Capítulo 6

Analisadores léxicos

A definição do autômato que reconhece se um símbolo pertence ou não a uma dada classe foi vista em detalhes na seção anterior. A questão que se coloca agora é: como isto é traduzido em um programa?

Não se pretende aqui entrar no mérito de como desenvolver um programa a partir de uma especificação inicial, que é objeto de estudo da área de Engenharia de Software. Neste capítulo, no entanto, serão abordados os principais aspectos envolvidos na construção do módulo de análise léxica de um compilador.

Uma boa forma de se iniciar a construção de um programa é partir de uma descrição do que ele deve fazer. Desta forma, evita-se chegar a um ponto no desenvolvimento no qual você já esqueceu o que estava fazendo... Neste caso, uma descrição inicial pode ser:

> O programa do analisador léxico recebe o nome de um arquivo que contém os *tokens* que devem ser analisados. O analisador submete ao autômato cada *token*. O autômato retorna a indicação se o *token* pertence à linguagem por ele reconhecida. Caso algum *token* não seja reconhecido, o analisador indica qual foi o *token* e encerra a execução.

O núcleo desse programa é a funcionalidade de reconhecimento de uma *string* (o *token*) pertencente a uma linguagem regular por um autômato. Os aspectos conceituais relativos ao reconhecimento do *token* pelo autômato foram apresentados no Capítulo 5. Na seqüência, apresenta-se um procedimento sistemático de como executar esse reconhecimento através de um programa.

6.1 Um algoritmo para o analisador

Tendo visto qual a fundamentação por trás do processo de reconhecimento de *tokens*, pode-se definir o núcleo do analisador léxico como uma implementação de um autômato finito, que reconhece *strings* como símbolos válidos de uma linguagem ou dá uma indicação de que a *string* não pertence à linguagem.

A implementação desse analisador léxico requer uma descrição do autômato que reconhece as sentenças da gramática ou expressão regular de interesse. A forma computacional mais simples para representar o autômato é através de uma tabela de transições, a partir da qual deve ser possível obter as seguintes informações:

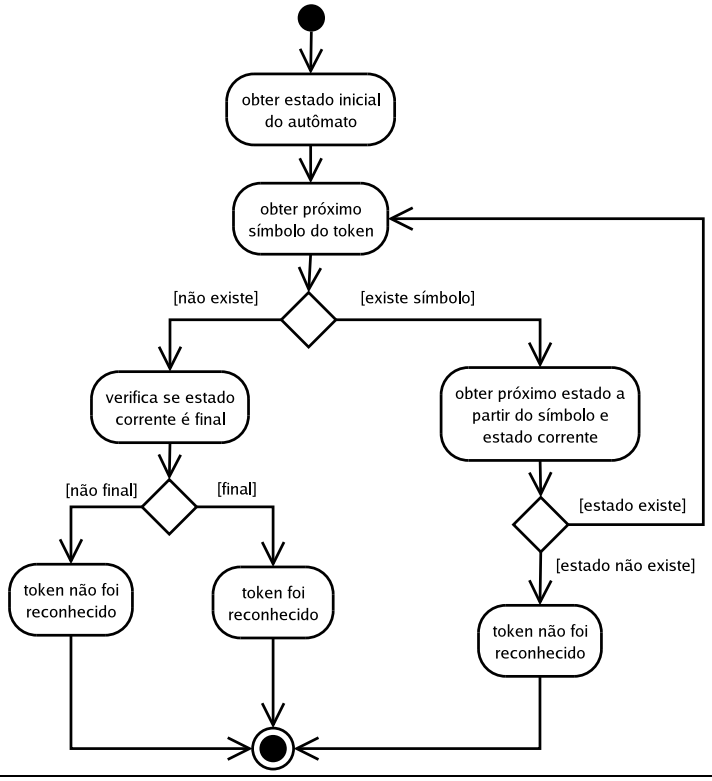
1. O estado inicial para o autômato;

2. Dado um estado qualquer, indicar se este é um estado final (condição de aceitação); e

3. Dado um estado qualquer e um símbolo, a indicação de qual é o próximo estado.

O procedimento de reconhecimento é apresentado graficamente na Figura 6.1, na forma de um diagrama de atividades UML. A inicialização do procedimento coloca o autômato num estado conhecido, correspondente ao seu estado inicial. A partir deste estado é que os símbolos da *string* serão analisados.

Figura 6.1 Diagrama de atividades para procedimento de reconhecimento de *token*



O procedimento prossegue com um laço que tentará analisar cada um dos símbolos da *string*. O início do laço obtém o próximo símbolo, que inicialmente é o primeiro símbolo, da *string*. Se não houver próximo símbolo, é preciso verificar se o estado corrente é um estado final. Se for, o procedimento retorna uma indicação

de que a *string* foi reconhecida pelo autômato. Se o estado corrente não for um estado final e não houver mais símbolos na sentença, então deve ser retornada uma indicação de que não houve reconhecimento.

Se houver símbolo a ser analisado, então o procedimento deve continuar o processo de reconhecimento. Para tanto, obtém o próximo estado correspondente à transição do estado atual pelo símbolo sob análise. Se não houver transição possível, então a *string* não foi reconhecida e o procedimento deve encerrar com essa indicação.

Uma descrição algorítmica desse procedimento é apresentada no Algoritmo 6.1, que determina se a *string* σ pertence à linguagem reconhecida pelo autômato M. O algoritmo utiliza uma variável s para denotar o estado corrente do autômato e uma variável c para o próximo símbolo de entrada da *string* σ.

Algoritmo 6.1 Algoritmo para um analisador léxico.

```
RECONHECE(M, σ)
1  s ← ESTADO-INICIAL(M)
2  while TEM-SÍMBOLOS(σ)
3  do c ← PRÓXIMO-SÍMBOLO(σ)
4     if EXISTE-PRÓXIMO-ESTADO(M, s, c)
5        then s ← PRÓXIMO-ESTADO(M, s, c)
6        else return false
7  if ESTADO-FINAL(M, s)
8     then return true
9     else return false
```

Além das variáveis, o algoritmo também faz usos de diversos procedimentos auxiliares. Para o autômato, esses procedimentos são ESTADO-INICIAL, que retorna um estado; ESTADO-FINAL, que retorna verdadeiro se o estado indicado fizer parte do conjunto de estados finais; EXISTE-PRÓXIMO-ESTADO, que retorna verdadeiro de houver transição possível a partir do estado e símbolo indicados; e PRÓXIMO-ESTADO, que retorna o estado decorrente dessa transição. Para a *string*, o procedimento TEM-SÍMBOLOS retorna verdadeiro se há na *string* símbolos ainda não processados; o procedimento PRÓXIMO-SÍMBOLO retorna o primeiro caráter da *string* que ainda não foi processado.

