DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS

Programação de Sistemas:
Uma Introdução

Ivan Luiz Marques Ricarte

`http://www.dca.fee.unicamp.br/~ricarte/`

2001

Impressão em 4 de março de 2002

Sumário

Prefácio

O material aqui apresentado foi desenvolvido a partir de uma necessidade real constatada nas disciplinas EA876 (Introdução a Software de Sistema) e EA877 (Mini e microcomputadores: software) da Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas (FEEC/UNICAMP). Seu objetivo é oferecer um material de apoio ao estudo do aluno que apresente de forma unificada os diversos tópicos apresentados nesse curso.

Essas disciplinas, oferecidas aos alunos de graduação dos cursos de Engenharia de Computação e de Engenharia Elétrica, têm por meta oferecer uma visão geral sobre o funcionamento e utilização do *software* que permite que usuários executem seus programas em computadores. Para os alunos de Engenharia de Computação, esse é um ponto de partida para temas que serão aprofundados em diversas outras disciplinas. Para os alunos de Engenharia Elétrica, pode significar o único contato com esse tema, essencial para o uso efetivo de computadores ao longo de sua vida profissional.

Uma das dificuldades no desenvolvimento desse material está em sua amplitude. Para oferecer essa visão geral do software de sistema, agrupou-se em uma disciplina temas tão amplos como compiladores, carregadores, ligadores e sistemas operacionais — cada um deles suficientemente complexo para ser o escopo de outras disciplinas. Adequar o grau de profundidade da apresentação dos tópicos ao tempo e aos objetivos do curso, sem tornar essa apresentação extremamente superficial, tem sido uma tarefa que vem requerendo a constante revisão desse material. Nessa tarefa, a realimentação dos usuários do material — alunos e instrutores das disciplinas citadas — tem sido um importante auxílio.

Em 2001, tomou-se a opção de retirar deste material a parte do texto sobre sistemas operacionais, que será coberto no oferecimento da disciplina através de outras referências. Houve também uma re-organização de capítulos e seções com o objetivo de tornar a seqüência do curso mais fluida e natural. A inclusão de sugestões de exercícios no texto, ao invés de disponibilizá-los através de listas de exercícios distribuídas em classes, busca atender uma solicitação daqueles que usaram versões anteriores do material.

Para o desenvolvimento deste texto, algumas opções foram norteadas pela prática corrente na área ou em outras disciplinas relacionadas da FEEC/UNICAMP. Nesta categoria inclui-se a opção pelos processadores da família Motorola 68K para os exemplos que utilizam linguagem *assembly*. Na primeira categoria, o destaque maior é o uso da linguagem C para os exemplos envolvendo linguagens de alto nível. C é a linguagem de programação de sistemas por excelência, devendo ocupar esse papel por um longo tempo ainda. Com o objetivo de não tornar o corpo principal desse texto extremamente carregado ou monótono para aqueles que detêm o conhecimento nessas duas linguagens, são oferecidos apêndices que as descrevem num grau de profundidade não mais que necessário para a compreensão dos exemplos.

Uma vez mais destacamos que este é um material em contínuo desenvolvimento, sendo a realimentação dos usuários uma componente fundamental nesse processo. Comentários e sugestões são bem vindos, podendo ser encaminhados por *e-mail* a `ricarte@dca.fee.unicamp.br`

Campinas, março de 2001

Ivan Luiz Marques Ricarte

Capítulo 1

Introdução

Nas últimas décadas, o computador deixou de ser uma ferramenta cara e exclusiva dos grandes centros de pesquisas e passou a ser presença constante no cotidiano de todos. Apesar dessa "quase onipresença," para muitos ainda há uma aura de mistério no que se refere ao funcionamento e à capacidade funcional de computadores.

Um computador tem essencialmente duas componentes, *hardware* e *software*. *Hardware* é o termo coletivo que representa todo o conjunto de circuitos que permite o funcionamento do computador. *Software* é o termo coletivo expressando o conjunto de programas que, juntamente com o *hardware*, permite a operação de computadores. Para a maior parte dos sistemas computacionais modernos, o *hardware* é genérico, sendo que o *software* é que determina quais são as funcionalidades do sistema oferecidas aos usuários finais.

1.1 Programas e processos

Um computador nada mais faz do que executar **programas**. Um programa é simplesmente uma seqüência de instruções definida por um programador. Assim, um programa pode ser comparado a uma receita indicando os passos elementares que devem ser seguidos para desempenhar uma tarefa. Cada instrução é executada no computador por seu principal componente, o processador ou CPU (de unidade central de processamento).

Cada processador entende uma linguagem própria, que reflete as instruções básicas que ele pode executar. O conjunto dessas instruções constitui a **linguagem de máquina** do processador. Cada instrução da linguagem de máquina é representada por uma seqüência de bits distinta, que deve ser decodificada pela unidade de controle da CPU para determinar que ações devem ser desempenhadas para a execução da instrução.

Claramente, seria extremamente desconfortável se programadores tivessem que desenvolver cada um de seus programas diretamente em linguagem de máquina. Programadores não trabalham diretamente com a representação binária das instruções de um processador. Existe uma descrição simbólica para as instruções do processador — a linguagem *assembly* do processador — que pode ser facilmente mapeada para sua linguagem de máquina.

No entanto, mesmo *assembly* é raramente utilizada pela maior parte dos programadores, que trabalha com **linguagens de programação de alto nível**. Em linguagens de alto nível, as instruções são expressas usando palavras, ou termos, que se aproximam daquelas usadas na linguagem humana, de forma a facilitar para os programadores a expressão e compreensão das tarefas que o programa deve executar. Várias linguagens de alto nível estão disponíveis para diversas máquinas distintas. Essa independência de um processador específico é uma outra vantagem no desenvolvimento de programas em linguagens de alto nível em relação ao uso de *assembly*. Em geral, cada linguagem de alto nível teve uma motivação para ser desenvolvida. BASIC foi desenvolvida para ensinar princípios de programação; Pascal, para o ensino de programação estruturada;

FORTRAN, para aplicações em computação científica; lisp e Prolog, para aplicações em Inteligência Artificial; Java, para o desenvolvimento de *software* embarcado e distribuído; e a linguagem C, para a programação de sistemas.

O fato de uma linguagem ter sido desenvolvida com uma aplicação em mente não significa que ela não seja adequada para outras aplicações. A linguagem C, juntamente com sua "sucessora" C++, é utilizada para um universo muito amplo de aplicações. Um dos atrativos da linguagem C é sua flexibilidade: um programador C tem à sua disposição comandos que permitem desenvolver programas com características de alto nível e ao mesmo tempo trabalhar em um nível muito próximo da arquitetura da máquina, de forma a explorar os recursos disponíveis de forma mais eficiente. Por este motivo, o número de aplicações desenvolvidas em C e C++ é grande e continua a crescer.

1.2 Desenvolvimento de *software*

Não há dúvidas hoje em dia quanto à importância do *software* no desenvolvimento dos mais diversos sistemas. Com esta evolução do papel do *software* em sistemas computacionais, veio também uma maior complexidade de programas e uma maior preocupação em desenvolver programas que pudessem ser facilmente entendidos e modificados (se necessário), não apenas pelo autor do programa mas também por outros programadores. A disciplina que estuda o desenvolvimento de programas com qualidade é conhecida como **Engenharia de *Software***.

A Engenharia de *Software* estabelece alguns princípios de desenvolvimento que independem da linguagem de programação adotada. Estes princípios são utilizados nas três grandes fases da vida de um programa, que são a especificação, o desenvolvimento e a manutenção de programas. A especificação inicia-se com o levantamento de requisitos (ou seja, o que deve ser feito pelo programa) e inclui a análise do sistema que deve ser desenvolvido. Na fase de desenvolvimento realiza-se o projeto do sistema, com descrições das principais estruturas de dados e algoritmos, sua codificação, com a implementação do projeto em termos de uma linguagem de programação, e testes dos programas desenvolvidos. Na fase de manutenção são realizadas modificações decorrentes da correção de erros e atualizações do programa.

Uma vez estabelecida a funcionalidade do programa que se deseja implementar na fase de definição, a fase de desenvolvimento propriamente dita pode ser iniciada. A fase de desenvolvimento costuma tomar a maior parte do ciclo de vida de criação de um programa. Nesta fase são tomadas decisões que podem afetar sensivelmente o custo e a qualidade do *software* desenvolvido. Esta fase pode ser dividida em três etapas principais, que são projeto, codificação e teste.

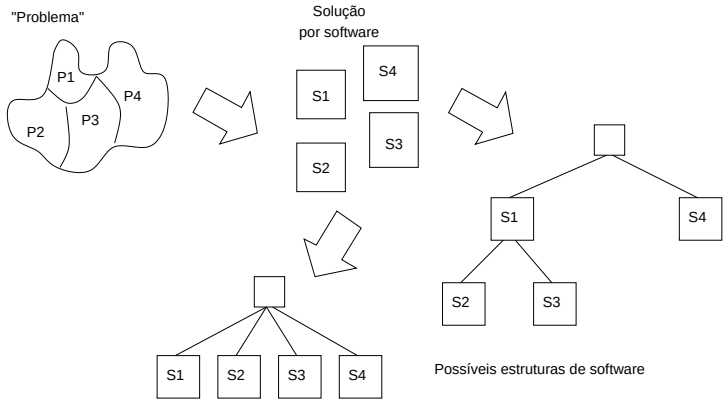
1.2.1 Projeto

O projeto de *software* pode ser subdividido em dois grandes passos, projeto preliminar e projeto detalhado. O projeto preliminar preocupa-se em transformar os requisitos especificados na fase de análise em arquiteturas de dados e de *software*. O projeto detalhado refina estas representações de arquitetura em estruturas de dados detalhadas e em representações algorítmicas do programa.

Uma das estratégias de projeto mais utilizadas é o desenvolvimento *top-down*. Neste tipo de desenvolvimento, trabalha-se com o conceito de refinamento de descrições do programa em distintos níveis de **abstração**. O conceito de abstração está relacionado com **esconder** informação sobre os detalhes. No nível mais alto de abstração, praticamente nenhuma informação é detalhada sobre como uma dada tarefa será implementada — simplesmente descreve-se qual é a tarefa. Em etapas sucessivas de refinamento, o projetista de *software* vai elaborando sobre a descrição da etapa anterior, fornecendo cada vez mais detalhes sobre como realizar a tarefa.

O desenvolvimento *top-down* estabelece o processo de passagem de um problema a uma estrutura de *software* para sua solução (Fig. 1.1), resultando em uma estrutura que representa a organização dos distintos componentes (ou **módulos**) do programa. Observe que a solução obtida pode não ser única: dependendo de como o projeto é desenvolvido e das decisões tomadas, distintas estruturas podem resultar.

Figura 1.1 Processo de evolução do *software*.



Outro aspecto tão importante quanto a estrutura de *software* é a **estrutura de dados**, que é uma representação do relacionamento lógico entre os elementos de dados individuais. Estruturas de dados são vistas em maiores detalhes no Capítulo 2.

Uma vez estabelecidas as estruturas de *software* e de dados do programa, o detalhamento do projeto pode prosseguir com o projeto procedimental, onde são definidos os detalhes dos algoritmos que serão utilizados para implementar o programa. Um **algoritmo** é uma solução passo-a-passo para a resolução do problema especificado que sempre atinge um ponto final. Em princípio, algoritmos poderiam ser descritos usando linguagem natural (português, por exemplo). Entretanto, o uso da linguagem natural para descrever algoritmos geralmente leva a ambigüidades, de modo que se utilizam normalmente linguagens mais restritas para a descrição dos passos de um algoritmo. Embora não haja uma representação única ou universalmente aceita para a representação de algoritmos, dois dos mecanismos de representação mais utilizados são o fluxograma e a descrição em pseudo-linguagens.

Um **fluxograma** é uma representação gráfica do fluxo de controle de um algoritmo, denotado por setas indicando a seqüência de tarefas (representadas por retângulos) e pontos de tomada de decisão (representados por losangos). A **descrição em pseudo-linguagem** combina descrições em linguagem natural com as construções de controle de execução usualmente presentes em linguagens de programação. As principais construções associadas a essas linguagens são descritas na seqüência.

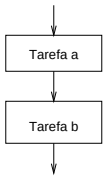
1.2.2 Programação estruturada

A programação estruturada estabelece uma disciplina de desenvolvimento de algoritmos que facilita a compreensão de programas através do número restrito de mecanismos de controle da execução de programas. Qualquer algoritmo, independentemente da área de aplicação, de sua complexidade e da linguagem de programação na qual será codificado, pode ser descrito através destes mecanismos básicos.

O princípio básico de programação estruturada é que um programa é composto por blocos elementares de código que se interligam através de três mecanismos básicos, que são **seqüência**, **seleção** e **iteração**. Cada uma destas construções tem um ponto de início (o topo do bloco) e um ponto de término (o fim do bloco) de execução.

Seqüência implementa os passos de processamento necessários para descrever qualquer programa. Por exemplo, um segmento de programa da forma "faça primeiro a *Tarefa a* e depois a *Tarefa b*" seria representado por uma seqüência de dois retângulos (Fig. 1.2a). A mesma contrução em pseudo-linguagem seria denotada pela expressão das duas tarefas, uma após a outra (Fig. 1.2b).

Figura 1.2 Construção estruturada: seqüência.



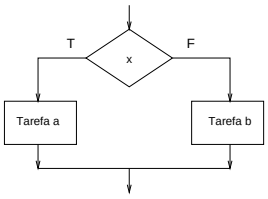
(a) fluxograma

```
SEQUENCIA()
1  Tarefa a
2  Tarefa b
```

(b) pseudo-linguagem

Seleção especifica a possibilidade de selecionar o fluxo de execução do processamento baseado em ocorrências lógicas. Há duas formas básicas de condição. A primeira forma é a construção `IF`, que permite representar fluxos da forma "se a condição lógica *x* for verdadeira, faça a *Tarefa a*; senão (isto é, se a condição *x* for falsa), faça a *Tarefa b*." Na representação em fluxograma (Figura 1.3a), as duas setas que saem do losango de condição recebem rótulos *T* e *F* para indicar o fluxo de execução quando a condição especificada é verdadeira ou falsa, respectivamente. O retângulo sob a seta rotulada *T* normalmente é denominado o **bloco then** da construção, enquanto que o outro retângulo é denominado **bloco else**.

Figura 1.3 Construção estruturada: seleção IF.



(a) fluxograma

```
SELECAOIF()
1  if x
2     then Tarefa a
3     else Tarefa b
```

(b) pseudo-linguagem

A outra forma de seleção estende o número de condições que podem ser avaliadas para definir o fluxo de execução. Esta construção, `SWITCH` (Fig. 1.4), permite representar fluxos da forma "se a variável *y* tem o valor *1*, faça a *Tarefa a*; se *y* tem o valor *2*, faça a *Tarefa b*; se tem o valor *0*, faça a *Tarefa c*; para qualquer outro valor, faça *Tarefa d*."

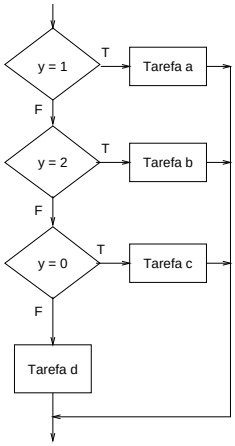
Observe que a construção `SWITCH` não é essencial, uma vez que ela pode ser representada em termos da seleção com `IF`, como em

```
SELECAOMULTIPLAIF()
1  if y = 1
2     then Tarefa a
3     else if y = 2
4             then Tarefa b
5             else if y = 0
6                     then Tarefa c
7                     else Tarefa d
```

Entretanto, a utilização de estruturas `SWITCH` simplifica a expressão de situações que ocorrem frequen-

temente em programas — por exemplo, selecionar ações dependendo de uma opção escolhida em um menu — sem ter que recorrer ao aninhamento excessivo de condições da forma `IF`. No entanto, essa condição está restrita a condições lógicas envolvendo exclusivamente testes de igualdade.

Figura 1.4 Construção estruturada: seleção SWITCH.



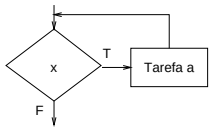
(a) fluxograma

```
SELECAOSWITCH()
1  switch y
2     case 1 :
3          Tarefa a
4     case 2 :
5          Tarefa b
6     case 0 :
7          Tarefa c
8     case default :
9          Tarefa d
```

(b) pseudo-linguagem

Iteração permite a execução repetitiva de segmentos do programa. Na forma básica de repetição, `WHILE` (Fig. 1.5), uma condição lógica é verificada. Caso seja verdadeira, o bloco de tarefas associado ao comando é executado. A condição é então reavaliada; enquanto for verdadeira, a tarefa é repetidamente executada.

Figura 1.5 Construção estruturada: repetição WHILE.



(a) fluxograma

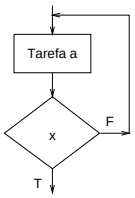
```
ITERACAOWHILE()
1  while x
2  do Tarefa a
```

(b) pseudo-linguagem

Uma variante dessa construção é apresentada na Fig. 1.6, onde inicialmente a tarefa é executada e apenas então a condição de repetição é avaliada; quando a condição torna-se verdadeira, a iteração é encerrada.

Tipicamente, a estratégia de desenvolvimento *top-down* é utilizada na descrição algorítmica de procedimentos. Neste caso, um retângulo ou uma linha de pseudo-código pode descrever uma tarefa tão complexa quanto necessário, sendo que esta tarefa pode ser posteriormente descrita em termos de outro(s) fluxograma(s) ou pseudo-código(s). Em geral, são aplicados tantos refinamentos quantos forem necessários até atingir um ponto em que uma tarefa possa ser facilmente descrita em termos das construções suportadas pela linguagem de codificação.

Figura 1.6 Construção estruturada: repetição REPEAT UNTIL.



(a) fluxograma

```
ITERACAOREPEAT()
1  repeat
2          Tarefa a
3    until x
```

(b) pseudo-linguagem

1.2.3 Codificação

A etapa de codificação traduz a representação do projeto detalhado em termos de uma linguagem de programação. Normalmente são utilizadas linguagens de alto nível, que podem então ser automaticamente traduzidas para a linguagem de máquina pelo processo de compilação. Neste texto, a linguagem C será utilizada nos exemplos e atividades práticas desenvolvidas.

Em uma linguagem de programação procedimental, como C, as construções da programação estruturadas podem ser expressas de forma quase imediata através dos comandos da própria linguagem. Por exemplo, a construção da Figura 1.2 é representada pela simples seqüência de expressões. Associando as tarefas do exemplo a funções, o trecho de código correspondente em C será:

```
/* uma seqüência de comandos */
tarefaA();  // executa esta função
tarefaB();  // depois de terminar A, executa B
```

Os pares de caracteres `/*` e `*/` delimitam comentários em C, sendo que esses comentários podem se estender por diversas linhas. A outra forma de comentário aceita pela linguagem é a barra dupla, `//`, que inicia um comentário que termina com o final da linha.

O uso de comentários deve ser usado como forma de documentação interna de um programa. O excesso de comentários não é recomendado, pois pode até prejudicar a leitura de um programa. Entretanto, há comentários que devem estar presentes em qualquer programa, tais como

Comentários de prólogo, que aparecem no início de cada módulo ou arquivo-fonte. Devem indicar a finalidade do módulo, uma história de desenvolvimento (autor e data de criação e modificações), e uma descrição das variáveis globais (se houver);

Comentários de procedimento, que aparecem antes da definição de cada função indicando seu propósito, uma descrição dos argumentos e valores de retorno;

Comentários descritivos, que descrevem blocos de código.

Comentários devem ser facilmente diferenciáveis de código, seja através do uso de linhas em branco, seja através de tabulações. É importante que comentários sejam corretos e coerentes com o código, uma vez que um comentário errôneo pode dificultar mais ainda o entendimento de um programa do que se não houvesse comentário nenhum.

Condições na forma IF são também diretamente suportadas em C. Se a condição *x* for mapeada para uma função que retorna um valor que será interpretado como verdadeiro ou falso, a construção da Figura 1.3 seria mapeada para o seguinte segmento de código C:

```
if ( x() ) {
   tarefaA();
}
else {
   tarefaB();
}
```

Similarmente, a construção da seleção por SWITCH (Figura 1.4) é suportada pela linguagem C, que oferece um comando para o qual as variáveis sendo testadas podem ser do tipo `char` ou `int`:

```
switch(y) {
case 1:
   tarefaA();
   break;
case 2:
   tarefaB();
   break;
case 0:
   tarefaC();
   break;
default:
   tarefaD();
}
```

Em C, há três comandos de controle de execução para indicar repetição. O primeiro, `while`, é diretamente correspondente à forma expressa na Figura 1.5:

```
while ( x() ) {
   tarefaA();
}
```

A segunda forma de repetição executa o bloco especificado pelo menos uma vez, repetindo-o se a condição testada ao final for verdadeira (ao contrário da construção da Figura 1.6):

```
do {
   tarefaA();
} while ( x() );
```

Finalmente, o comando `for` combina a inicialização, o teste e a atualização da variável (ou variáveis) de condição em uma única expressão. Por exemplo,usando uma variável inteira `i` para executar uma tarefa 10 vezes:

```
for (i = 0 ; i < 10 ; i = i+1) {
  tarefaA();
}
```

Essa construção equivale a

```
i = 0;
while (i < 10) {
  tarefaA();
  i = i+1;
}
```

A tradução de uma especificação de um programa para uma linguagem de programação pode resultar em programas incompreensíveis se não houver um cuidado na preservação da informação presente. As linhas gerais que estão a seguir buscam estabelecer uma disciplina de codificação que, se seguida, facilita o entendimento e manutenção de programas.

Código deve ser acima de tudo claro. Os compiladores modernos fazem um ótimo trabalho de otimização de código, de forma que não há necessidade do programador ficar se preocupando em usar pequenos "truques" para economizar algumas instruções no código de máquina — provavelmente o compilador já faria isto para o programador[1]. A preocupação com a otimização de código só deve existir após a detecção da necessidade real desta otimização, e ela deve se restringir ao segmento do programa onde o problema de desempenho foi localizado.

Na maior parte das linguagens de programação modernas, a posição exata (em qual coluna) a instrução está localizada é irrelevante. Assim, na codificação das construções estruturadas, diferentes níveis de tabulação devem ser usados para indicar blocos de distintos níveis. Evite sempre que possível o uso de condições de teste complicadas e o aninhamento muito profundo de condições de teste. Use parênteses para deixar claro como expressões lógicas e aritméticas serão computadas.

Com relação a instruções de entrada e saída, todos os dados de entrada devem ser validados. O formato de entrada deve ser tão simples quanto possível e, no caso de entrada interativa, o usuário deve receber uma indicação de que o programa está aguardando dados. Da mesma forma, a apresentação de resultados deve ser clara, com indicação sobre o que está sendo apresentado. O valor de retorno de qualquer rotina que possa retornar um erro deve sempre ser testado, tomando-se as ações que forem necessárias para minimizar o efeito do erro.

Nomes de identificadores (variáveis, procedimentos) devem denotar claramente o significado do identificador. Linguagens de programação modernas raramente limitam o número de caracteres que um identificador pode ter, de forma que não faz sentido restringir este tamanho. Compare as duas linhas de código a seguir e observe como uma boa escolha no nome de variáveis nomes claros pode facilitar a compreensão de um programa:

```
d = v * t;
distancia = velocidade * tempo;
```

1.3 *Software* de sistema

Um programa, desde sua criação em uma linguagem de alto nível, é manipulado por um grande conjunto de outros programas que traduzem seu código para linguagem de máquina e controlam sua execução no computador. Este conjunto de programas recebe a denominação genérica de ***software* de sistema** e é o objeto de estudo deste texto.

No desenvolvimento de programas, o *software* de sistema é extensamente utilizado, com as várias etapas inter-relacionadas para a criação e execução de um programa (Fig. 1.7). Tipicamente, esse relacionamento dá-se de forma transparente para o programador.

Programas são usualmente descritos em linguagens de alto nível. O **compilador** é o programa do sistema que traduz um programa descrito através de uma linguagem de alto nível específica para um programa equivalente em linguagem *assembly*. Esse processo de tradução é denominado compilação.

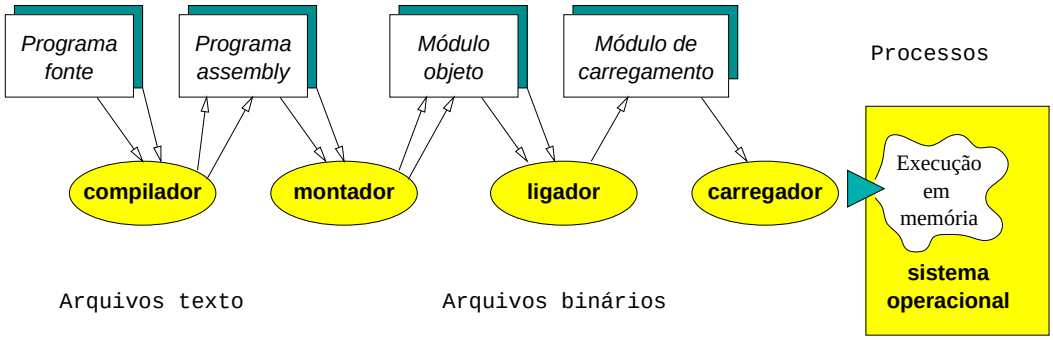
O **montador** (*assembler*) é o programa do sistema responsável por traduzir um programa *assembly* para o código de máquina. Esse processo de tradução de um programa-fonte *assembly* para um programa em código de máquina é denominado montagem; o resultado da montagem é um **módulo objeto** contendo pelo menos o código binário que será posteriormente executado.

[1] Veja a Seção 3.6.

Programas complexos raramente são descritos através de um único arquivo-fonte, mas sim organizados em módulos objetos interrelacionados. Tais módulos podem agregar funcionalidades da aplicação sendo desenvolvida ou recursos comuns do sistema que devem ser integrados à aplicação. O programa do sistema **ligador** é o responsável por interligar os diversos módulos de um programa para gerar o programa que será posteriormente carregado para a memória. Essa etapa de preparação de um programa para sua execução é denominada ligação.

Para que um programa possa ser executado, seu código de máquina deve estar presente na memória. O **carregador** é o programa do sistema responsável por transferir o código de máquina de um módulo objeto para a memória e encaminhar o início de sua execução. O processo de transferir o conteúdo de um módulo objeto para a memória principal é denominado carregamento. A execução de qualquer programa deve ser precedida por seu carregamento.

Figura 1.7 Etapas para execução de programa.



A execução de cada programa se dá sob o controle do **sistema operacional**. A um programa em execução dá-se o nome de **processo**. Além das instruções do programa, um processo necessita de todo um conjunto de informações adicionais para o controle de sua execução. O estado corrente dessas informações associadas a cada programa em execução constitui o **estado do processo**. O sistema operacional é o responsável por gerenciar cada processo no computador, estabelecendo como será realizada sua execução. Ele também atua como um programa supervisor que estabelece uma camada de controle entre o *hardware* do computador e as aplicações de usuários. Uma de suas funções é estabelecer uma interface de *software* uniforme entre o computador, outros programas do sistema e programas de aplicação de usuários. Outra função fundamental de um sistema operacional é gerenciar os recursos de um computador de forma a promover sua eficiente utilização. Exemplos de sistemas operacionais são MS-DOS, Windows NT, OS/2, Linux e Solaris — estes dois implementações do sistema operacional Unix.

Capítulo 2

Estruturas de dados

Em diversos contextos, disciplinas associadas à programação recebem a denominação de "processamento de dados". Esta denominação não é gratuita — de fato, embora seja possível criar procedimentos que não manipulem nenhum dado, tais procedimentos seriam de pouco valor prático.

Uma vez que procedimentos são, efetivamente, processadores de dados, a eficiência de um procedimento está muito associada à forma como seus dados são organizados. **Estrutura de dados** é o ramo da computação que estuda os diversos mecanismos de organização de dados para atender aos diferentes requisitos de processamento.

As estruturas de dados definem a organização, métodos de acesso e opções de processamento para a informação manipulada pelo programa. A definição da organização interna de uma estrutura de dados é tarefa do projetista da estrutura, que define também qual a API[1] para a estrutura, ou seja, qual o conjunto de procedimentos que podem ser usados para manipular os dados na estrutura. É esta API que determina a visão funcional da estrutura de dados, que é a única informação relevante para um programador que vá utilizar uma estrutura de daos pré-definida.

Neste capítulo são apresentadas algumas estruturas de dados, com ênfase naquelas que são utilizadas posteriormente no decorrer do texto. Assim, algumas estruturas de importância para outros tipos de aplicações — como a representação de matrizes esparsas, fundamental para a área de computação científica — não estão descritas aqui.

2.1 Tipos de dados

A organização de um estrutura de dados é construída a partir de alguns blocos básicos, presentes na maior parte das linguagens de programação de alto nível. A partir desses blocos elementares (os tipos escalares) e de operadores das linguagens de programação, construções mais complexas podem ser definidas, tais como arranjos e estruturas.

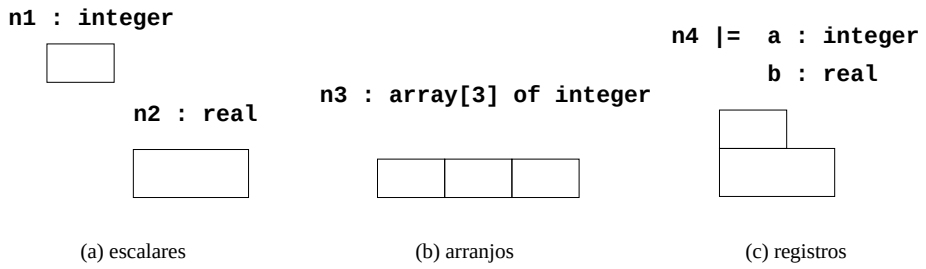
2.1.1 Escalares

A mais simples estrutura para representar um dado é um **escalar** (Fig. 2.1a), que representa um elemento de informação que pode ser acessado através de um identificador.

A representação elementar de um dado se dá através de um escalar. Em *assembly*, tipicamente todos os valores escalares são representados na forma de *bytes* ou *words*. Diferentes linguagens de alto nível determinam tipos diferenciados de escalares que são compreendidos pela linguagem.

[1] *Application Programming Interface.*

Figura 2.1 Estruturas elementares de dados.



(a) escalares (b) arranjos (c) registros

Internamente, todos os dados são representados no computador como seqüências de bits[2]. Esta é a forma mais conveniente para manipular os dados através de circuitos digitais, que podem diferenciar apenas entre dois estados (*on* ou *off*, *verdade* ou *falso*, 0 ou 1). Uma seqüência de N bits pode representar uma faixa com 2^N valores distintos.

O formato de representação interna (ou seja, como uma seqüência de bits é traduzida para um valor) pode variar de computador para computador, embora haja um esforço crescente para uniformizar a representação de tipos básicos. Assim, um caráter usualmente ocupa um byte com conteúdo definido por algum padrão de codificação (EBCDIC, ASCII, UniCode, ISO); um número inteiro tem uma representação binária inteira (sinal e magnitude, complemento de um ou complemento de dois); e um valor real é usualmente representado tipicamente no formato sinal, mantissa e expoente.

A linguagem C suporta escalares através da declaração de variáveis em um dos seus tipos de dados básicos[3], que são:

`char`	caráter
`int`	inteiro
`float`	real
`double`	real de precisão dupla

Observe que, ao contrário de outras linguagens de alto nível, em C não há um tipo booleano para construir variáveis que assumem o valor verdadeiro ou falso. Em C, em uma expressão booleana qualquer valor inteiro diferente de 0 é interpretado como verdadeiro, enquanto que o valor 0 é interpretado como falso.

