em detalhes no Capítulo 3.

A segunda e a terceira fase, de análise sintática e semântica, respectivamente, já foram também introduzidas na Seção 1.2. A análise sintática impõe uma estrutura hierárquica ao fluxo de *tokens*, a qual iremos retratar através de árvores sintáticas, como na Fig. 1.11(a). Uma estrutura de dados típica para a árvore é mostrada na Fig. 1.11(b), na qual um nó interior é um registro com um campo para o operador e dois campos contendo apontadores para os registros dos filhos à esquerda e à direita. Uma folha é um registro com dois ou mais campos, um para identificar o *token* que está à folha, e os outros para registrar informações sobre o *token*. Informações adicionais sobre as construções da linguagem podem ser mantidas através da adição de novos campos aos registros dos nós. Discutimos a análise sintática e a análise semântica nos Capítulos 4 e 6, respectivamente.

Geração de Código Intermediário

Após as análises sintática e semântica, alguns compiladores geram uma representação intermediária explícita do programa fonte. Podemos pensar dessa representação intermediária como um programa para uma máquina abstrata. Essa representação intermediária deveria possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo.

A representação intermediária pode ter uma variedade de formas. No Capítulo 8, consideramos uma forma intermediária chamada "código de três endereços", que é como uma linguagem de montagem para uma máquina, na qual cada localização de memória possa atuar como um registrador. O código de três endereços consiste em uma seqüência de instruções, cada uma delas possuindo no máximo três operandos. O programa fonte em (1.1) poderia ser expresso no código de três endereços como

```
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2                    (1.3)
id1 := temp3
```

Esta forma intermediária possui várias propriedades. Primeiro, cada instrução de três endereços possui, no máximo, um operador, além do de atribuição. Então, ao gerar essas instruções, o compilador precisa decidir sobre a ordem em que as mesmas devam ser realizadas; a multiplicação precede a adição no programa fonte de (1.1). Segundo, o compilador precisa gerar um nome temporário para receber o valor computado em cada instrução. Terceiro, algumas instruções de três endereços possuem menos do que três operandos, por exemplo, a primeira e a última instruções em (1.3).

No Capítulo 8, cobrimos as principais representações intermediárias usadas nos compiladores. Em geral, essas representações precisam fazer mais do que computar expressões; precisam também tratar construções do fluxo de controle e chamadas de procedimentos. Os Capítulos 5 e 8 apresentam algoritmos para a geração do código intermediário das construções típicas das linguagens de programação.

Otimização de Código

A fase de otimização tenta melhorar o código intermediário, de tal forma que venha resultar um código de máquina mais rápido em tempo de execução. Algumas otimizações são triviais. Por exemplo, um algoritmo natural gera o código intermediário (1.3) após a análise semântica, usando uma instrução para cada operador na representação em árvore, ainda que exista uma maneira melhor de se realizar a mesma computação, usando-se as duas instruções

```
temp1 := id3 * 60.0
id1 := id2 + temp1                      (1.4)
```

Não existe nada de errado com esse algoritmo simples, dado que o problema pode ser corrigido durante a fase de otimização de código. Ou seja, o compilador pode deduzir que a conversão de `60`, da representação de inteiro para a de real, pode ser realizada uma vez, e para todo o sempre, em tempo de compilação, e, por conseguinte, a operação `inttoreal` pode ser eliminada. Além do mais, `temp3` é usada para transmitir seu valor a `id1` uma única vez. Torna-se seguro, então, substituir `temp3` por `id1` e isso feito torna o último enunciado de (1.3) desnecessário, resultando no código de (1.4).

Existe uma grande variação na quantidade de otimizações de código que cada compilador executa. Naqueles que mais a realizam, chamados de "compiladores otimizantes", uma porção significativa de seus tempos é gasta nessa fase. Entretanto, existem otimizações simples que melhoram significativamente o tempo de execução do programa alvo, sem alongar o tempo de compilação. Muitas delas são discutidas no Capítulo 9, enquanto que o Capítulo 10 fornece a tecnologia usada pelos compiladores otimizantes mais poderosos.

Geração de Código

A fase final do compilador é a geração do código alvo, consistindo normalmente de código de máquina relocável ou código de montagem. As localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Então, as instruções intermediárias são, cada uma, traduzidas numa seqüência de instruções de máquina que realizam a mesma tarefa. Um aspecto crucial é a atribuição das variáveis aos registradores.