6.2 Representação do autômato

A implementação do autômato demanda a utilização de algumas estruturas de dados que permitam armazenar a sua representação e, a partir dela, obter as informações necessárias para realizar as operações descritas no algoritmo do analisador.

O primeiro tipo de informação é simples e pode ser armazenado em uma variável escalar — é a informação sobre qual é o estado inicial do autômato. Como existe

apenas um estado inicial para cada autômato, não é preciso usar nenhuma estrutura mais complexa para esse armazenamento. Com essa informação, o procedimento ESTADO-INICIAL pode ser facilmente implementado.

Para os demais procedimentos do autômato, outras estruturas devem ser utilizadas. Essas estruturas de dados, genericamente denominadas *containers* associativos, lidam com coleções de valores e como acessar a informação armazenada usando para isto parte da própria informação — uma chave de busca. Tais estruturas e sua implementação na STL de C++ são descritas na seqüência.

6.2.1 Conjuntos

Uma estrutura de dados do tipo conjunto implementa funcionalidades associadas ao conceito matemético de conjunto. Em outras palavras, com uma estrutura deste tipo é possível determinar se um elemento pertence ou não ao conjunto. Para o analisador léxico, esse tipo de estrutura suporta a implementação do procedimento ESTADO-FINAL.

Em C++, conjuntos são implementados na biblioteca STL por *containers* do tipo `set`. Assim como para os *containers* descritos na Seção 2.6, a definição de um conjunto precisa da especificação de qual tipo de conteúdo será armazenado no conjunto. Por exemplo, para criar um conjunto de *strings*, o seguinte comando seria utilizado:

```
#include <set>
#include <string>
...
set<string> finais;
```

Assim como os *containers* descritos na Seção 2.6, é possível utilizar os métodos `empty` e `size` para obter informações sobre a quantidade de elementos em um conjunto. Da mesma forma, o método `insert` é utilizado para acrescentar um elemento ao conjunto. Para descobrir se um elemento pertence ou não ao conjunto, o método `find` é utilizado.

O `set` também suporta operações cujos argumentos são outros conjuntos do mesmo tipo. As operações de união e interseção, por exemplo, são suportadas através dos métodos `set_union` e `set_intersection`, respectivamente.

Ao implementar o conceito matemático de um conjunto, `set` não armazena elementos de mesmo valor mais de uma vez. Assim, uma operação de `insert` com um argumento cujo valor já existe no conjunto não tem efeito. Uma implementação variante, `multiset`, admite a presença de elementos duplicados.

6.2.2 Mapas

Um mapa é um *container* associativo, assim como um conjunto. No entanto, mapas armazenam como elementos pares de valores, dos quais o primeiro é utilizado como a chave de acesso para o segundo. No caso do autômato, esta seria a estrutura

adequada para representar a tabela de transições, com uma chave composta pelo estado e pelo símbolo correntes que leva ao valor que é o próximo estado. Com esta informação, é possível implementar os procedimentos auxiliares EXISTE-PRÓXIMO-ESTADO e PRÓXIMO-ESTADO.

Em C++, mapas são implementados através da classe parametrizada `map`. Como dois valores são utilizados, na instanciação de um `map` é preciso especificar dois parâmetros, que indicam respectivamente o tipo da chave e o tipo do valor. Por exemplo, para criar um mapa no qual o primeiro elemento de cada par (a chave) é um valor inteiro e o segundo elemento é uma *string*, os seguintes comandos deveriam estar presentes no código:

```
#include <map>
#include <string>
...
map<int,string> meuMapa;
```

Para manipular o mapa, os métodos usuais de um *container* podem ser utilizados. Adicionalmente, `map` sobrecarrega o operador de indexação `[]` usando a chave como índice de acesso. Esta forma pode ser utilizada tanto para inserir novos elementos como para encontrar elementos já existentes.

Da mesma forma que `set`, as chaves de um `map` não admitem valores duplicados. Se um par for armazenado com um valor de chave que já existe no mapa, o valor do segundo elemento original é perdido. Há no entanto um *container* `multimap` que aceita chaves iguais em elementos distintos.

6.2.3 Aspectos de implementação de *containers* associativos

Há várias formas possíveis de se implementar estruturas do tipo *container* associativo. A forma mais simples utiliza uma estrutura linear, como o vetor apresentado na Seção 2.6.1. Neste caso, o armazenamento de novos elementos seria rápido, pois bastaria incluí-lo na última posição. No entanto, a busca por elementos seria ineficiente, pois poderia demandar a comparação elemento a elemento em todas as posições do vetor.

Assim, é interessante conhecer técnicas mais eficientes para suportar esses tipos de coleção de elementos. Esta seção apresenta algumas dessas técnicas.

Vetores ordenados

Uma melhoria simples à abordagem de representação de *containers* por um vetor, com o objetivo de obter uma busca mais eficiente, é a manutenção dos seus elementos em ordem. Deste modo, é possível utilizar no momento da busca uma estratégia análoga àquela utilizada ao procurar uma palavra no dicionário:

1. Faz-se uma estimativa da posição aproximada da palavra e abre-se na página estimada.

2. Se a palavra não foi encontrada nessa página, pelo menos sabe-se em que direção buscar, se mais adiante ou mais atrás no dicionário. O processo de estimar a nova página de busca repete-se na parte do dicionário que pode conter a palavra.

Esse mecanismo de busca só é possível porque existe uma ordenação possível das palavras com base na precedência das letras no alfabeto (a chamada ordem lexicográfica) e esta ordenação é utilizada no dicionário. Se o dicionário mantivesse as palavras sem nenhuma ordem, esse tipo de busca não seria possível. Com base nesse mesmo princípio, a busca em um vetor pode ser melhorada se seu conteúdo puder ser ordenado.

O algoritmo de **busca binária** utiliza esse princípio. No caso, a estimativa que é feita para a posição a ser buscada é a posição mediana do restante do vetor que ainda precisa ser analisado. No início, este "restante" é o vetor completo; assim, a sua posição central é a primeira a ser analisada. Se seu conteúdo não for a entrada buscada, analisa-se a metade que resta, ou a inferior (se a chave encontrada tem valor maior que a procurada) ou a superior (em caso contrário). O procedimento assim se repete, até que se encontre a entrada desejada ou que a busca se esgote sem que esta seja encontrada.

O Algoritmo 6.2 descreve essa forma de busca. Os dois argumentos são o vetor T e o elemento que será utilizado como chave de busca c, cujo tipo é genericamente aqui denotado como ELEMENT. O retorno é uma indicação se o elemento está presente ou não nesta coleção. As variáveis bot, mid e top referem-se a posições no vetor — respectivamente o início, o meio e o fim da área ainda por ser pesquisada. A notação $\lfloor x \rfloor$ denota o maior inteiro cujo valor é menor ou igual a x.