Uma variável do tipo `char` ocupa um byte com o valor binário da representação ASCII de um caráter. O formato ASCII básico (Tabela 2.1) permite representar 128 caracteres distintos (valores entre 0 e 127), entre os quais estão diversos caracteres de controle (tais com ESC, associado à tecla de *escape*, e CR, associado ao *carriage return*) e de pontuação. Outros formatos além de ASCII inclui ISO8859, que padroniza a representação associada à faixa de valores entre 128 e 255 organizando-a em diversos subconjuntos, dos quais o ISO8859-1 (Latin-1) é o mais utilizado. UniCode integra várias famílias de caracteres em uma representação unificada (de um ou dois bytes), sendo que a base dessa proposta engloba as codificações ASCII e ISO8859-1.

Caracteres ASCII são denotados em C entre aspas simples, tais como `'A'`. Cada caráter ASCII corresponde também a uma representação binária usada internamente — por exemplo, o caráter ASCII A equivale a uma seqüência de bits que corresponde ao valor hexadecimal 41_H ou decimal 65.

Além dos caracteres alfanuméricos e de pontuação, que podem ser representados em uma função diretamente pelo símbolo correspondente entre aspas, C também define representações para caracteres especiais de controle do código ASCII através de seqüências de escape iniciados pelo símbolo \ (contrabarra). As principais seqüências são apresentadas na Tabela 2.2.

[2] Um *bit* é um dígito binário *(binary digit)*, usualmente representado pelos símbolos 0 e 1.
[3] Variáveis usadas em uma função C devem obrigatoriamente ser declaradas.

Tabela 2.1 Codificação ASCII.

Cód	0x-0	0x-1	0x-2	0x-3	0x-4	0x-5	0x-6	0x-7	0x-8	0x-9	0x-A	0x-B	0x-C	0x-D	0x-E	0x-F
0x0-	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
0x1-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
0x2-	[esp]	!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
0x3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x6-	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x7-	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Tabela 2.2 Caracteres representados por seqüências de escape em C.

`\n`	nova linha
`\t`	tabulação
`\b`	retrocesso
`\r`	retorno de carro
`\f`	alimentação de formulário
`\\`	contrabarra
`\'`	apóstrofo
`\"`	aspas
`\0`	o caráter NUL
`\xxx`	qualquer padrão de bits `xxx` em octal

Seqüências de caracteres ocorrem tão frequentemente em computação que recebem um nome específico, *(string)*. Em C, *strings* são representadas por arranjos de caracteres, com o fim da *string* sendo marcado pelo caráter NUL (`'\0'`).

O tipo `int` representa um valor inteiro que pode ser positivo ou negativo. O número de bytes ocupado por este tipo (e conseqüentemente a faixa de valores que podem ser representados) refletem o tamanho “natural” do inteiro na máquina onde o programa será executado. Usualmente, quatro bytes (32 bits) são reservados para o tipo `int` nos computadores atuais, permitindo representar valores na faixa entre -2^{31} a $+2^{31}-1$, ou $-2\,147\,483\,648$ a $2\,147\,483\,647$.

Para declarar uma variável escalar de nome `i` do tipo inteiro em um programa C, a seguinte expressão seria utilizada:

```
int i;
```

Essa declaração reserva na memória um espaço para a variável `i`, suficiente para armazenar a representação binária em complemento de dois do valor associado à variável, que inicialmente é indefinido. Caso se desejasse ter um valor inicial definido para a variável, a declaração seria acompanhada da inicialização, como em

```
int i = 0;
```

Variáveis inteiras podem ser qualificadas na sua declaração como `short` ou `long` e `unsigned`. Um tipo `unsigned int` indica que a variável apenas armazenará valores positivos; o padrão é que variáveis inteiras utilizem a representação em complemento de dois, com valores positivos e negativos (Apêndice A).

Os modificadores `short` e `long` modificam o espaço reservado para o armazenamento da variável. Um tipo `short int` indica que (caso seja possível) o compilador deverá usar um número menor de bytes para representar o valor numérico — usualmente, dois bytes são alocados para este tipo. Uma variável do tipo `long int` indica que a representação mais longa de um inteiro deve ser utilizada, sendo que usualmente quatro bytes são reservados para variáveis deste tipo.

Estas dimensões de variáveis denotam apenas uma situação usual definida por boa parte dos compiladores, sendo que não há nenhuma garantia quanto a isto. A única coisa que se pode afirmar com relação à dimensão de inteiros em C é que uma variável do tipo `short int` não terá um número maior de bits em sua representação do que uma variável do tipo `long int`[4].

Em C, números inteiros podem ter representação decimal, octal ou hexadecimal. Números com representação decimal são denotados por qualquer seqüência de algarismos entre `0` e `9` que inicie com um algarismo diferente de `0` — 10, 127, 512, etc. Números em octal são seqüências de algarismos entre `0` e `7` iniciadas por `0` — 012 (decimal 10), 077 (decimal 63), etc. Números em hexadecimal são seqüências de algarismos entre `0` e `F` iniciadas com o prefixo `0x` — `0xF` (decimal 15), `0x1A` (decimal 26). As representações octal e hexadecimal são atrativas como formas compactas de representação de valores binários — cada algarismo da representação octal pode ser diretamente expandido para uma seqüência de três bits, e da representação hexadecimal para seqüências de quatro bits. Assim, tanto `127` quanto `0177` quanto `0x7F` correspondem a uma mesma seqüência de bits, 01111111.

Os tipos `float` e `double` representam valores em *ponto flutuante*, limitados apenas pela precisão da máquina que executa o programa. O tipo `float` oferece seis dígitos de precisão enquanto que `double` suporta o dobro da precisão de um `float`.

Valores com representação em ponto flutuante (reais) são representados em C através do uso do ponto decimal, como em `1.5` para representar o valor um e meio. A notação exponencial também pode ser usada, como em `1.2345e-6` ou em `0.12E3`.

2.1.2 Ponteiros

Em C, é possível ter acesso não apenas ao valor de uma variável mas também ao seu endereço. O operador unário `&`, aplicado a uma variável, retorna o endereço de memória associado à variável. A manipulação de endereços dá-se através da utilização de **ponteiros**. Ponteiros constituem um dos recursos mais utilizados na programação C. Eles fornecem um mecanismo poderoso, flexível e eficiente de acesso a variáveis. Há computações que só podem ser expressas através do uso de ponteiros.

Para definir que uma variável vai guardar um endereço, o operador unário `*` é utilizado na declaração, como em

```
/* define um ponteiro para um inteiro */
int *ap;
/* algumas variaveis inteiras */
int x, y;
```

Neste exemplo, `ap` é uma variável do tipo *ponteiro para inteiro*, ou seja, ela irá receber um endereço de uma variável inteira. Para se obter o endereço de uma variável, o operador unário & pode ser utilizado. No exemplo acima,

```
/* ap recebe o endereco de x */
ap = &x;
```

Após esta instrução, diz-se que `ap` *aponta para* `x`. É possível acessar o valor da variável `x` através do ponteiro usando o operador unário `*`, como em

```
/* y recebe o conteudo da variavel apontada por ap */
y = *ap;
```

[4]Estes qualificadores refletem o estado atual do padrão ANSI-C. Dependendo do computador e do compilador disponível, outros qualificadores podem ser suportados, tais como `long long` e `long double` para representar inteiros de 64 bits e reais de precisão estendida (80 bits), respectivamente. Entretanto, não há ainda padronização neste sentido.

Observe que a combinação `*ap` é um inteiro, que pode assim ser atribuído a outro inteiro. Esta associação pode facilitar a compreensão da declaração de ponteiros. No exemplo, `int *ap` pode ser lido como "`*ap` é um inteiro."

Também seria possível definir o valor de `x` através do ponteiro, como em

```
/* o conteudo da variavel apontada por ap recebe y */
*ap = y;
```

Ponteiros podem tomar parte em expressões aritméticas. Assim, a seguinte expressão é perfeitamente válida:

```
y = *ap + 1;
```

ou seja, `y` receberia o valor de `x` (variável apontada por `ap`) incrementado de um.

Observe que a expressão `ap + 1` expressa um incremento do ponteiro (endereço) e não do conteúdo. Neste caso, estaria indicando o inteiro que estivesse armazenado na posição de memória seguinte ao inteiro armazenado em `ap`. As únicas operações aritméticas envolvendo ponteiros que são permitidas são a soma ou subtração de um valor inteiro a um ponteiro e a diferença entre dois ponteiros. Para essas operações, a unidade corresponde ao tamanho em bytes do tipo manipulado pelo ponteiro. Assim, se o valor de `ap` é 1000 e em uma máquina X o inteiro é representado em dois bytes, `ap+1` resulta em 1002. Porém, a mesma expressão em uma máquina Y com inteiros em quatro bytes retornaria 1004. Desse modo, o programador não precisa se preocupar com os detalhes do armazenamento, trabalhando simplesmente com "o próximo inteiro."

O **ponteiro nulo** é um valor que representa "nenhuma posição de memória", sendo útil para indicar condições de erro e evitar o acesso a áreas inválidas de memória. Em C, o ponteiro nulo é representado pelo valor 0.

2.1.3 Tipos agregados

As linguagens de programação normalmente oferecem facilidades para construir tipos compostos por agregados de outros elementos. Estruturas agregadas uniformes, onde todos os elementos são de um mesmo tipo arranjados em uma área contígua de memória, constituem **arranjos** ou vetores seqüenciais (Fig. 2.1b); seus elementos podem ser acessados através de um índice representando a posição desejada.

Em C, arranjos são definidos e acessados através do operador de indexação `[]`, como em:

```
int elem[5];   // definição: arranjo com cinco inteiros
...
elem[0] = 1;   // acesso ao primeiro elemento do arranjo
```

Neste exemplo, um arranjo de nome `elem` é definido na função `main`. Este arranjo tem espaço para armazenar cinco valores inteiros, que serão referenciados no programa como `elem[0]`, `elem[1]`, `elem[2]`, `elem[3]` e `elem[4]`. Observe através desse exemplo que o primeiro elemento de um arranjo em C é sempre o elemento de índice 0 (`elem[0]`). Conseqüentemente, para um arranjo com `N` elementos o último elemento é o de índice `N-1` (`elem[4]`, no exemplo).

A implementação de arranjos está intimamente ligada ao conceito de ponteiros. Quando se cria, como acima, um arranjo com cinco inteiros, reserva-se o espaço na memória para armazená-los e atribui-se um nome para o endereço inicial dessa área — em outras palavras, um ponteiro. O acesso ao valor `i`-ésimo elemento do arranjo `elem`, `elem[i]`, equivale à expressão

```
*(elem + i)
```

Em C, as duas formas de indicar o elemento acessado podem ser usadas indistintamente.

Agregados não-uniformes, com componentes de tipos distintos, constituem **registros** (Fig. 2.1c). Em C, registros são definidos através do uso da construção `struct`, como em:

```
struct {
  int a;
  float b;
} x;            // definição da estrutura
```

A forma geral de definição de uma estrutura C é

```
struct tag {
    /* declaracao de componentes: */
    ...
} var1, var2, ..., varN;
```

A palavra chave `struct` inicia a definição da estrutura. A etiqueta (`tag`) é opcional, porém deve estar presente caso se deseje referenciar esta estrutura em algum momento posterior. Da mesma forma, a lista de variáveis declaradas (`var1`, ... , `varN`) também não precisa estar presente — a não ser que a etiqueta não esteja presente.

Quando a etiqueta está presente na definição de uma estrutura, variáveis com essa mesma estrutura podem ser definidas posteriormente. O exemplo acima poderia ter separado a definição da estrutura, como em:

```
struct simples {
     int a;
     float b;
};
```

da declaração da variável com esse tipo de estrutura, como em

```
struct simples x;
```

Muitas vezes, definições de estruturas estão associadas ao uso do comando `typedef` (definição de tipo), que associa um nome simbólico a algum tipo básico da linguagem C (ver Seção C.2.6). No exemplo, um nome de tipo `Simples` poderia estar associado à estrutura definida através da expressão:

```
typedef struct simples Simples;
```

e então declarar `x` como

```
Simples x;
```

Uma vez que uma estrutura tem componentes internos que devem ser acessados para processamento, algum mecanismo de acesso deve ser fornecido pela linguagem. C oferece dois operadores que permitem acessar elementos de estruturas. O operador básico de acesso a elementos de estruturas é o operador `.` (ponto). Por exemplo, para atribuir o valor 1 ao componente `a` de `x`, a expressão usada é

```
x.a = 1;        // acesso
```

A outra forma de expressar acesso aos elementos de uma estrutura está associada ao conceito de ponteiros para estruturas — neste caso, o operador `->` (seta) é utilizado. Considere o exemplo

```
Simples *y;        // um ponteiro para uma estrutura
y = &x;
y->a = 1;          // equivalente ao exemplo anterior
```

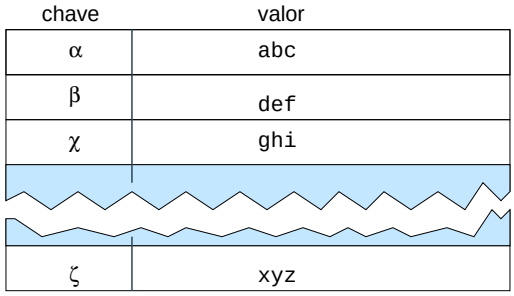
Observe que a expressão `y->a` equivale à `(*y).a`, sendo simplesmente uma notação mais confortável para denotar a mesma operação.

2.2 Tabelas

Tabela é uma das estruturas de dados de fundamental importância para o desenvolvimento de *software* de sistema. Um de seus usos principais é na construção de tabelas de símbolos, amplamente utilizadas em compiladores e montadores. Tabelas são também amplamente utilizadas em sistemas operacionais para a manutenção de informação de apoio à execução de programas, como as tabelas de processos e as tabelas de arquivos.

Uma tabela é um agregado de elementos armazenados na forma de pares *chave*,*valor*, de tal forma que é possível ter acesso a *valor* (que pode eventualmente ser uma estrutura complexa) a partir da *chave* (Figura 2.2). Assim, deve ser possível obter o valor `def` a partir da especificação da chave β, ou o valor `xyz` a partir de ζ.

Figura 2.2 Visão conceitual de uma tabela.



Em uma tabela de símbolos, a chave é usualmente o nome de uma variável e o valor é o conjunto de informações sobre a variável, tais como o seu tipo, endereço de memória e local de definição. Já em uma tabela de arquivos, a chave pode ser um identificador inteiro (por exemplo, o terceiro arquivo aberto pelo processo) e o valor o conjunto de informaçòes sobre o arquivo, tais como a posição corrente no arquivo.

Sob o ponto de vista funcional, a operação essencial associada a uma tabela é a busca de um valor a partir da especificação de uma chave:

GETVALUE(tabela,chave) retorna o valor correspondente à chave especificada, se esta estiver presente na tabela; caso contrário, retorna uma indicação (por exemplo, um valor nulo) de que a chave não existe na tabela indicada.

Caso a tabela seja estática, ou seja, tenha um conteúdo que é determinado no momento da construção da tabela e não é alterado durante seu processamento, essa é a única operação definida para a manipulação da estrutura de dados, pois tais tabelas são utilizadas exclusivamente para consultas. Exemplos de tabelas desse tipo são as tabelas que descrevem as instruções de um processador, usadas em montadores.

Para a manipulação de tabelas que podem ter seu conteúdo construído e modificado ao longo da execução de um programa faz-se necessário definir outras operações, tais como:

INSERT(tabela,chave,valor) é o procedimento para inserir uma nova informação na tabela;

REMOVE(tabela,chave) é o procedimento para retirar da tabela o elemento que tem a chave especificada.

2.2.1 Organização interna

A forma mais simples de organizar a estrutura interna de uma tabela é através da criação de um arranjo de registros, onde cada registro tem pelo menos dois campos, um deles correspondente à chave e o outro, ao valor. Para auxiliar a descrição e manipulação desses dois campos de cada elemento da tabela, são definidos alguns tipos de dados abstratos e suas propriedades correspondentes.

A chave será descrita por um tipo KEY, para o qual pelo menos um operador é definido, que é o teste de igualdade ($=$) entre duas chaves. Se a chave pertence a um domínio parcialmente ordenado, então os operadores de comparação de ordem (maior que, $>$, e menor que, $<$) são também definidos. Naturalmente, os operadores complementares ($\neq$, $\leq$, $\geq$) podem também ser definidos.

Para os objetivos desta descrição, o valor é descrito por um tipo VALUE, sem nenhuma propriedade específica exceto pelo acesso ao seu conteúdo para leitura e escrita.

Uma entrada em uma tabela será representada pelo tipo ENTRY, definido como um registro com um KEY e um VALUE:

$$\begin{array}{rcl} \text{ENTRY} \models & \text{c} & : \text{KEY} \\ & \text{v} & : \text{VALUE} \end{array}$$

Com essas definições, a estrutura interna para uma tabela poderia ser representada como um registro com o número de elementos na tabela e um arranjo de entradas:

$$\begin{array}{rcl} \text{TABLE} \models & \text{n} & : \texttt{integer} \\ & \text{e} & : \texttt{array}[\text{n}]\ \texttt{of}\ \text{ENTRY} \end{array}$$

2.2.2 Aspectos de implementação

Embora a linguagem C não permita efetivamente a criação de novos tipos, através da construção `typedef` é possível criar programas em C com um nível de abstração bem próximo àquele dos procedimentos descritos de forma conceitual.

Por exemplo, seria possível definir um tipo `Table`, associado a uma estrutura C, com capacidade para 100 posições através da seguinte construção:

```
#define TABLESIZE 100
typedef struct {
  int    n;
  Entry  e[TABLESIZE];
} Table;
```

O tipo `Entry` seria similarmente definido:

```
typedef struct {
  Key   c;
  Value v;
} Entry;
```

Finalmente, `Key` e `Value` seriam definidos da mesma forma. Por exemplo, se a chave for um valor inteiro sem sinal, `Key` poderia ser definido como

```
typedef unsigned int Key;
```

`Value` poderia ser definido similarmente, tanto no caso de um valor complexo (usando um `struct`) como no caso de um valor simples (como exemplificado para `Key`).

2.3 Busca

A operação GETVALUE, apresentada como a operação essencial na manipulação de uma tabela, é a implementação de um procedimento de busca, essencial para qualquer estrutura de dados. Embora detalhes desse procedimento sejam específicos de cada estrutura de dados, alguns princípios são comuns.

Em geral, estruturas cujo conteúdo são mantidos sem ordem serão mais simples de criar, porém demandarão mais tempo de busca. Na seqüência serão apresentados os dois mecanismos básicos de busca no contexto de tabelas, a busca linear e a busca binária. Os dois procedimentos poderiam ser utilizados para a implementação de GETVALUE, mas a busca binária demanda que as entradas na tabela sejam mantidas em ordem pelo valor da chave.

2.3.1 Busca linear

No caso da implementação mais simples de uma tabela, as suas entradas poderiam estar em qualquer ordem — por exemplo, a ordem em que foram criadas. Neste caso, a operação de busca por um valor tem que ser exaustiva, ou seja, eventualmente todas as entradas deverão ser pesquisadas a fim de determinar se a chave está ou não presente na tabela.

Este procedimento de **busca linear** está descrito de forma simplificada no Algoritmo 2.1. Neste algoritmo, os argumentos de entrada são a tabela onde a busca será realizada e a chave de busca.

Algoritmo 2.1 Busca linear em tabela.

```
LINEARSEARCH(TABLE T, KEY c)
1  declare pos : INTEGER
2  for pos ← 0 to T.n − 1
3  do if c = T.e[pos].c
4        then return T.e[pos].v
5  return NIL
```

Esse algoritmo começa a analisar a tabela a partir da primeira posição, podendo potencialmente analisar todas as posições da tabela — a máxima quantidade de posições analisáveis está restrita ao número de elementos na tabela. A chave associada a cada entrada da tabela é comparada com a chave de busca. Caso as duas chaves sejam iguais, a entrada foi encontrada e o algoritmo encerra retornando o valor associado à chave para essa entrada. Caso a busca chegue ao final da tabela sem que a chave especificada tenha sido encontrada, o algoritmo encerra retornando o valor especial NIL.

O atrativo desse procedimento é a simplicidade. Porém, seu uso está restrito a tabelas pequenas, pois para tabelas grandes ele é muito ineficiente. O tempo de pesquisa cresce linearmente com o número de entradas na tabela, ou seja, o algoritmo apresenta complexidade temporal $O(n)$.

2.3.2 Busca binária

Um mecanismo mais eficiente de busca é análogo àquele utilizado ao procurar uma palavra no dicionário. Faz-se uma estimativa da posição aproximada da palavra e abre-se na página estimada. 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 até que a página com a palavra desejada seja encontrada. Esse mecanismo de busca só é possível porque as palavras estão ordenadas no dicionário. Se o dicionário mantivesse as palavras sem nenhuma ordem, apenas a busca linear seria possível. Da mesma forma, a busca em uma tabela pode ser melhorada se seu conteúdo estiver ordenado.

O algoritmo de **busca binária** utiliza exatamente esse princípio. No caso, a estimativa que é feita para a posição a ser buscada é a posição mediana do restante da tabela que falta para ser analisado. No início, o restante é a tabela toda; assim, a primeira posição analisada é a entrada no meio da tabela. Se não for a entrada buscada, analisa-se a metade que resta, ou a inferior (se a chave encontrada tem valor maior que a que se busca) ou a superior (em caso contrário). O procedimento assim se repete, até que se encontre a chave buscada ou que a busca se esgote sem encontrar essa chave.

O Algoritmo 2.2 descreve essa forma de busca. Os dois argumentos são, como para o algoritmo de busca linear, a tabela e a chave de busca. Também como no caso anterior, o valor retornado é o valor associado à chave na tabela ou NIL se a chave não for encontrada. As variáveis *bot*, *mid* e *top* referem-se a posições na tabela — respectivamente o início, o meio e o fim da área da tabela ainda por ser pesquisada.

Algoritmo 2.2 Busca binária em tabela.

```
BINARYSEARCH(TABLE T, KEY c)
 1  declare bot, mid, top : INTEGER
 2  bot ← 1
 3  top ← T.n
 4  while bot ≤ top
 5  do mid ← ⌊(bot + top)/2⌋
 6     if c > T.e[mid].c
 7        then bot ← mid + 1
 8        else if c < T.e[mid].c
 9                then top ← mid − 1
10                else return T.e[mid].v
11  return NIL
```

Uma diferença entre os dois algoritmos de busca apresentados é que neste assume-se que a tabela está ordenada. A manutenção da ordem em uma tabela dá-se através de procedimentos auxiliares de **ordenação**, uma das áreas relevantes de estudos em estruturas de dados.

As situações especiais que podem ocorrer nesse processamento de busca incluem o tratamento de mais de uma entrada na tabela com a mesma chave e a situação na qual nenhuma entrada é encontrada para a chave. Tais situações devem ser tratadas, em princípio, pela aplicação, que saberá qual ação deve ser tomada em cada um desses casos.

2.3.3 Usando rotinas de busca em C

A biblioteca padrão C oferece uma função genérica, `bsearch`, para realizar a busca binária de uma chave em um arranjo ordenado. O protótipo dessa rotina está definido em `stdlib.h`:

```
void *bsearch(const void *key, const void *base, size_t nmemb,
       size_t size, int (*compar)(const void *, const void *));
```

A função `bsearch` manipula elementos de qualquer tipo especificado pelo programador. Para tanto, faz uso extensivo de ponteiros especificados genericamente no programa através do tipo `void*`.

A função irá procurar a chave cujo valor é apontado por `key` no arranjo cujo início é apontado por `base`. Para poder percorrer o arranjo corretamente, é preciso indicar quantos elementos estão nele contidos (`nmemb`) e qual o tamanho de cada elemento em bytes (`size`).

O programador precisa também informar como deve ser realizada a comparação entre dois elementos desse arranjo, de forma a determinar em que segmento do arranjo a busca deve continuar após cada comparação. Para tanto, o programador deve fornecer uma rotina que compare dois elementos, sendo que o primeiro será um ponteiro para a chave (`key`) e o segundo um ponteiro para o membro do arranjo que está sendo pesquisado. O valor de retorno dessa rotina deve ser um inteiro que será igual a 0 se os dois valores forem iguais; menor que 0 se o valor da chave for menor que o valor pesquisado; ou maior que 0 se o valor da chave for maior que o valor pesquisado no arranjo.

Por exemplo, considere a busca de um valor inteiro em um arranjo do tipo `int`. A função de comparação poderia ser definida como:

```
int compara(int *s1, int *s2) {
  return *s1 - *s2;
}
```

Por exemplo, considere que um arranjo de elementos inteiros foi definido e contém uma série de valores ordenados:

```
#define SIZE 100
... // definição da função
 int array[SIZE];
... // preenchimento do arranjo
```

Para procurar se o arranjo contém um elemento cuja valor é `1900`, a invocação seria da forma:

```
int busca = 1900;
int *found;
found = bsearch(&busca,array,SIZE,sizeof(int),compara);
if (found)
   printf("Encontrou %d na posicao %d\n", busca, found-array);
else
   printf("Nao encontrou %d\n", busca);
```

2.4 Ordenação

Por simplicidade, é assumido nessa apresentação que as tabelas que serão ordenadas estão sempre contidas em memória. A classe de 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)*.

2.4.1 Ordenação por comparação

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 chave na tabela. Uma vez que seja definido que entrada é essa, troque-a com a primeira entrada da tabela. Repita então o procedimento para o restante da tabela, excluindo os elementos que já estão na posição correta. Esse procedimento é descrito no Algoritmo 2.3, que recebe como argumento a tabela a ser ordenada com base nos valores da chave de cada entrada. No procedimento, *pos1* e *pos2* são as posições da tabela sendo correntemente analisadas e *min* corresponde à posição que contém a chave com menor valor dentre as restantes na tabela. A variável *sml* contém o valor da menor chave encontrada.

Neste algoritmo, o laço de iteração mais externo indica o primeiro elemento na tabela não ordenada a ser analisado — no início, esse é o primeiro elemento da tabela. As linhas de 4 a 8 são responsáveis por procurar, no restante da tabela, o elemento com menor valor de chave. Na linha 9, o procedimento auxiliar SWAP, descrito no Algoritmo 2.4, inverte o conteúdo das duas entradas nas posições especificadas.

Este tipo de algoritmo de ordenação é razoável para manipular tabelas com um pequeno número de elementos, mas à medida que o tamanho da tabela cresce ele passa a se tornar inviável — sua complexidade temporal é $O(n^2)$, conseqüência do duplo laço de iteração que varre a tabela até o final. Felizmente, há outros algoritmos de ordenação com melhores características.

Algoritmo 2.3 Ordenação de tabela pela busca do menor valor.

```
MINENTRYSORT(TABLE T)
1  declare min, pos1, pos2 : INTEGER
2  declare sml : KEY
3  for pos1 ← 1 to T.n − 1
4  do sml ← +∞
5     for pos2 ← pos1 to T.n
6     do if T.e[pos2].c < sml
7           then sml ← T.e.[pos2].c
8                min ← pos2
9     SWAP(T, pos1, min)
```

Algoritmo 2.4 Troca de conteúdos de duas posições de uma tabela.

```
SWAP(TABLE T, INTEGER p1, INTEGER p2)
1  declare aux : ENTRY
2  if p1 ≠ p2
3    then aux ← T.e[p1]
4         T.e[p1] ← T.e[p2]
5         T.e[p2] ← aux
```

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 2.5), os argumentos determinam as posições inicial e final do segmento da tabela a ser ordenado, *init* e *end*, respectivamente.

O segredo deste algoritmo está na forma de realizar a partição — um elemento é escolhido como **pivô**, ou separador de partições. Nesse procedimento, a posição do pivô resultante é estabelecida pela posição *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 tabela. 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

Algoritmo 2.5 Ordenação de tabela por *quicksort*.

```
QUICKSORT(TABLE 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.e[pos1].c < T.e[init].c
 7            do pos1 ← pos1 + 1
 8            while T.e[pos2].c > T.e[init].c
 9            do pos2 ← pos2 − 1
10            if pos1 < pos2
11              then part ← pos1
12                   SWAP(T, pos1, pos2)
13              else part ← pos2
14                   break
15         SWAP(T, init, part)
16         QUICKSORT(T, init, part − 1)
17         QUICKSORT(T, part + 1, end)
```

de ordenação ocorre à medida que elementos são definidos.

2.4.2 Ordenação por contagem

Outra classe de algoritmos de ordenação é aquela na qual a definição da ordem se dá por contagem. 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 de valor i cada vez que esse valor for encontrado. Ao final dessa varredura conhece-se exatamente quantas posições serão necessárias para cada valor de chave; os elementos são transferidos para as posições corretas na nova coleção, ordenada.

Claramente, a aplicação desse princípio básico de contagem a domínios com muitos valores tornaria-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 aplicando 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 2.6. Para essa descrição, assumiu-se que a chave da tabela 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 da chave está sendo analisada, iniciando pelos R bits menos significativos; *pos*, que indica qual posição da tabela está sendo analisada; *radixValue*, o valor da parte da chave (entre 0 e $2^R - 1$) no passo atual; *count*, a tabela de contadores; e T_{aux}, uma cópia da tabela ordenada segundo a raiz sob análise ao final de cada passo.

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 o tamanho da chave em bits.

Algoritmo 2.6 Ordenação de tabela por *radixsort*.

```
RADIXSORT(TABLE T, INTEGER R)
 1  declare pass, pos, radixValue : INTEGER
 2  declare count : array[2^R] of INTEGER
 3  declare T_aux : TABLE
 4  for pass ← 1 to ⌈SIZEOFBITS(KEY)/R⌉
 5  do for radixValue ← 0 to 2^R − 1
 6     do count[radixValue] ← 0
 7     for pos ← 0 to T.n − 1
 8     do radixValue ← BINARYAND(BINARYSHIFT(T.e[pos].c, 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.n − 1 to 0
13     do radixValue ← BINARYAND(BINARYSHIFT(T.e[pos].c, R × (pass − 1)), 2^R − 1)
14        T_aux.e[count[radixValue]] ← T.e[pos]
15        count[radixValue] ← count[radixValue] − 1
16     T ← T_aux
```

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), a tabela é percorrida para avaliar o valor da raiz em cada posição (linha 8) 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 na tabela. Essa informação permitirá que, no próximo laço (linhas 12 a 15), a tabela T_{aux} seja preenchida colocando cada entrada da tabela 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 a próxima raiz com o mesmo valor seja colocada na posição correta, anterior à última ocupada. Finalmente, a tabela 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), a tabela estará completamente 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 $O(n)$.