Por exemplo, usando-se os registradores 1 e 2, a tradução do código de (1.4) poderia se tornar

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1                    (1.5)
ADDF R2, R1
MOVF R1, id1
```

O primeiro e o segundo operandos de cada instrução especificam o emissor e o receptor, respectivamente. O F em cada instrução nos informa que as instruções lidam com números em ponto flutuante. Esse código copia o conteúdo do endereço[1] `id3` no registrador 2 e, então, multiplica-o pela constante 60.0 . O # significa que `60.0` deve ser tratado como uma constante. A terceira instrução copia o conteúdo de `id2` no registrador 1 e o adiciona ao valor previamente computado no registrador 2. Finalmente, o valor no registrador 1 é copiado para o endereço de `id1` , de tal forma que o código implementa a atribuição na Fig. 1.10. O Capítulo 9 cobre a geração de código.

1.4 OS PRIMOS DO COMPILADOR

Como vimos na Fig. 1.3, a entrada para o compilador pode ser produzida por um ou mais pré-processadores e pode ser necessário processamento posterior da saída do compilador, antes do código de máquina ser obtido. Nesta seção, discutimos o contexto no qual um compilador tipicamente opera.

Pré-processadores

Os pré-processadores produzem entrada para compiladores. Podem realizar as seguintes funções:

1. *Processamento de macros.* Um pré-processador pode permitir que um usuário defina macros que sejam abreviações para construções mais longas.

2. *Inclusão de arquivos.* Um pré-processador pode incluir arquivos no papel de cabeçalhos do texto do programa. Por exemplo, o pré-processador C faz com que o conteúdo do arquivo `<global.h>` substitua o enunciado `#include <global.h>` ao processar um arquivo contendo tal enunciado.

3. *Pré-processadores "racionais".* Tais pré-processadores expandem as linguagens mais antigas com facilidades modernas de controle de fluxo e estruturação de dados. Por exemplo, tal pré-processador poderia providenciar, ao usuário, macros embutidas para construções, tais como comandos *while* ou *if*, quando os mesmos não existissem na linguagem de programação em si.

4. *Extensores de linguagens.* Tentam conferir maior poder às linguagens, através de macros embutidas. Por exemplo, Equel (Stonebraker *et al.* [1976]) é uma linguagem de interrogação de banco de dados embutida em C. Os enunciados começando por `##` são considerados pelo pré-processador como comandos de acesso a banco de dados, irrelacionados com a linguagem C, e traduzidos em chamadas de procedimentos para rotinas que realizam tal acesso.

Os processadores de macros lidam com dois tipos de enunciados: definição e uso de macros. As definições são normalmente indicadas por algum caractere único ou palavra-chave, como `define` ou `macro`. Consistem em um nome para a macro sendo definida e em um *corpo*, formando a definição. Freqüentemente, os processadores de macros permitem *parâmetros formais* em suas definições, ou seja, símbolos a serem substituídos por valores (um "valor" é uma cadeia de caracteres, nesse contexto). O uso de uma macro consiste na designação de uma macro, através de seu nome, e no fornecimento dos *parâmetros atuais*, isto é, valores para seus parâmetros formais. O processador de macros substitui os parâmetros formais pelos atuais no corpo da macro; por conseguinte, o corpo substituído substitui a macro em si.

Exemplo 1.2. O sistema de composição tipográfica TEX, mencionado na Seção 1.2, contém uma facilidade generalizada para macros. As definições de macros tomam a forma

```
\define <nome da macro> <gabarito> {<corpo>}
```

Um nome de macro é qualquer cadeia de letras precedida por uma barra invertida. O gabarito é qualquer cadeia de caracteres, com as cadeias de forma `#1, #2, ..., #9` consideradas como parâmetros formais. Esses símbolos podem também figurar no corpo, qualquer número de vezes. Por exemplo, a macro seguinte define uma citação para o *Journal of the ACM.*

```
\define\JACM #1; #2; #3.
  { {\sl J. ACM} {\bf #1} : #2, pp. #3.}
```

O nome da macro é `\JACM` e o gabarito é `"#1;#2;#3."`; os pontos-e-vírgulas separam os parâmetros e o último é seguido por um ponto. Um uso dessa macro precisa tomar a forma do gabarito, exceto que cadeias arbitrárias devem substituir os parâmetros formais.[2] Podemos, então, escrever