Algoritmo 6.2 Busca binária em um vetor.

```
BINARYSEARCH(VECTOR T, ELEMENT c)
 1  declare bot, mid, top : INTEGER
 2  bot ← 0
 3  top ← T.size() − 1
 4  while bot ≤ top
 5  do mid ← ⌊(bot + top)/2⌋
 6     if c > T[mid]
 7        then bot ← mid + 1
 8        else  if c < T[mid]
 9                 then top ← mid − 1
10                 else  return true
11  return false
```

A manutenção da ordem em um vetor dá-se através de procedimentos auxiliares de **ordenação**, uma das áreas relevantes de estudos em estruturas de dados. Por simplicidade, é assumido nessa apresentação que os conteúdos que serão ordenados

estão sempre contidos em memória. Os algoritmos de ordenação que trabalham com essa restrição são denominados **algoritmos de ordenação interna**. Algoritmos de ordenação externa manipulam conjuntos de valores que podem estar contidos em arquivos maiores, armazenados em discos ou outros dispositivos de armazenamento externos à memória principal. Os algoritmos de ordenação interna (em memória) são convencionalmente baseados em estratégias de comparação *(quicksort, heapsort)* ou em estratégias de contagem *(radixsort)*.

Um algoritmo básico de ordenação é o algoritmo de **ordenação pelo menor valor**, que pode ser sucintamente descrito como a seguir. Inicialmente, procure a entrada que apresenta o menor valor de todo o vetor. Uma vez que seja definido que posição contém esse valor, troque seu conteúdo com aquele da primeira posição do vetor; desta forma, o menor valor estará na posição correta. Repita então o procedimento para o restante do vetor, excluindo os elementos que já estão na posição correta.

Esse procedimento é descrito no Algoritmo 6.3, que recebe como argumento o vetor a ser ordenado. No procedimento, $pos1$ e $pos2$ são as posições sob análise. A variável min corresponde à posição onde foi encontrado o menor valor até o momento, mantido na variável sml.

Algoritmo 6.3 Ordenação de vetor pela busca do menor valor.

```
MINENTRYSORT(VECTOR T)
1  declare min, pos1, pos2 : INTEGER
2  declare sml : ELEMENT
3  for pos1 ← 0 to T.size() − 2
4  do sml ← +∞
5     for pos2 ← pos1 to T.size() − 1
6     do if T[pos2] < sml
7           then sml ← T[pos2]
8                min ← pos2
9     T.swap(pos1, min)
```

Neste algoritmo, o laço de iteração mais externo indica o primeiro elemento no vetor não ordenado a ser analisado — no início, esse é o seu primeiro elemento. As linhas de 4 a 8 são responsáveis por procurar, no restante do vetor, o elemento com menor valor. Na linha 9, o método $swap$ troca o conteúdo das duas entradas nas posições especificadas.

Este tipo de algoritmo de ordenação é razoável para manipular coleções com um pequeno número de elementos, mas à medida que esse tamanho cresce o desempenho torna seu uso inviável — sua complexidade temporal é $O(n^2)$, conseqüência do duplo laço de iteração que varre o vetor até o final. Felizmente, há outros algoritmos de ordenação com melhor comportamento de desempenho em situações onde a quantidade de elementos cresce.

Quicksort é baseado no princípio de "dividir para conquistar:" o conjunto de elementos a ser ordenado é dividido em dois subconjuntos (partições), que sendo menores irão requerer menor tempo total de processamento que o conjunto total, uma vez que o tempo de processamento para a ordenação não é linear com a quantidade de elementos. Em cada partição criada, o procedimento pode ser aplicado recursivamente, até um ponto onde o tamanho da partição seja pequeno o suficiente para que a ordenação seja realizada de forma direta por outro algoritmo.

Nesta descrição do procedimento QUICKSORT (Algoritmo 6.4), os seus argumentos determinam as posições inicial e final do segmento do vetor a ser ordenado, $init$ e end, respectivamente.

Algoritmo 6.4 Ordenação de vetor por *quicksort*.

```
QUICKSORT(VECTOR T, INTEGER init, INTEGER end)
 1  declare pos1, pos2, part : INTEGER
 2  if init < end
 3    then pos1 ← init + 1
 4         pos2 ← end
 5         while true
 6         do while T[pos1] < T[init]
 7            do pos1 ← pos1 + 1
 8            while T[pos2] > T[init]
 9            do pos2 ← pos2 − 1
10            if pos1 < pos2
11              then part ← pos1
12                   T.swap(pos1, pos2)
13              else part ← pos2
14                   break
15         T.swap(init, part)
16         QUICKSORT(T, init, part − 1)
17         QUICKSORT(T, part + 1, end)
```

O ponto crítico deste algoritmo está na forma de realizar a partição — um elemento é escolhido como **pivô**, ou separador de partições. No Algoritmo 6.4, a posição desse pivô é mantida na variável *part*. Todos os elementos em uma partição têm valores menores que o valor do pivô, enquanto todos os elementos de outra partição têm valores maiores que o valor do pivô. Os dois laços internos (iniciando nas linhas 6 e 8) fazem a busca pelo pivô tomando como ponto de partida o valor inicial. Um melhor desempenho poderia ser obtido obtendo-se o valor médio de três amostras como ponto de partida para o pivô — por exemplo, entre os valores no início, meio e fim da partição sob análise. Dessa forma, haveria melhores chances de obter como resultado partições de tamanhos mais balanceados, característica essencial para atingir um bom desempenho para esse algoritmo.

Quicksort é um algoritmo rápido em boa parte dos casos onde aplicado, com complexidade temporal média $O(n \log n)$. Entretanto, no pior caso essa complexidade pode degradar para $O(n^2)$. Mesmo assim, implementações genéricas desse algoritmo são usualmente suportadas em muitos sistemas — por exemplo, pela rotina `qsort` da biblioteca padrão da linguagem C.

Entre os principais atrativos de *quicksort* destacam-se o fato de que na maior parte dos casos sua execução é rápida e de que é possível implementar a rotina sem necessidade de espaço de memória adicional. Uma desvantagem de *quicksort* é que todos os elementos **devem** estar presentes na memória para poder iniciar o processo de ordenação. Isto inviabiliza seu uso para situações de ordenação *on the fly*, ou seja, onde o processo de ordenação ocorre à medida que elementos são obtidos.

Outra classe de algoritmos de ordenação é aquela na qual a definição da posição ordenada de um elemento se dá pela contagem do número de elementos com cada valor. O princípio básico é simples. Considere por exemplo uma coleção de elementos a ordenar onde as chaves podem assumir N valores diferentes. Cria-se então uma tabela com N contadores e varre-se a coleção do início ao fim, incrementando-se o contador correspondente à chave i cada vez que esse valor for encontrado. Ao final dessa varredura conhece-se exatamente quantas posições serão necessárias para cada valor; os elementos são transferidos para as posições corretas na nova coleção, agora ordenada.