2.4.3 Usando rotinas de ordenação em C

A biblioteca padrão C oferece implementações genéricas da rotina de ordenação *quicksort*, a função `qsort`, cujo protótipo está definido no arquivo de cabeçalho `stdlib.h`:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *))
```

Os argumentos de `qsort` indicam o início do arranjo que deve ser ordenado (`base`), a quantidade de elementos (`nmemb`) e o tamanho de cada elemento do arranjo (`size`). O último argumento é o ponteiro para a função que sabe como comparar dois elementos do arranjo.

Assim como para `bsearch` (Seção 2.3.3), é necessário que o programador indique como deve ser realizada a comparação entre dois elementos da estrutura que será ordenada. Para tanto, uma função deve ser definida

que receba dois argumentos que são referências para cada um dos elementos. Essa função retorna um valor inteiro, que deve ser:

menor que 0 se o primeiro elemento preceder o segundo elemento no critério de ordenação estabelecido;

maior que 0 se o primeiro elemento suceder o segundo elemento no critério de ordenação estabelecido; ou

igual a 0 quando os dois elementos forem iguais.

Por exemplo, para a comparação de elementos do tipo inteiro a função de comparação poderia ser a mesma já apresentada na Seção 2.3.3. Usando um exemplo similar àquele, para utilizar `qsort` para ordenar o arranjo de `SIZE` inteiros poderia se utilizar a invocação

```
qsort(array,SIZE,sizeof(int),compara);
```

2.5 Tabelas *hash*

Na apresentação da estrutura de tabelas, a busca por uma chave ocorre sempre através de comparações. Uma alternativa de busca em tabelas dá-se através do cálculo da posição que uma chave ocupa na tabela através de uma função. Esse tipo de função que mapeia um símbolo para valores inteiros é denominado **função *hash*** e o tipo de tabela 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.

Idealmente, cada chave processada por uma função *hash* geraria uma posição diferente na tabela. 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.

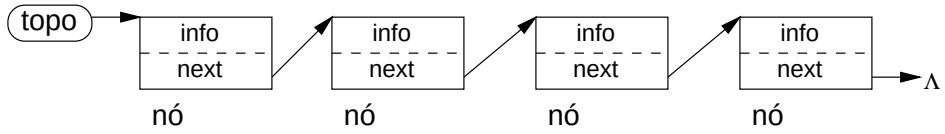
Na técnica de tratamento de colisão por encadeamento, para cada posição onde ocorre colisão cria-se uma área de armazenamento auxiliar, externa à área inicial da tabela *hash*. Normalmente essa área é organizada como uma **lista ligada** (Seção 2.6) que contém todas as chaves que foram mapeadas para 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.

2.6 Listas ligadas

Uma **lista ligada** é uma estrutura que corresponde a uma seqüência lógica de entradas ou **nós**. Tipicamente, em uma lista ligada há um ou dois pontos conhecidos de acesso — normalmente o **topo** da lista (seu primeiro elemento) e eventualmente o fim da lista (seu último elemento). Cada nó armazena também a localização do próximo elemento na seqüência, ou seja, de seu **nó sucessor**. Desse modo, o armazenamento de uma lista não requer uma área contígua de memória. A Figura 2.3 representa graficamente uma estrutura de lista ligada.

Figura 2.3 Representação de uma lista ligada.



2.6.1 Manipulação de nó

Como pode-se observar nessa figura, um nó é essencialmente uma estrutura com dois campos de interesse: *info*, o conteúdo do nó, e *next*, uma referência para o próximo nó da lista. A entrada que determina o topo da lista deve ser registrada à parte da lista. Essa informação é tipicamente mantida em um **nó descritor** da lista. A entrada que marca o fim da lista não precisa de indicação especial — tipicamente, o ponteiro nulo como valor de *next* marca o final da lista.

Para fins de descrição dos procedimentos de uma lista ligada, considera-se aqui que o nó de uma lista é um registro com a seguinte estrutura:

$$\begin{array}{lllll} \text{NODE} \models & \text{info} & : & \text{ENTRY} \\ & \text{next} & : & \text{NODE} \end{array}$$

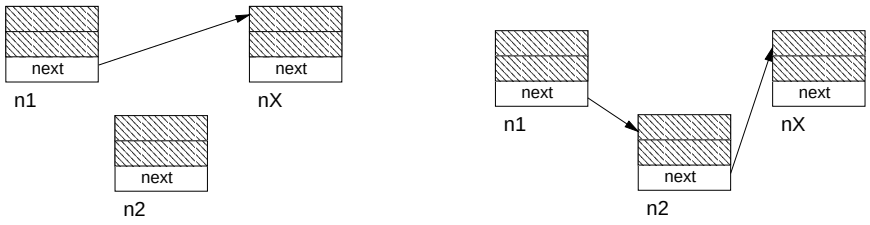
Como listas são estruturas dinâmicas, normalmente são definidos procedimentos que permitem criar e remover nós na memória. Neste texto, a criação e remoção de um nó estarão associadas respectivamente aos procedimentos:

CREATENODE(ENTRY e). Aloca recursos (área de memória) para guardar a informação especificada nos argumentos. Retorna uma referência para o nó criado, do tipo NODE; e

DELETENODE(NODE n). Libera os recursos usados pelo nó.

Estabelecer a conexão entre dois nós é uma operação simples e freqüente na manipulação de listas. Para estabelecer a ligação entre um nó já pertencente a uma lista e um novo nó, basta fazer com que o novo nó referencie no campo *next* o nó que anteriormente era referenciado pelo nó original — mesmo que esse campo tenha o valor nulo. Para concluir a conexão, o nó original deve ter atualizado o campo *next* para referenciar o novo nó. O efeito dessa conexão é ilustrado na Figura 2.4.

Figura 2.4 Efeito da aplicação do procedimento LINKNODE.



(a) Antes da ligação (b) Após a ligação

O procedimento LINKNODE, apresentado no Algoritmo 2.7, descreve como estabelecer essa ligação entre os dois nós que são passados como argumento.

Algoritmo 2.7 Ligação de dois nós.

LINKNODE(NODE $n1$, NODE $n2$)
1 $n2.next \leftarrow n1.next$
2 $n1.next \leftarrow n2$

2.6.2 Manipulação de lista

A informação sobre a estrutura de uma lista ligada está distribuída ao longo de seus nós. Assim, a única informação adicional requerida para manter a lista é a especificação de seu nó descritor:

$$\text{LIST} \models \quad \text{top} \quad : \quad \text{NODE}$$

Na criação de uma lista, o nó descritor está inicialmente vazio, de forma que seu campo *next* tem o valor nulo. Assim, um procedimento para verificar se a lista está vazia deve verificar o valor desse campo. Esse procedimento está descrito no Algoritmo 2.8.

Para a inserção de um novo nó em uma lista há duas possibilidades que podem ser consideradas, dependendo da opção de se inserir o novo nó no início (antes do primeiro elemento) ou no final (após o último elemento) da lista.

Algoritmo 2.8 Verificação se a lista ligada está vazia.

```
ISEMPTY(LIST l)
1  if l.top.next = NIL
2     then return true
3     else  return false
```

A primeira dessas possibilidades está representada através do procedimento INSERT, que recebe como argumentos as referências para a lista e para o nó a ser inserido. O Algoritmo 2.9 apresenta esse procedimento.

Algoritmo 2.9 Inserção de nó no topo da lista ligada.

```
INSERT(LIST l, NODE n)
1  LINKNODE(l.top, n)
```

O procedimento que acrescenta um nó ao final da lista necessita varrer a lista completamente em busca do último nó, aquele cujo campo *next* tem o valor nulo. Para tanto, requer a utilização de uma variável interna que indica qual o nó está atualmente sendo analisado. No momento em que o campo *next* desse nó tiver o valor nulo, então sabe-se que o último nó da lista foi localizado. Esse procedimento, APPEND, está descrito no Algoritmo 2.10.

Algoritmo 2.10 Inserção de nó no final da lista ligada.

```
APPEND(LIST l, NODE n)
1  declare curr : NODE
2  curr ← l.top
3  while curr.next ≠ NIL
4  do curr ← curr.next
5  LINKNODE(curr, n)
```

De forma similar, o procedimento para retirar um nó do início da lista é mais simples que aquele para retirar um nó do fim da lista, pois este requer a varredura completa da lista. Nos dois casos, o valor de retorno é uma referência ao nó retirado; a partir dessa referência, a aplicação pode determinar o que deseja fazer com o nó, se manipular a informação nele contida ou simplesmente liberar os recursos com o procedimento DELETENODE. Um valor de retorno nulo indica que a operação foi especificada sobre uma lista vazia.

O procedimento para retirar o nó do início da lista é apresentado no Algoritmo 2.11. Embora a linha 5 desse algoritmo não seja absolutamente necessária, ela garante que não há meios de acesso aos nós da lista exceto pelos procedimentos definidos. Se ela não estivesse presente, seria possível que a aplicação, ao obter o nó com a informação de endereço do seu antigo sucessor, tivesse acesso a um nó da lista de forma direta.

O procedimento para a retirada de um nó do fim da lista é descrito no Algoritmo 2.12. Da mesma forma que no procedimento de remoção do primeiro elemento da lista, a situação de manipulação de uma lista vazia deve receber tratamento especial. Como no procedimento de inserção, uma varredura de toda a lista é feita mantendo-se uma referência para o nó sob análise; adicionalmente, mantém-se uma referência para o nó anterior a este de forma a permitir a atualização da indicação do fim da lista.

Outro procedimento usual na manipulação de uma lista ligada é aquele que permite procurar o nó que satisfaz um determinado critério de busca — por exemplo, cuja chave em seu campo de informação seja igual a um argumento fornecido. Esse procedimento de busca é apresentado através do Algoritmo 2.13, que retorna

Algoritmo 2.11 Retirada do primeiro nó da lista ligada.

```
REMOVEFIRST(LIST l)
1  declare first : NODE
2  first ← l.top.next
3  if first ≠ NIL
4    then LINKNODE(l.top, first.next)
5         first.next ← NIL
6  return first
```

Algoritmo 2.12 Retirada do último nó da lista ligada.

```
REMOVELAST(LIST l)
1  declare pred, curr : NODE
2  pred ← l.top
3  curr ← l.top.next
4  if curr ≠ NIL
5    then while curr.next ≠ NIL
6         do pred ← curr
7            curr ← curr.next
8         pred.next ← NIL
9  return curr
```

uma referência para o campo de informação do nó encontrado ou o valor nulo se não for encontrado nenhum nó que satisfaça a condição de busca.

Algoritmo 2.13 Busca de nó com chave especificada em lista ligada.

```
FIND(LIST l, KEY k)
1  declare curr : Node
2  curr ← l.top.next
3  while curr ≠ NIL
4  do if curr.info.c = k
5       then return curr.info
6       else curr ← curr.next
7  return NIL
```

Como se observa nesse caso, a varredura de uma lista em busca de uma dada chave é um procedimento seqüencial, similar em conceito à busca linear. Essa é a contra-partida à flexibilidade de manipulação de uma lista — não há como realizar uma busca binária, por exemplo, nesse tipo de estrutura. Assim, o procedimento deve analisar a lista nó a nó até encontrar o elemento procurado.

Dependendo da aplicação, outros procedimentos podem ser associados à manipulação de uma lista ligada, tais como obter o número de nós na lista, SIZE(); concatenar ou combinar duas listas, CONCAT(); inserir elemento em posição específica da lista, INSERTAT(); ou remover elemento em posição específica, REMOVEAT().

2.6.3 Filas e pilhas

Filas e pilhas são estruturas usualmente implementadas através de listas, retringindo a política de manipulação dos elementos da lista.

Uma **fila** *(queue)* tipicamente estabelece uma política FIFO — *first in, first out* — de acesso aos dados. Em outras palavras, a ordem estabelecida na lista é a ordem de inserção. No momento de retirar um nó da lista, o nó mais antigo (o primeiro que entrou) é o primeiro a ser retirado.

Como as políticas de inserção e remoção são pré-definidas, para esse tipo de estrutura as operações são descritas de forma genérica, INSERT e REMOVE. Há duas possibilidades para implementar as operações de uma fila usando os procedimentos descritos para listas:

1. restringir a inserção ao procedimento INSERT e a remoção ao procedimento REMOVELAST, ou
2. restringir a inserção ao procedimento APPEND e a remoção ao procedimento REMOVEFIRST.

Uma estrutura de **pilha** *(stack)*, por outro lado, estabelece uma política LIFO — *last in, fisrt out*. Uma estrutura de pilha também oferece basicamente duas operações de manipulação, PUSH, para inserção no topo da pilha, e POP, para retirada do topo da pilha.

Embora também fosse possível implementar uma pilha através de lista usando os procedimentos que acrescentam e removem os nós no final da lista, por razões óbvias de desempenho uma pilha é usualmente implementada usando os procedimentos INSERT e REMOVEFIRST, que não requerem a varredura da lista para estabelecer essa política de manipulação de dados.

2.6.4 Manipulação de listas em C

Tradicionalmente, listas em C são implementadas através de estruturas (asociadas aos nós) armazenadas na memória dinâmica.

A estrutura que implementa um nó de uma lista ligada deve incluir, além do contéudo da informação do nó, um ponteiro para o próximo nó. Tipicamente, a definição da estrutura é da forma:

```
typedef struct node {
   /* conteudo */
   ...
   /* proximo no */
   struct node* next;
} Node;
```

É possível criar um conjunto de rotinas para a manipulação de listas genéricas se o conteúdo for um ponteiro para qualquer tipo de dado, `void*`, que pode referenciar até mesmo outras estruturas complexas. Uma variável ponteiro para `void` denota um endereço genérico, que pode ser atribuído a um ponteiro para qualquer outro tipo sem problemas de conversão.

A linguagem C permite que o programador utilize a área de memória dinâmica através de suas rotinas de alocação dinâmica de memória, que estão presentes na biblioteca padrão da linguagem. Há duas atividades básicas relacionadas à manipulação desta área:

1. o programa pode requisitar uma área de memória dentro do espaço livre disponível; ou
2. o programa pode liberar uma área de memória que tenha sido previamente requisitada do espaço livre e que não seja mais necessária.

Em C, a alocação de memória é suportada pela rotina `malloc`. Esta rotina recebe um argumento, que é a dimensão em bytes da área necessária. O valor de retorno é o endereço do início da área que o sistema operacional alocou para este pedido, um ponteiro para o tipo `void`. Por exemplo, para reservar o espaço

necessário para o nó de uma lista, seria necessário ter inicialmente declarado a variável que receberá o ponteiro para um nó:

```
Node *n;
```

A definição do valor do ponteiro dá-se através da invocação de `malloc`:

```
n = malloc(sizeof(struct node));
```

O conteúdo inicial da área alocada é indefinido. Uma variante da rotina de alocação, chamada `calloc`, inicializa o conteúdo da área alocada para 0's. Esta rotina recebe dois argumentos ao invés de um: o primeiro argumento é o número de itens que será alocado, e o segundo é o tamanho em bytes de cada item. Por exemplo, o trecho de código para criar um nó do tipo `Node` em uma rotina CREATENODE poderia ser escrito usando `calloc`:

```
1     #include <stdlib.h>
2        Node *createNode(void* info) {
3           Node *newnode;          /* ponteiro para inicio da area */
4           newnode = calloc(1, sizeof(struct node));
5           /* define conteudo */
6           newnode->entry = info;
7           return newnode;
8         }
```

Para liberar uma área alocada por `malloc` ou `calloc`, a rotina `free` deve ser utilizada. Assim, o exemplo acima poderia ser complementado da seguinte forma:

```
1         void deleteNode(Node *oldnode) {
2           free(oldnode);
3         }
```

O que a rotina `free` faz é retornar o espaço que havia sido pré-alocado dinamicamente para o serviço do sistema operacional que gerencia o espaço livre, que pode assim reutilizar esta área para atender a outras requisições.

Há ainda uma quarta rotina associada à gerência de espaço livre em C, `realloc`, que permite alterar a dimensão de uma área pré-alocada. Esta rotina recebe dois argumentos, o primeiro sendo o ponteiro para a área que já havia sido alocada e o segundo sendo a nova dimensão para esta área em bytes. O valor de retorno é o endereço (que pode eventualmente ser o mesmo que o original) para a área com a nova dimensão requisitada. Caso o endereço seja diferente, `realloc` se encarrega de copiar o conteúdo original para a nova área alocada.

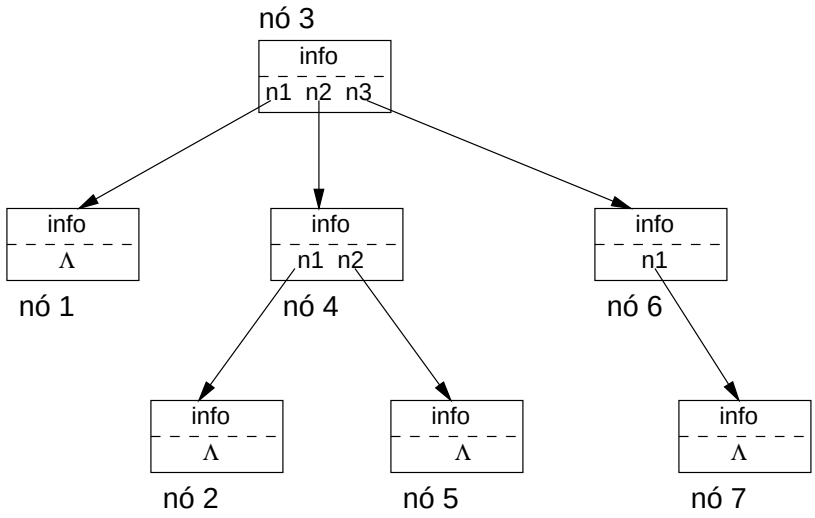
Nesses exemplos, não se comentou o que aconteceria caso não houvesse espaço disponível em memória para atender à requisição de alocação dinâmica. Quando `malloc` (`calloc`, `realloc`) não consegue obter o espaço requisitado, o valor retornado pela função é o ponteiro nulo. Este é um comportamento típico em C para indicar erros em rotinas que retornam ponteiros.

2.7 Árvores

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 2.5). A representação esquemática de árvores usualmente coloca a raiz no topo, com a árvore crescendo para baixo.

Em uma definição mais formal, 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

Figura 2.5 Representação esquemática de uma estrutura de árvore.



em 0 ou mais conjuntos disjuntos onde cada um desses conjuntos é em si uma árvore, que recebe o nome de sub-árvore.

É possível descrever a árvore representada na Figura 2.5 através da aplicação dessa definição como se segue. A árvore T 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 2.6 ilustra alguns exemplos de árvores binárias.

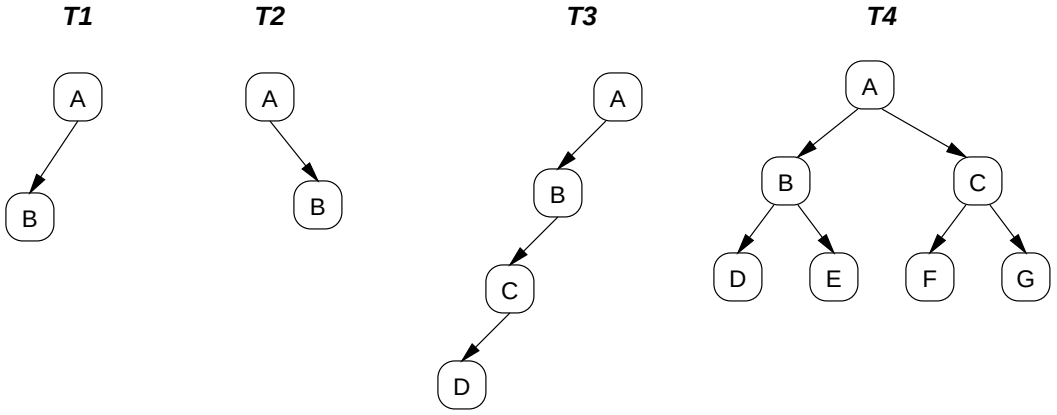
Observe na figura que $T1$ e $T2$ são árvores binárias distintas pois, ao contrário da definição genérica de árvores, há diferença de tratamento para a árvore binária entre a sub-árvore direita e a sub-árvore esquerda. Outra diferença de definição para árvores binárias é que elas podem eventualmente ser vazias, algo que a definição de árvore genérica não permite. $T3$ é uma árvore binária degradada, 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;

Figura 2.6 Árvores binárias



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 2.6), 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.

2.8 Manipulação de arquivos

Um arquivo é um recurso manipulado pelo sistema operacional ao qual a linguagem de programação tem acesso. É uma abstração que permite acessar convenientemente dispositivos externos de entrada e saída de dados. Embora tipicamente um arquivo seja associado a um espaço de armazenamento em disco, outros dispositivos de entrada e saída são manipulados como arquivos, tais como o teclado e a tela de um monitor.

Usualmente, para trabalhar com um arquivo, ele deve ser inicialmente aberto. Sua abertura indica ao sistema operacional que o arquivo será manipulado, de forma que informação que agiliza o acesso ao arquivo é mantida em memória. Após aberto, um arquivo é acessado através de um **descritor**, que basicamente indica onde o sistema operacional mantém a informação sobre o arquivo. Nas descrições de algoritmos, o descritor está associado a um tipo FILE.

A abertura de um arquivo é o passo inicial para a realização dessas operações. O objetivo dessa operação é, inicialmente, traduzir um nome simbólico que identifica o arquivo para um descritor que o programa pode manipular de forma mais eficiente. Nesse procedimento de abertura, é necessário também verificar se quem deseja realizar a manipulação do arquivo tem as autorizações necessárias.

Para realizar a abertura de um arquivo, o sistema operacional oferece uma rotina de serviço que recebe pelo menos um argumento que é o nome do arquivo e retorna o descritor. Outros argumentos que tipicamente estão presentes indicam o modo de operação do arquivo e ações que devem ser tomadas em algumas situações específicas. O modo de operação é designado como leitura ou escrita, podendo adicionalmente ser especificado se o conteúdo deve ser tratado como seqüências de caracteres (modo texto) ou dados (modo binário). As indicações de ação determinam, por exemplo, o que fazer se o arquivo especificado para escrita não existe (criar ou gerar erro), ou caso o arquivo já exista se o conteúdo anterior deve ser sobrescrito (*truncate*) ou mantido (*append*).

Por simplicidade, o procedimento usado para abertura de arquivos nas descrições de algoritmo omitem o segundo argumento:

OPENFILE(): recebe como argumento uma *string* com o nome do arquivo; retorna uma referência para FILE, que é usada nas operações de manipulação do arquivo, ou o valor nulo em caso de falha na operação.

Uma vez que o arquivo esteja aberto, o sistema operacional mantém em suas estruturas internas todas as informações necessárias para sua manipulação; essa informação é obtida através do descritor do arquivo. Por exemplo, quando um arquivo é aberto para leitura a "posição atual" para a próxima operação é o início do arquivo. Ao ler uma linha do arquivo, essa posição é atualizada para o início da linha seguinte e assim sucessivamente.

Quando o arquivo não precisa mais ser manipulado, esses recursos mantidos pelo sistema operacional podem ser liberados. Novamente, uma rotina do sistema oferece esse serviço, que será descrito nos algoritmos pela rotina

CLOSEFILE(): recebe como argumento uma referência para FILE; não tem valor de retorno, mas simplesmente fecha o arquivo que havia sido aberto com OPENFILE().

As rotinas de manipulação de conteúdo que são utilizadas dependem do tipo de conteúdo e do modo de operação do arquivo. No nível mais básico, qualquer arquivo pode ser visto como uma seqüência de bytes; nesse nível, uma rotina de leitura de um byte e outra de escrita de um byte devem existir. A partir dessas, outras rotinas com nível de abstração mais alto podem ser oferecidas.

Para a leitura de arquivos texto, duas rotinas são de interesse:

READCHAR(): recebe como argumento uma referência para FILE; retorna o próximo caráter do arquivo, ou um valor especial EOF para sinalizar que a posição atual alcançou o final do arquivo.

READLINE(): recebe como argumento uma referência para FILE; retorna a próxima linha do arquivo, ou o valor nulo se não houver próxima linha.

2.8.1 Arquivos em C

Na linguagem de programação C, a informação sobre um arquivo é acessada através de um descritor cuja estrutura é definida no arquivo de cabeçalho `stdio.h`. Um programa C que vá manipular arquivos deve então incorporar ao início de seu programa fonte a linha de inclusão desse arquivo de cabeçalho:

```
#include <stdio.h>
```

Esse arquivo de cabeçalho define o nome de tipo `FILE` associado a essa estrutura. Não é necessário conhecer o formato interno dessa estrutura para manipular arquivos. O programador C, para acessar arquivos, define variáveis ponteiros para este tipo, `FILE *`, que são manipuladas diretamente pelas funções da biblioteca padrão de entrada e saída. Tais variáveis são usualmente chamadas de **manipuladores de arquivo**.

Assim, a função que vai manipular o arquivo deve incluir a declaração de uma variável manipulador de arquivo, como em:

```
FILE *arqFonte;
```

O objetivo de manipular um arquivo é realizar operações de leitura e escrita sobre seu conteúdo. Para que essas operações de transferência de dados tenham sucesso, é preciso que haja a permissão adequada para a operação. Por exemplo, um teclado seria um dispositivo que não aceita saída de dados (escrita), mas apenas entrada (leitura).

Para abrir um arquivo em C, a rotina `fopen` é invocada recebendo dois parâmetros. O primeiro é uma *string* com o nome do arquivo que será aberto. O segundo parâmetro é outra *string* que especifica o modo de acesso, que pode conter os caracteres `r` (leitura), `w` (escrita), `a` (escrita ao final — *append*), e `b` (acesso em modo binário). O valor de retorno é o manipulador alocado para o arquivo aberto.

Por exemplo, para realizar a leitura do conteúdo de um arquivo chamado `teste.asm`, a seguinte expressão poderia ser usada no programa:

```
arqFonte = fopen("teste.asm", "r");
```

Caso o arquivo não possa ser aberto, a função `fopen` retorna o ponteiro nulo. Assim, para verificar de o arquivo foi aberto sem problemas, é necessário testar o valor de retorno:

```
if (arqFonte != 0) {
    /*  tudo bem */
}
else {
    /* erro */
}
```

Encerradas as operações sobre um arquivo, ele deve ser fechado. Isso permite que o sistema operacional libere o espaço ocupado pelas informações sobre o arquivo para que esse mesmo espaço possa ser reocupado para a manipulação de outros arquivos. Esta liberação é importante, uma vez que sistemas operacionais tipicamente limitam a quantidade de arquivos que podem ser abertos simultaneamente devido a restrições de espaço alocado para essas estruturas auxiliares.

Para fechar um arquivo previamente aberto, a rotina `fclose` pode ser usada. Ela recebe como argumento o manipulador do arquivo e não retorna nenhum valor. Assim, após encerrada a operação com o arquivo a expressão `fclose(arqFonte);` fecha-o.

2.8.2 Acesso seqüencial

Quando o arquivo é aberto, a posição corrente (mantida internamente pelo sistema) é o início do arquivo. A cada operação executada sobre o arquivo, essa posição é atualizada. O valor da posição corrente pode ser obtido pela função `ftell`. A função `feof` retorna um valor verdadeiro (inteiro diferente de 0) se a posição corrente para o arquivo indicado é o final do arquivo, ou falso (inteiro igual a 0) em caso contrário.

Na maior parte dos exemplos analisados neste texto, os arquivos estarão sendo manipulados de forma seqüencial. Assim, na leitura de um arquivo contendo texto, após a leitura de um caráter a posição corrente do arquivo estará indicando o próximo caráter; após a leitura de uma linha, a posição indicada será o início da próxima linha.

A rotina C para obter um caráter de um arquivo é `fgetc`:

```
int fgetc(FILE *stream);
```

O valor de retorno de `fgetc` é um inteiro, que pode conter o código ASCII do caráter ou o valor `EOF` (definido em `stdio.h`), que indica o final do arquivo ou a ocorrência de alguma condição de erro.

Uma linha de um arquivo texto nada mais é do que uma seqüência de caracteres seguido por um caráter terminador de linha *(newline)*. Tipicamente, o caráter terminador de linha adotado é o CR (ASCII 0x0D), embora alguns sistemas operacionais adotem o par CR/LF (o par de valores 0x0D e 0x0A) como terminador de linha. A linguagem C traduz o terminador de linha para o caráter `'\n'`.

Para ler uma linha de um arquivo texto, a função da biblioteca padrão C `fgets` pode ser utilizada:

```
char *fgets(char *s, int size, FILE *stream);
```

Essa função recebe três argumentos. O primeiro é o endereço de um arranjo de caracteres que irá receber a linha lida; esse arranjo deve ter capacidade para pelo menos `size` caracteres. O segundo é o número máximo de caracteres que deve ser lido da linha, caso a linha tenha mais caracteres que essa quantidade. O terceiro parâmetro é o manipulador do arquivo de onde a linha será lida. O retorno é um ponteiro para o início do arranjo com a linha, ou o ponteiro nulo caso o final do arquivo tenha sido atingido. Se houver espaço para o terminador de linha no arranjo, ele será incluído. Após o último caráter lido, a rotina inclui o terminador de *string* `'\0'`.

Operações correspondentes para a escrita em arquivos são oferecidas pela biblioteca padrão de C. Para escrever um caráter na posição corrente de um arquivo, a rotina `fputc` é usada:

```
int fputc(int c, FILE *stream);
```

Para escrever uma *string*, a rotina `fputs` pode ser utilizada:

```
int fputs(const char *s, FILE *stream);
```

Neste caso, a *string* apontada por `s` (sem o terminador de *string* `'\0'`) é escrita no arquivo.

A função `fseek` permite modificar a posição corrente para um ponto arbitrário do arquivo, se tal operação for permitida. O primeiro argumento dessa função é o manipulador do arquivo; o segundo, um valor `long` indicando o deslocamento desejado; e o terceiro, um valor inteiro inidcando a referência para o deslocamento, que pode ser o início do arquivo (`SEEK_SET`), a posição corrente (`SEEK_CUR`) ou o final do arquivo (`SEEK_END`). Um valor de retorno 0 indica que a operação foi completada com sucesso. A função `rewind` retorna a posição corrente para o início do arquivo, sem valor de retorno.

2.9 Exercícios

2.1 Em uma função C, as seguintes variáveis foram declaradas:

```
int a, b, c[10], *d, *e;
```

Com base nestas declarações, indique se cada uma das linhas abaixo é válida ou não, justificando suas respostas:

(a) `b = 078;`

(b) `d = c;`

(c) `a = b && d;`

(d) `c = e;`

2.2 Considere o seguinte programa em C:

```
#include <stdio.h>
int main( ) {
    int valor=12;
    char str[10];

    printf("Entre uma string: ");
    if (gets(str) == 0) {
        fprintf(stderr, "Sinto muito, algo saiu errado!\n");
        return 1;
    }
    printf("Sua string foi %s e meu valor secreto era %d\n",
            str, valor);
    return 0;
}
```

Quando compilado e executado, em uma das execuções a seguinte situação foi observada:

```
Entre uma string: Pindamonhangaba
Sua string foi Pindamonhangaba e meu valor secreto era 6382177
```

Explique esse comportamento do programa e indique como o código seria modificado para evitar a situação que gerou esse comportamento.

2.3 Implemente um par de programas em C que permitam *(i)* definir valores para um arranjo de 20 inteiros e gravar esses valores em um arquivo em disco em formato texto; e *(ii)* ler esses valores e apresentá-los na saída padrão.

2.4 Repita o exercício anterior usando um arquivo binário para gravar os valores dos elementos do arranjo. Use as funções para manipulação de arquivos binários descritas na Seção C.5.2.

2.5 Modifique o programa de apresentação dos exercícios acima para que a saída seja apresentada por ordem crescente de valores; porém, após cada valor deve ser indicado o índice daquele valor no arranjo original. Por exemplo, se a saída original (para cinco elementos) fosse

```
36
15
28
68
13
```

a nova saída seria

```
13 [4]
15 [1]
28 [2]
36 [0]
68 [3]
```

2.6 Uma função *hash* está sendo definida para mapear *strings* para a faixa de valores entre 0 e 63 da seguinte maneira: o resultado intermediário é a somatória dos valores ASCII de cada caráter não-nulo da *string*; esse resultado é dividido por 64 e o resto da divisão é o valor de *hash*.

(a) Implemente uma função C que realize essa função.

(b) Avalie o resultado dessa função para as seguintes *strings*:

- FEEC
- Universidade Estadual de Campinas
- EA876
- FA875
- Estruturas de dados

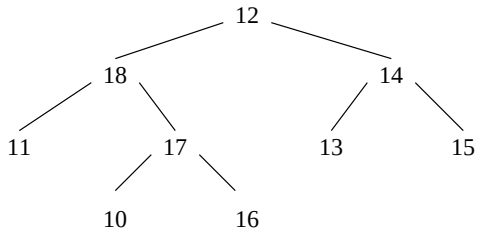
2.7 Repita o exercício anterior substituindo o cômputo do valor de *hash* por divisão pelo método do meio do quadrado.

2.8 Repita os exercícios anteriores substituindo, no cálculo da função *hash*, a operação de soma pela operação ou-exclusivo.

2.9 Descreva os algoritmos para as operações INSERT e REMOVE para a manipulação de fila usando operações de manipulação de lista.

2.10 Descreva os algoritmos para as operações PUSH e POP para a manipulação de pilha usando operações de manipulação de lista.

2.11 Apresente a seqüência de números correspondente à varredura da árvore binária abaixo quando esta for percorrida usando *(i)* pré-ordem, *(ii)* intra-ordem e *(iii)* pós-ordem.



Capítulo 3

Compiladores

Um compilador é um programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Em geral, um compilador não produz diretamente o código de máquina mas sim um programa em linguagem simbólica *(assembly)* semanticamente equivalente ao programa em linguagem de alto nível. O programa em linguagem simbólica é então traduzido para o programa em linguagem de máquina através de montadores.

Para desempenhar suas tarefas, um compilador deve executar dois tipos de atividade. A primeira atividade é a **análise** do código fonte, onde a estrutura e significado do programa de alto nível são reconhecidos. A segunda atividade é a **síntese** do programa equivalente em linguagem simbólica. Embora conceitualmente seja possível executar toda a análise e apenas então iniciar a síntese, em geral estas duas atividades ocorrem praticamente em paralelo.

Para apresentar um exemplo das atividades que um compilador deve desempenhar, considere o seguinte trecho de um programa em C:

```
1    int a, b, valor;
2    a = 10;  b = 20;
3    valor = a * (b + 20);
```

Para o compilador, este segmento nada mais é do que uma seqüência de caracteres em um arquivo texto. O primeiro passo da análise é reconhecer que agrupamentos de caracteres têm significado para o programa — por exemplo, saber que `int` é uma palavra-chave da linguagem e que `a` e `b` serão elementos individuais neste programa. Posteriormente, o compilador deve reconhecer que a seqüência `int a` corresponde a uma declaração de uma variável inteira cujo identificador recebeu o nome `a`.

As regras de formação de elementos e frases válidas de uma linguagem são expressos na **gramática** da linguagem. O processo de reconhecer os comandos de uma gramática é conhecido como **reconhecimento de sentenças**. Gramáticas e reconhecimento de sentenças serão estudados na Seção 3.1.

A aplicação do conceito de reconhecimento de sentenças para agrupar as seqüências de caracteres em "palavras" é a **análise léxica**. Os elementos reconhecidos nessa primeira etapa da compilação são denominados **itens léxicos** ou *tokens*. Para o exemplo acima, a seguinte lista de *tokens* seria reconhecida pelo compilador:

No. token	Valor token	No. token	Valor token
1	int	14	20
2	a	15	;
3	,	16	valor
4	b	17	=
5	,	18	a
6	valor	19	*
7	;	20	(
8	a	21	b
9	=	22	+
10	10	23	20
11	;	24	)
12	b	25	;
13	=		

Para desempenhar a análise léxica, o compilador deve ter conhecimento de quais são os *tokens* válidos da linguagem, assim como suas palavras chaves e regras para formação de identificadores. Por exemplo, a declaração

```
int 1var;
```

deve resultar em erro nessa etapa da análise, pois `1var` não é uma constante ou identificador válido. Se um programa contendo esta declaração fosse compilado usando o compilador `gcc`, por exemplo, a esta linha seria associado o erro *"nondigits in number and not hexadecimal"*, mostrando que o compilador esperava que `1var` fosse um número, já que inicia com um algarismo. A análise léxica será estudada na Seção 3.2.

O segundo passo da análise desempenhado pelo compilador é a **análise sintática**, onde a estrutura do programa é analisada a partir do agrupamento de *tokens*. Nesta etapa, que será estudada na Seção 3.4, o compilador deverá reconhecer que a seqüência de *tokens* obtida do segmento de código apresentado no início da seção corresponde a quatro comandos, sendo o primeiro deles um comando de declaração de variáveis e os três restantes de atribuição. Adicionalmente, deverá reconhecer que o último dos comandos de atribuição contém subexpressões que deverão ser avaliadas para completar a atribuição na execução do programa.

Para que este passo possa ser realizado pelo compilador, ele deve ter conhecimento de como são formados os comandos válidos da linguagem. Por exemplo, um comando de declaração de variáveis como

```
int int x;
```

deve resultar em erro nesta etapa — o compilador `gcc` indicaria o erro com uma mensagem *"two or more data types in declaration of 'x' "*.

Após realizada a análise, na fase de **síntese** o compilador deverá gerar o código em linguagem simbólica equivalente ao código analisado. Por exemplo, para um compilador C gerando código para a linguagem simbólica do processador 68000 os seguintes mapeamentos poderiam estar preparados:

C	Assembly 68K		
int V	V	DS.W	1
L + R		MOVE.W	L,D1
		ADD.W	R,D1
		MOVE.W	D1,VTMP
L * R		MOVE.W	L,D1
		MULS.W	R,D1
		MOVE.W	D1,VTMP
L = R		MOVE.W	R,D1
		MOVE.W	D1,L

onde `L` e `R` seriam substituídos pelo compilador pelas variáveis usadas internamente na síntese do programa.

Na verdade, compiladores não trabalham diretamente com o código de um processador específico. Normalmente o código gerado nessa fase é expresso em alguma linguagem intermediária, próxima do *assembly* mas independente de processador, que depois pod ser mapeada para diversos processadores distintos.

Com base nesse mapeamento, o compilador poderia gerar o seguinte trecho de código em linguagem simbólica para o exemplo apresentado:

```
1     VTMP    DS.W    1
2     a       DS.W    1
3     b       DS.W    1
4     valor   DS.W    1
5             MOVE.W  #10,D1
6             MOVE.W  D1,a
7             MOVE.W  #20,D1
8             MOVE.W  D1,b
9             MOVE.W  b,D1
10            ADD.W   #20,D1
11            MOVE.W  D1,VTMP
12            MOVE.W  a,D1
13            MULS.W  VTMP,D1
14            MOVE.W  D1,VTMP
15            MOVE.W  VTMP,D1
16            MOVE.W  D1,valor
```

Este é um código correto sob o ponto de vista de que as tarefas executadas correspondem semanticamente ao programa C original. Entretanto, uma simples observação demonstra que o código gerado poderia ser muito mais sucinto através da eliminação de alguns comandos desnecessários

A melhoria do código gerado pelo compilador é a fase final da compilação, que é a **otimização de código**. No exemplo acima, por simples inspeção verifica-se que há algumas linhas de código redundantes, como as linhas 8 e 9 e as linhas 14 e 15. O mesmo código poderia ser reduzido para o seguinte programa equivalente:

```
1     a       DS.W    1
2     b       DS.W    1
3     valor   DS.W    1
4             MOVE.W  #10,D1
5             MOVE.W  D1,a
6             MOVE.W  #20,D2
7             MOVE.W  D2,b
8             ADD.W   #20,D2
9             MULS.W  D1,D2
10            MOVE.W  D2,valor
```

Em programas mais complexos, há muitas outras situações onde o código gerado automaticamente pelo compilador pode ser melhorado. O objetivo da fase de otimização de é detectar tais situações automaticamente e aplicar estratégias heurísticas para permitir a melhoria desse código. A geração e otimização de código são objetos da Seção 3.6.

3.1 Gramáticas

Uma linguagem consiste essencialmente de uma seqüência de *strings* ou símbolos com regras para definir quais seqüências de símbolos são válidas na linguagem, ou seja, qual a **sintaxe** da linguagem. A interpretação

do significado de uma seqüência válida de símbolos corresponde à **semântica** da linguagem.

Existem meios formais para definir a sintaxe de uma linguagem — a definição semântica é um problema bem mais complexo. A sintaxe de linguagens é expressa na forma de uma **gramática**, que será introduzida na seqüência.

3.1.1 Terminologia

Conjuntos são representados por seqüências de elementos entre chaves, como em

$$\{1, 2, 3\}$$

Nomes de conjuntos serão usualmente representados por letras maiúsculas,

$$A = \{1, 2, 3\}$$

Operações entre conjuntos incluem união ($\cup$), interseção ($\cap$) e diferença ($-$),

$$\{1, 2\} \cup \{2, 3\} = \{1, 2, 3\}$$
$$\{1, 2\} \cap \{2, 3\} = \{2\}$$
$$\{1, 2\} - \{2, 3\} = \{1\}$$

Os relacionamentos de inclusão entre conjuntos são inclui ($\supset, \supseteq$) e é incluído ($\subset, \subseteq$),

$$\{1, 2, 3\} \supset \{1\}$$
$$\{1, 2, 3\} \supseteq \{1\}$$
$$\{1, 2, 3\} \subseteq \{1, 2, 3\}$$
$$\{1, 2\} \subset \{1, 2, 3\}$$

O símbolo $\in$ é usado para expressar o relacionamento de pertinência de elemento em conjunto,

$$1 \in \{1, 2, 3\}$$

O conjunto vazio é denotado pelo símbolo $\emptyset$, como em

$$\{1, 2\} \cap \{3, 4\} = \emptyset$$

Um conjunto pode também ser definido por um predicado, como

$$A = \{x \mid x \in \mathbb{N} \text{ e } x < 4\}$$

Um **alfabeto** (ou um vocabulário) é um conjunto de símbolos. Se A é um alfabeto, então a clausura de A, denotada A^*, é o conjunto de todas as *strings* — incluindo a *string* vazia, denotada ε — compostas a partir de símbolos de A. Por exemplo, para $A = \{0, 1\}$

$$A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$$

A^+ é o conjunto de todas *strings* compostas a partir de A sem incluir a *string* vazia, ou seja,

$$A^+ = A^* - \varepsilon$$

Uma **linguagem** é uma especificação de quais *strings*, dentre todas possíveis associadas a um conjunto de símbolos, são válidas (aceitáveis) para uma aplicação.

Linguagens simples podem ser definidas diretamente em termos de notação de conjuntos. Por exemplo, a linguagem

$$L = \{0^n 1^n \mid n \geq 0\}$$

denota que sentenças válidas na linguagem L estão no conjunto de todas as *strings* compostas de 0's e 1's iniciadas por n 0's seguidos por igual número de 1's. Observe que a *string* vazia está incluída nesta linguagem, ou seja, é uma sentença válida em L, uma vez que n pode ser 0.

Linguagens mais complexas, como aquelas envolvidas na programação de computadores, dificilmente podem ser representadas de forma tão simples como esta. Para tanto, é preciso especificar a linguagem através de uma gramática. Uma das partes de uma gramática é o conjunto de regras que indica como gerar sentenças em uma linguagem. Por exemplo, para a mesma linguagem L as seguintes regras podem ser utilizadas:

$$S \to 0S1$$
$$S \to \varepsilon$$

A interpretação destas regras é a seguinte. Para gerar uma sentença nesta linguagem, comece com o símbolo S e substitua-o por $0S1$ ou por ε. Cada vez que o símbolo S estiver presente na sentença gerada, ele pode ser novamente substituído por uma das duas opções. Qualquer *string* produzida dessa forma e que não contenha S será uma sentença da linguagem.

O processo de substituir o símbolo S pelo lado direito de uma regra é denominado uma **derivação**. Por exemplo, a sentença 0011 pode ser construída pela seqüência de derivações

$$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 0011$$

Todas as sentenças em L podem ser geradas a partir dessas duas regras. Qualquer *string* que não possa ser gerada a partir delas não será uma sentença em L.

3.1.2 Definição formal

Formalmente, a definição de uma linguagem é uma quádrupla (V_T, V_N, P, S) onde

V_T é um alfabeto cujos símbolos são conhecidos como **símbolos terminais**;

V_N é um alfabeto de **símbolos não-terminais**, sendo que $V_T \cap V_N = \emptyset$ e $V_T \cup V_N = V$, onde V é o conjunto de símbolos da linguagem;

P é um conjunto de regras ou **produções**, normalmente expressos na forma $\alpha \to \beta$, onde $\alpha \in V^+$ e $\beta \in V^*$. Normalmente, α é denominado o **lado esquerdo** da produção e β, o **lado direito**.

$S \in V_N$ é o ponto de partida na produção de qualquer sentença na linguagem, denominado **símbolo sentencial**, **símbolo não-terminal inicial** ou simplesmente **axioma**.

A gramática G_L para a linguagem L pode ser formalmente descrita por

$$G_L = (\{0, 1\}, \{S\}, P, S)$$

onde

$$P = \{S \to 0S1, S \to \varepsilon\}$$

Cada *string* que pode ser derivada a partir do símbolo sentencial é denominada uma **forma sentencial**. Por exemplo, em

$$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 000S111 \Longrightarrow 000111$$

as *strings* $0S1, 000S111, 000111$ são formas sentenciais. Uma **sentença** é uma forma sentencial sem símbolos não-terminais, como 000111.

Uma forma sentencial δ de uma gramática G é dita ser derivável de outra forma sentencial γ,

$$\gamma \underset{G}{\overset{+}{\Longrightarrow}} \delta$$

quando δ pode ser obtida a partir de γ a partir da aplicação de uma ou mais produções de G. Quando está claro a qual gramática a derivação se refere, a indicação da gramática pode ser omitida,

$$\gamma \overset{+}{\Longrightarrow} \delta$$

Assim, no exemplo acima pode-se afirmar que $0S1 \overset{+}{\Longrightarrow} 000S111$.

Similarmente, quando γ permite derivar δ por zero ou mais aplicações de produções de uma gramática, pode-se expressar este relacionamento por

$$\gamma \overset{*}{\Longrightarrow} \delta$$

onde a gramática a que se refere a derivação pode ser explicitada, se preciso,

$$\gamma \underset{G}{\overset{*}{\Longrightarrow}} \delta$$

No exemplo, $0S1 \overset{*}{\Longrightarrow} 000S111$ e $0S1 \overset{*}{\Longrightarrow} 0S1$.

Quando a forma sentencial δ de G é obtida de γ pela aplicação de exatamente uma produção, então diz-se que δ é **imediatamente derivável** de γ,

$$\gamma \underset{G}{\Longrightarrow} \delta$$

ou omitindo a gramática,

$$\gamma \Longrightarrow \delta$$

No exemplo, $0S1 \Longrightarrow 00S11$ e $000S111 \Longrightarrow 000111$

Em geral, buscar-se-á denotar símbolos terminais por *strings* de letras minúsculas e símbolos não-terminais por *strings* de letras maiúsculas. Formas sentenciais serão representadas por letras gregas.

Uma mesma linguagem pode ser gerada por mais de uma gramática. Quando duas gramáticas geram a mesma linguagem diz-se que elas são **equivalentes**.

3.1.3 Classificação

Pelo exemplo que foi apresentado acima pode parecer que o lado esquerdo de produções de uma gramática está restrito a um único símbolo não-terminal. Entretanto, da definição de gramáticas deve estar claro que este não é o caso, pois em uma produção

$$\alpha \rightarrow \beta$$

definiu-se que $\alpha \in V^+$ e $\beta \in V^*$.

Assim, a gramática $G_1 = (\{a\}, \{S, N, Q, R\}, P, S)$, onde os elementos de P são

$$\begin{aligned} S &\rightarrow QNQ \\ QN &\rightarrow QR \\ RN &\rightarrow NNR \\ RQ &\rightarrow NNQ \\ N &\rightarrow a \\ Q &\rightarrow \varepsilon \end{aligned}$$

é uma gramática perfeitamente válida, gerando sentenças na linguagem $\{a^{2^n} \mid n \geq 0\}$. Derivações típicas incluem

$$S \Rightarrow QNQ \stackrel{+}{\Rightarrow} a$$
$$S \Rightarrow QNQ \Rightarrow QRQ \Rightarrow QNNQ \stackrel{+}{\Rightarrow} aa$$

Restringindo o tipo de produções que podem aparecer em uma gramática, é possível definir classes especiais de gramáticas, tal como proposto na **hierarquia de Chomsky** descrita a seguir.

Uma gramática definida sem restrições quanto ao tipo de produções que ela pode conter é dita ser uma **gramática de tipo 0**.

Uma **gramática de tipo 1** obedece à propriedade $\mid \alpha \mid \leq \mid \beta \mid$ para todas as produções da forma $\alpha \rightarrow \beta$, onde $\mid \alpha \mid$ denota o comprimento (ou número de símbolos) em α e similarmente para $\mid \beta \mid$. Gramáticas de tipo 1 são também denominadas **gramáticas sensíveis ao contexto**.

Se a uma gramática de tipo 1 impõe-se a restrição adicional de que todos os lados esquerdos de produções tenham um único símbolo não-terminal, ou seja, $\mid \alpha \mid = 1$ para todas produções $\alpha \rightarrow \beta$ da gramática, então diz-se que a gramática é de **tipo 2**. Gramáticas de tipo 2 são também denominadas **gramáticas livres de contexto**.

Gramáticas de tipo 3 ou **gramáticas regulares** são gramáticas de tipo 2 onde as produções são apenas da forma

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aA \end{aligned}$$

quando a gramática é linear direita, ou da forma

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow Aa \end{aligned}$$

quando a gramática é linear esquerda.

Por simplicidade de notação, quando um lado esquerdo de uma produção pode levar a duas (ou mais) formas sentenciais distintas no lado direito, pode-se representar estas regras de forma compacta usando o símbolo $\mid$, lido **ou**, como em $A \rightarrow a \mid aA$ ou $A \rightarrow a \mid Aa$ correspondendo aos exemplos de gramáticas lineares direita e esquerda, respectivamente.

À hierarquia de gramáticas corresponde uma hierarquia de linguagens. Assim, se uma linguagem pode ser gerada por uma gramática de tipo 3 ela é dita ser uma linguagem de tipo 3.

3.1.4 Expressões regulares

Gramáticas de tipo 3 são adequadas para representar algumas "características locais" de linguagens de programação, tais como definição de formatos de constantes, identificadores e palavras da linguagem. Por

exemplo, uma típica definição de identificadores pode ser expressa em uma gramática

$$G = (\{letter, digit\}, \{IDENT, REST\}, P, IDENT)$$

com as seguintes produções em P:

$$\begin{aligned} IDENT &\rightarrow letter \mid letter\ REST \\ REST &\rightarrow letter \mid digit \mid letter\ REST \mid digit\ REST \end{aligned}$$

Uma forma equivalente de definir símbolos ou sentenças que podem ser gerados por gramáticas de tipo 3 é através de **expressões regulares**. Dado um alfabeto A, então são expressões regulares em A:

1. um elemento de A ou a *string* vazia.

   Se P e Q são expressões regulares em A, então também o serão:

2. PQ (P seguido de Q),
3. $P \mid Q$ (P ou Q),
4. P^* (0 ou mais ocorrências de P).

Expressões regulares podem ser vistas como expressões criadas a partir da aplicação de três operadores, concatenação, $\mid$ e $*$, sendo que $*$ tem a maior precedência e $\mid$, a menor. O operador $\mid$ é comutativo e associativo, enquanto que concatenação é associativa mas não comutativa.

No exemplo de identificadores, a expressão regular que descreve um identificador é

$$letter(letter \mid digit)^*$$

onde parênteses são utilizados para adequar a precedência dos operadores.

Diz-se que uma expressão regular gera um **conjunto regular**. Um dado conjunto regular pode ser gerado por mais de uma expressão regular. Entre as propriedades que tornam expressões regulares atraentes do ponto de vista de programadores de sistema estão

1. Dada uma *string*, existe um algoritmo para determinar se ela é parte de um dado conjunto regular, e
2. Existe um algoritmo para determinar se duas expressões regulares geram o mesmo conjunto regular.

Entretanto, expressões regulares têm muitas limitações. Por exemplo, não é possível expressar através de uma gramática de tipo 3 ou de uma expressão regular sentenças na forma de delimitadores balanceados, tais como (()(())). Entretanto, tais sentenças podem ser facilmente descritas por gramáticas livres de contexto, tal como

$$\begin{aligned} S &\rightarrow (S) \\ S &\rightarrow SS \\ S &\rightarrow \varepsilon \end{aligned}$$

Tais tipos de situações são recorrentes em linguagens de programação, onde pares de delimitadores — tais como (e), { e }, [e] em C, ou **begin** e **end** em Pascal — devem ocorrer de forma balanceada.

3.1.5 Gramáticas livres de contexto

Tradicionalmente, gramáticas livres de contexto têm sido utilizadas para realizar a análise sintática de linguagens de programação. Nem sempre é possível representar neste tipo de gramática restrições necessárias a algumas linguagens — por exemplo, exigir que todas as variáveis estejam declaradas antes de seu uso ou verificar se os tipos envolvidos em uma expressão são compatíveis. Entretanto, há mecanismos que podem ser incorporados às ações durante a análise — por exemplo, interações com tabelas de símbolos — que permitem complementar a funcionalidade da análise sintática.

A principal propriedade que distingüe uma gramática livre de contexto de uma gramática regular é a **auto-incorporação**. Uma gramática livre de contexto que não contenha auto-incorporação pode ser convertida em uma gramática regular.

Auto-incorporação é definida como se segue. Se uma gramática G tiver um símbolo não-terminal A para o qual

$$A \overset{*}{\underset{G}{\Longrightarrow}} \alpha_1 A \alpha_2$$

onde α_1 e α_2 são *strings* não vazias de símbolos terminais e/ou não-terminais, então diz-se que a gramática tem a propriedade da auto-incorporação. Esse tipo de regra é necessário para expressar sentenças com parênteses, colchetes ou delimitadores de blocos balanceados.

Representação de gramáticas livres de contexto

Toda gramática livre de contexto é equivalente a uma gramática na **Forma Normal de Chomsky**, onde todas as produções podem ser colocadas nas formas

$$A \to BC$$
$$A \to a$$

É possível representar esse tipo de gramática usando outros mecanismos de especificação no lugar da notação matemática de conjuntos. Dois mecanismos tipicamente utilizados são a notação BNF e a notação de grafos sintáticos.

A **notação BNF** (*Backus-Naur Form*) introduz uma forma de representação textual para descrever gramáticas livres de contexto. BNF foi inicialmente desenvolvida para especificar a linguagem Algol 60, uma das predecessoras da linguagem C.

O principal operador dessa linguagem é o operador binário ::=, que permite descrever as produções da gramática. O operando do lado esquerdo do operador é o símbolo não-terminal; do lado direito, a sua expansão, que pode conter símbolos terminais e não-terminais.

Na notação BNF, os símbolos não-terminais são delimitados por colchetes angulares, < e >. Por exemplo, a regra

$$< expr > \quad ::= \quad (< expr >)$$

define que uma expressão entre parênteses (o lado direito do operador) pode ser reduzido simplesmente a uma expressão (o símbolo não-terminal do lado esquerdo).

A notação BNF também introduz um conjunto de operadores destinados a simplificar a representação das regras da gramática, assim como `lex` introduz operadores para facilitar a especificação de expressões regulares.

O operador | (ou) permite expressar em uma mesma regra produções alternativas. Por exemplo, a regra

$$< S > \quad ::= \quad A|B$$

equivale às duas regras

$$< S > ::= A$$
$$< S > ::= B$$

O operador [ ] (opcional) permite expressar zero ou uma ocorrência do símbolo especificado. Por exemplo, a regra

$$< S > \quad ::= [\alpha]$$

equivale a

$$< S > ::= \varepsilon$$
$$< S > ::= \alpha$$

onde ε representa a *string* vazia.

O operador (|) (fatoração) permite expressar a combinação de símbolos alternativos, como em

$$< S > \quad ::= a(b \mid c)d$$

que equivale a

$$< S > ::= abd$$
$$< S > ::= acd$$

O operador * (repetição), assim como para expressões regulares, expressa 0 ou mais ocorrências do símbolo. Por exemplo, a regra

$$< S > \quad ::= \alpha^*$$

equivale a

$$< S > ::= \varepsilon$$
$$< S > ::= \alpha \ < S >$$

Assim, a ocorrência do padrão no lado direito de uma produção equivale a $\varepsilon \mid \alpha \mid \alpha\alpha \mid \alpha\alpha\alpha \mid \ldots$

O operador $\{\|\}^*$ (concatenação) permite expressar a repetição múltipla de símbolos concatenados. Por exemplo, a regra

$$< S > \quad ::= \{A \parallel B\}^*$$

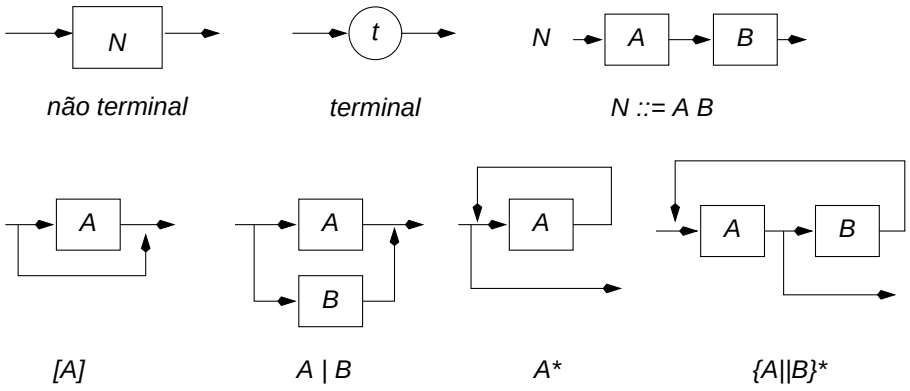
equivale a

$$< S > ::= A < t >$$
$$< t > ::= \varepsilon$$
$$< t > ::= BA < t >$$

Outro tipo de notação usual para gramáticas é a notação de **grafos sintáticos**. Esta notação tem o mesmo poder de expressão de BNF, porém define uma representação visual para as regras de uma gramática livre de contexto.

Na notação de grafos sintáticos, símbolos terminais são representados por círculos e símbolos não-terminais, por retângulos. Setas são utilizadas para indicar a seqüência de expansão de um símbolo não-terminal. A Figura 3.1 apresenta alguns exemplos de regras já descritas para o BNF expressas através dessa notação.

Figura 3.1 Grafos sintáticos.



3.2 Análise léxica

A análise léxica pode ser encarada como a primeira etapa do processo de compilação. Nesta etapa, o programa fonte é encarado como uma seqüência de caracteres que deverão ser agrupados e identificados como palavras reservadas da linguagem (em C, por exemplo, **main**, **int**, **for**), constantes (`123`, `0x1F`, `'A'`), identificadores (`myvar`, `Str1`).

Inicialmente serão apresentados alguns aspectos genéricos no reconhecimento de símbolos. Posteriormente será apresentada uma ferramenta amplamente utilizada na programação de sistemas para gerar automaticamente analisadores léxicos.

3.2.1 Autômatos finitos

Os símbolos que deverão ser reconhecidos na análise léxica são representáveis por expressões regulares (Seção 3.1.4) ou equivalentemente por gramáticas regulares (tipo 3). Existe também uma correspondência unívoca entre expressões (ou gramáticas) regulares e *autômatos finitos*, máquinas que podem ser utilizadas para reconhecer *strings* de uma dada linguagem.

Um autômato finito tem um conjunto de **estados**, alguns dos quais são denominados estados finais. À medida que caracteres da *string* de entrada são lidos, o controle da máquina passa de um estado a outro, segundo um conjunto de **regras de transição** especificadas para o autômato. Se após o último carácter o autômato encontra-se em um dos estados finais, a *string* foi reconhecida (ou seja, pertence à linguagem). Caso contrário, a *string* não pertence à linguagem aceita pelo autômato.

Formalmente, um autômato é descrito por cinco características,

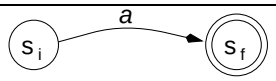
1. um conjunto finito de estados, K,
2. um alfabeto de entrada finito, Σ,
3. um conjunto de transições, δ,
4. um estado inicial, S, onde $S \in K$,
5. um conjunto de estados finais, F, onde $F \subseteq K$

sendo portanto representável por uma quíntupla $M = (K, \Sigma, \delta, S, F)$.

As transições são representadas por triplas (s_i, Σ_T, s_f), onde s_i é um estado inicial da transição, Σ_T é o conjunto de símbolos do alfabeto (caracteres) que disparam esta transição quando o estado corrente é s_i e s_f será o novo estado corrente do autômato após a transição.

Autômatos são usualmente representados na forma de um grafo dirigido, onde estados são representados por círculos, sendo que estados finais são representados por círculos duplos, e as transições por arestas rotuladas com os símbolos que disparam a transição entre os dois estados conectados (Figura 3.2).

Figura 3.2 Representação gráfica de $(s_i, \{a\}, s_f)$, denotando a transição de s_i (estado não final) para s_f (um estado final) disparada pelo símbolo a.



Uma outra forma de representar um autômato, mais apropriada para fins de processamento automático, é através de **tabelas de transição**. Uma tabela de transição é uma matriz na qual as colunas representam os estados do autômato e as linhas os símbolos do alfabeto. Cada entrada na matriz indica qual o estado final de uma transição a partir do estado indicado na coluna através do símbolo indicado na linha. Assim, a transição expressa na Figura 3.2 poderia ser equivalentemente representada por

	...	s_i	...
a	...	s_f	...
...	...	...	...

Assim, colunas da tabela de transição representam o estado corrente e as linhas, o símbolo corrente. Uma entrada na tabela (cruzamento entre linha e coluna) pode ser vazia, indicando que não há transição possível, ou ter o próximo estado que é atingido pela transição corrente.

Autômatos finitos podem ser determinísticos, quando para cada combinação de estado e entrada existe uma única transição aplicável, ou não-determinísticos em caso contrário. É possível mostrar que a cada autômato finito não-determinístico corresponde um autômato finito determinístico que aceita a mesma linguagem.

3.2.2 Construção do autômato finito não-determinístico

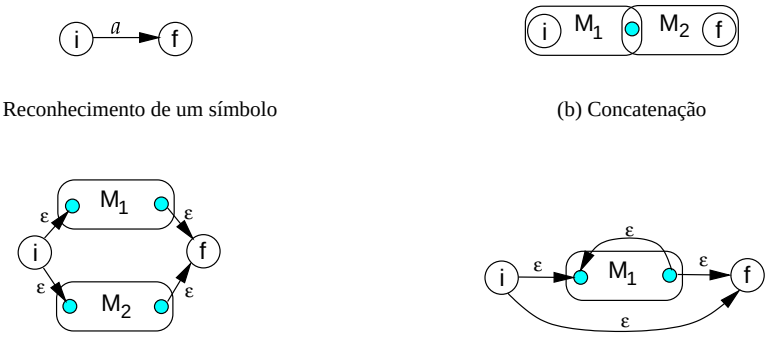
Na seqüência, apresenta-se o **Algoritmo de Thompson** para a construção de um autômato finito para reconhecer uma dada expressão regular. Esse procedimento determina como construir um autômato finito não-determinístico para reconhecer sentenças de uma gramática regular. Posteriormente, apresenta-se o procedimento para converter esse autômato para uma máquina determinística.

O primeiro passo do procedimento é decompor a expressão regular que define as sentenças que deverão ser reconhecidas em termos de suas relações elementares:

- um símbolo do alfabeto da linguagem;
- concatenação, RS;
- alternativa, $R|S$;
- repetição, R^*.

Uma vez que a expressão regular tenha sido estruturada em termos das relações elementares, é preciso construir um autômato para reconhecer cada uma das partes da expressão. Para cada relação elementar, a estrutura de um autômato correspondente é determinada.

Figura 3.3 Autômatos elementares para a construção de Thompson.



(a) Reconhecimento de um símbolo

(b) Concatenação

(c) Alternativa

(d) Repetição

Para reconhecer um símbolo a do alfabeto da linguagem Σ, o autômato correspondente é composto simplesmente por um estado inicial que atinge um estado final através de uma transição pela ocorrência do símbolo a (Figura 3.3a).

Se a relação elementar for a concatenação de duas expressões regulares, RS, é preciso compor as duas máquinas M_1 e M_2 que reconhecem R e S, respectivamente. Para a máquina composta, o estado final de M_1 é combinado com o estado inicial de M_2 (Figura 3.3b).

Para reconhecer a relação elementar que estabelece a alternativa entre duas expressões regulares, $R|S$, a forma de compor as duas respectivas máquinas M_1 e M_2 é através da introdução de um novo estado inicial. Este estado tem transições para os estados iniciais de cada uma das máquinas M_1 e M_2 através da *string* vazia. Similarmente, um novo estado final é introduzido, o qual pode ser atingido com transições pela *string* vazia a partir dos estados finais das duas máquinas originais (Figura 3.3c).

A última relação elementar a ser considerada é a repetição, R^*, cuja máquina de reconhecimento deve ser derivada da máquina M_1 que reconhece R. Também neste caso novos estados inicial e final são introduzidos. Para reconhecer zero ocorrências de R, há uma transição pela *string* vazia direta do novo estado inicial para o novo estado final. Para reconhecer uma ocorrência de R, há transições pela *string* vazia entre o novo estado inicial e o estado inicial original, assim como entre o estado final original e o novo estado final. Finalmente, o reconhecimento de várias ocorrências de R dá-se através de uma transição pela *string* vazia do estado final para o estado inicial da máquina original M_1 (Figura 3.3d).

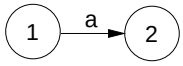
A título de exemplo, considere a construção de um um autômato finito não-determinístico para reconhecer sentenças descritas pela expressão regular $R = (a|b)^*abb$. O primeiro passo é decompor a expressão em termos de suas relações elementares:

$$\begin{aligned} R_1 &= a & R_2 &= b \\ R_3 &= R_1|R_2 & R_4 &= {R_3}^* \\ R_5 &= R_4R_1 & R_6 &= R_5R_2 \end{aligned}$$

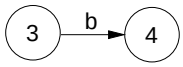
$$R = R_6R_2$$

Uma vez determinadas as expressões regulares elementares que compõem a expressão regular sob análise, é possível construir os autômatos que reconhecem cada uma dessas expressões elementares.

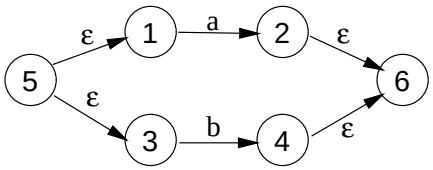
Para reconhecer a expressão R_1, constrói-se a máquina M_1 que reconhece o símbolo a, usando a construção apresentada na Figura 3.3a:



Similarmente, para reconhecer R_2, constrói-se a máquina M_2 que reconhece o símbolo b:

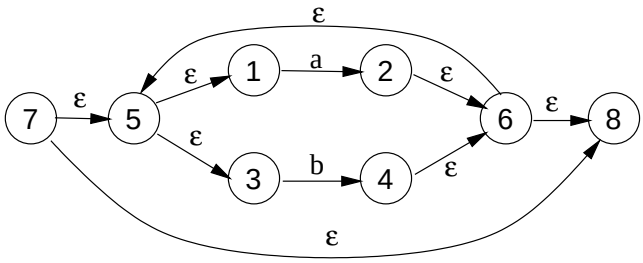


A expressão regular R_3 é a composição pela alternativa das expressões R_1 e R_2. O autômato para reconhecer R_3 é construído pela combinação das máquinas M_1 e M_2 conforme a estratégia apresentada na Figura 3.3c:



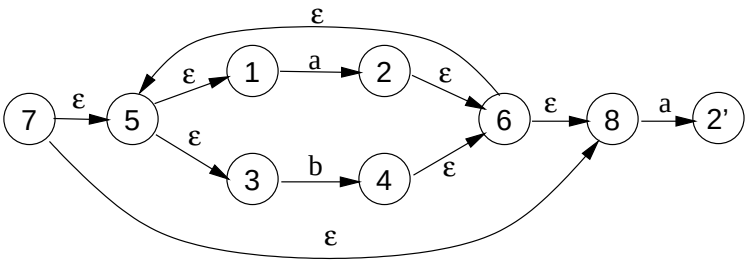
Neste caso, os estados 5 e 6 são os novos estados introduzidos respectivamente como o estado inicial e o estado final da nova máquina M_3.

O reconhecimento da expressão regular R_4 é feito pelo autômato que reconhece zero ou mais ocorrências de R_3, ou seja, a máquina M_4 é construída a partir da máquina M_3 conforme a estratégia apresentada na Figura 3.3d:



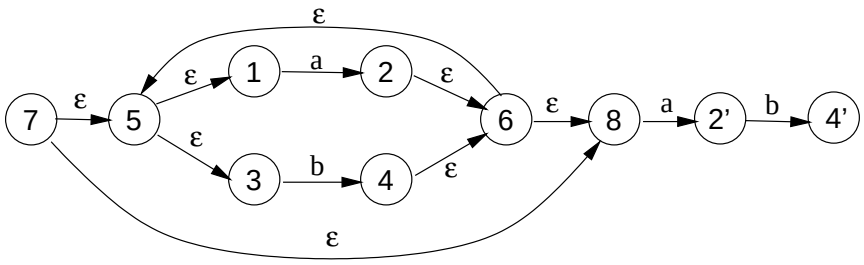
Observe-se que os novos estados inicial e final de M_4 passam a ser os estados 7 e 8, respectivamente.

Como a expressão R_5 é formada pela concatenação de R_4 com R_1, a máquina M_5 deve combinar a máquina M_4 com uma nova instância da máquina M_1 segundo a estratégia apresentada na Figura 3.3b:



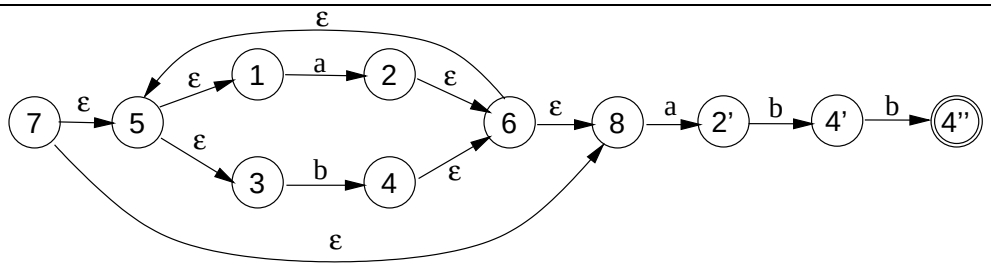
Observe que o estado inicial da nova máquina M_1', $1'$, foi combinado com o estado final da máquina M_4, enquanto que o estado final de M_1', $2'$, passou a ser o estado final de M_5.

Similarmente, R_6 é uma concatenação de R_5 e R_2. Combinando a máquina M_5 com uma nova instância da máquina que reconhece R_2, obtém-se para a máquina M_6:



Finalmente, a expressão completa é uma concatenação de R_6 com R_2. Combinando M_6 com uma máquina M_2'' obtém-se a máquina que reconhece a expressão regular completa, que é apresentada na Figura 3.4. Para essa máquina, o estado inicial é o estado inicial de M_6, ou seja, o estado 7, e o estado final é o estado final de M_2'', o estado $4''$.

Figura 3.4 Autômato não-determinístico que reconhece $(a|b)^*abb$



3.2.3 Conversão para autômato finito determinístico

Autômatos finitos não-determinísticos precisam lidar com situações de ambigüidade, como no caso de um estado a partir do qual parte mais de uma transiçao vazia. É possível eliminar essas ambigüidades através da construção de um autômato finito determinístico que é equivalente a um autômato finito não-determinístico.

O procedimento aqui apresentado é freqüentemente denominado de **construção de subconjuntos**. Na descrição a seguir, o termo "estado original" refere-se a um estado do autômato não-determinístico, enquanto o termo "novo estado" refer-se a estados do autômato determinístico.

A base desse procedimento é criar novos estados que representem todas as possibilidades de estados originais em um dado momento da análise da sentença em processo de reconhecimento. Para tal, define-se o conjunto ϵ^* associado a um conjunto de estados de um autômato não determinístico. A ϵ^* (lê-se épsilon-clausura) é o conjunto que inclui cada um dos estados indicados e todos os estados alcançáveis a partir dele através de transições por *strings* vazias. O resultado é um conjunto de estados que irá representar um novo estado no autômato determinístico.

O procedimento da construção de subconjuntos começa pela computação da ϵ^* do conjunto que contém o estado inicial original. Neste caso, obtém-se um conjunto de estados que irá representar o novo estado inicial, pois o conjunto resultante inclui o estado inicial original.

Cada estado novo que é criado precisa ser posteriormente analisado. Para tanto, constrói-se uma lista de estados não-analisados que, inicialmente, contém apenas o novo estado inicial.

O procedimento prossegue com a análise dos novos estados ainda não analisados. Para cada novo estado s nessa lista, é preciso analisar o que acontece quando o próximo símbolo da sentença for α, onde α é cada um dos símbolos do alfabeto sob consideração na expressão regular. Por exemplo, se α_1 é um dos símbolos que pode ocorrer na sentença, então analisa-se, para cada um dos estados originais em s, quais seriam os estados

originais resultantes pela transição pelo símbolo α_1. A ϵ^* desse conjunto de estados resultantes gera um novo estado t (que eventualmente já pode ser um estado novo já existente). O autômato finito determinístico irá conter a transição $s \xrightarrow{\alpha} t$.

Em outras palavras, para obter t, obtenha o conjunto T_α dos estados originais que podem ser alcançados através de uma transição pelo símbolo α a partir de cada estado original no conjunto s e compute $\epsilon^*(T_\alpha)$. Se t for um novo estado, inclua-o na lista de estados não-analisados. Se algum elemento de t for um estado final no autômato original, então t será um estado final no novo autômato.

A análise dos novos estados ainda não analisados deve prosseguir até que essa lista torne-se vazia. Quando isso ocorre, o procedimento está encerrado e o autômato finito determinístico está definido.

Como exemplo, considere a conversão do autômato não-determinístico da Figura 3.4 para um autômato determinístico através da aplicação desse procedimento.

O conjunto de estados originais daquele autômato é $T_0 = \{7\}$. Portanto, o novo estado inicial, s_0, será dado por $\epsilon^*(T_0)$, ou

$$s_0 = \{1, 3, 5, 7, 8\}$$

A lista de estados não-analisados contém s_0. Para esse estado, é preciso analisar as transições resultantes para cada um dos dois símbolos do alfabeto, a e b.

O estado s_0 contém dois estados originais a partir dos quais existem transições com o símbolo a, 1 e 8. Os estados originais atingidos por essas transições são 2 e $2'$. Portanto, $T_1 = \{2, 2'\}$ e o estado atingido a partir de s_0 pela transição através do símbolo a será $s_1 = \epsilon^*(T_1)$, ou

$$s_1 = \{1, 2, 2', 3, 5, 6, 8\}$$

Similarmente, a partir de s_0 pelo símbolo b atinge-se o conjunto de estados $T_2 = \{4\}$ e, portanto, $s_2 = \epsilon^*(T_2)$ resulta em

$$s_2 = \{1, 3, 4, 5, 6, 8\}$$

O estado s_0 foi retirado da lista de estados não-analisados, mas dois novos estados — s_1 e s_2 — foram incluídos. Esses dois novos estados precisam então ser analisados.

Da análise de s_1, obtém-se que a transição pelo símbolo a também levará ao estado s_1, enquanto que a transição pelo símbolo b leva aos estados originais $T_3 = \{4, 4'\}$, resultando em um novo estado, s_3, gerado por $\epsilon^*(T_3)$,

$$s_3 = \{1, 3, 4, 4', 5, 6, 8\}$$

A análise de s_2 indica que a transição pelo símbolo a leva ao estado s_1, enquanto que a transição pelo símbolo b leva ao próprio estado s_2. Nenhum novo estado é criado.

O estado s_3 permanece na lista de estados não-analisados. A transição a partir dele pelo símbolo a leva também ao estado s_1. Pelo símbolo b, o conjunto de estados originais resultantes é $T_4 = \{4, 4''\}$; assim, um novo estado, s_4, é gerado a partir do cômputo de $\epsilon^*(T_4)$,

$$s_4 = \{1, 3, 4, 4'', 5, 6, 8\}$$

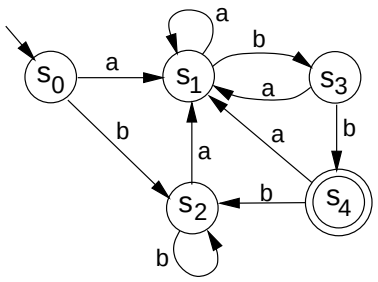
Esse estado, que é incluído na lista de estados não-analisados, é um estado final, pois contém o estado original final $4''$.

Finalmente, a análise de s_4 indica que a transição pelo símbolo a leva ao estado s_1, enquanto que a transição pelo símbolo b leva ao estado s_2.

Após a análise de s_4, a lista de estados não-analisados ficou vazia, indicando a conclusão do processo de conversão do autômato. O autômato determinístico resultante, que reconhece sentenças expressas por $(a|b)^*abb$, é apresentado na Figura 3.5a.

Para fins computacionais, a representação mais adequada para autômatos é aquela na forma de tabelas de transição. Para esse autômato, a tabela correspondente é apresentada na Figura 3.5b.

Figura 3.5 Autômato finito determinístico que reconhece $(a|b)^*abb$



(a) Representação gráfica

	s_0	s_1	s_2	s_3	s_4
a	s_1	s_1	s_1	s_1	s_1
b	s_2	s_3	s_2	s_4	s_2

(b) Tabela de transição

3.2.4 Minimização de estados

Muitas vezes é possível ter mais de um autômato reconhecendo o mesmo conjunto de sentenças, em geral através da presença de estados redundantes. O procedimento apresentado nesta seção permite detectar tais situações de redundância, reduzindo o número de estados do autômato ao mínimo possível.

O procedimento de minimização de estados de um autômato $M = (S, \Sigma, \delta, s_0, F)$ inicia-se pela criação de uma partição inicial Π_0 do conjunto de estados S contendo dois subconjuntos, um com todos os estados finais, F, e outro contendo os estados não-finais, $S - F$.

O procedimento prossegue com o refinamento da partição corrente Π_i para a criação de uma nova partição Π_{i+1}. Para tanto, analisa-se cada um dos conjuntos que compõem Π_i buscando a criação de novas partições que sejam equivalentes sob transições. Partindo de cada grupo G de estados em Π_i, um novo subgrupo G' conterá dois estados s e t se e somente se, para todos os símbolos do alfabeto de entrada, s e t tenham transições para um mesmo grupo de Π_i.

A partir desse processo de refinamento dos grupos de Π_i, o conjunto dos subgrupos resultantes forma uma nova partição Π_{i+1}. O procedimento deve ser repetido até que a nova partição gerada seja igual à partição corrente, situação que indica que novos refinamentos não são mais possíveis.

Para construir o autômato com o número mínimo de estados, associa-se a cada subgrupo da partição final atingida por esse procedimento um estado da nova máquina M'. Eventualmente, o procedimento pode gerar "estados mortos", que não são estados finais e contêm transição apenas para si próprios, ou estados não-alcançáveis a partir do estado inicial. Tais estados podem ser eliminados do autômato resultante.

Aplicando esse procedimento ao autômato da Figura 3.5, tem-se que a partição inicial é

$$\Pi_0 = \{\{s_0, s_1, s_2, s_3\}, \{s_4\}\}$$

A segunda partição, $G_2 = \{s_4\}$, é irredutível e não será mais analisada — este será o estado final do autômato minimizado. Para aplicar o procedimento de refinamento à primeira partição, $G_1 = \{s_0, s_1, s_2, s_3\}$, deve-se analisar para cada estado qual o grupo resultante pela transição para cada um dos dois símbolos do alfabeto, a e b:

	s_0	s_1	s_2	s_3
a	G_1	G_1	G_1	G_1
b	G_1	G_1	G_1	G_2

Observa-se que o grupo G_1 pode ser particionado em dois novos grupos, $G_3 = \{s_0, s_1, s_2\}$ e $G_4 = \{s_3\}$. Assim, a nova partição a ser analisada é

$$\Pi_1 = \{\{s_0, s_1, s_2\}, \{s_3\}, \{s_4\}\}$$

Novamente há um grupo que não pode ser mais reduzido, mas o grupo G_3 precisa ainda ser analisado. Aplicando novamente o procedimento de refinamento considerando a nova partição $\Pi_1 = \{G_3, G_4, G_2\}$, o resultado para o refinamento do grupo G_3 é

	s_0	s_1	s_2
a	G_3	G_3	G_3
b	G_3	G_4	G_3

Nesse grupo, s_0 e s_2 permanecem com comportamento equivalente, portanto fazem parte do mesmo grupo, G_5; porém, s_1 faz parte de um novo grupo G_6. Assim, a partição resultante é

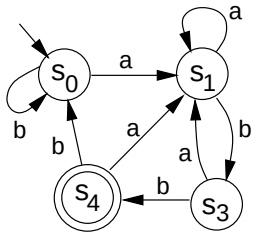
$$\Pi_2 = \{\{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\}\}$$

Como $\Pi_1 \neq \Pi_2$, o grupo G_5 (o único que ainda pode ser refinado) precisa ser analisado. A análise dos estados desse grupo resulta em

	s_0	s_2
a	G_6	G_6
b	G_5	G_5

Como não há novo particionamento, o procedimento de refinamento está concluído. O resultado obtido é que o estado s_2 pode ser incorporado ao estado s_0, resultando no autômato equivalente com número mínimo de estados. Esse autômato é apresentado na Figura 3.6.

Figura 3.6 Autômato com número mínimo de estados que reconhece $(a|b)^*abb$



3.3 Analisadores léxicos

Um **analisador léxico**, ou *scanner*, é um programa que implementa um autômato finito, reconhecendo (ou não) *strings* como símbolos válidos de uma linguagem.

A implementação de um analisador léxico requer uma descrição do autômato que reconhece as sentenças da gramática ou expressão regular de interesse. Com essa descrição, é possível oferecer os seguintes procedimentos auxiliares para o analisador léxico:

ESTADO-INICIAL, que recebe como argumento a referência para o autômato e retorna o seu estado inicial;

ESTADO-FINAL, que recebe como argumentos a referência para o autômato e a referência para o estado corrente. O procedimento retorna *true* se o estado especificado é elemento do conjunto de estados finais do autômato, ou *false* caso contrário; e

PRÓXIMO-ESTADO, que recebe como argumento a referência para o autômato, para o estado corrente e para o símbolo sendo analisado. O procedimento consulta a tabela de transições e retorna o próximo estado do autômato, ou o valor nulo se não houver transição possível.

A sentença a ser reconhecida é estruturada como uma lista de símbolos, que é passada como argumento para o analisador léxico juntamente com a referência para o autômato.

O analisador léxico inicia sua operação definindo o estado inicial como o estado corrente. Obtém então o próximo símbolo (que inicialmente é o primeiro símbolo) da sentença σ. Se não houver próximo símbolo, é preciso verificar se o estado corrente é um estado final. Se for, o procedimento retorna *true*, indicando que a sentença foi reconhecida pelo autômato. Se o estado corrente não for um estado final e não houver mais símbolos na sentença, então não houve reconhecimento e o procedimento retorna *false*.

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 sentença não foi reconhecida e o procedimento deve encerrar, retornando *false*.

Esse algoritmo é apresentado no Algoritmo 3.1, que determina se a *string* σ pertence à linguagem reconhecida pelo autômato M.

Algoritmo 3.1 Algoritmo do analisador léxico.