```
\JACM 17;4;715-728.
```

e esperar ver

J. *ACM* **17**:4 , pp. 715-728

A parte do corpo `{\sl J. ACM}` chama por um "*J. ACM*" em itálico ("inclinado"). A expressão `{\bf #1}` informa que o primeiro parâmetro atual deve ficar em negrito; esse parâmetro está destinado a ser o número do volume.

TEX permite que qualquer pontuação ou cadeia de texto separe o volume, número e numeração de página na definição da macro `\JACM`. Poderíamos mesmo não ter usado pontuação alguma, caso em que TEX tomaria cada parâmetro atual como sendo constituído por um único caractere ou uma cadeia envolvida por { }. □

Montadores

Alguns compiladores produzem um código de montagem, como em (1.5), que é passado a um montador para processamento posterior. Outros compiladores realizam a tarefa do montador, produzindo um código de máquina relocável, que pode ser passado diretamente para um carregador/editor de ligações. Assumimos que o leitor tenha alguma familiaridade com o que uma linguagem de montagem se pareça e o que um montador faça. Aqui, iremos rever o relacionamento entre o código de montagem e o código de máquina.

O *código de montagem* é uma versão mnemônica do código de máquina, na qual são usados nomes em lugar do código binário para as

[1] Deixamos de lado o importante tema da reserva de memória para os identificadores no programa fonte. Como veremos no Capítulo 7, a organização de memória em tempo de execução depende da linguagem sendo compilada. As decisões sobre a reserva de memória ou são tomadas durante a geração do código intermediário ou durante a geração de código.

[2] Bem, cadeias quase arbitrárias, pois no uso da macro se dá um esquadrinhamento simples da esquerda para a direita e tão logo no texto seja encontrado um símbolo que se iguale ao que vem em seguida a um símbolo `#i` no gabarito, considera-se a cadeia precedente emparelhada com `#i`. Então, se tentássemos substituir `#1 por ab; cd` encontraríamos que somente `ab` ter-se-ia emparelhado com `#1`, tendo `cd` sido emparelhada com `#2`.

operações e fornecidos nomes aos endereços de memória. Uma seqüência típica de instruções de montagem seria

```
MOV a, R1
ADD #2, R1                    (1.6)
MOV R1, b
```

Este código copia o conteúdo do endereço `a` no registrador 1, adiciona a constante 2 ao mesmo, tratando o conteúdo do registrador 1 como um número em ponto fixo, e, finalmente, armazena o resultado na localização denominada `b`. Computa, então, `b := a + 2`.

É usual que as linguagens de montagem possuam facilidades de macros, que são similares àquelas dos pré-processadores de macros discutidas acima.

Montagem em Duas Passagens

A forma mais simples do montador realiza duas passagens sobre a sua entrada, onde uma *passagem* consiste na leitura do arquivo de entrada uma única vez. Na primeira, todos os identificadores que denotam localizações de memória são localizados e armazenados numa tabela de símbolos (separada do compilador). São associadas localizações de memória aos identificadores à medida que os mesmos sejam encontrados pela primeira vez, de tal forma que após ler (1.6), por exemplo, a tabela de símbolos poderia conter as entradas mostradas na Fig. 1.12. Nela, assumimos que seja reservada uma palavra, consistindo em quatro *bytes* para cada identificador e que os endereços sejam atribuídos começando-se pelo *byte* 0.

Na segunda passagem, o montador esquadrinha a entrada de novo. Desta vez, traduz tanto cada operação em seqüências de *bits*, representando àquela operação em linguagem de máquina, quanto cada identificador, representando uma localização no endereço atribuído ao mesmo na tabela de símbolos.

A saída da segunda passagem é usualmente um código de máquina *relocável*, significando que pode ser carregado começando em qualquer localização L na memória, isto é, se L for adicionado a todos os endereços no código, então todas as referências estarão corretas. A saída do montador precisa distinguir, então, aquelas partes das instruções que se refiram a endereços que possam ser relocados.

Exemplo 1.3. O que se segue é o código de uma máquina hipotética, no qual poderiam ser traduzidas as instruções de montagem (1.6).