Claramente, a aplicação desse princípio básico de contagem a domínios com muitos valores torna-se inviável. Por exemplo, se os elementos são inteiros de 32 bits, o algoritmo de contagem básico precisaria de uma tabela com cerca de quatro bilhões (2^{32}) de contadores.

Radix sort é um algoritmo baseado neste conceito de ordenação por contagem que contorna este problema ao aplicar o princípio da ordenação por contagem a uma parte da representação do elemento, a **raiz**. O procedimento é repetido para a raiz seguinte até que toda a representação dos elementos tenha sido analisada. Por exemplo, a ordenação de chaves inteiras com 32 bits pode ser realizada em quatro passos usando uma raiz de oito bits, sendo que a tabela de contadores requer apenas 256 (2^8) entradas.

O procedimento para execução de *radix sort* é descrito no Algoritmo 6.5. Para essa descrição, assumiu-se que elementos do vetor são inteiros positivos, que serão analisados em blocos de R bits a cada passo. As variáveis internas ao procedimento são $pass$, que controla o número de passos executados e também qual parte do elemento está sob análise, iniciando pelos R bits menos significativos; pos, que indica qual posição do vetor está sob análise; $radixValue$, o valor da parte do elemento (entre 0 e $2^R - 1$) no passo atual; $count$, a tabela de contadores; e T_{aux}, uma cópia do vetor ordenado segundo a raiz sob análise ao final de cada passo. A notação $\lceil x \rceil$ denota o menor inteiro cujo valor é maior ou igual a x.

O laço mais externo do algoritmo RADIXSORT (linhas 4 a 16) é repetido tantas vezes quantas forem necessárias para que a chave toda seja analisada em blocos de tamanho R bits. Utiliza-se na linha 4 um operador SIZEOFBITS para determinar

Algoritmo 6.5 Ordenação de vetor por *radixsort*.

```
RADIXSORT(VECTOR T, INTEGER R)
 1  declare pass, pos, radixValue : INTEGER
 2  declare count : array[2^R] of INTEGER
 3  declare T_aux : VECTOR
 4  for pass ← 1 to ⌈SIZEOFBITS(INTEGER)/R⌉
 5  do for radixValue ← 0 to 2^R − 1
 6     do count[radixValue] ← 0
 7     for pos ← 0 to T.size() − 1
 8     do radixValue ← (T[pos] >> (R × (pass − 1))) & (2^R − 1)
 9        count[radixValue] ← count[radixValue] + 1
10     for radixValue ← 1 to 2^R − 1
11     do count[radixValue] ← count[radixValue] + count[radixValue − 1]
12     for pos ← T.size() − 1 to 0
13     do radixValue ← (T[pos] >> (R × (pass − 1))) & (2^R − 1)
14        T_aux[count[radixValue]] ← T[pos]
15        count[radixValue] ← count[radixValue] − 1
16     T ← T_aux
```

o tamanho do elemento em bits. Em C ou C++, este seria implementado com o operador `sizeof`, que retorna o tamanho do tipo em bytes, e a informação sobre quantos bits há em um byte.

O primeiro laço interno do algoritmo (linhas 5 e 6) simplesmente inicializa o arranjo de contadores, pois este será reutilizado em todos os demais passos do laço. No laço seguinte (linhas 7 a 9), o vetor é percorrido para avaliar o valor da raiz em cada posição (linha 8, usando os operadores binários SHIFT, >> e AND, &) e assim atualizar a contagem de valores (linha 9). Na seqüência (linhas 10 e 11), gera-se a soma acumulada de contadores, o que permite avaliar quantas chaves com raiz de valor menor que o indicado existem. Essa informação permitirá que, no próximo laço (linhas 12 a 15), o vetor auxiliar T_{aux} seja preenchido colocando cada entrada do vetor T na nova posição que lhe corresponde segundo esse valor de raiz; cada vez que uma entrada é colocada na tabela, o valor do contador associado deve ser decrementado (linha 15) para que o elemento com a próxima raiz de mesmo valor seja colocada na posição correta, anterior à última ocupada. Finalmente, o vetor T recebe a tabela ordenada segundo a raiz e o procedimento é repetido para o bloco de bits seguinte da chave. Após a varredura do último bloco (o mais significativo), o vetor estará completamente ordenada.

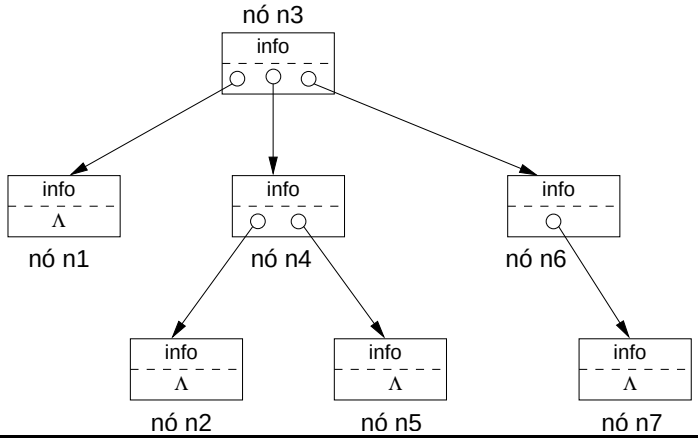
Este algoritmo requer, para seu correto funcionamento, que a ordenação utilizada em cada etapa intermediária seja estável, ou seja, que dois elementos com mesmo valor mantenham suas posições relativas na nova seqüência ordenada. *Radix sort* é um algoritmo rápido, mas apresenta como principal desvantagem a necessidade de espaço adicional de memória — uma área do mesmo tamanho ocupado pelo conjunto

de elementos sendo ordenado é necessária para receber os dados re-ordenados após cada contagem. Quando o espaço de memória não é um recurso limitante, *radix sort* é um algoritmo atrativo, com complexidade temporal linear $O(n)$.

Árvores binárias

Outra estrutura extensivamente utilizada na programação de sistemas é a estrutura de árvore, que esquematicamente pode ser visualizada como uma extensão de uma lista ligada na qual um nó pode ter mais de um sucessor (Figura 6.2). A representação esquemática de árvores usualmente coloca a raiz no topo, com a árvore crescendo para baixo.

Figura 6.2 Representação esquemática de uma estrutura de árvore.



Uma **árvore** é uma estrutura que contém um conjunto finito de um ou mais nós, sendo que um dos nós é especialmente designado como o **nó raiz** e os demais nós são particionados em 0 ou mais conjuntos disjuntos onde cada um desses conjuntos é em si uma árvore, que recebe o nome de sub-árvore.