```
SCANNER(M, σ)
 1  s ← ESTADO-INICIAL(M)
 2  while true
 3  do if ISEMPTY(σ)
 4        then if ESTADO-FINAL(M, s)
 5                then return true
 6                else  return false
 7        else  c ← REMOVEFIRST(σ)
 8              s ← PRÓXIMO-ESTADO(M, s, c)
 9              if s = NIL
10                then return false
```

3.3.1 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 3.1, 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.

3.3.2 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 3.1.4, 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 2.2). 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`.

3.3.3 Integração com código C

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`, conforme definido para a manipulação de arquivos em C (Seção 2.8). 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 3.2 poderia ter sido utilizado.

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

Algoritmo 3.2 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 }
```

Considere como exemplo que a especificação `lex` apresentada no Algoritmo 3.2 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 na Seção 4.6.1.)

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.

3.4 Análise sintática

Na Seção 3.2 mostrou-se como proceder para o reconhecimento de seqüências de símbolos que satisfazem a uma gramática regular (tipo 3) usando autômatos finitos. Este tipo de procedimento é adequado para identificar os símbolos básicos que compõem uma linguagem, mas não para identificar a forma como esses símbolos

devem ser combinados para que façam sentido na linguagem — o processo de *análise sintática*. Para tanto, é preciso usar gramáticas com maior poder de expressão.

Gramáticas tipo 2, ou gramáticas livres de contexto, são adequadas para representar boa parte das características de linguagens de programação. Embora não todas construções de programação sejam passíveis de representação por esse tipo de gramática, através de reconhecedores de sentenças livres de contexto e algumas estratégias heurísticas é possível automatizar a análise sintática. O uso e a construção de programas analisadores sintáticos, ou *parsers*, são os objetos desta seção.

3.4.1 Reconhecimento de sentenças

O reconhecimento de sentenças, ou *parsing*, é o procedimento que verifica se uma dada sentença pertence à linguagem gerada por uma gramática. Este procedimento é essencial para um compilador, que deve reconhecer e validar expressões de diversos tipos — declarações, expressões aritméticas, construções de controle de execução — no processo de construção de um código executável equivalente à expressão reconhecida.

Considere uma gramática que define um subconjunto de expressões aritméticas aceitas por alguma linguagem de programação, apresentada na Figura 3.7.

Figura 3.7 Gramática para reconhecimento de expressões.

E	$\rightarrow$	$E + T$	1
E	$\rightarrow$	T	2
T	$\rightarrow$	$T \times F$	3
T	$\rightarrow$	F	4
F	$\rightarrow$	(E)	5
F	$\rightarrow$	id	6

A primeira regra dessa gramática determina que a soma de uma expressão e um termo é também uma expressão. Pela segunda regra, um único termo é também uma expressão. Pela terceira regra, o produto de um termo e um fator é um termo válido. A quarta regra estabelece que um único fator é também um termo válido. Pela quinta regra, uma expressão entre parênteses é um fator. Finalmente, a sexta regra estabelece que um identificador é um fator.

Nessa gramática, id é um símbolo terminal, ou seja, um *token*, assim como os símbolos $+$, $\times$, (e). Através de uma gramática regular seria possível determinar o que é um identificador para a linguagem; por exemplo,

$$id \rightarrow L\ L*$$
$$L \rightarrow x|y|z$$

Considere o procedimento para o reconhecimento da expressão $(x + y)z$. No procedimento de análise léxica, a expressão sob análise seria traduzida para $(id+id)id$. O procedimento de análise sintática deve então aplicar as regras da gramática para verificar se a sentença é válida ou não. Se é possível derivar a sentença a partir do símbolo sentencial (no caso, E), então a sentença é reconhecida. Caso contrário, a sentença é inválida e deve ser rejeitada.

Usando inicialmente um procedimento informal para esse reconhecimento, é possível estabelecer que a sentença $(id + id)id$ pode ser obtida a partir de $(F + F)F$ pela aplicação da sexta regra, ou seja,

$$(F + F)F \stackrel{+}{\Longrightarrow} (id + id)id$$

Pela aplicação da quarta regra, obtém-se que a expressão pode ainda ser reduzida a $(T + T)F$, ou seja,

$$(T + T)F \stackrel{+}{\Rightarrow} (F + F)F$$

Pela segunda regra,

$$(E + T)F \Rightarrow (T + T)F$$

e pela primeira regra,

$$(E)F \Rightarrow (E + T)F$$

Finalmente, pela quinta regra, obtém-se

$$FF \Rightarrow (E)F$$

Nesse momento, não há mais nenhuma regra que possa ser aplicada para obter a redução da sentença até o símbolo sentencial. Assim, o analisador indicaria que a sentença $(x + y)z$ não faz parte da linguagem descrita pela gramática especificada.

Por outro lado, para a sentença $(x + y) \times z$, o analisador reconheceria $(id + id) \times id$ como uma sentença válida:

$$\begin{aligned} E &\Rightarrow T \\ &\Rightarrow T \times F \\ &\Rightarrow F \times F \\ &\Rightarrow (E) \times F \\ &\Rightarrow (E + T) \times F \\ &\Rightarrow (T + T) \times F \\ &\Rightarrow (T + F) \times F \\ &\Rightarrow (F + F) \times F \\ &\Rightarrow (F + F) \times id \\ &\Rightarrow (F + id) \times id \\ &\Rightarrow (id + id) \times id \end{aligned}$$

Nesse caso, a sentença é reconhecida como válida pois foi possível derivar a sentença a partir do símbolo sentencial usando a aplicação das regras 2, 3, 4, 5, 1, 2, 4, 4, 6, 6, e 6. Essa seqüência de regras aplicadas na derivação da sentença sob análise é conhecida como a **seqüência de reconhecimento** da sentença.

3.4.2 Derivações canônicas

Mesmo considerando expressões simples como essa do exemplo, há diversas opções para a seqüência de aplicação das regras que poderiam levar ou não ao reconhecimento da sentença, o que dificultaria a automatação desse processo. Portanto, é importante ter formas sistemáticas de aplicações dessas regras que levem a uma conclusão sobre a validade ou não da sentença.

Essa forma sistemática de aplicação das regras de uma gramática é estabelecida através das derivações canônicas. Duas formas de derivação canônica são estabelecidas, as derivações mais à esquerda e mais à direita.

Na **derivação mais à esquerda** (*leftmost derivation*), a opção é aplicar uma regra da gramática ao símbolo não-terminal mais à esquerda da forma sentencial sendo analisada. A correspondente seqüência de regras aplicadas é denominada a **seqüência de reconhecimento mais à esquerda**, ou *leftmost parse*.

Similarmente, na **derivação mais à direita** (*rightmost derivation*) o símbolo não-terminal mais à direita é sempre selecionado para ser substituído usando alguma regra da gramática. No caso desse tipo de derivação, a

seqüência de reconhecimento mais à direita, ou *rightmost parse*, é o reverso da seqüência de regras associada à derivação.

A Figura 3.10 apresenta as duas derivações canônicas para a sentença $(id + id) \times id$ segundo a gramática da Figura 3.7. A Figura 3.10a apresenta a derivação mais à esquerda, destacando em cada derivação qual o símbolo não-terminal que está sendo analisado. Para esse caso, a seqüência de reconhecimento mais à esquerda associada é

$$2, 3, 4, 5, 1, 2, 4, 6, 4, 6, 6.$$

Por sua vez, a seqüência de reconhecimento mais à direita, associada à derivação canônica da Figura 3.10b, é dada por

$$6, 4, 2, 6, 4, 1, 5, 4, 6, 3, 2.$$

Figura 3.10 Derivações canônicas para a sentença $(id + id) \times id$.

$$
\begin{aligned}
\mathbf{E} &\Rightarrow \mathbf{T} \\
&\Rightarrow \mathbf{T} \times F \\
&\Rightarrow \mathbf{F} \times F \\
&\Rightarrow (\mathbf{E}) \times F \\
&\Rightarrow (\mathbf{E} + T) \times F \\
&\Rightarrow (\mathbf{T} + T) \times F \\
&\Rightarrow (\mathbf{F} + T) \times F \\
&\Rightarrow (id + \mathbf{T}) \times F \\
&\Rightarrow (id + \mathbf{F}) \times F \\
&\Rightarrow (id + id) \times \mathbf{F} \\
&\Rightarrow (id + id) \times id
\end{aligned}
$$

(a) derivação mais à esquerda

$$
\begin{aligned}
\mathbf{E} &\Rightarrow \mathbf{T} \\
&\Rightarrow T \times \mathbf{F} \\
&\Rightarrow \mathbf{T} \times id \\
&\Rightarrow \mathbf{F} \times id \\
&\Rightarrow (\mathbf{E}) \times id \\
&\Rightarrow (E + \mathbf{T}) \times id \\
&\Rightarrow (E + \mathbf{F}) \times id \\
&\Rightarrow (\mathbf{E} + id) \times id \\
&\Rightarrow (\mathbf{T} + id) \times id \\
&\Rightarrow (\mathbf{F} + id) \times id \\
&\Rightarrow (id + id) \times id
\end{aligned}
$$

(b) derivação mais à direita

Observe que uma dada produção é utilizada o mesmo número de vezes nas duas derivações canônicas. Assim, a regra 1 foi usada uma vez; a regra 2, duas; a regra 3, uma; a regra 4, três; a regra 5, uma; e a regra 6, três vezes.

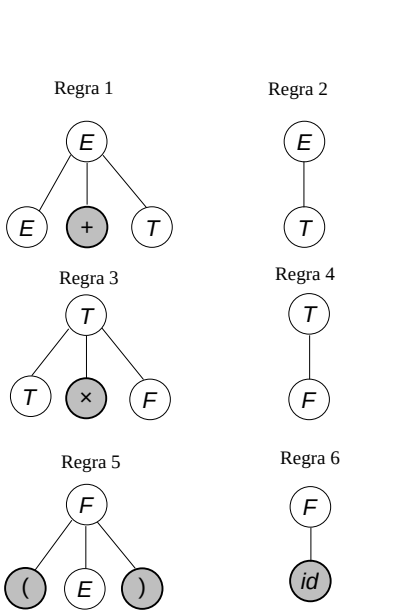
3.4.3 Árvores gramaticais

Uma outra forma de representar a derivação associada ao reconhecimento de uma sentença é através da utilização de uma **árvore gramatical**. Nesse tipo de representação

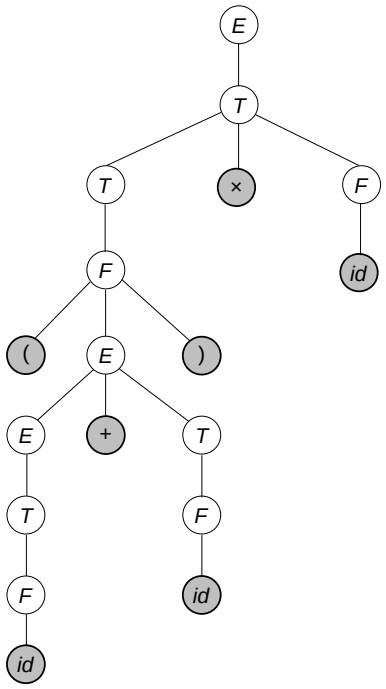
1. a raiz da árvore é o símbolo sentencial;
2. as folhas da árvore são símbolos terminais, na seqüência em que ocorrem na sentença analisada; e
3. os nós intermediários da árvore correspondem a símbolos não terminais, onde um nó cujo rótulo é P com filhos $s_1, s_2, \ldots, s_n$ pode ocorrer apenas se houver uma regra $P \rightarrow s_1\ s_2 \ldots\ s_n$ na gramática.

Considere como exemplo a gramática da Figura 3.7. Uma árvore gramatical para uma sentença dessa gramática só poderá conter nós na forma especificada na Figura 3.11a, onde os nós correspondentes aos símbolos terminais estão destacados. A árvore gramatical para a sentença $(id + id) \times id$ é apresentada na Figura 3.11b.

Figura 3.11 Árvore gramatical.



(a) Nós para a gramática de expressões



(b) Árvore para $(id + id) \times id$

O problema de reconhecimento de uma sentença em uma gramática é essencialmente um problema de, dada uma sentença composta por *tokens* da linguagem, encontrar uma seqüência de reconhecimento mais à esquerda *(leftmost parse)*, ou uma seqüência de reconhecimento mais à direita *(rightmost parse)*, ou uma árvore gramatical para a expressão.

Na maior parte das situações, seqüências de reconhecimento e árvores gramaticais são únicas. Quando uma sentença pode ser gerada em uma gramática por mais de uma derivação mais à esquerda ou mais à direita ou tem mais de uma árvore gramatical, diz-se que a gramática que gera esta sentença é **ambígua**.