```
0001 01 00 00000000 *
0011 01 10 00000010           (1.7)
0010 01 00 00000100 *
```

Examinamos aqui uma pequenina palavra de instrução, na qual os primeiros quatro *bits* são o código de instrução, com `0001`, `0010` e `0011` significando carregar *(load)*, armazenar *(store)* e adicionar *(add)*, respectivamente. Por carregar e armazenar significamos cópias a partir da memória para um registrador e vice-versa. Os dois *bits* seguintes designam um registrador, e `01` designa o de número 1, em cada uma das três instruções acima. Os dois *bits* após representam um "descritor", com `00` significando modo de endereçamento ordinário, e os últimos oito *bits* se referem a endereços de memória. O descritor `10` significa modo imediato, onde os últimos oito *bits* são tomados literalmente como um operando. Esse modo figura na segunda instrução de (1.7).

IDENTIFICADOR	ENDEREÇO
a	0
b	4

Fig. 1.12 Uma tabela de símbolos de um montador com identificadores de (1.6).

Observamos, também, em (1.7), um * associado à primeira e terceira instruções. Aquele asterisco representa o *bit de relocação* que está associado a cada operação no código relocável de máquina. Suponhamos que o espaço de endereçamento contendo os dados deva ser carregado iniciando-se pela localização L. A presença do * significa que L precisa ser adicionado ao endereço na instrução. Então, se L = `00001111`, isto é, 15, então `a` e `b` estariam nas localizações 15 e 19, respectivamente, e as instruções de (1.7) apareceriam como

```
0001 01 00 00001111
0011 01 10 00000010           (1.8)
0010 01 00 00010011
```

sob a forma de um código de máquina *absoluto* ou irrelocável. Note que, como não há * associado à segunda instrução em (1.7), L não foi adicionado ao seu endereço em (1.8), o que está exatamente correto, porque seus *bits* representam a constante 2, não a localização 2.

Carregadores e Editores de Ligação

Usualmente, um programa chamado *carregador* realiza as duas funções de carga e de edição de ligações. O processo de carga consiste em se tomar um código relocável de máquina, alterar os endereços relocáveis, como discutido no Exemplo 1.3, e em se colocar as instruções alteradas e os dados na memória nas localizações apropriadas.

O editor de ligações nos permite criar um único programa a partir de diversos arquivos de código relocável de máquina. Esses arquivos podem ter sido o resultado de diversas compilações diferentes e um ou mais deles podem ser arquivos de bibliotecas de rotinas, providenciadas pelo sistema, e disponíveis a qualquer programa que delas necessite.

Se os arquivos estão destinados a serem usados juntos numa forma útil, devem existir algumas *referências externas*, nas quais o código de um arquivo se refira a uma localização em outro arquivo. A referência pode ser para uma localização de dados definida em um arquivo, e usada num outro, ou pode ser um ponto de entrada de um procedimento que figure no código para um arquivo, e seja chamado a partir de um outro arquivo. O código relocável de máquina precisa reter a informação da tabela de símbolos para cada localização de dados ou rótulo de instrução que seja referido externamente. Se não soubermos antecipadamente o que será referido, precisaremos incluir, com efeito, toda a tabela de símbolos de montagem como parte do código relocável de máquina.

Por exemplo, o código de (1.7) seria precedido por