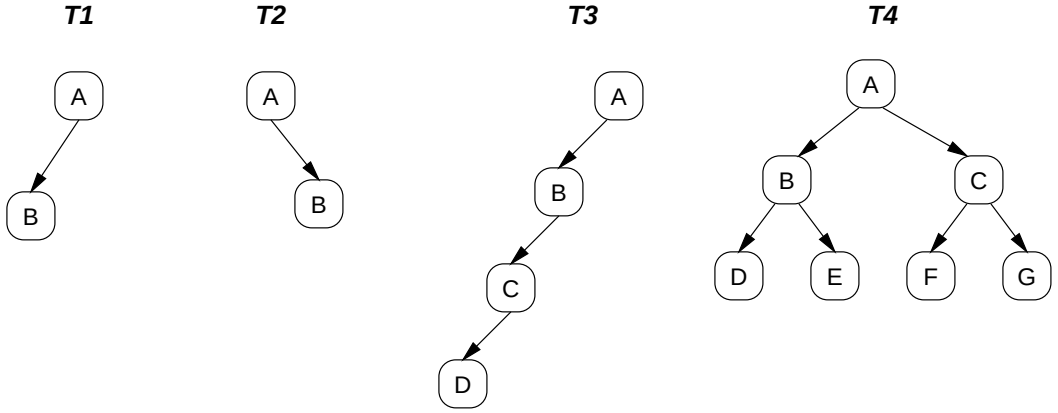
A Figura 6.2 ilustra um exemplo de uma árvore T, que tem o nó raiz $n3$ e as sub-árvores T_1, T_2 e T_3. A sub-árvore T_1 tem o nó raiz $n1$ e não contém sub-árvores; sub-árvore T_2 tem o nó raiz $n4$ e sub-árvores T_4 e T_5; e a sub-árvore T_3 tem o nó raiz $n6$ e sub-árvore T_6. No próximo nível, as sub-árvores T_4, T_5 e T_6 têm respectivamente os nós raízes $n2$, $n5$ e $n7$ e não têm sub-árvores.

O número de sub-árvores de um nó é o **grau do nó**. No exemplo, o nó $n3$ tem grau 3; $n4$, 2; e $n5$, 0. O **grau da árvore** é o maior valor de grau de nó entre todos os nós da árvore; no exemplo, a árvore T tem grau 3. Um nó que não tem sub-árvores, ou seja, cujo grau é 0, é normalmente denominado **nó folha** da árvore. No exemplo, a árvore T tem folhas $n1$, $n2$, $n5$ e $n7$. Os nós raízes das sub-árvores de um nó são usualmente chamados de **nós filhos** desse nó, que recebe também o nome de **nó pai** daqueles nós. Em uma estrutura de árvore, cada nó tem apenas um nó pai.

Um tipo especial de árvore é a **árvore binária**. Uma árvore binária tem um nó raiz e no máximo duas sub-árvores, uma sub-árvore esquerda e uma sub-árvore

direita. Em decorrência dessa definição, o grau de uma árvore binária é limitado a dois. A Figura 6.3 ilustra alguns exemplos de árvores binárias.

Figura 6.3 Árvores binárias



Observe na figura que $T1$ e $T2$ são árvores binárias distintas pois, ao contrário da definição genérica de árvores, há diferença de tratamento para a árvore binária entre a sub-árvore direita e a sub-árvore esquerda. Outra diferença de definição para árvores binárias é que elas podem eventualmente ser vazias, algo que a definição de árvore genérica não permite. $T3$ é uma árvore binária degradada (equivalente a uma lista linear), enquanto $T4$ é uma árvore binária completa e balanceada (com sub-árvores de igual tamanho).

Uma das principais aplicações de árvores binárias é a manutenção de estruturas nas quais a ordem é importante. Para manter a ordem dos nós de uma árvore binária, três estratégias podem ser utilizadas:

Pré-ordem é a estratégia de varredura de uma árvore binária na qual o primeiro nó é o nó raiz, seguido pela sub-árvore esquerda em pré-ordem e finalmente pela sub-árvore direita em pré-ordem;

Intra-ordem é a estratégia de varredura de árvore binária na qual lê-se primeiro a sub-árvore esquerda em intra-ordem, seguido pelo nó raiz e finalmente pela sub-árvore direita em intra-ordem;

Pós-ordem é a estratégia de varredura na qual primeiro lê-se os nós da sub-árvore esquerda em pós-ordem, depois os nós da sub-árvore direita em pós-ordem e finalmente o nó raiz.

Aplicando essas estratégias à árvore $T4$ (Figura 6.3), com pré-ordem a seqüência de nós da árvore seria A, B, D, E, C, F, G; com intra-ordem, D, B, E, A, F, C, G; e com a pós-ordem, D, E, B, F, G, C, A.

A estratégia intra-ordem é usualmente utilizada para a manutenção de coleções ordenadas por valores. Todos os valores na sub-árvore esquerda de um nó precedem

o valor armazenado nesta raiz; similarmente, todos os valores na sub-árvore direita são maiores que esse valor. Deste modo, a busca por um valor na árvore inicia-se pelo nó raiz. Se o valor armazenado for igual ao buscado, encerra-se a busca. Caso o valor buscado seja menor que a raiz, a busca é repetida na sub-árvore esquerda; caso seja mair, na sub-árvore direita.

Para inserir um novo valor na árvore binária, estratégia similar é utilizada. Se a árvore estiver vazia, o elemento é inserido na raiz. Caso a raiz exista, o elemento é inserido na sub-árvore esquerda, se seu valor for menor ou igual àquela armazenado na raiz, ou na sub-árvore direita, caso contrário.

Hashing

Na apresentação da estrutura de *containers*, a busca por uma chave ocorre sempre através de comparações. Uma alternativa de busca em coleções dá-se através do cálculo da posição que uma chave ocupa numa tabela através de uma função. Esse tipo de função que mapeia um símbolo ou uma *string* de símbolos para valores inteiros é denominado **função *hash*** e o tipo de estrutura manipulada dessa forma é uma **tabela *hash***.

A grande vantagem na utilização da tabela *hash* está no desempenho — enquanto a busca linear tem complexidade temporal $O(N)$ e a busca binária tem complexidade $O(\log N)$, o tempo de busca na tabela *hash* é praticamente independente do número de chaves armazenadas na tabela, ou seja, tem complexidade temporal $O(1)$. Aplicando a função *hash* no momento de armazenar e no momento de buscar a chave, a busca pode se restringir diretamente àquela posição da tabela gerada pela função. Outro aspecto importante é que as estratégias mais eficientes baseadas em comparação, com vetores ordenados ou árvores binárias, demandam que o conjunto de valores assumido pelas chaves seja ordenável, algo que não é necessário em *hashing*.