Considere a produção em uma gramática que representa um comando condicional de uma linguagem de programação, onde a cláusula *else* é opcional:

$$cond \rightarrow \text{if } (\ expr\ )\ cmnd$$
$$cond \rightarrow \text{if } (\ expr\ )\ cmnd \text{ else } cmnd$$

onde adicionalmente uma das definições possíveis de $cmnd$ fosse estabelecida pela produção

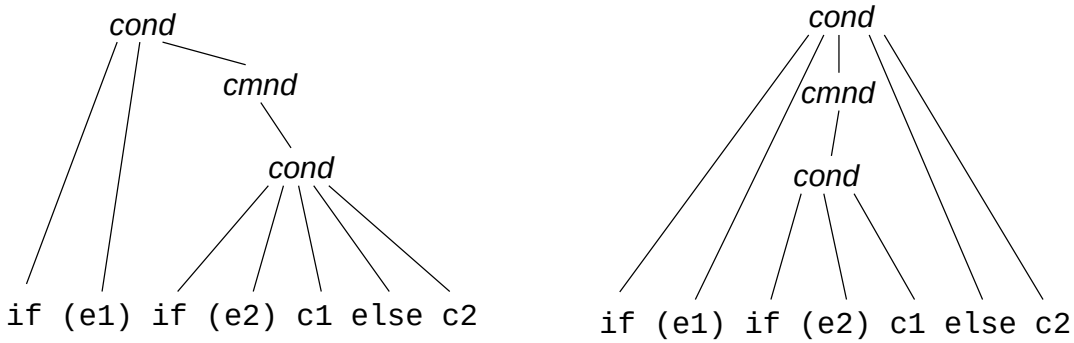
$$cmnd \rightarrow cond$$

Para essas produções, o trecho de código

```
if ( e1 ) if ( e2 ) c1 else c2
```

poderia dar margem a duas árvores gramaticais distintas, uma associada ao padrão adotado em linguagens de programação que associa o `else` ao último `if` possível (Figura 3.12a) e outra, não-usual porém gramaticalmente correta, que assume que o último `if` é que tem a cláusula *else* vazia (Figura 3.12b).

Figura 3.12 Árvores gramaticais para comando *if-else*.



(a) Interpretação usual

(b) Outra interpretação possível

Em alguns casos é possível transformar uma gramática ambígua em uma gramática sem ambiguidades. Entretanto, dada uma gramática qualquer não existe um algoritmo que indique se a gramática é ambígua ou não.

3.5 Analisadores sintáticos

Um analisador sintático para uma gramática G é um programa que aceita como entrada uma sentença (uma lista de símbolos α) e constrói para a sentença sua árvore gramatical (ou equivalentemente uma seqüência de derivação) ou, caso a sentença não pertença à linguagem descrita por G, uma indicação de erro.

Duas técnicas básicas para a contrução de analisadores sintáticos são a construção ascendente ou a construção descendente. Na **contrução ascendente** (*bottom-up*), o analisador sintático varre a sentença buscando a aplicar produções que permitam substituir seqüências de símbolos da sentença pelo lado esquerdo das produções, até alcançar como único símbolo restante o símbolo sentencial.

O Algoritmo 3.3 ilustra a estratégia de reconhecimento de sentença baseado em construção ascendente da árvore sintática. Esse algoritmo recebe como entrada uma representação da gramática G e a lista α de símbolos terminais que compõem a sentença. A saída é uma indicação se a sentença pertence *(true)* ou não *(false)* à gramática G. Para a descrição desse algoritmo, as seguintes funções são definidas:

Σ, que recebe uma gramática G como argumento e retorna o seu símbolo sentencial; e

MATCH(G, α) retorna uma nova lista de símbolos gerada a partir da aplicação de alguma regra de G à sentença α. Para tanto, esse procedimento analisa se para a sentença α, composta pela seqüência de símbolos $\beta x_1 x_2 \ldots x_n \mu$ (onde eventualmente β e μ podem ser a *string* vazia), há na gramática alguma regra aplicável $\chi \rightarrow x_1 x_2 \ldots x_n$. Se houver, o valor de retorno do procedimento é a lista $\beta\chi\mu$; caso contrário, o procedimento retorna uma lista vazia.

Algoritmo 3.3 Contrução ascendente.

```
ASCENDINGPARSER(G, α)
1  declare s : Symbol
2  while true
3  do s ← REMOVEFIRST(α)
4     if s = Σ(G) ∧ ISEMPTY(α)
5        then return true
6        else INSERT(α, s)
7             α ← MATCH(G, α)
8             if ISEMPTY(α)
9               then return false
```

Esse algoritmo apresenta duas condições de término possíveis: a primeira quando a sentença pode ser reduzida ao símbolo sentencial da gramática (condição de sucesso) e a segunda quando a sentença não está reduzida ao símbolo sentencial e não há mais regras aplicáveis à sentença (condição de rejeição).

Na **construção descendente** (*top-down*), o objetivo é iniciar a análise com uma lista que contém inicialmente apenas o símbolo sentencial; a partir da análise dos símbolos presentes na sentença, busca-se aplicar regras que permitam expandir os símbolos na lista até alcançar a sentença desejada.

Na construção descendente, o objetivo é obter uma derivação mais à esquerda para uma sentença. Em termos de árvores gramaticais, a construção descendente busca a construção de uma árvore a partir da raiz usando pré-ordem para definir o próximo símbolo não-terminal que deve ser considerado para análise e expansão.

Pela forma como a técnica de construção descendente opera, ela não pode ser aplicada a gramáticas com produções recursivas à esquerda, ou seja, que contenham regras da forma

$$A \rightarrow A\beta$$

A limitação é que a análise descendente de tal tipo de produção poderia levar a uma recursão infinita na análise pela tentativa de expandir sempre a mesma regra sem consumir símbolo algum da entrada.

É possível transformar uma produção recursiva à esquerda em uma recursiva à direita que descreve as mesmas sentenças através da seguinte técnica. Sejam β e δ duas seqüências de símbolos que não sejam iniciadas pelo símbolo não-terminal A e sejam as produções para A:

$$A \rightarrow A\ \beta$$
$$A \rightarrow \delta$$

Através da introdução de um novo símbolo não-terminal A', as mesmas sentenças descritas pelas produções acima podem ser descritas pelas produções recursivas à direita:

$$A \rightarrow \delta\ A'$$
$$A' \rightarrow \beta\ A'$$
$$A' \rightarrow \varepsilon$$

Nos dois casos, as sentenças são formadas por uma ocorrência de δ no início seguida por zero ou mais ocorrências de β.

Os primeiros compiladores usavam essencialmente dois tipos de analisadores sintáticos. Analisadores baseados em **precedência de operadores** utilizam a técnica de construção ascendente combinada com informação sobre a precedência e associatividade de operadores da linguagem para guiar suas ações, sendo adequados à análise de expressões aritméticas. Analisadores do tipo **descendentes recursivos** implementam a técnica de construção descendente através de um conjunto de rotinas mutualmente recursivas para realizar a análise, sendo normalmente utilizados para outros comandos que não expressões aritméticas.

3.5.1 Analisador sintático preditivo

Esta seção descreve a construção de um analisador sintático baseado na técnica de construção descendente. Este programa, que realiza a análise sintática preditiva não recursiva, recebe como argumentos uma descrição da gramática G e a sentença α, expressa na forma de uma lista de símbolos terminada com um símbolo delimitador \$, não-pertencente aos símbolos da gramática.

O ponto crítico nesse procedimento é saber escolher, dado um símbolo não-terminal que pode ser expandido e os próximos símbolos da sentença, qual deve ser a produção da gramática que deve ser aplicada na expansão. A tabela sintática para a gramática, cuja construção é descrita na seqüência, contém essa informação essencial à execução do algoritmo.

Construção da tabela sintática

A **tabela sintática** é a estrutura de apoio ao reconhecimento de sentenças pela técnica de construção descendente que tem como chave um par de símbolos. O primeiro componente da chave é um símbolo não-terminal, que corresponde ao símbolo que estará sendo analisado pelo algoritmo de reconhecimento da sentença. O segundo componente da chave é um símbolo da sentença, ou seja, um símbolo terminal ou o delimitador de fim de seqüência \$. O valor associado a esse par de símbolos é a produção da gramática a ser aplicada para prosseguir com o reconhecimento da sentença.

Para construir a tabela sintática para uma gramática qualquer G, deve-se analisar cada uma das produções $A \rightarrow \alpha$ de G. Inicialmente, deve-se obter o conjunto de símbolos terminais que podem iniciar uma cadeia a partir de α. Se α for um símbolo terminal, então esse conjunto é composto apenas por esse próprio símbolo. Caso contrário, as possíveis expansões de α devem ser analisadas até que os símbolos terminais ou a *string* vazia sejam alcançados.

Caso símbolos terminais sejam alcançados, a tabela sintática recebe a entrada com o valor $A \rightarrow \alpha$ para cada chave A, t onde t é cada um dos símbolos terminais que podem ser alcançados desde α. Caso a *string* vazia seja um dos resultados possíveis para a expansão de α, é preciso analisar também as possíveis expansões dos símbolos à direita do símbolo corrente na produção.

Dois procedimentos auxiliares são definidos para a construção dessa tabela. O primeiro, STF, computa os símbolos terminais associados ao início das expansões de cada um dos símbolos X da gramática. Esse procedimento é descrito no Algoritmo 3.4.

Algoritmo 3.4 Cômputo dos primeiros símbolos terminais de um símbolo gramatical.

```
STF(G, X)
 1  declare s, t : List
 2  declare i : Integer
 3  if TERMINAL(G, X)
 4    then INSERT(s, X)
 5    else if X → ε é uma produção de G
 6           then INSERT(s, ε)
 7         if X → Y1Y2...Yk é uma produção de G
 8           then i ← 1
 9                while i ≤ k ∧ REMOVEFIRST(STF(G, Yi)) = ε
10                do i ← i + 1
11                if i = k + 1
12                  then INSERT(s, ε)
13                  else CONCAT(s, STF(G, Yi))
14  return s
```

O cômputo de STF pode também ser aplicado a uma cadeia de símbolos, sendo que neste caso o valor resultante é o primeiro cômputo de STF aplicado a cada símbolo da seqüência tal que o resultado não tenha sido ε. Caso o cômputo de STF para todos os símbolos da cadeia resulte em ε, este também será o resultado final.

O outro procedimento auxiliar deve computar, para cada símbolo não-terminal da gramática G, o conjunto de símbolos terminais que podem estar imediatamente à direita do símbolo especificado em alguma forma sentencial. Essa informação é mantida em uma lista $seguinte(\alpha)$, onde α é o símbolo de interesse. Para construir essas listas, as seguintes regras devem ser aplicados até que se esgotem as possibilidades de acrescentar algo às listas:

Regra 1: O símbolo sentencial da gramática pode ter como próximo símbolo o delimitador de fim de sentença; insira o símbolo \$ na lista $seguinte(\Sigma(G))$.

Regra 2: Se existir uma produção em G da forma $A \to \alpha B \beta$, então todos os símbolos terminais que podem iniciar a expansão de β podem aparecer após B; insira em $seguinte(B)$ o conteúdo de STF(β) sem incluir ε, se estiver presente.

Regra 3: Se existir uma produção em G da forma $A \to \alpha B$, então B termina a expansão de A. O mesmo pode ocorrer para uma produção da forma $A \to \alpha B \beta$ onde a expansão de β pode levar à *string* vazia ε. Em qualquer um desses casos, tudo que está em $seguinte(A)$ deve ser incluído em $seguinte(B)$.

Com esses procedimentos, a construção da tabela sintática para uma gramática G procede como se segue. Para cada produção $A \to \chi$ em G:

1. Compute STF(G, χ). Para cada símbolo x dessa lista, acrescente a produção $A \to \chi$ como o valor da tabela para o par de chaves $[A, x]$.
2. Caso STF(G, χ) contenha ε, acrescente a produção $A \to \chi$ à tabela para o par de chaves $[A, y]$ para cada y em $seguinte(A)$.

Como exemplo, considere a construção do analisador sintático preditivo para a gramática da Figura 3.7. Como essa gramática é recursiva à esquerda, o primeiro passo nessa construção é construir a gramática equivalente sem esse tipo de recursão. O resultado da aplicação da técnica para eliminar a recursão à esquerda resulta na seguinte gramática:

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \\
E' &\to \varepsilon \\
T &\to FT' \\
T' &\to \times FT' \\
T' &\to \varepsilon \\
F &\to (E) \\
F &\to id
\end{aligned}
$$

Para esta gramática, o cômputo de STF() para cada um dos símbolos não-terminais resulta em:

$$
\begin{aligned}
\text{STF}(E) = \text{STF}(T) = \text{STF}(F) &= (, id \\
\text{STF}(E') &= +, \varepsilon \\
\text{STF}(T') &= \$, \varepsilon
\end{aligned}
$$

Como STF() para um símbolo terminal é o próprio símbolo, esses valores não são aqui apresentados.

A aplicação das regras para a construção das listas $seguinte()$ resulta, para cada símbolo não-terminal:

$$seguinte(E) = seguinte(E') = \$,)$$
$$seguinte(T) = seguinte(T') = +, \$,)$$
$$seguinte(F) = \times, +, \$,)$$

A construção da tabela sintática para essa gramática analisa cada uma das suas produções:

P1. $E \rightarrow T\ E'$
Para essa produção, $\text{STF}(TE') = \text{STF}(T)$, que resulta na lista com os símbolos (e id. Portanto, na tabela sintática as entradas $[E, (]$ e $[E, id]$ farão referência à produção P1.

P2. $E' \rightarrow +\ T\ E'$
O cômputo de $\text{STF}(+TE')$ resulta em $+$, ou seja, haverá uma referência para P2 na entrada $[E', +]$ da tabela sintática.

P3. $E' \rightarrow \varepsilon$
$\text{STF}(\varepsilon) = \varepsilon$; portanto, a segunda regra para a construção da tabela deve ser aplicada. Como $seguinte(E') = \$,)$, as entradas correspondentes à $[E', \$]$ e $[E',)]$ farão referência à produção P3.

P4. $T \rightarrow F\ T'$
Como $\text{STF}(FT') = \text{STF}(F) = (, id$, as entradas para $[T, (]$ e $[T, id]$ farão referência à P4.

P5. $T' \rightarrow \times\ F\ T'$
Neste caso, $\text{STF}(\times FT') = \times$. Portanto, a entrada $[T', \times]$ terá a referência para P5.

P6. $T' \rightarrow \varepsilon$
Novamente a segunda regra deve ser aplicada. Como $seguinte(T') = +, \$,)$, as entradas com referência à P6 na tabela serão $[T', +]$, $[T', \$]$ e $[T',)]$.

P7. $F \rightarrow (\ E\ )$
Apenas a entrada $[F, (]$ fará referência a esta produção, pois $\text{STF}((E)) = (.$

P8. $F \rightarrow id$
Apenas a entrada $[F, id]$ fará referência a esta produção, pois $\text{STF}(id) = id$.

A Tabela 3.1 apresenta esses resultados na forma de um arranjo bidimensional, resumindo os resultados encontrados. Nessa tabela, a primeira coluna contém os símbolos não-terminais da gramática, que corresponderão aos símbolos que estarão no topo da pilha do analisador sintático. A primeira linha contém os símbolos que podem ser encontrados na sentença, ou seja, os símbolos terminais da gramática e o símbolo indicador de fim de sentença.

Tabela 3.1 Tabela sintática para o analisador preditivo.

	$+$	$\times$	$($	$)$	id	\$
E			P1		P1	
E'	P2			P3		P3
T			P4		P4	
T'	P6	P5		P6		P6
F			P7		P8	

Para algumas gramáticas, pode ser que a tabela sintática apresente mais que uma produção por chave, o que reduz a aplicabilidade desse tipo de analisador — seria necessário manter um registro do estado do analisador em pontos de múltipla escolha para eventualmente retornar a esse estado e tentar a outra alternativa, em caso de insucesso na escolha anterior. Gramáticas ambíguas e gramáticas recursivas à esquerda são exemplos de gramáticas que produziriam tabelas sintáticas com múltiplas produções para uma chave de par de símbolos.

Gramáticas com tabelas sintáticas sem múltiplas definições são denominadas **gramáticas LL(1)**, indicando que a varredura da sentença ocorre da esquerda para a direita *(Left-to-right)* e que é utilizada a derivação canônica mais à esquerda *(Leftmost derivation)*. O número entre parênteses indica quantos símbolos da sentença precisam ser analisados *(lookahead)* para a tomada de decisão no processo de reconhecimento.

Uma gramática com duas produções $A \to \alpha$ e $A \to \beta$ é LL(1) se apresentar as seguintes propriedades:

1. α e β não podem derivar ao mesmo tempo seqüências que tenham início pelo mesmo símbolo terminal;
2. Apenas um dos dois, α ou β, podem derivar ε; e
3. Se uma das produções deriva ε, a outra não pode derivar qualquer seqüência de símbolos que tenha início com um símbolo presente em $seguinte(A)$.

Algoritmo de reconhecimento de sentença

O Algoritmo 3.5 descreve o analisador sintático preditivo não-recursivo que reconhece sentenças para a gramática especificada. Durante o processamento, o programa utiliza uma estrutura de pilha para acomodar os símbolos sob análise.

O programa utiliza dois procedimentos auxiliares. O primeiro, SELECT, recebe como argumentos a descrição da gramática, um símbolo não-terminal (que está sob análise para expansão) e o próximo símbolo da sentença, retornando uma lista com os símbolos do lado direito da produção que deve ser aplicada para expandir o símbolo analisado, ou o valor nulo no caso de erro. Internamente, esse procedimento faz uso da informação contida na tabela sintática.

O outro procedimento auxiliar é TERMINAL, que recebe como argumentos a descrição da gramática e um de seus símbolos, retornando verdadeiro se este for um símbolo terminal ou falso, se for não-terminal.

A seqüência de produções usadas para reconhecer a sentença é registrada em uma lista ℓ, que é retornada ao final do algoritmo. Um valor de retorno nulo indica que a sentença não foi reconhecida para a gramática indicada.

Considere o reconhecimento da sentença $(id + id) \times id$ usando esse algoritmo. Na condição inicial, a pilha contém o delimitador \$ e o símbolo sentencial E; a lista com a sentença contém os sete símbolos terminais e o delimitador \$:

Pilha	Lista
$\$\ E$	$(\ id\ +\ id\ )\ \times\ id\ \$$

O analisador, verificando que o topo da pilha (na representação, o símbolo mais à direita) é um símbolo não-terminal, busca uma produção apropriada para a entrada $[E, (]$, pois (é o primeiro elemento da lista que contém a sentença. Em consulta à tabela sintática, ele verifica que P1, $E \to TE'$, é essa produção. Assim, os símbolos E' e T são colocados na pilha, enquanto a sentença permanece inalterada:

Produção	Pilha	Lista
$E \to TE'$	$\$\ E'\ T$	$(\ id\ +\ id\ )\ \times\ id\ \$$

Da mesma forma, as iterações seguintes do algoritmo levam a

Produção	Pilha	Lista
$T \to FT'$	$\$\ E'\ T'\ F$	$(\ id\ +\ id\ )\ \times\ id\ \$$
$F \to (\ E\ )$	$\$\ E'\ T'\ )\ E\ ($	$(\ id\ +\ id\ )\ \times\ id\ \$$

Algoritmo 3.5 Analisador sintático baseado na técnica de construção descendente.

```
PREDPARSER(G, α)
 1  declare t, f : Symbol
 2  declare s : Stack
 3  declare r, ℓ : List
 4  PUSH(s, $)
 5  PUSH(s, Σ(G))
 6  repeat
 7        t ← POP(s)
 8        f ← REMOVEFIRST(α)
 9        if TERMINAL(G, t) ∨ t = $
10          then if t ≠ f
11                 then return NIL
12          else r ← SELECT(G, t, f)
13               if r = NIL
14                 then return NIL
15                 else INSERT(α, f)
16                      APPEND(ℓ, t → r)
17                      repeat
18                            PUSH(s, REMOVELAST(r))
19                        until ISEMPTY(r)
20    until t = $
21  return ℓ
```

Na iteração seguinte, o programa encontra no topo da pilha um símbolo terminal. Como esse símbolo é idêntico ao primeiro símbolo da sentença, essa não é uma condição de erro; os dois símbolos são simplesmente retirados de suas respectivas estruturas:

Produção	Pilha	Lista
—	$\$\ E'\ T'\ )\ E$	$id\ +\ id\ )\ \times\ id\ \$$

Para as iterações seguintes, o topo da pilha contém símbolos não terminais e a expansão continua com a utilização das produções P1, P4 e P8:

Produção	Pilha	Lista
$E \to TE'$	$\$\ E'\ T'\ )\ E'\ T$	$id\ +\ id\ )\ \times\ id\ \$$
$T \to FT'$	$\$\ E'\ T'\ )\ E'\ T'\ F$	$id\ +\ id\ )\ \times\ id\ \$$
$F \to id$	$\$\ E'\ T'\ )\ E'\ T'\ id$	$id\ +\ id\ )\ \times\ id\ \$$

Novamente a condição de remoção de símbolos acontece, levando a

Produção	Pilha	Lista
—	$\$\ E'\ T'\ )\ E'\ T'$	$+\ id\ )\ \times\ id\ \$$

O resultado das iterações seguintes é apresentado na Tabela 3.2, até a condição final de aceitação quando resta apenas o símbolo delimitador no topo da pilha e na sentença.

3.5.2 Analisador de deslocamento e redução

A estratégia de análise sintática por deslocamento e redução é baseada na técnica de reconhecimento de sentenças por construção ascendente. Nessa estratégia, símbolos terminais da sentenças são lidos um a um;

Tabela 3.2 Final da seqüência de reconhecimento da sentença.

Produção	Pilha	Lista
$T' \rightarrow \varepsilon$	$\$\ E'\ T'\ )\ E'$	$+\ id\ )\ \times\ id\ \$$
$E' \rightarrow +\ T\ E'$	$\$\ E'\ T'\ )\ E'\ T\ +$	$+\ id\ )\ \times\ id\ \$$
—	$\$\ E'\ T'\ )\ E'\ T$	$id\ )\ \times\ id\ \$$
$T \rightarrow F\ T'$	$\$\ E'\ T'\ )\ E'\ T'\ F$	$id\ )\ \times\ id\ \$$
$F \rightarrow id$	$\$\ E'\ T'\ )\ E'\ T'\ id$	$id\ )\ \times\ id\ \$$
—	$\$\ E'\ T'\ )\ E'\ T'$	$)\ \times\ id\ \$$
$T' \rightarrow \varepsilon$	$\$\ E'\ T'\ )\ E'$	$)\ \times\ id\ \$$
$E' \rightarrow \varepsilon$	$\$\ E'\ T'\ )$	$)\ \times\ id\ \$$
—	$\$\ E'\ T'$	$\times\ id\ \$$
$T' \rightarrow \times\ F\ T'$	$\$\ E'\ T'\ F\ \times$	$\times\ id\ \$$
—	$\$\ E'\ T'\ F$	$id\ \$$
$F \rightarrow id$	$\$\ E'\ T'\ id$	$id\ \$$
—	$\$\ E'\ T'$	$\$$
$T' \rightarrow \varepsilon$	$\$\ E'$	$\$$
$E' \rightarrow \varepsilon$	$\$$	$\$$

a cada símbolo lido, o analisador decide se prossegue com a leitura (desloca) ou se é possível aplicar uma produção aos símbolos previamente lidos para substituí-los por um símbolo não-terminal da gramática (reduz). O procedimento conclui com sucesso se toda a sentença foi lida e apenas o símbolo sentencial resulta da aplicação de todas as reduções.

Da mesma forma que para o analisador preditivo, a decisão a ser tomada pelo analisador é apoiada em uma estrutura de dados gerada a partir da análise da gramática. A construção dessa estrutura é apresentada a seguir e seu uso no reconhecimento de sentenças, na seqüência.

Construção da tabela SR

A base para a operação de reconhecimento neste tipo de analisador é a **Tabela de Deslocamento e Redução** ou Tabela SR *(shift-reduce)*, as duas ações básicas desempenhadas pelo analisador durante a análise de uma sentença. Essa tabela determina, a partir do último símbolo resultante das ações efetuadas sobre os símbolos já lidos (que pode ser tanto um símbolo terminal como um não-terminal) e do próximo símbolo terminal presente na sentença, se o próximo passo da análise é ler o próximo símbolo, reduzir os símbolos já lidos ou se não há ação a ser tomada.

Para construir essa tabela, as produções da gramática são analisadas para obter as **relações de precedência simples** entre os símbolos gramaticais, acrescidos do delimitador de sentenças \$. Para dois símbolos X e Y, as relações de precedência definidas são $X \preceq Y$ (X confere precedência a Y) e $X \succ Y$ (X tem precedência sobre Y).

Para obter as relações "confere precedência" entre os símbolos de uma gramática G, as seguintes regras são aplicadas:

Regra 1: $\$ \preceq \Sigma(G)$, ou seja, o símbolo delimitador de sentença confere precedência ao símbolo sentencial da gramática.

Regra 2: $X \preceq Y$ se existe alguma produção de G na forma $\alpha \rightarrow \beta X Y \mu$, ou seja, onde X aparece à esquerda de Y no lado direito da produção.

Regra 3: $X \preceq Y$ se $X \preceq \alpha$, onde α é um símbolo não-terminal, e existe alguma produção para α em G onde Y é o primeiro símbolo do lado direito, $\alpha \rightarrow Y\mu$.

As relações "tem precedência sobre" entre os símbolos de G são obtidas pela aplicação das seguintes regras:

Regra 4: $\Sigma(G) \succ \$$, ou seja, o símbolo sentencial da gramática tem precedência sobre o delimitador de sentença.

Regra 5: $X \succ Y$ se, para algum símbolo não-terminal α, $\alpha \preceq Y$ e existe uma produção para α em G cujo último símbolo é X, $\alpha \rightarrow \beta X$.

Regra 6: $X \succ Y$ se, para algum símbolo não-terminal α, $\alpha \succ Y$ e existe uma produção para α em G cujo último símbolo é X, $\alpha \rightarrow \beta X$.

Regra 7: $X \succ Y$ se, para algum símbolo não-terminal α, $X \succ \alpha$ e existe uma produção para α em G cujo primeiro símbolo é Y, $\alpha \rightarrow Y\mu$.

Deve-se observar que $X \preceq Y$ não implica que $Y \succ X$. Também pode ser verdade para alguma gramática que $X \preceq Y$ ao mesmo tempo que $Y \preceq X$ ou $X \succ Y$. Assim, não se deve confundir as propriedades dessas relações, apesar da semelhança de notação, com aquelas das bem conhecidas relações de ordem "menor ou igual" e "maior".

Uma vez determinado o conjunto completo das relações de precedência simples, é possível construir diretamente a tabela de deslocamento e redução. Nessa tabela, a chave é composta por dois símbolos $[X, a]$ da gramática estendida com o delimitador de sentença. O primeiro estará associado ao estado corrente da análise da sentença, podendo portanto ser um símbolo qualquer. O segundo símbolo da chave é um símbolo terminal, que estará associado ao próximo símbolo da sentença. Se $X \preceq a$, então o valor na tabela para a chave $[X, a]$ conterá a indicação de que a próxima ação deve ser a leitura do próximo símbolo da sentença. Se $X \succ a$, então o valor da chave indicará que a próxima ação do analisador deve ser a redução dos últimos símbolos lidos pela aplicação de uma produção da gramática.

Considere como exemplo a construção da tabela de deslocamento e redução para a gramática da Figura 3.7. Pela Regra 1, obtém-se a primeira relação de precedência,

$$\$ \preceq E$$

A Regra 2 estabelece que há a mesma relação de precedência entre símbolos contíguos no lado direito das produções da gramática. A aplicação dessa regra deriva as relações

$$\begin{array}{lll} E \preceq + & + \preceq T & T \preceq \times \\ \times \preceq F & (\preceq E & E \preceq) \end{array}$$

Para a aplicação da Regra 3, é preciso analisar todas as relações, obtidas pela aplicação das Regras 1 e 2 e da própria Regra 3, onde o símbolo do lado direito é um símbolo não-terminal, até que novas relações não possam mais ser estabelecidas. Por exemplo, como $\$ \preceq E \wedge E \rightarrow T$, então $\$ \preceq T$; como esta relação tem do lado direito um símbolo não-terminal, deve-se analisar quais símbolos iniciam o lado direito das produções para T, o que leva à obtenção da relação $\$ \preceq F$.

A aplicação da Regra 3 gera as seguintes relações de precedência:

$$\begin{array}{llll} \$ \preceq E & \$ \preceq T & + \preceq T & + \preceq F \\ \times \preceq (& \times \preceq id & (\preceq E) & (\preceq T \\ \$ \preceq F & + \preceq (& + \preceq id & (\preceq F \\ \$ \preceq (& \$ \preceq id & (\preceq (& (\preceq id \end{array}$$

Destas relações, três já haviam sido anteriormente derivadas pelas outras regras.

As regras seguintes permitem estabelecer as relações "tem precedência sobre", que estarão associadas a reduções durante a análise. A primeira dessas regras, Regra 4, deriva a relação

$$E \succ \$$$

Para a aplicação da Regra 5, é preciso analisar as relações "confere precedência a" que tenham símbolos não-terminais do lado esquerdo, que são $E \preceq +$, $T \preceq \times$ e $E \preceq)$, e quais símbolos terminam as produções para esses símbolos não-terminais. Essa análise permite derivar as relações

$$T \succ + \qquad F \succ \times \qquad T \succ)$$

A aplicação da Regra 6 requer a análise das relações "tem precedência sobre" derivadas da aplicação das regras 4 e 5 e da própria Regra 6. As relações de interesse são aquelas que têm do lado esquerdo um símbolo não-terminal. A aplicação da regra deriva as relações

$$\begin{array}{llll} T \succ \$ & F \succ + &) \succ \times & id \succ \times \\ F \succ) & F \succ \$ &) \succ +) & id \succ + \\ ) \succ) & id \succ) &) \succ \$ & id \succ \$ \end{array}$$

A última regra, Regra 7, deriva novas relações de "tem precedência sobre" a partir dessas relações onde há um símbolo não-terminal no lado direito. Particularmente para esse exemplo não há nenhuma relação dessa forma e portanto nenhuma nova relação pode ser derivada.

Concluída a análise das relações de precedência para a gramática, é possível construir a sua tabela de deslocamento e redução (Tabela 3.3) usando as relações $X \preceq Y$ ou $X \succ Y$ onde Y é um símbolo terminal. Nessa tabela, a entrada "S" *(shift)* indica que a ação deve ser de leitura do próximo símbolo da sentença, enquanto que a entrada "R" determina a redução dos símbolos já lidos. Para as entradas em branco não há uma ação que possa ser tomada que leve ao reconhecimento da sentença.

Tabela 3.3 Tabela de deslocamento e redução.

	id	$+$	$\times$	$($	$)$	$\$$
$\$$	S			S		
E		S		S	S	R*
T		R	S		R	R
F		R	R		R	R
id		R	R		R	R
$+$	S			S		
$\times$	S			S		
$($	S			S		
$)$		R	R		R	R

Observe na Tabela 3.3 que a entrada para $[E, \$]$ recebeu uma marcação especial, pois essa situação — os símbolos já analisados resultaram no símbolo sentencial e a sentença chegou ao fim — determina a condição de reconhecimento da sentença.

Para algumas gramáticas, a construção da tabela de deslocamento e redução pode levar a situações onde mais de uma ação poderia ser tomada para um dado estado e próximo símbolo da sentença. Essa tabela não terá entrada duplicadas se a gramática for uma **gramática de operadores**, para a qual nenhuma produção tem do lado direito dois símbolos não-terminais adjacentes, e se nenhuma produção tiver ε do lado direito.