```
a       0
b       4
```

Se um arquivo, carregado com o conteúdo de (1.7) se referisse a `b`, então tal referência seria substituída por 4 mais o deslocamento pelo qual as localizações no arquivo (1.7) fossem relocadas.

1.5 O AGRUPAMENTO DAS FASES

A discussão das fases na Seção 1.3 lida com a organização lógica do compilador. Numa implementação, as atividades de mais de uma fase são freqüentemente agrupadas.

Interfaces de Vanguarda e Retaguarda

Freqüentemente, as fases são coletadas numa interface de *vanguarda* ou de *retaguarda*. A interface de vanguarda consiste naquelas fases, ou partes de fases, que dependem primariamente da linguagem fonte e são amplamente independentes da máquina alvo. Dentre essas fases são normalmente incluídas a análise léxica e a sintática, a criação da tabela

de símbolos, a análise semântica, e a geração do código intermediário. Uma certa quantidade de otimizações de código pode ser feita igualmente pela interface de vanguarda. A interface de vanguarda também inclui o tratamento de erros que está associado a essas fases.

A interface de retaguarda inclui aquelas partes do compilador que dependem da máquina alvo e que, geralmente, não dependem da linguagem fonte, tão-só da linguagem intermediária. Na interface de retaguarda encontramos alguns aspectos das fases de otimização e de geração de código, juntamente com as operações de tratamento de erro e manipulação da tabela de símbolos necessárias.

Tem se tornado uma praxe tomar a interface de vanguarda de um compilador e refazer sua interface de retaguarda associada, de forma a produzir um compilador para a mesma linguagem fonte numa máquina diferente. Se a interface de retaguarda tiver sido projetada cuidadosamente, pode nem ser mesmo necessário reprojetar muito de sua interface de retaguarda; esse assunto é discutido no Capítulo 9. É também atraente recompilar várias diferentes linguagens na mesma linguagem intermediária e usar uma interface de retaguarda comum para as diferentes interfaces de vanguarda, obtendo vários compiladores para a mesma máquina. No entanto, em decorrência das diferenças sutis de enfoques nas diferentes linguagens, tem havido um sucesso apenas limitado nessa direção.

Passagens

Várias fases da compilação são usualmente implementadas numa única *passagem,* consistindo na leitura de um arquivo de entrada e da escrita de um arquivo de saída. Na prática, existe grande variação na forma em que as fases de um compilador são agrupadas em passagens e, por conseguinte, preferimos organizar nossa discussão da compilação em torno das fases, ao invés das passagens. O Capítulo 12 discute alguns compiladores representativos e menciona a forma com que foram estruturadas as fases em passagens.

Como mencionamos, é comum que várias fases sejam agrupadas numa única passagem e que as atividades dessas fases estejam entremeadas durante a mesma. Por exemplo, a análise léxica, a análise sintática, a análise semântica e a geração de código intermediário poderiam ser agrupadas numa passagem. Se assim o forem, o fluxo de *tokens* após a análise léxica pode ser traduzido diretamente em código intermediário. Mais detalhadamente, podemos pensar no analisador sintático como sendo "a chefia". O analisador sintático tenta descobrir a estrutura gramatical nos *tokens* que enxerga; obtém os *tokens,* à medida que deles necessita, através de chamadas ao analisador léxico, a fim de que encontre o próximo *token.* À medida que a estrutura gramatical é descoberta, o analisador sintático chama o gerador de código intermediário para realizar a análise semântica e gerar uma parte do código. Um compilador organizado dessa forma é apresentado no Capítulo 2.

Reduzindo o Número de Passagens

É desejável se ter relativamente poucas passagens, dado que toma tempo ler e gravar arquivos intermediários. Por outro lado, se agrupamos várias fases numa única passagem, podemos ser forçados a manter todo o programa na memória, porque uma fase pode precisar de informações numa ordem diferente da que a fase anterior produziu. A forma interna do programa pode ser consideravelmente maior do que o programa fonte e também do que o programa alvo e, dessa forma, esse espaço não deve ser considerado um assunto trivial.

Para algumas fases, o agrupamento em uma passagem apresenta uns poucos problemas. Por exemplo, como mencionado acima, a interface entre o analisador léxico e o sintático pode ser freqüentemente limitada a um único *token.* Por outro lado, é freqüentemente muito difícil realizar a geração de código antes que a representação intermediária tenha sido completamente gerada. Por exemplo, linguagens como PL/I e Algol 68 permitem que as variáveis sejam usadas antes de serem declaradas. Não podemos gerar o código alvo para uma construção se não conhecemos os tipos das variáveis envolvidas na mesma. Similarmente, a maioria das linguagens permite desvios que saltem para adiante no código. Não podemos determinar o endereço alvo de tais saltos até que tenhamos visto o código fonte interveniente e o código alvo gerado para o mesmo.

Em alguns casos, é possível deixar um espaço vazio para a informação ausente e preenchê-lo quando a mesma se tornar disponível. Em particular, a geração do código intermediário e do código alvo podem ser freqüentemente combinadas numa única passagem usando-se uma técnica chamada de "retrocorreção". Conquanto não possamos explicar todos os detalhes até que tenhamos visto a geração de código intermediário no Capítulo 8, podemos ilustrar a retrocorreção em termos de um montador. Relembremos que, na secção anterior, discutimos um montador de duas passagens, onde na primeira eram determinados todos os identificadores que representavam localizações de memória e se deduziam seus endereços à medida que fossem descobertos. Uma segunda passagem substituía, então, os identificadores pelos endereços.

Podemos combinar a ação das passagens como se segue. Ao se encontrar um enunciado de montagem que seja uma referência posterior, digamos