Idealmente, cada chave processada por uma função *hash* gera uma posição diferente na tabela, um vetor com capacidade pré-definida. No entanto, na prática existem **sinônimos** — chaves distintas que resultam em um mesmo valor de *hashing*. Quando duas ou mais chaves sinônimas são mapeadas para a mesma posição da tabela, diz-se que ocorre uma **colisão**.

Uma boa função *hash* deve apresentar duas propriedades básicas: seu cálculo deve ser rápido e deve gerar poucas colisões. Além disso, é desejável que ela leve a uma ocupação uniforme da tabela para conjuntos de chaves quaisquer. Duas funções *hash* usuais são descritas a seguir:

Meio do quadrado. Nesse tipo de função, a chave é interpretada como um valor numérico que é elevado ao quadrado. Os r bits no meio do valor resultante são utilizados como o endereço em uma tabela com 2^r posições.

Divisão. Nesse tipo de função, a chave é interpretada como um valor numérico que é dividido por um valor M. O resto dessa divisão inteira, um valor entre 0 e $M - 1$, é considerado o endereço em uma tabela de M posições. Para reduzir colisões, é recomendável que M seja um número primo.

Como nem sempre a chave é um valor inteiro que possa ser diretamente utilizado, a técnica de *folding* pode ser utilizada para reduzir uma chave a um valor inteiro. Nessa técnica, a chave é dividida em segmentos de igual tamanho (exceto pelo último deles, eventualmente) e cada segmento é considerado um valor inteiro. A soma de todos os valores assim obtidos será a entrada para a função *hash*.

O processamento de tabelas *hash* demanda a existência de algum mecanismo para o tratamento de colisões. As formas mais usuais de tratamento de colisão são por endereçamento aberto ou por encadeamento.

Na técnica de tratamento de colisão por endereçamento aberto, a estratégia é utilizar o próprio espaço da tabela que ainda não foi ocupado para armazenar a chave que gerou a colisão. Quando a função *hash* gera para uma chave uma posição que já está ocupada, o procedimento de armazenamento verifica se a posição seguinte também está ocupada; se estiver ocupada, verifica a posição seguinte e assim por diante, até encontrar uma posição livre. (Nesse tipo de tratamento, considera-se a tabela como uma estrutura circular, onde a primeira posição sucede a última posição.) A entrada é então armazenada nessa posição. Se a busca termina na posição inicialmente determinada pela função *hash*, então a capacidade da tabela está esgotada e uma mensagem de erro é gerada.

No momento da busca, essa varredura da tabela pode ser novamente necessária. Se a chave buscada não está na posição indicada pela função *hashing* e aquela posição está ocupada, a chave pode eventualmente estar em outra posição na tabela. Assim, é necessário verificar se a chave não está na posição seguinte. Se, por sua vez, essa posição estiver ocupada com outra chave, a busca continua na posição seguinte e assim por diante, até que se encontre a chave buscada ou uma posição sem nenhum símbolo armazenado.

Na técnica de tratamento de colisão por encadeamento, para cada posição onde ocorre colisão cria-se uma área de armazenamento auxiliar, externa à área inicial da tabela *hash*. Normalmente essa área é organizada como uma **lista ligada** (Seção 2.6.3) que contém todas as chaves que foram mapeadas para a mesma posição da tabela. No momento da busca, se a posição correspondente à chave na tabela estiver ocupada por outra chave, é preciso percorrer apenas a lista ligada correspondente àquela posição até encontrar a chave ou alcançar o final da lista.

Hashing é uma técnica simples e amplamente utilizada na programação de sistemas. Quando a tabela *hash* tem tamanho adequado ao número de chaves que irá armazenar e a função *hash* utilizada apresenta boa qualidade, a estratégia de manipulação por *hashing* é bastante eficiente.

6.3 Obtenção de *tokens*

Além de realizar o processamento associado ao autômato, onde as estruturas de dados acima descritas são utilizadas, o analisador léxico tem a função de obter os símbolos a partir do arquivo com o código-fonte. Em geral, esses símbolos são oriundos de um arquivo com formato interno de texto (seqüencias de caracteres)

armazenados em disco.

Os recursos para a manipulação de arquivos em disco na linguagem C++ são especificados no arquivo de cabeçalho `fstream`, que deve ser incorporado ao programa que utiliza esses recursos através do comando para o pré-processador `include`.

Um arquivo cujo conteúdo de texto será lido por um programa C++ corresponde a um objeto da classe `ifstream`. Há duas formas de especificar o nome do arquivo ao objeto `ifstream` correspondente:

1. Como um argumento na própria declaração da variável, ou seja, usando um construtor com argumentos para objetos dessa classe, como em

```
ifstream arq("learq.cpp");
```

2. Através da aplicação do método `open` à variável, como em

```
ifstream arq;
...
arq.open("learq.cpp");
```

Em qualquer das duas situações, a *string* constante poderia ter sido substituída por uma variável com o nome do arquivo, como em

```
ifstream arq(argv[1]);
```

Também, um segundo argumento pode ser especificado na abertura do arquivo, que corresponde ao modo de operação. Para leitura de arquivos texto, não há nenhuma especificação de modo de operação a associar. Para a leitura de arquivos com conteúdo binário, o especificador `ios::binary` deve ser utilizado.

Uma vez identificado e aberto o arquivo sobre o qual as operações serão realizadas, o próximo passo é realizar as operações de leitura. Para arquivos de texto, são utilizados o operador `>>` ou os métodos `get` e `getline`, já apresentados para arquivos padrões na Seção 2.10.

O operador `>>` faz a leitura, com interpretação de formato, a partir da entrada especificada à sua esquerda para a variável indicada à sua direita. Por exemplo, a linha

```
char ch;
...
arq >> ch;
```

lê um caráter do arquivo associado ao objeto `arq` e armazena-o na variável `ch`. Outro método importante na leitura de arquivos é `eof`, que retorna um valor lógico verdadeiro quando o final do arquivo foi alcançado.

Um primeiro programa para apresentar, na saída padrão (`cout`), o conteúdo de um arquivo cujo nome é especificado na linha de comando, pode então ser apresentado como se segue:

```
1  #include <fstream>
2  #include <iostream>
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6    ifstream arq(argv[1]);
7    char ch;
8
9    while (! arq.eof()) {
10     arq >> ch;
11     cout << ch;
12   }
13 }
```

No entanto, ao executar este programa, a saída obtida não será aquela desejada, mas algo como:

```
#include<fstream>#include<iostream>usingnamespacest...
```

O motivo é que o operador de leitura `>>` tem como comportamento padrão ignorar os chamados "espaços em branco": o espaço, o caráter de tabulação (`'\t'`) e o caráter de mudança de linha (`'\n'`). Características de formatação podem ser alteradas com os métodos de definição de formato de entrada e saída `setf` e `unsetf`.