Algoritmo do analisador de deslocamento e redução

O analisador de deslocamento e redução trabalha com duas estruturas de dados auxiliares, além da tabela de deslocamento e redução. A primeira delas é a lista de símbolos terminais a analisar, que contém inicialmente a sentença submetida à análise delimitada ao final pelo símbolo $. A outra estrutura é uma pilha com os símbolos já analisados, os quais podem ter sido eventualmente substituídos por símbolos não-terminais pela aplicação de produções da gramática. Portanto, a pilha pode conter qualquer símbolo, terminal ou não-terminal, do alfabeto da gramática.

Há dois principais procedimentos auxiliares utilizados nesse algoritmo. O primeiro é NEXTACTION(), que determina qual a próxima ação em função do estado corrente do analisador e da consulta à tabela de deslocamento e redução. Seus argumentos são, além da referência à gramática, um símbolo que corresponde ao topo da pilha e o próximo símbolo da sentença. Seu valor de retorno foi definido ser do tipo *Action*, que pode assumir os valores S, R, R^* ou o valor nulo para indicar uma situação de erro.

O outro procedimento auxiliar usado na descrição do algoritmo é REDUCE(), que recebe como argumentos a referência à gramática e à pilha. Esse procedimento retira do topo da pilha os símbolos que podem ser utilizados para combinar, na ordem correta, com o lado direito da produção aplicável no estado atual. Seu valor de retorno é o símbolo resultante, aquele do lado esquerdo da produção aplicada.

O Algoritmo 3.6, que descreve o procedimento do analisador, recebe como argumentos a descrição da gramática G e a lista α com a sentença a ser analisada. O valor de retorno é verdadeiro, se a sentença pertence à gramática, ou falso, caso contrário.

Algoritmo 3.6 Analisador sintático por deslocamento e redução.

```
PARSESR(G, α)
 1  declare t, f : Symbol
 2  declare p : Stack
 3  declare n : Action
 4  t ← $
 5  f ← REMOVEFIRST(α)
 6  n ← NEXTACTION(G, t, f)
 7  while n ≠ R*
 8  do PUSH(p, t)
 9     if n = S
10        then t ← f
11             f ← REMOVEFIRST(α)
12        else if n = R
13                then t ← REDUCE(G, p)
14                else return false
15     n ← NEXTACTION(G, t, f)
16  return true
```

A aplicação desse algoritmo é ilustrada através do reconhecimento da sentença $(id + id) \times id$. No estado inicial, a pilha está vazia e α contém todos os símbolos da sentença:

$$\alpha \ : \ (\ id + id\ ) \times id\ \$$$

Antes do início da primeira iteração, t recebe o delimitador \$ e f recebe o primeiro símbolo da sentença:

$$\begin{aligned} t &: \$ \\ f &: (\\ \alpha &: id + id\ ) \times id\ \$ \end{aligned}$$

Para esses valores de t e f, a ação associada é de deslocamento (S). Assim, na primeira iteração o símbolo em t é inserido no topo da pilha e os demais símbolos são deslocados, isto é, t recebe o valor de f e f recebe o primeiro elemento de α.

$$\begin{aligned} p &: \$ \\ t &: (\\ f &: id \\ \alpha &: +id\ ) \times id\ \$ \end{aligned}$$

Novamente, para esses valores de t e f a ação é de deslocamento, que leva a

$$\begin{aligned} p &: \$\ (\\ t &: id \\ f &: + \\ \alpha &: id\ ) \times id\ \$ \end{aligned}$$

A entrada na tabela de deslocamento e redução para $[id, +]$, pesquisada pelo procedimento NEXTACTION(), indica que a ação é de redução. Assim, no início da iteração o estado da pilha é modificado (isso é independente do tipo de ação):

$$p : \$\ (\ id$$

Logo em seguida, no segundo bloco *if*, o procedimento REDUCE() retira o símbolo id da pilha e atribui o resultado de sua redução pela produção $F \rightarrow id$, ou seja, o símbolo F, à variável t:

$$\begin{aligned} p &: \$\ (\\ t &: F \end{aligned}$$

O valor de f permanece inalterado; assim, os argumentos para o procedimento NEXTACTION() correspondem à chave $[F, +]$, que novamente leva a uma redução para a próxima iteração. As iterações prosseguem da mesma forma até que a condição de reconhecimento seja alcançada; se a sentença fosse inválida, o algoritmo terminaria no momento em que não houvesse uma ação aplicável para os valores de t e f.

O processo de reconhecimento dessa sentença, partindo do estado inicial até alcançar o estado de aceitação da sentença, é apresentado na Tabela 3.4.

Métodos de análise ascendente são quase sempre determinísticos, mas há situações em que o analisador deve decidir entre dois possíveis movimentos. Uma delas é a situação de conflito *reduzir ou deslocar* e a outra, quando pelo menos duas regras são aplicáveis em uma situação de redução, é a situação de conflito *reduzir ou reduzir*.

O método LR de análise é o mais geral que pode ser aplicado a todas as linguagens e gramáticas passíveis de análise determinística. Seu nome deriva-se do fato de que a análise é realizada a partir de uma leitura dos símbolos da esquerda para a direita (*Left to right*) e que a derivação canônica mais à direita é obtida (*Rightmost derivation*).

Tabela 3.4 Reconhecimento da sentença $(id + id) \times id$ por deslocamento e redução.

p	t	f	α	n
			$($ $id + id$ $)$ $\times$ id \$	
	\$	$($	$id + id$ $)$ $\times$ id \$	S
\$	$($	id	$+$ id $)$ $\times$ id \$	S
\$ $($	id	$+$	id $)$ $\times$ id \$	R
\$ $($ id				
\$ $($	F			R
\$ $($ F				
\$ $($	T			R
\$ $($ T				
\$ $($	E			S
\$ $($ E	$+$	id	$)$ $\times$ id \$	S
\$ $($ E $+$	id	$)$	$\times id$ \$	R
\$ $($ E $+$ id				
\$ $($ E $+$	F			R
\$ $($ E $+$ F				
\$ $($ E $+$	T			R
\$ $($ E $+$ T				
\$ $($	E			S
\$ $($ E	$)$	$\times$	id \$	R
\$ $($ E $)$				
\$	F			R
\$ F				
\$	T			S
\$ T	$\times$	id	\$	S
\$ T $\times$	id	\$		R
\$ T $\times$ id				
\$ T $\times$	F			R
\$ T $\times$ F				
\$	T			R
\$ T				
\$	E			R*

Uma gramática LR(k), usada como base de um analisador ascendente, é uma na qual as situações de conflito podem ser resolvidas pela verificação dos símbolos já lidos até o momento e pela visão de uma quantidade limitada a no máximo k símbolos adiante (o chamado *lookahead*).

Na prática, o valor de k é geralmente limitado a 0 ou 1 sem perda de generalidade na aplicação do método. Embora haja gramáticas LR(2) que não são gramáticas LR(1), há um resultado teórico que diz que toda linguagem gerada por uma gramática LR(k) pode ser também gerada por uma gramática LR(1).

3.5.3 Geradores de analisadores sintáticos

Como ocorre na construção de analisadores léxicos, a construção de programas analisadores sintáticos é usualmente suportada por ferramentas para a geração automática de programas a partir de uma especificação.

Uma tradicional ferramenta de criação de analisadores sintáticos é `yacc` *(Yet Another Compiler-Compiler)*, oriunda do ambiente de desenvolvimento de *software* do sistema operacional Unix. Assim como a ferramenta `lex` (Seção 3.3.1), `yacc` recebe como entrada um arquivo de especificação de uma gramática e gera como saída um módulo com código-fonte em C contendo uma rotina que realiza o reconhecimento de sentenças segundo essa gramática.

Especificação da gramática

O arquivo de entrada para `yacc`, que por convenção recebe a extensão `.y`, é estruturado em três seções. Como na definição de arquivos `lex`, essas três seções — definições, regras da gramática e código do usuário — são separadas pelos símbolos `%%`.

A especificação das regras da gramática utiliza uma notação próxima de BNF (Seção 3.1.5). Cada produção é expressa na forma

```
simb  :   exp  ;
```

onde `simb` é um símbolo não terminal e `exp` é a sua expansão em termos de outros símbolos da gramática. A expansão pode conter símbolos terminais e não-terminais, que por convenção são representados usando letras maiúsculas e minúsculas, respectivamente.

Pelas características de gramáticas livres de contexto, a expansão pode ser recursiva, isto é, conter o próprio símbolo que está sendo definido, como em

```
expr : expr '+' expr ;
```

Porém, pelo menos uma expansão para esse símbolo deve ser não-recursiva:

```
expr : IDENT ;
```

Em caso de definição recursiva, pelas características do analisador gerado (LR(1)) recomenda-se optar quando possível pela recursão à esquerda.

Produções para um mesmo símbolo podem ser agrupadas usando o símbolo `'|'`,

```
expr  :  expr + expr
      |  IDENT
      ;
```

Expansões para a *string* vazia podem ser definidas; por convenção e para tornar mais clara a definição, essa expansão é destacada na forma de um comentário C:

```
retv  :  /* empty */
      |  expr
      ;
```

O símbolo sentencial da gramática pode ser estabelecido na seção de definições através da declaração `start`, como em

```
%start  expr
```

Na ausência dessa declaração, o símbolo não-terminal cuja expansão é definida na primeira produção da seção de regras da gramática é assumido ser o símbolo sentencial.

Outros tipos de declaração que podem estar presentes na primeira seção são declarações em C, colocadas entre os símbolos `%{` e `%}`, e a definição dos nomes de tipos de *tokens*, os quais serão usados posteriormente nas expansões das produções.

Tokens que são representados por um único caráter, como `'+'` ou `';'`, não precisam ser declarados e podem ser usados dessa forma (como constantes do tipo caráter em C) nas expansões; os demais *tokens* precisam ser explicitamente declarados. Para tanto, a declaração `token` pode ser utilizada, como em

```
%token IDENT
```

Alternativamente, *tokens* para operadores podem ser definidos com uma especificação de associatividade usando, ao invés de `token`, as declarações `left`, `right` ou `nonassoc`. Uma declaração

```
%left OP
```

determina que uma expressão `A OP B OP C` será interpretada como `(A OP B) OP C`, enquanto que se a declaração tivesse sido

```
%right OP
```

a interpretação seria `A OP (B OP C)`. A declaração

```
%nonassoc OP
```

determinaria que a expressão `A OP B OP C` estaria incorreta, pois o operador não é associativo.

A precedência dos operadores também é definida através dessas declarações. Operadores definidos através da mesma linha de declaração, como

```
%left OP1 OP2
```

têm a mesma precedência. Para aqueles definidos em linhas distintas, as últimas declarações têm maior precedência.

O símbolo terminal `error` é pré-definido, podendo ser utilizado como a última expansão de um símbolo caso a aplicação deseje determinar um curso de ação específico em uma situação de não-reconhecimento de uma sentença a partir das expansões previamente definidas para o símbolo.

Manipulação das sentenças reconhecidas

Reconhecer que uma seqüência de símbolos é uma sentença válida em uma gramática é parte essencial do processo de compilação, porém pouco uso teria se simplesmente uma indicação de validade fosse retornada sem nenhuma possibilidade de manipulação adicional das expressões. No caso de `yacc`, essa possibilidade está associada à definição de ações semânticas.

Uma **ação semântica** em `yacc` é definida através de um bloco de expressões em C associado à definição de produções para um símbolo não-terminal:

```
symb  :  expansão    { ação }  ;
```

A definição do corpo da ação pode conter referências aos valores semânticos de cada um dos símbolos da produção. O **valor semântico** de um *token* está associado a um valor associado ao símbolo, que pode ser por exemplo um valor numérico de uma constante ou a *string* associada a um identificador.

O valor semântico do *token* pode ser referenciado na expressão C através de pseudo-variáveis com nome `$`i, onde i determina a posição do *token* na expansão. A variável `$$` referencia o valor semântico resultante para o símbolo sendo definido. Por exemplo,

```
expr  :  expr '+' expr
         { $$ = $1 + $3 }
      ;
```

atribui à expressão reduzida o valor semântico que é a soma dos valores semânticos do primeiro e do terceiro componentes da expansão, que estão separados pelo segundo componente, `'+'`.

Se nenhuma ação for definida para uma produção, a ação semântica padrão — `{ $$ = $1; }` é assumida.

O tipo associado a valores semânticos é definido pela macro `YYSTYPE`, que é inicialmente definida como `int`. Para modificar esse padrão, pode-se modificar essa definição através de uma declaração C na primeira seção do arquivo que define a gramática, como por exemplo

```
%{
 #define  YYSTYPE  double
%}
```

Em aplicações que necessitem manipular *tokens* com diferentes tipos de valores semânticos, a declaração `union` deve ser utilizada para definir quais são os tipos de valores possíveis. Por exemplo, em uma aplicação que manipula valores inteiros e reais a seguinte declaração estaria presente:

```
%union {
   int    ival;
   double fval;
}
```

Essa declaração determina que a coleção de tipos de valores permitidos é composta por valores com nome `ival` ou `fval`, respectivamente para valores inteiros e reais — no código C, uma união (ver Seção C.2.4) será criada. Esses mesmos nomes são utilizados para qualificar a definição de *tokens* da gramática, como em

```
%token <ival>  INTEGER
%token <fval>  REAL
```

Quando uma coleção de tipos de valores é utilizada, é preciso determinar também qual o tipo para o símbolo não-terminal para o qual a expressão está sendo reduzida. Para esse fim, `yacc` define a declaração `type`:

```
%type <fval> expr
```

Desenvolvimento de uma aplicação

Nesta seção descreve-se o procedimento para desenvolver uma aplicação, usando a ferramenta `bison`, que realiza a análise sintática de um arquivo de entrada. A ferramenta `bison` é uma implementação de `yacc` disponível para diversas plataformas e distribuída, assim como `flex`, sob a licença de *software* GNU da *Free Software Foundation*.

A execução de `bison` requer como argumento o nome do arquivo com a gramática especificada. Se esse arquivo recebeu, por exemplo, o nome `mygram.y`, então a linha de comando

```
bison mygram.y
```

gera um módulo-fonte em C de nome `mygram.tab.c` que contém a definição das tabelas para a análise sintática e a rotina de reconhecimento, cujo nome é `yyparse()`.

A rotina `yyparse()` deve ser invocada pela aplicação para que a análise sintática do arquivo de entrada seja realizada. Esta rotina retorna um valor inteiro, que será 0, se toda a entrada pode ser reconhecida sem erros pela gramática especificada, ou 1, caso algum erro sintático tenha sido detectado no arquivo de entrada.

A rotina `yyparse()` irá invocar uma rotina `yylex()` que irá varrer o arquivo de entrada e retornar os *tokens* para o analisador sintático. Essa rotina não é criada automaticamente e deve ser fornecida pelo usuário. Tipicamente, mas não necessariamente, essa rotina é gerada por uma ferramenta `lex`.

Caso a rotina `yylex()` fornecida seja simples o suficiente para ser definida manualmente, então seu código pode ser incluído na seção de usuário do arquivo de especificação da gramática e todas as definições de tipos de *tokens* podem ser usadas diretamente. Caso contrário — por exemplo, se `flex` for ser utilizada para gerar a rotina de análise léxica — então é preciso transportar essas definições para os demais módulos da aplicação. Para tanto, utiliza-se a chave de execução `-d`, que gera um arquivo de cabeçalho com essas definições:

```
bison -d mygram.y
```

Com essa opção, além do arquivo-fonte C um arquivo de nome `mygram.tab.h` é gerado com as definições necessárias, podendo ser incluído em outros módulos para realizar a integração. Por exemplo, um arquivo `mylex.l` contendo a especificação para reconhecimento de *tokens* usando `flex` poderia conter, na sua seção de definições, a declaração

```
%{
 #include "mygram.tab.h"
%}
%%  /* definições das expressões regulares: */
```

Outro aspecto importante na integração das rotinas `yyparse()` e `yylex()` é na forma de definição dos valores semânticos e dos tipos dos *tokens*. A definição do tipo de *token* é determinada pelo valor de retorno de `yylex()`. Por exemplo, se no arquivo `mygram.y` houvesse a declaração

```
%token INTEGER
```

então a ação associada ao reconhecimento de um padrão regular que reconhecesse esse tipo de *token* deveria concluir com a expressão

```
return INTEGER;
```

O outro ponto de ligação entre as duas rotinas é a definição do valor semântico, que é realizado através de uma variável global `yylval`, definida no módulo `mygram.tab.c`. Para aplicações que trabalham com um único tipo de valor, basta atribuir na ação associada ao reconhecimento do *token* o valor semântico resultante, como em

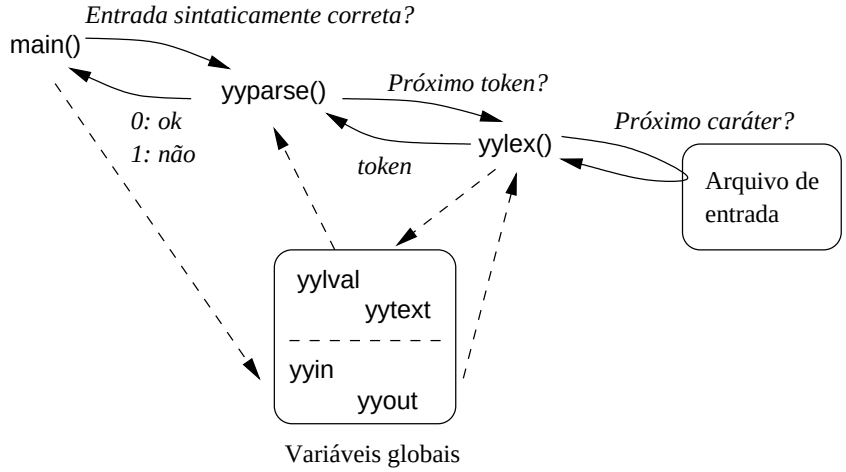
```
...
yylval = atoi(yytext);
return INTEGER;
}
```

Se a aplicação opera com múltiplos tipos de valores semânticos, então será preciso qualificar a atribuição, indicando qual componente da união de tipos está sendo utilizada:

```
...
yylval.ival = atoi(yytext);
return INTEGER;
}
```

A Figura 3.13 ilustra a forma de operação integrada das rotinas `yyparse()` e `yylex()`. A partir do código principal da aplicação, a rotina `yyparse()` é invocada. Eventualmente, o código `main()` pode alterar os valores das variáveis globais `yyin` e `yyout`, definidas no módulo do analisador léxico, para determinar quais arquivos serão utilizados para a entrada e a saída de `yylex()`. A rotina `yyparse()` irá solicitar de rotina `yylex()` que obtenha os *tokens* do arquivo de entrada; para tanto, `yylex()` lê os caracteres desse arquivo, compondo os tokens segundo as expressões regulares definidas. Uma vez reconhecido um *token*, `yylex()` atribui seu valor semântico (através da variável global `yylval`, definida no módulo do analisador sintático) e qual a *string* correspondente (através da variável global `yytext`, definida pelo analisador léxico).

Figura 3.13 Operação integrada dos analisadores léxicos e sintáticos.



3.6 Geração de código e otimização

As Seções 3.2 e 3.4 apresentaram técnicas com forte embasamento teórico e conceitual para permitir reconhecer os símbolos e as expressões tipicamente utilizadas em linguagens de programação de alto nível. No entanto, a operação de um compilador requer mais que o simples reconhecimento da validade de um programa; é preciso gerar o código equivalente que será efetivamente executado pelos processadores.

Além das análises léxica e sintática, as demais tarefas de um compilador são apoiadas por heurísticas e não mais por formalismos. O objetivo deste capítulo é oferecer uma visão geral dessas técnicas usadas para concluir a obtenção do código *assembly* equivalente ao programa de alto nível que foi processado pelo compilador.

3.6.1 Análise semântica

Embora a análise sintática consiga verificar se uma expressão obedece às regras de formação de uma dada gramática, seria muito difícil expressar através de gramáticas algumas regras usuais em linguagem de programação, como "todas as variáveis devem ser declaradas" e situações onde o contexto em que ocorre a expressão ou o tipo da variável deve ser verificado.

O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre partes distintas do programa. As tarefas básicas desempenhada durante a análise semântica incluem a verificação de tipos, a verificação do fluxo de controle e a verificação da unicidade da declaração de variáveis. Dependendo da linguagem de programação, outros tipos de verificações podem ser necessários.

Considere o seguinte exemplo de código em C:

```
int f1(int a, float b) {
  return a%b;
}
```

A tentativa de compilar esse código irá gerar um erro detectado pelo analisador semântico, mais especificamente pelas regras de **verificação de tipos**, indicando que o operador módulo % não pode ter um operador real. No compilador `gcc`, essa mensagem é

```
In function ‘f1’:
...: invalid operands to binary %
```

Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador. Por exemplo, na expressão em C

```
a = x - ’0’;
```

a constante do tipo caráter `'0'` é automaticamente convertida para inteiro para compor corretamente a expressão aritmética na qual ela toma parte; todo `char` em uma expressão é convertido pelo compilador para um `int`. Esse procedimento de conversão de tipo é denominado **coerção** *(cast)*. Em C, a seguinte seqüência de regras determina a realização automática de coerção em expressões aritméticas:

1. `char` e `short` são convertidos para `int`, `float` para `double`;
2. se um dos operandos é `double`, o outro é convertido para `double` e o resultado é `double`;
3. se um dos operandos é `long`, o outro é convertido para `long` e o resultado é `long`;
4. se um dos operandos é `unsigned`, o outro é convertido para `unsigned` e o resultado é `unsigned`;
5. senão, todos os operandos são `int` e o resultado é `int`.

Em outras situações, a conversão deve ser indicada explicitamente pelo programador através do operador de molde, com o nome do tipo entre parênteses na frente da expressão cujo resultado deseja-se converter. Por exemplo, com as declarações

```
int a;
int *p;
```

a expressão

```
a = p;
```

geraria a seguinte mensagem do compilador:

```
warning: assignment makes integer from pointer without a cast
```

Porém, se o programador indicar que sabe que está fazendo uma conversão “perigosa” através do operador de molde,

```
a = (int) p;
```

então nenhuma mensagem é gerada.

Algumas linguagens de programação, como C++, permitem definir comportamentos diferenciados a operadores segundo o tipo de argumento que recebem. Por exemplo, na expressão

```
c << x;
```

o operador `<<` será interpretado como o comando de deslocamento de bits à esquerda se `c` e `x` forem inteiros, mas será uma operação de saída se `c` for uma referência para um arquivo. Esse mecanismo de adequar o comportamento do operador segundo o tipo de seus operandos é denominado **sobrecarga de operadores**. Em geral, essas linguagens permitem também aplicar o mesmo tipo de mecanismo a rotinas. Através da **sobrecarga de funções**, o compilador seleciona entre rotinas que têm o mesmo nome aquela cuja quantidade e lista de tipos estão adequadas à forma de invocação.

Outro exemplo de erro detectado pela análise semântica, neste caso pela **verificação de fluxo de controle**, é ilustrado pelo código

```
void f2(int j, int k) {
  if (j == k)
    break;
  else
    continue;
}
```

Nesse caso, o compilador gera as mensagens:

```
In function ‘f2’:
...: break statement not within loop or switch
...: continue statement not within a loop
```

ou seja, ele reconhece que o comando `break` só pode ser usado para quebrar a seqüência de um comando de iteração *(within loop)* ou para indicar o fim de um `case` *(within switch)*. Da mesma forma, um comando `continue` só pode ser usado em um comando de iteração.

A **verificação de unicidade** detecta situações tais como duplicação em declarações de variáveis, de componentes de estruturas e em rótulos do programa. Por exemplo, a compilação do seguinte código

```
void f3(int k) {
  struct {
    int a;
    float a;
  } x;
  float x;
  switch (k) {
  case 0x31: x.a = k;
  case ’1’: x = x.a;
  }
}
```

causaria a geração das seguintes mensagens de erro pelo compilador `gcc`:

```
In function ‘f3’:
...: duplicate member ‘a’
...: previous declaration of ‘x’
...: duplicate case value
```

A primeira mensagem detecta que dois membros de uma mesma definição de estrutura recebem o mesmo nome, `a`, o que não é permitido. Na segunda mensagem a indicação refere-se às duas variáveis de mesmo nome, `x`. A terceira mensagem indica que dois `case`s em uma expressão `switch` receberam o mesmo rótulo, o que também não é permitido. Observe que, mesmo embora a forma de expressar o valor nos `case`s tenha sido diferente, o compilador verificou que `0x31` e `’1’` referiam-se ao mesmo valor e acusou a situação de erro.

3.6.2 Geração de código

A tradução do código de alto nível para o código do processador está associada à traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa. Embora tal atividade possa ser realizada para a árvore completa após a conclusão da análise sintática, em geral ela é efetivada através das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. Este procedimento é denominado **tradução dirigida pela sintaxe**.

Em geral, a geração de código não se dá diretamente para a linguagem *assembly* do processador-alvo. Por conveniência, o analisador sintático gera código para uma máquina abstrata, com uma linguagem próxima a *assembly* porém independente de processadores específicos. Em uma segunda etapa de geração de código, esse código intermediário é traduzido para a linguagem *assembly* desejada. Dessa forma, grande parte do compilador é reaproveitada para trabalhar com diferentes tipos de processadores.

Código intermediário

A linguagem utilizada para a geração de um código em formato intermediário entre a linguagem de alto nível e a linguagem *assembly* deve representar, de forma independente do processador para o qual o programa será gerado, todas as expressões do programa original. Duas formas usuais para esse tipo de representação são a notação posfixa e o código de três endereços.

Código de três endereços

O código de três endereços é composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome "três endereços" está associado à especificação, em uma instrução, de no máximo três variáveis: duas para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly* (Seção 4.1.1) e, conseqüentemente, de mais fácil conversão para a linguagem-alvo.

Uma possível especificação de uma linguagem de três endereços envolve quatro tipos básicos de instruções: expressões com atribuição, desvios, invocação de rotinas e acesso indexado e indireto.

Instruções de atribuição são aquelas nas quais o resultado de uma operação é armazenado na variável especificada à esquerda do operador de atribuição, aqui denotado por :=. Há três formas para esse tipo de instrução. Na primeira, a variável recebe o resultado de uma operação binária:

```
x := y op z
```

O resultado pode ser também obtido a partir da aplicação de um operador unário:

```
x := op y
```

Na terceira forma, pode ocorrer uma simples cópia de valores de uma variável para outra:

```
x := y
```

Por exemplo, a expressão em C

```
a = b + c * d;
```

seria traduzida nesse formato para as instruções:

```
_t1 := c * d
  a := b + _t1
```

As **instruções de desvio** podem assumir duas formas básicas. Uma instrução de desvio incondicional tem o formato

```
goto L
```

onde `L` é um rótulo simbólico que identifica uma linha do código. A outra forma de desvio é o desvio condicional, com o formato

```
if x opr y goto L
```

onde `opr` é um operador relacional de comparação e `L` é o rótulo da linha que deve ser executada se o resultado da aplicação do operador relacional for verdadeiro; caso contrário, a linha seguinte é executada.

Por exemplo, a seguinte iteração em C

```
while (i++ <= k)
  x[i] = 0;
x[0] = 0;
```

poderia ser traduzida para

```
_L1: if i > k goto _L2
     i := i + 1
     x[i] := 0
     goto _L1
_L2: x[0] := 0
```

A **invocação de rotinas** ocorre em duas etapas. Inicialmente, os argumentos do procedimento são "registrados" com a instrução `param`; após a definição dos argumentos, a instrução `call` completa a invocação da rotina. A instrução `return` indica o fim de execução de uma rotina. Opcionalmente, esta instrução pode especificar um valor de retorno, que pode ser atribuído na linguagem intermediária a uma variável como resultado de `call`.

Por exemplo, considere a chamada de uma função `f` que recebe três argumentos e retorna um valor:

```
f(a, b, c);
```

Neste exemplo em C, esse valor de retorno não é utilizado. De qualquer modo, a expressão acima seria traduzida para

```
param a
param b
param c
_t1 := call f,3
```

onde o número após a vírgula indica o número de argumentos utilizados pelo procedimento `f`. Com o uso desse argumento adicional é possível expressar sem dificuldades as chamadas aninhadas de procedimentos.

O último tipo de instrução para códigos de três endereços refere-se aos modos de endereçamento indexado e indireto. Para atribuições indexadas, as duas formas básicas são

```
x := y[i]
x[i] := y
```

As atribuições associadas ao modo indireto permitem a manipulação de endereços e seus conteúdos. As instruções em formato intermediário também utilizam um formato próximo àquele da linguagem C:

```
x := &y
w := *x
*x := z
```

A representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas. Na abordagem que utiliza **quádruplas** (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado. Por exemplo, a tradução da expressão `a=b+c*d;` resultaria no seguinte trecho da tabela:

	operador	arg 1	arg 2	resultado
1	*	c	d	_t1
2	+	b	_t1	a

Para algumas instruções, como aquelas envolvendo operadores unários ou desvio incondicional, algumas das colunas estariam vazias.

Na outra forma de representação, por **triplas**, evita a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos. Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre implicitamente associado à linha da tabela. No mesmo exemplo apresentado para a representação interna por quádruplas, a representação por triplas seria

	operador	arg 1	arg 2
1	*	c	d
2	+	b	(1)

Notação posfixa

A notação tradicional para expressões aritméticas, que representa uma operação binária na forma `x+y`, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixa. Uma notação alternativa para esse tipo de expressão é a notação posfixa, também conhecida como notação polonesa[1], na qual o operador é expresso após seus operandos.

O atrativo da notação posfixa é que ela dispensa o uso de parênteses. Por exemplo, as expressões

```
a*b+c;
a*(b+c);
(a+b)*c;
(a+b)*(c+d);
```

seriam representadas nesse tipo de notação respectivamente como

```
a b * c +
a b c + *
a b + c *
a b + c d + *
```

Instruções de desvio em código intermediário usando a notação posfixa assumem a forma

```
 L jump
 x y L jcc
```

[1] O criador desse tipo de notação, J. Lukasiewicz, era polonês.

para desvios incondicionais e condicionais, respectivamente. No caso de um desvio condicional, a condição a ser avaliada envolvendo `x` e `y` é expressa na parte `cc` da própria instrução. Assim, `j`*cc* pode ser uma instrução entre `jeq` (desvio ocorre se `x` e `y` forem iguais), `jne` (se diferentes), `jlt` (se `x` menor que `y`), `jle` (se `x` menor ou igual a `y`), `jgt` (se `x` maior que `y`) ou `jge` (se `x` maior ou igual a `y`).

Expressões em formato intermediário usando a notação posfixa podem ser eficientemente avaliadas em máquinas baseadas em pilhas, também conhecidas como máquinas de zero endereços. Nesse tipo de máquinas, operandos são explicitamente introduzidos e retirados do topo da pilha por instruções `push` e `pop`, respectivamente. Além disso, a aplicação de um operador retira do topo da pilha seus operandos e retorna ao topo da pilha o resultado de sua aplicação.

Por exemplo, a avaliação da expressão `a*(b+c)` em uma máquina baseada em pilha poderia ser traduzida para o código

```
push a
push b
push c
add
mult
```

3.6.3 Otimização de código

A etapa final na geração de código pelo compilador é a fase de otimização. Como o código gerado através da tradução orientada a sintaxe contempla expressões independentes, diversas situações contendo seqüências de código ineficiente podem ocorrer. O objeto da etapa de otimização de código é aplicar um conjunto de heurísticas para detectar tais seqüências e substituí-las por outras que removam as situações de ineficiência.

As técnicas de otimização que são usadas em compiladores devem, além de manter o significado do programa original, ser capazes de capturar a maior parte das possibilidades de melhoria do código dentro de limites razoáveis de esforço gasto para tal fim. Em geral, os compiladores usualmente permitem especificar qual o grau de esforço desejado no processo de otimização. Por exemplo, em `gcc` há opções na forma `-O...` que dão essa indicação, desde `-O0` (nenhuma otimização) até `-O3` (máxima otimização, aumentando o tempo de compilação), incluindo também uma opção `-Os`, indicando que o objetivo é reduzir a ocupação de espaço em memória.