```
GOTO alvo
```

geramos o esqueleto de uma instrução, com a operação de máquina para o `GOTO` e espaços para o endereço. Todas as instruções com espaços em branco para os endereços de `alvo` são mantidas numa lista associada à entrada da tabela de símbolos para `alvo`. Os espaços em branco serão preenchidos quando finalmente encontrarmos uma instrução tal como

```
alvo: MOV valor, R1
```

e determinarmos o valor de `alvo;` é o endereço da instrução corrente. Retrocorrigimos, então, percorrendo a lista para `alvo`, para todas as instruções que necessitem de seu endereço, substituindo pelo endereço de `alvo` os espaços em branco dos campos de endereço daquelas instruções. Essa abordagem é fácil de implementar se as instruções puderem ser mantidas na memória até que todos os endereços alvo sejam determinados.

Esse enfoque é razoável para um montador que possa manter toda a sua saída na memória. Como as representações finais de código para um montador são grosseiramente as mesmas e certamente de tamanho aproximadamente igual, a retrocorreção sobre o tamanho de todo o programa de montagem não é inviável. Entretanto, num compilador com um código intermediário consumidor de espaço, podemos precisar nos precaver quanto à distância sobre a qual a retrocorreção atua.

1.6 FERRAMENTAS PARA A CONSTRUÇÃO DE COMPILADORES

O escritor de um compilador, como qualquer outro programador, pode usar, vantajosamente, ferramentas de *software,* tais como depuradores, gerenciadores de versões, customizadores e assim por diante. No Capítulo 11, veremos como algumas delas podem ser usadas para implementar um compilador. São mencionadas apenas brevemente nesta secção; são cobertos, em detalhe, nos capítulos apropriados.

Logo após a escrita dos primeiros compiladores, surgiram os sistemas para auxiliar esse processo. Foram freqüentemente referidos como *compiladores de compiladores, geradores de compiladores e sistemas de escrita de tradutores.* São amplamente orientados em torno de um modelo particular de linguagem e mais adequados para gerar compiladores de linguagens similares ao modelo.

Por exemplo, é tentador assumir que os analisadores léxicos sejam essencialmente os mesmos para todas as linguagens, exceto para

as palavras-chave e símbolos reconhecidos. Muitos compiladores de compiladores produzem realmente rotinas de análise léxica fixas para uso no compilador gerado. Essas rotinas diferem somente na lista de palavras-chave reconhecida e essa lista precisa ser fornecida pelo usuário. O enfoque é válido, mas pode se tornar inviável se for requerido reconhecer *tokens* não padrão, tais como identificadores que possam incluir certos caracteres além de letras e dígitos.

Algumas ferramentas gerais foram criadas para o projeto automático de componentes específicos do compilador. Essas ferramentas usam linguagens especializadas para especificar e implementar o componente e muitas usam algoritmos um tanto sofisticados. As ferramentas de maior sucesso são aquelas que escondem os detalhes do algoritmo de geração e produzem componentes que podem ser facilmente integrados à parte restante do compilador. O que se segue é uma lista de algumas ferramentas úteis para a construção de compiladores:

1. *Geradores de analisadores gramaticais.* Produzem analisadores sintáticos, normalmente a partir de entrada baseada numa gramática livre de contexto. Nos primeiros compiladores, a análise sintática consumia uma grande parte não só do tempo de execução de um compilador mas, também, do esforço intelectual para se escrevê-lo. Essa fase é agora considerada uma das mais fáceis de se implementar. Muitas das "pequenas linguagens" usadas para composição de tipos deste livro, tais como PIC (Kernigham [1982]) e EQN foram implementadas nuns poucos dias usando-se o gerador de *parsers* descrito na Seção 4.7. Muitos geradores de *parsers* usam algoritmos de análise gramatical que são muito complexos para serem realizados a mão.