Neste exemplo, o que se deseja é retirar *(unset)* a característica de ignorar espaços em branco *(white spaces)*, associado ao especificador `ios::skipws`. Assim, a inclusão de uma linha com o comando

```
arq.unsetf(ios::skipws);
```

antes do laço de leitura dos caracteres fará com que todos os espaços, tabulações e quebras de linhas sejam preservados.

O método `get` faz a leitura não-formatada dos dados da entrada. Na sua forma de uso sem argumentos, obtém o próximo caráter da entrada e retorna-o como um valor inteiro. No exemplo acima, ao substituir o interior do laço de leitura e apresentação dos caracteres por uma única linha com

```
cout << arq.get();
```

a saída obtida da execução seria algo como

```
351051109910811710010132601021151161141019710962l0...
```

onde 35 é o valor do código ASCII para o símbolo #, 105 para a letra `i`, 110 para a letra `n`, 99 para `c` e assim sucessivamente.

Para apresentar os valores inteiros como caracteres, o mecanismo de coerção *(cast)* deve ser utilizado. Há duas formas de especificar esse mecanismo:

1. A forma funcional, como em

```
cout << char(arq.get());
```

2. Com o operador de *cast*, como em

```
cout << static_cast<char>(arq.get());
```

Esta última forma, apesar de mais extensa, facilita a identificação no código dos pontos onde o mecanismo de coerção está sendo utilizado. Como este pode ser uma causa comum de falhas no código, essa propriedade é interessante e, portanto, esta forma de coerção deve ser utilizada em geral.

A segunda forma de utilizar o método `get` dá-se através da especificação de um argumento do tipo `char` para onde o caráter será lido, como em

```
while (! arq.eof()) {
  arq.get(ch);
  cout << ch;
}
```

Para ilustrar a terceira forma do método `get`, o programa exemplo será modificado para incluir um contador para registrar quantas vezes o método foi invocado. Considere a última versão do programa:

```
1  #include <fstream>
2  #include <iostream>
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6    ifstream arq(argv[1]);
7    char ch;
8    int inv = 0;
9
10   while (! arq.eof()) {
11     arq.get(ch);
12     ++inv;
13     cout << ch;
14   }
15   cout << "get invocado " << inv << " vezes." << endl;
16 }
```

Ao final da execução, tendo o próprio arquivo como argumento, a mensagem indica:

```
get invocado 279 vezes.
```

Na terceira forma de uso do método `get`, o objetivo é obter vários caracteres em uma única invocação. Para tanto, o programador deve especificar uma área para a leitura (um arranjo de caracteres) e quantos caracteres podem ser lidos de uma vez. Um terceiro argumento indica um terminador para a leitura. Tipicamente, para a leitura de linhas, o terminador é o caráter `'\n'`, que será implicitamente assumido pelo método se o terceiro argumento for omitido.

O método `getline` tem uma única forma, equivalente a esta última do método `get`. A diferença é que `getline` remove o terminador da entrada, enquanto que o método `get` preserva o terminador na entrada. Neste caso, invocações subseqüentes do método `get` retornariam o conteúdo da área de leitura vazio, pois o terminador seria lido de imediato; seria preciso consumi-lo explicitamente. A forma `getline` já faz isso. Assim, o exemplo modificado fica:

```
1  #include <fstream>
2  #include <iostream>
3  using namespace std;
4
5  int main(int argc, char* argv[]) {
6    ifstream arq(argv[1]);
7    const int tamanhoBuffer = 100;
8    char chbuf[tamanhoBuffer];
9    int inv = 0;
10
11   while (! arq.eof()) {
12     arq.getline(chbuf,tamanhoBuffer,'\n');
13     ++inv;
14     cout << chbuf << endl;
15   }
16
17   cout << "getline invocado " << inv
18        << " vezes." << endl;
19 }
```

A mensagem ao final indicaria:

```
getline invocado 20 vezes.
```

6.4 Geradores de analisadores léxicos

Embora seja possível implementar analisadores léxicos a partir da construção do autômato finito para a expressão regular e a aplicação do Algoritmo 6.1, pode-se imaginar que para linguagens mais complexas essa estratégia de implementação seria extremamente trabalhosa. Como essa complexidade é freqüente na programação de sistemas, diversas ferramentas de apoio a esse tipo de programação foram desenvolvidas.

Uma classe dessas ferramentas são os geradores de analisadores léxicos, que automatizam o processo de criação do autômato e o processo de reconhecimento de sentenças regulares a partir da especificação das expressões regulares correspondentes.

Uma das ferramentas mais tradicionais dessa classe é o programa `lex`, originalmente desenvolvido para o sistema operacional Unix. O objetivo de `lex` é gerar uma rotina para o *scanner* em C a partir de um arquivo de especificação contendo a especificação das expressões regulares e trechos de código C do usuário que serão executados quando sentenças daquelas expressões forem reconhecidas. Atualmente há diversas implementações de `lex` para diferentes sistemas, assim como ferramentas similares que trabalham com outras linguagens de programação que não C.

6.4.1 Especificação das sentenças regulares

O ponto de partida para a criação de um analisador léxico usando `lex` é criar o arquivo com a especificação das expressões regulares que descrevem os itens léxicos que são aceitos. Este arquivo é composto por até três seções: definições, regras e código do usuário. Essas seções são separadas pelos símbolos `%%`.

A seção mais importante é a seção de regras, onde são especificadas as expressões regulares válidas e as correspondentes ações do programa. Cada regra é expressa na forma de um par padrão-ação,

```
pattern   action
```

Para a descrição do padrão, `lex` define uma linguagem para descrição de expressões regulares. Esta linguagem mantém a notação para expressões regulares apresentada na Seção 4.3.1, ou seja, a presença de um caráter a indica a ocorrência daquele caráter; se R é uma expressão regular, $R*$ indica a ocorrência dessa expressão zero ou mais vezes; e se S também é uma expressão regular, então RS é a concatenação dessas expressões e $R|S$ indica a ocorrência da expressão R ou da expressão S. Além dessas construções, a linguagem oferece ainda as seguintes extensões:

`.`	qualquer caráter exceto `\n`
`[xyz]`	uma classe de caracteres, `'x'` ou `'y'` ou `'z'`
`[a-f]`	classe de caracteres, qualquer caráter entre `'a'` e `'f'`
`[^xyz]`	classe de caracteres negada, qualquer caráter *exceto* `'x'` ou `'y'` ou `'z'`
`R+`	uma ou mais ocorrências da expressão regular `R`
`R?`	0 ou uma ocorrência da expressão regular `R`
`R{4}`	exatamente quatro ocorrências da expressão regular `R`
`R{2,}`	pelo menos duas ocorrências da expressão regular `R`
`R{2,4}`	entre duas e quatro ocorrências da expressão regular `R`
`^R`	a expressão regular `R` ocorrendo apenas no início de uma linha

`R$`	a expressão regular `R` ocorrendo apenas no final de uma linha
`<<EOF>>`	fim de arquivo

Caso deseje-se usar um dos caracteres que têm significado especial nessa linguagem como um caráter da expressão, é possível usar a construção `\X`; por exemplo, `\.` permite especificar a ocorrência de um ponto na expressão regular. Se $X \in \{0, a, b, f, n, r, t, v\}$, então o caráter recebe a mesma interpretação associada às definições da linguagem C (Tabela 2.1). Outra forma de indicar que uma *string* deve ser interpretada literalmente é representá-la entre aspas na regra.