2. *Geradores de analisadores léxicos.* Geram automaticamente analisadores léxicos, normalmente a partir de uma especificação baseada em expressões regulares, discutidas no Capítulo 3. A organização básica do analisador léxico resultante é, com efeito, um autômato finito. Um gerador de *scanners* típico e sua implementação são discutidos nas Seções 3.5 e 3.8.

3. *Dispositivos de tradução dirigida pela sintaxe.* Produzem coleções de rotinas que percorrem uma árvore gramatical, tal como a da Fig. 1.4, gerando código intermediário. A idéia básica é que uma ou mais "traduções" sejam associadas a cada nó da árvore gramatical e que cada tradução seja definida em termos das traduções de seus nós vizinhos na árvore. Tais dispositivos são discutidos no Capítulo 5.

4. *Geradores automáticos de código.* Tal ferramenta toma uma coleção de regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina da máquina alvo. Tais regras precisam incluir detalhamento suficiente para que possamos lidar com os diferentes métodos de acesso possíveis para os dados; por exemplo, as variáveis podem estar em registradores, numa localização fixa (estática) de memória ou podem ser reservadas numa posição de uma pilha. A técnica básica é a de "correspondência de gabaritos". Os enunciados do código intermediário são substituídos por "gabaritos" que representam as seqüências das instruções de máquina, de uma forma tal que as suposições sobre o armazenamento das variáveis se correspondem de gabarito a gabarito. Como existem usualmente muitas opções relacionadas a onde as variáveis devam ser colocadas (por exemplo, em um dentre vários registradores ou na memória), existem diversas maneiras de se "moldar" o código intermediário com um dado conjunto de gabaritos, e é necessário selecionar uma boa moldagem sem uma explosão combinatória no tempo de execução do compilador. As ferramentas dessa natureza são cobertas no Capítulo 9.

5. *Dispositivos de fluxo de dados.* Muito da informação necessitada para se realizar uma boa otimização de código envolve a "análise do fluxo de dados", capturando-se a informação sobre como os valores são transmitidos de uma parte do programa para outra. As diferentes tarefas dessa natureza podem ser realizadas essencialmente pela mesma rotina, com o usuário fornecendo os detalhes do relacionamento entre os enunciados do código intermediário e a informação sendo capturada. Uma ferramenta dessa natureza é descrita na Secção 10.11.

NOTAS BIBLIOGRÁFICAS

Ao escrever, em 1962, sobre a história da escrita dos compiladores, Knuth [1962] observou que: "Nesse campo tem havido uma quantidade de inusitada de descobertas paralelas da mesma técnica, por pessoas trabalhando independentemente." Continuou, observando que vários indivíduos isolados haviam, de fato, descoberto "vários aspectos de uma técnica e que a mesma foi polida através dos anos num algoritmo muito agradável, que nenhum dos originadores concretizou totalmente". Reivindicar a autoria intelectual de técnicas permanece uma tarefa perigosa; as notas bibliográficas servem aqui meramente como um auxílio para estudo posterior da literatura.

As notas históricas no desenvolvimento das linguagens de programação e compiladores até a chegada de Fortran podem ser encontradas em Knuth e Trabb Pardo [1977]. Wexelblat [1981] contém depoimentos históricos, sobre várias linguagens de programação, fornecidos por participantes de seus desenvolvimentos.

Alguns artigos iniciais, fundamentais na compilação, foram coletados em Rosen [1967] e Pollack [1972]. A edição de janeiro de 1961 do *Communications of ACM* fornece um retrato do estado da arte da escrita de compiladores àquela época. Uma prestação de contas detalhada de um compilador Algol inicial é feita por Randell e Russell [1964].

Começando ao início dos anos 60, com o estudo da sintaxe, os estudos teóricos tiveram uma profunda influência no desenvolvimento da tecnologia de compiladores e, talvez, no mínimo, tanta influência quanto em qualquer outra área da ciência da computação. O aspecto fascinante da sintaxe há muito tempo se desvaneceu, mas a compilação como um todo continua a ser assunto de uma pesquisa viva. Os frutos dessa pesquisa se tornarão evidentes quando examinarmos a compilação em mais detalhes nos próximos capítulos.