A ação associada a cada padrão na regra é um bloco de código C definido pelo usuário. Esse código será incorporado ao código do analisador léxico, sendo executado quando a seqüência de caracteres de entrada for reconhecida pelo padrão especificado. Como qualquer bloco em C, se apenas uma linha de código for especificada então as chaves de início e fim de bloco podem ser omitidas; caso contrário, devem obrigatoriamente estar presentes.

O exemplo abaixo de especificação de regras no padrão `lex` determina o reconhecimento de constantes numéricas inteiras, segundo o padrão da linguagem C:

```
1  %%
2  [1-9][0-9]*     printf("Dec");
3  0[0-7]*         printf("Oct");
4  0x[0-9A-Fa-f]+  printf("Hex");
```

A primeira linha do arquivo começa com o separador de seções, ou seja, a seção de definições, assim como a seção de código do usuário, é vazia.

O analisador léxico gerado por esse arquivo de especificação irá receber como entrada *strings* que eventualmente irão conter constantes inteiras. Se uma constante inteira for encontrada, ela será substituída na saída pela *string* correspondente especificada na ação, para cada um dos formatos de constantes reconhecidos. Assim, se a *string* for

```
abc 10 def 017 ghi 0xAF0
```

o analisador léxico gerará a *string*

```
abc Dec def Oct ghi Hex
```

Esse exemplo ilustra a utilização da **regra padrão** — quando nenhum padrão especificado combina com a seqüência de caracteres analisada, então esses caracteres são simplesmente ecoados para a saída.

A seção de definições permite criar representações simbólicas que podem ser posteriormente utilizadas nas seções de regras. Por exemplo, se nessa seção houver a definição

```
DIGIT    [0-9]
```

então o padrão da regra que reconhece constantes decimais poderia ter sido escrito na forma

```
[1-9]{DIGIT}*
```

A forma `{name}` é substituída no padrão pela expansão da definição `name`.

Outro tipo de especificação que pode estar presente na seção de definições são trechos de código C que devem ser incluídos ao início do arquivo. Esse código, especificado nessa seção entre os símbolos `%{` e `%}`, normalmente é utilizado para incluir diretrizes para o pré-processador C, como `#define` e `#include`.

6.4.2 Integração com código de aplicação

O resultado da aplicação do programa `lex`, tendo como entrada um arquivo de especificação de expressões regulares e respectivas ações, é a criação de um arquivo-fonte contendo o código C que implementa o correspondente analisador léxico. Este está associado à rotina de nome `yylex()`, que é invocada pela aplicação para fazer o reconhecimento dos itens léxicos na seqüência de caracteres da entrada.

A rotina `yylex()` não recebe nenhum argumento e pode retornar um valor inteiro, que no processo de análise léxica pode ser associado a um tipo de *token*. Essa rotina lê os caracteres de entrada de um arquivo especificado pela variável global `yyin` e envia os resultados de sua análise para o arquivo especificado pela variável global `yyout`; essas duas variáveis são ponteiros para `FILE`, para a manipulação de arquivos em C. Adicionalmente, a última *string* que foi reconhecida pelo analisador léxico é referenciada pela variável global `yytext`, do tipo ponteiro para caracteres.

A definição padrão das variáveis `yyin` e `yyout` associa-as respectivamente ao arquivo de entrada padrão (teclado) e ao arquivo de saída padrão (tela do monitor). Essa definição pode ser modificada pela especificação presente na seção de código do usuário do arquivo `lex`.

Se nenhum código for definido nessa seção, o código de aplicação utilizado é o fornecido na biblioteca de rotinas do `lex`, tipicamente algo da forma

```
int main() {
    yylex();
    return 0;
}
```

Para modificar as definições padronizadas, o código C que altera esse comportamento deve estar especificado nessa seção. Por exemplo, para que o analisador léxico que reconhece as constantes inteiras pudesse ter a opção de obter sua entrada de um arquivo especificado na linha de comando, o arquivo de especificação `lex` apresentado como o Algoritmo 6.6 poderia ter sido utilizado.

6.4.3 Geração da aplicação

Nesta seção ilustra-se a utilização de uma das implementações do programa `lex`, que é o aplicativo `flex`, disponível para diversas plataformas computacionais. A

Algoritmo 6.6 Arquivo de especificação `lex`.

```
1  DIGIT [0-9]
2  %%
3  [1-9]{DIGIT}*   printf("Dec");
4  0[0-7]*         printf("Oct");
5  0x[0-9A-Fa-f]+  printf("Hex");
6  <<EOF>>         return 0;
7  %%
8  int main(int argc, char *argv[]) {
9    FILE *f_in;
10
11   if (argc == 2) {
12     if (f_in = fopen(argv[1],"r"))
13       yyin = f_in;
14     else
15       perror(argv[0]);
16   }
17   else
18     yyin = stdin;
19
20   yylex();
21   return(0);
22 }
```

sintaxe dos comandos apresentadas correspondem à utilização do aplicativo com o sistema operacional Unix.

Considere como exemplo que a especificação `lex` apresentada no Algoritmo 6.6 foi escrita em um arquivo que recebeu o nome `unsint.l`, onde `.l` é uma extensão padrão para esse tipo de arquivo. Para gerar o analisador léxico, `flex` é invocado recebendo esse arquivo como entrada:

```
> flex  unsint.l
```

A execução desse comando gera um arquivo-fonte C de nome `lex.yy.c`, que implementa os procedimentos do analisador léxico. Para gerar o código executável, este programa deve ser compilado e ligado com a biblioteca `libfl`, que contém os procedimentos internos padrões de `flex`

```
> gcc -o aliss lex.yy.c -lfl
```

(Bibliotecas são descritas no Capítulo 11.)

O arquivo executável `aliss` conterá o analisador léxico para inteiros sem sinal. Se invocado sem argumentos, `aliss` irá aguardar a entrada do teclado para proceder à análise das *strings*; o término da execução será determinado pela entrada do caráter

`control-D`. Se for invocado com um argumento na linha de comando, `aliss` irá interpretar esse argumento como o nome de um arquivo que conterá o texto que deve ser analisado, processando-o do início ao fim.