Algumas heurísticas de otimização são sempre aplicadas pelos compiladores. Por exemplo, se a concatenação de código gerado por duas expressões no programa original gerou uma situação de desvio incondicional para a linha seguinte, como em

```
     <a>
     goto _L1
_L1: <b>
```

esse código pode ser seguramente reduzido com a aplicação da técnica de **eliminação de desvios desnecessários**, resultando em

```
     <a>
_L1: <b>
```

Outra estratégia de otimização elimina os rótulos não referenciados por outras instruções do programa. Assim, se o rótulo `_L1` estivesse sendo referenciado exclusivamente por essa instrução de desvio, ele poderia ser eliminado em uma próxima aplicação das estratégias de otimização.

As técnicas de otimização podem ser classificadas como independentes de máquina, quando podem ser aplicadas antes da geração do código na linguagem *assembly*, ou dependentes de máquina, quando aplicadas na geração do código *assembly*.

A otimização independente de máquina tem como requisito o levantamento dos blocos de comandos que compõem o programa. Essa etapa da otimização é conhecida como a análise de fluxo, que por sua vez contempla a análise de fluxo de controle e a análise de fluxo de dados. Estratégias que podem ser aplicadas analisando um único bloco de comandos são denominadas estratégias de otimização local, enquanto aquelas que envolvem a análise simultânea de dois ou mais blocos são denominadas estratégias de otimização global.

Algumas estratégias básicas de otimização, além da já apresentada eliminação de desvios desnecessários, são apresentadas a seguir.

A estratégia de **eliminação de código redundante** busca detectar situações onde a tradução de duas expressões gera instruções cuja execução repetida não tem efeito. Por exemplo, em

```
x := y
  ...
x := y
```

se não há nenhuma mudança no valor de `y` entre as duas instruções, então a segunda instrução poderia ser eliminada. O mesmo aconteceria se a segunda instrução fosse

```
y := x
```

e o valor de `x` não fosse alterado entre as duas instruções.

Outra estratégia básica é a **eliminação de código não-alcançável**, ou "código morto". Por exemplo, se a seguinte seqüência de código fosse gerada por um conjunto de expressões,

```
    ...
  goto _L1
  x := y
_L1:  ...
```

a instrução contendo a atribuição de `y` a `x` nunca poderia ser executada, pois é precedida de um desvio incondicional e não é o destino de nenhum desvio, pois não contém um rótulo na linha. Assim, essa linha poderia ser eliminada e provocar posteriormente a aplicação da estratégia de eliminação de desvios desnecessários.

O **uso de propriedades algébricas** é outra estratégia de otimização usualmente aplicada. Nesse caso, quando o compilador identifica que uma expressão aritmética foi reduzida a $x + 0$ ou $0 + x$ ou $x - 0$ ou $x \times 1$ ou $1 \times x$ ou $x/1$, então ele substitui a expressão simplesmente por x. Outra classe de propriedades algébricas que é utilizada tem por objetivo substituir operações de alto custo de execução por operações mais simples, como

$$2.0 \times x = x + x$$
$$x^2 = x \times x$$
$$x/2 = 0.5 \times x$$

Particularmente no último caso, se `x` for uma variável inteira a divisão por dois pode ser substituída por um deslocamento da representação binária à direita de um bit. Genericamente, a divisão inteira por 2^n equivale ao deslocamento à direita de n bits na representação binária do dividendo. Da mesma forma, a multiplicação inteira por potências de dois pode ser substituída por deslocamento de bits à esquerda.

A utilização de propriedades algébricas permite também o reuso de subexpressões já computadas. Por exemplo, a tradução direta das expressões

```
a = b + c;
e = c + d + b;
```

geraria o seguinte código intermediário:

```
    a := b + c
  _t1 := c + d
    e := _t1 + b
```

No entanto, o uso da comutatividade e associatividade da adição permite que o código gerado seja reduzido usando a **eliminação de expressões comuns**, resultando em

```
    a := b + c
    e := a + d
```

Diversas oportunidades de otimização estão associadas à análise de comandos iterativos. Uma estratégia é a **movimentação de código**, aplicado quando um cálculo realizado dentro do laço na verdade envolve valores invariantes na iteração. Por exemplo, o comando

```
while (i < 10*j) {
  a[i] = i + 2*j;
  ++i;
}
```

estaria gerando um código intermediário equivalente a

```
_L1:  _t1 := 10 * j
      if i >= _t1 goto _L2
      _t2 := 2 * j
      a[i] := i + _t2
      i := i + 1
      goto _L1
_L2:   ...
```

No entanto, a análise de fluxo de dados mostra que o valor de `j` não varia entre iterações. Assim, o compilador poderia mover as expressões que envolvem exclusivamente constantes na iteração para fora do laço, substituindo esse código por

```
      _t1 := 10 * j
      _t2 := 2 * j
_L1:  if i >= _t1 goto _L2
      a[i] := i + _t2
      i := i + 1
      goto _L1
_L2:   ...
```

Um exemplo de otimização dependente de máquina pode ser apresentado para a expressão

```
x := y + K
```

onde `K` é alguma constante inteira. Na tradução para o código *assembly* de um processador da família 68K, a forma genérica poderia ser algo como

```
MOVE.L  y,D0
ADDI.L #K,D0
MOVE.L D0,x
```

No entanto, se o compilador verificasse que o valor de `K` fosse uma constante entre 1 e 8, então a segunda instrução *assembly* poderia ser substituída por `ADDQ.L`, que utilizaria em sua codificação apenas dois bytes ao invés dos seis bytes que seriam requeridos pela instrução `ADDI.L`.

3.7 Exercícios

3.1 Descreva em termos de diagramas de grafos sintáticos a seguinte gramática expressa em notação `yacc`:

```
%token  T1, T2, T3, T4
%%
n1   :  n2 T1 n2 T2
     |  T2
     ;
n2   :  n3
     |  n4
     ;
n3   :  n4 T3 n4
     |  T3 n4
     ;
n4   :  n4 T4
     |  T4
     ;
```

3.2 Dada a expressão regular $(xx) * (y|z)z*$

(a) Construa, usando o método da construção de Thompson, o autômato finito não-determinístico que reconhece as seqüências de símbolos que formam sentenças na linguagem regular correspondente.

(b) Converta o autômato do item *(a)* para um autômato finito determinístico usando o método da construção de subconjuntos.

(c) Converta o autômato finito determinístico obtido para outro equivalente com o menor número possível de estados.

3.3 Desenvolva o autômato finito determinístico com mínimo número de estados para reconhecer sentenças da gramática regular $G_2 = \{V_n, V_t, P, A\}$, com $V_n = \{A, B, C\}$, $V_t = \{x, y\}$ e produções $P = \{A \rightarrow xB,\ A \rightarrow yB,\ B \rightarrow xC,\ C \rightarrow xC, C \rightarrow y\}$.

3.4 O seguinte arquivo em formato `lex` especifica *tokens* que serão aceitos em uma dada aplicação (a ação correspondente não é mostrada aqui):

```
%%
0[01]{2}0
1(0{2}|1{2})[01]
```

Dada esta especificação, indique para cada um dos tokens a seguir se ele seria ou não aceito na aplicação, justificando sua resposta.

(a) 0000

(b) 00120

(c) 0001

(d) 0010

(e) 0120

(f) 1000

(g) 1010

(h) 1020

(i) 1111

(j) 100110

3.5 Dada a gramática $G_1 = \{V_n, V_t, P, S\}$, com $V_n = \{A, S\}$, $V_t = \{a, b\}$ e as produções $P = \{S \rightarrow A,\ A \rightarrow aAb,\ A \rightarrow ab\}$

(a) Qual a classe de linguagem definida por essa gramática?

(b) Mostre a árvore gramatical para a sentença $aabbb$. Esta pertence à linguagem definida por essa gramática? Justifique.

(c) Dê dois exemplos de sentenças geradas por essa gramática, apresentando para cada um dos exemplos a árvore sintática e a derivação canônica mais à esquerda.

(d) Monte a tabela de deslocamento e redução para esta gramática. Mostre como as sentenças dos itens *(b)* e *(c)* seriam processadas através do analisador de deslocamento e redução correspondente.

3.6 Considere a gramática G com $V_{nt} = \{S, A, B, C\}$, $V_t = \{x, y, z\}$, símbolo sentencial S e as seguintes produções (ε é a *string* vazia):

R1: $S \rightarrow AxByC$
R2: $A \rightarrow xAx$
R3: $A \rightarrow \varepsilon$
R4: $B \rightarrow By$
R5: $B \rightarrow \varepsilon$
R6: $C \rightarrow zAz$

Para a sentença $x\ x\ x\ y\ y\ z\ x\ x\ z$, apresente

(a) a derivação canônica mais à direita;

(b) a derivação canônica mais à esquerda; e

(c) a árvore gramatical, incluindo as ocorrências da *string* vazia como folhas da árvore.

3.7 Considere a gramática $G = (\{E, T, F\}, \{x, y, +, \times, (,)\}, P, E)$ onde P é o conjunto com as seguintes produções:

(a) $E \rightarrow E + T$
(b) $E \rightarrow T$
(c) $T \rightarrow T \times F$
(d) $T \rightarrow F$
(e) $F \rightarrow (E)$
(f) $F \rightarrow x$
(g) $F \rightarrow y$

(a) Classifique a gramática pela hierarquia de Chomsky, justificando sua resposta.

(b) A sentença $x + x \times y + y$ é descrita por esta gramática? Justifique sua resposta mostrando se existe ou não uma árvore sintática para a sentença **e** uma derivação canônica.

Capítulo 4

Carregadores e ligadores

O resultado do processo de compilação é um arquivo contendo um programa em *assembly* equivalente ao programa originalmente descrito em linguagem de alto nível. Um programa em linguagem *assembly*, ou linguagem simbólica, contém seqüências de instruções mnemônicas que representam as operações que devem ser realizadas pelo processador. Essas instruções são definidas pelos projetistas do processador; o conjunto de todas as instruções definidas para um processador contitui seu **jogo de instruções**.

O **montador** *(assembler)* é o programa do sistema responsável por traduzir o código *assembly* em linguagem de máquina, traduzindo cada instrução do programa para a seqüência de bits que codifica a instrução de máquina. Como cada processador tem sua própria linguagem, montadores são específicos para processadores. Montadores são objetos de estudo da Seção 4.1.

Neste capítulo, serão ainda descritas as atividades do sistema necessárias para que o programa montado possa efetivamente ser executado — a **ligação**, que resolve as referências que tenham sido feitas a dados e rotinas em outros programas, e o **carregamento**, que transfere o programa montado para a memória principal e dá início à sua execução.

4.1 Montadores

O processo de montagem recebe como entrada um arquivo texto com o código fonte do programa em *assembly* e gera como saída um arquivo binário, o **módulo objeto**, contendo o código de máquina e outras informações relevantes para a execução do código gerado.

Em geral, montadores oferecem facilidades além da simples tradução de código *assembly* para código de máquina. Além das instruções do processador, um programa fonte para o montador pode conter diretivas ou **pseudo-instruções** definidas para o montador (e não para o processador), assim como macro-instruções, uma seqüência de instruções que será inserida no código ao ser referenciada pelo nome. Um montador que suporte a definição e utilização de macro-instruções é usualmente denominado um **macro-montador** (*macro-assembler*). Um **montador multiplataforma** *(cross-assembler)* é um montador que permite gerar código para um processador-alvo diferente daquele no qual o montador está sendo executado.

Na seqüência apresenta-se brevemente as atividades relacionadas ao processo de montagem, partindo da descrição do formato de entrada esperado até a geração do módulo-objeto de saída.

4.1.1 Programas *assembly*

O montador recebe como entrada um arquivo texto cujas linhas são instruções *assembly*. Embora o formato específico de um arquivo-fonte em *assembly* possa sofrer ligeiras variações de acordo com o sistema, a descrição a seguir cobre a maior parte dos aspectos relevantes para a operação do montador.

O seguinte trecho de código apresenta um típico programa *assembly*:

```
POS      DS.W      1
; Busca 0 na sequencia de inteiros
SRCH0    MOVEA.L   #DATUM,A0   ; (DATUM) definido alhures
         MOVE.L    #DATUM,D0   ; guarda inicio
         CLR.W     D1
LOOP     CMP.W     (A0)+,D1
         BNE       LOOP
         SUB.L     A0,D0
         MOVE.W    D0,POS
         RTS
         END
```

Cada linha desse programa pode conter instruções ou comentários; uma linha é de comentário quando contém no início da linha o caráter `;` (ponto-e-vírgula).

As linhas de instrução contém até quatro campos. A primeira coluna pode apresentar um **rótulo** opcional. A função básica do rótulo é criar uma identificação para poder referenciar simbolicamente a linha de código rotulada. O montador pode reconhecer rótulos pela presença de caracteres distintos de `;` a partir da primeira posição da linha.

A segunda coluna contém o **campo de operação**, que especifica a instrução que será montada. A operação pode ser tanto uma instrução de máquina, a exemplo de `MOVE` e `RTS`, como uma pseudo-instrução, como `DS`. Os sufixos às instruções presentes no exemplo indicam o tamanho do operando — no caso dos processadores da família 68K, `.B` *(byte)*, `.W` *(word)* ou `.L` *(long word)* respectivamente para um, dois ou quatro bytes. Se for omitido, o tamanho *word* é usado como padrão.

Dependendo da instrução presente na segundo coluna, o montador sabe se deve esperar zero, um ou dois operandos na terceira coluna, que corresponde ao **campo de operandos**. Operandos podem fazer referências a registradores do processador (no caso do 68K, registradores de dados `D0` a `D7`, de endereços `A0` a `A7`, contador de programas `PC` e, para algumas instruções, de códigos de condição `CCR`), a símbolos definidos pelos programas através de rótulos e a valores constantes.

A especificação do valor de uma constante que representa um operando imediato, indicado pelo prefixo #, pode se dar sob diversas formas de representação. Por exemplo, nas instruções

```
MOVE.B    #48,D0
MOVE.B    #$30,D0
MOVE.B    #@60,D0
MOVE.B    #%110000,D0
MOVE.B    #'0',D0
```

o operando imediato é sempre o mesmo valor, representado respectivamente como um número decimal (sem prefixo adicional), hexadecimal (prefixo `$`), octal (prefixo `@`), binário (prefixo `%`) e ASCII (entre aspas simples). Qualquer que fosse a forma selecionada, o código de máquina gerado para essa instrução seria o mesmo:

```
00010000 00111100
00000000 00110000
```

A especificação do código de máquina para as principais instruções da família 68K usadas neste texto são apresentadas no Apêndice B.

Seqüências de caracteres *(strings)* são definidas também entre aspas simples. Por exemplo, `'ABC'` define uma seqüência de três bytes com valores $41, $42 e $43.

A quarta e última coluna, também opcional, corresponde ao **campo de comentários**. No exemplo, cada comentário é iniciado pelo caráter `;`, após o qual todo o restante da linha pode ser ignorado pelo montador.

Através do uso de pseudo-instruções e dos rótulos, é possível fazer referências a posições de memória e a variáveis através de identificadores simbólicos. Isso permite que o programador possa usar esses identificadores simbólicos como operandos de suas instruções sem ter que necessariamente saber a qual posição de memória a variável ou instrução está alocada. A regra para a composição de tais identificadores pode apresentar diferenças entre montadores distintos. Em geral, identificadores podem incluir letras minúsculas ou maiúsculas, dígitos e o caráter _, mas não podem ser iniciados por um dígito.

Estrutura de programa *assembly*

Um programa *assembly* é tipicamente composto por pelo menos dois segmentos, um **segmento de dados** que define o espaço associado ao armazenamento das variáveis e constantes usadas pelo programa; e um **segmento de instruções**, onde o código do programa é armazenado. Além dessas duas seções, um programa-fonte *assembly* pode conter uma **seção de definições**, que são usadas na descrição dos programas não produzem nenhum efeito no código gerado.

Por conveniência da leitura do código fonte, a seção de definições é tradicionalmente alocada ao início do código. Assim, quando o código for lido por um ser humano ele terá noção do significado das constantes simbólicas usadas ao longo do programa. Com relação aos segmentos de dados e de instruções, não há um posicionamento fixo. Na prática, um programa pode ter vários segmentos associados.

Para o montador, o posicionamento dos diferentes trechos de programa no código fonte deve ser irrelevante. Na verdade, é necessário que o montador seja capaz de manipular representações simbólicas antes que elas tenham sido definidas. Considere o seguinte exemplo de um trecho de programa:

```
1       START     ADD.L     D0,D1
2                 JMP       NEXT
3       LOOP      ADD.L     #1,D1
4       NEXT      CLR.L     D5
5                 JMP       LOOP
```

Na linha 2 desse trecho de programa há uma referência a um símbolo, `NEXT`, cujo valor ainda não havia sido determinado — essa definição só acontecerá na linha 4. Há duas possibilidades de lidar com essas referências futuras.

A primeira possibilidade é deixar uma lacuna reservada no código gerado associada ao operando da instrução da linha 2. Posteriormente, quando houvesse uma definição desse valor — provavelmente quando o fim do arquivo com o código fonte fosse alcançado — essa lacuna seria preenchida. Neste caso, seria possível gerar o código de máquina realizando um único passo (uma única leitura) sobre o arquivo. Entretanto, haveria um maior custo na complexidade de implementação do montador, que deveria manter referências a todas as lacunas que devem ser preenchidas ao final da montagem.

A outra possibilidade, conceitualmente mais simples, é realizar o processo montagem em dois passos. O primeiro passo simplesmente lê o arquivo com o objetivo de criar a Tabela de Símbolos, ou seja, obter os valores associados a todas as constantes simbólicas definidas no programa. No segundo passo, uma nova leitura sobre o arquivo é realizada para gerar o código de máquina; nesse passo, a informação da tabela de símbolos criada no primeiro passo é utilizada.

Pseudo-Instruções

Além das instruções do processador, um programa *assembly* preparado para um montador também pode conter pseudo-instruções que estabelecem a conexão entre referências simbólicas e valores a serem efetivamente referenciados. Cada montador pode oferecer um conjunto de pseudo-instruções diferenciado. As pseudo-instruções descritas a seguir representam um subconjunto significativo de facilidades oferecidas por montadores.

A pseudo-instrução de **substituição simbólica**, `EQU`, associa um valor definido pelo programador a um símbolo. Por exemplo, a linha de instrução

```
SIZE    EQU     100
```

associa o valor decimal 100 ao símbolo SIZE, que pode ser posteriormente referenciado em outras instruções, como em

```
        MOVE    #SIZE,D0
```

Para essa pseudo-instrução, o rótulo deve estar sempre presente e o operando pode ser qualquer expressão que, quando avaliada, defina o valor para o símbolo. Essa expressão pode envolver outros símbolos já definidos.

A pseudo-instrução `EQU` define símbolos que serão usados durante o processo de montagem, mas que não farão parte do módulo-objeto. Para definir constantes para a execução do código, ou seja, que ocuparão algum espaço em memória durante a execução do programa gerado, as pseudo-instruções `DC` e `DS` devem ser usadas.

A **definição de variável inicializada**, isto é, com algum valor constante definido no momento da alocação de espaço para a variável, dá-se através da pseudo-instrução `DC`, como nos exemplos

```
CONTADOR  DC.L    100
ARR1      DC.W    0,1,1,2,3,5,8,13
MENSAGEM  DC.B    'Alo, pessoal!'
```

O rótulo deve estar presente nessa instrução para permitir referenciar a posição de memória de cada variável ao longo do código. O sufixo no código de operação indica o tamanho em bytes para cada variável, seguindo o padrão das instruções da família 68K. Assim, `CONTADOR` fará referência a uma posição de memória cujos quatro bytes seguintes terão inicialmente a representação para o valor 100. `ARR1` é uma referência para a posição de memória que dá início a um bloco contíguo de oito palavras de dois bytes, cada uma delas com o valor especificado na posição correspondente no operando. De forma similar, `MENSAGEM` faz referência ao início de um bloco de treze bytes representados em ASCII no operando.

Outra forma de reservar um espaço de memória para armazenar valores é através da pseudo-instrução de **declaração de variáveis**, `DS`, que reserva a quantidade de espaço indicada mas não inicializa seu conteúdo. Por exemplo,

```
VALUE   DS.W    1
```

associa ao símbolo `VALUE` uma referência para um endereço de memória que tem espaço suficiente para armazenar valores de uma variável de tamanho dois bytes (word).

A pseudo-instrução `ORG` determina a **origem do segmento**. Um segmento é um conjunto de palavras de máquina que deve ocupar um espaço contíguo na memória principal. A posição de memória (endereço) associada ao início do segmento é denominada a sua origem. O módulo-objeto gerado pelo montador contém tipicamente pelo menos dois segmentos, um segmento de código de máquina e um segmento de dados.

O efeito da pseudo-instrução `ORG` depende do tipo de montador que irá interpretá-la. Em alguns casos, pode ser uma definição de um endereço absoluto de memória no qual a origem do segmento deve ser posicionada. Em outros, é apenas a definição de um nome para referências futuras ao segmento quando sua posição de origem for definida. A forma genérica aqui adotada para a instrução será

```
        ORG ident
```

onde `ident` é um identificador que pode ser um valor constante já definido (no caso dos montadores absolutos) ou estar sendo definido como o nome de um segmento (nos demais casos).

Por exemplo, no trecho de programa a seguir

```
SEG1    EQU     $1000
        ORG     SEG1
        MOVE.W  DATA,D0
        MOVE.W  D0,DATA+2
        RTS
```

indica que a primeira instrução `MOVE.W` (na terceira linha) estará alocada à posição $1000 da memória, dando início ao segmento de código do módulo-objeto que será gerado.

Uma outra pseudo-instrução importante é `END`, que indica ao montador o **fim do programa** *assembly*. Seu formato geral é

```
        END     ident
```

onde o identificador no operando está associado a um rótulo do início do programa que está sendo encerrado por essa pseudo-instrução. Assim, esse identificador só deve estar presente uma única vez no código, mesmo que o programa fonte esteja distribuído entre diversos arquivos — em geral, está associado a um "módulo principal". Nos demais módulos, a pseudo-instrução `END` aparece sem argumentos, sempre na última linha.

No caso de um programa cujo código-fonte está distribuído entre diversos segmentos, pode ser preciso fazer referências desde um segmento a variáveis definidas em outros arquivos-fontes. Para possibilitar essa conexão de referências, a pseudo-instrução `GLOB` é usada para indicar que cada um dos símbolos indicados pode ser **referenciável externamente**, ou seja, torna o símbolo visível globalmente. Seu formato genérico é

```
        GLOB    idents
```

onde `idents` é a lista de identificadores (separados por vírgulas, se mais de um estiver presente) definidos nesse segmento com a pseudo-instrução que podem ser referenciados a partir de outros módulos. Os demais símbolos definidos no segmento são considerados de escopo local, ou seja, são invisíveis para os módulos externos.

Alguns montadores definem pseudo-instruções tais como `EXTERN` para indicar que o símbolo que está sendo usado no módulo foi definido externamente, em outro módulo. No entanto, essa pseudo-instrução é desnecessária se for assumido que todos os símbolos referenciados mas não definidos localmente devem estar definidos externamente; esse comportamento é adotado pelo montador GNU, o programa `as`.

Macro-instruções

Uma macro-instrução é um sinônimo para um grupo de instruções que pode ser usado como uma instrução ao longo do código-fonte. O uso de macros facilita a especificação de trechos repetitivos de código, que podem ser invocados pelo programador como um única linha no programa. Por esse motivo, diversos montadores apresentam extensões com funcionalidades para a definição e utilização de macros.

Na sua forma mais simples, uma macro é simplesmente uma abreviatura para um grupo de instruções. A forma geral de definição de uma macro é

```
nome    MACRO   [argumentos]
        corpo
        ENDM
```

A pseudo-instrução `MACRO` marca o início da definição da macro-instrução. Toda macro tem um **nome**, especificado como o rótulo da pseudo-instrução e que será utilizado pelo programador para invocar a macro, e um **corpo**, que será usado pelo macro-montador para substituir o nome usado pelo programador pela seqüência de instruções nele especificados. A pseudo-instrução `ENDM` marca o fim da definição.

Uma macro pode opcionalmente receber argumentos, que serão usados para adaptar a expansão do corpo da macro. Assim, a seqüência de instruções especificadas no corpo da macro podem ser parametrizadas pelos argumentos. Os parâmetros formais na definição de uma macro-instrução são precedidos pelo símbolo `&`.

A definição de `TOLOWER`, exemplo apresentado abaixo, cria uma macro-instrução com dois parâmetros. O primeiro, `&IN`, é interpretado como a referência a um endereço de memória de um byte cujo conteúdo será copiado para o registrador `D0`, onde o sexto bit será setado. O conteúdo resultante será copiado para a posição indicada pelo segundo argumento, `&OUT`:

```
TOLOWER  MACRO     &IN,&OUT
         MOVE.B    &IN,D0
         ORI.B     32,D0
         MOVE.B    D0,&OUT
         ENDM
```

Uma vez que uma macro esteja definida, seu nome pode ser utilizado como se fosse uma operação válida do montador. A associação entre os argumentos da invocação da macro e os parâmetros formais da definição é feita pela posição da variável na declaração e invocação, assim como ocorre em subrotinas nas linguagens de alto nível.

Considerando a definição acima, o uso da macro `TOLOWER` dar-se-ia como em

```
SIZE     EQU       5
CHARS_I  DC.B      'EA876'
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     TOLOWER   (A0),(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Após passar pela etapa de processamento de macros, o código acima seria expandido para o seguinte código *assembly*:

```
SIZE     EQU       5
CHARS_I  DC.B      'EA876'
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     MOVE.B    (A0),D0
         ORI.B     32,D0
         MOVE.B    D0,(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Outra facilidade geralmente associada a macro-instruções é a possibilidade de definir expansões condicionais de trechos de código. Para tanto, a pseudo-instrução `AIF` é definida. Assim, é possível definir trechos da definição da macro que poderão não estar incluídos na respectiva expansão.

O formato dessa pseudo instrução é

```
        AIF     cond  .mrot
```

onde `cond` é a condição que deve ser avaliada e `.mrot` é o rótulo de macro onde a expansão deverá continuar se a condição for verdade; caso contrário, a expansão continua na linha seguinte. A condição envolve tipicamente operadores relacionais de comparação de *strings*, aqui denotados `EQ` e `NE` para expressar "igual a" e "diferente de", respctivamente. O rótulo de macro é sempre iniciado por um ponto, uma forma de diferenciá-lo dos rótulos que serão incluídos no código expandido.

Para ilustrar esse conceito, considere novamente a definição da macro `TOLOWER`, que usa o registrador `D0` para realizar a operação desejada. Se um dos argumentos para a macro for esse registrador, uma das instruções `MOVE` não deve ser incluída na sua expansão. Usando `AIF`, uma nova definição para essa macro que considera essa possibilidade é

```
TOLOWER  MACRO     &IN,&OUT
         AIF       (&IN EQ 'D0') .PULA
         MOVE.B    &IN,D0
.PULA    ORI.L     32,D0
         AIF       (&OUT EQ 'D0') .FIM
         MOVE.L    D0,&OUT
.FIM     ENDM
```

No Apêndice C.7 apresenta-se o pré-processador C, uma facilidade para definição de macros associada a uma linguagem de alto nível.

4.1.2 Montagem

O processo de montagem de um código *assembly* pode apresentar pequenas diferenças em função das opções adotadas no projeto do montador, mas em linhas gerais as funcionalidades a seguir são suportadas.

Uma etapa inicial que pode ser suportada é o pré-processamento do código, onde informação não relevante pode ser eliminada. Por exemplo, o pré-processador do montador `as` elimina comentários e converte constantes em formato caráter para as correspondentes constantes em valores numéricos. Na seqüência, o montador realiza o pré-processamento de macros, obtendo um código pronto para a criação do módulo objeto.

Funcionalidades básicas

O montador estará recebendo como entrada um arquivo em formato texto, do qual ele deverá ler cada linha para fazer o processamento que for necessário. Assim, um dos primeiros grupos de funcionalidades que se faz necessário é a manipulação de arquivos, descritas na Seção 2.8.

Uma vez obtida a linha do arquivo, a tarefa de extrair de cada linha o campo de interesse estará representada através dos seguintes procedimentos, todos recebendo como argumento uma referência para a linha a ser processada:

GETLABEL(): extrai o rótulo da linha, se presente; caso contrário, returna o valor nulo.

GETOPERATION(): extrai da linha o mnemônico de operação, que pode ser de uma instrução de máquina ou de uma pseudo-instrução.

GETOPERANDS(): obtém uma lista de operandos, que eventualmente pode ser vazia, a partir do conteúdo do campo de operandos da linha.

Processamento de macro-instruções

De maneira geral, um processador de macro deve realizar quatro tarefas básicas:

1. Reconhecer as definições de macros;
2. Salvar as definições de macros de forma possibilitar a posterior expansão;
3. Reconhecer as invocações a macros; e
4. Expandir as invocações, possivelmente substituindo argumentos e verificando condições.

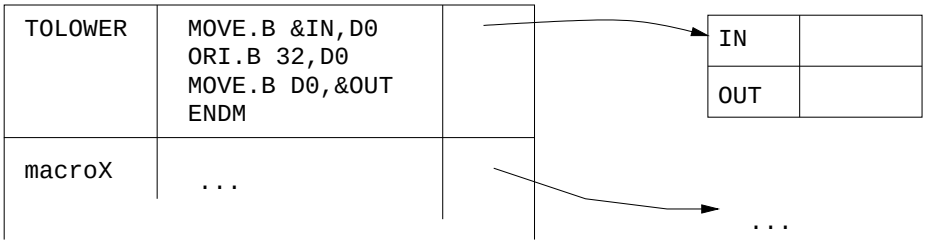
O processador de macros pode ser visto como um programa independente do montador, que é invocado antes do processo de montagem propriamente dito. Sua implementação mais simples pode ser realizada em dois passos.

No primeiro passo, cada linha do arquivo com o programa fonte (já sem comentários) é lida. Caso contenha na coluna do campo de operação a pseudo-instrução `MACRO`, então o que se segue é uma definição de macro, que deve ser armazenada. Uma estrutura de dados, a **Tabela de Definição de Macro**, é usada para guardar essas definições. A chave nessa tabela é o nome da macro, definido no campo de rótulo dessa mesma linha.

Associado a cada nome de macro-instrução, a tabela de definição de macro contém dois valores. Um valor é o corpo da definição da macro e o outro a sua lista de parâmetros formais.

Para obter a lista de parâmetros formais, o processador de macro verifica se essa lista está presente no campo de operandos da linha. Se estiver, cada membro da lista será associado a uma entrada em outra estrutura de dados auxiliar, a **Tabela da Lista de Argumentos**, que será referenciada na tabela de definição de macro (Figura 4.1).

Figura 4.1 Estruturas de dados no processamento de macros.



Para armazenar o corpo da macro, a linha a seguir é lida e copiada literalmente para a tabela. Verifica-se então se o campo de operação da linha copiada era `ENDM`; se sim, então a definição dessa macro é concluída. Caso contrário, o procedimento repete para a linha seguinte.

O primeiro passo do processador de macro é encerrado quando a pseudo-instrução `END` é encontrada, sinalizando o fim do código fonte.

No segundo passo, cada linha de entrada é novamente lida e o campo de operação é obtido. Se a operação especificada for `MACRO`, esta linha e todas as que a seguem, até aquela que contenha a operação `ENDM`, são ignoradas. Caso contrário, verifica-se se a operação está presente na tabela de definição de macro. Se não estiver, a linha é copiada para o arquivo de saída na sua forma original. Caso contrário, a linha contém uma invocação de macro-instrução que deve ser expandida.

Na expansão da macro, verifica-se se a tabela da lista de argumentos contém algum elemento. Caso haja argumentos, as *strings* com os nomes dos argumentos são associadas, como valores na tabela, aos parâmetros, que são as chaves nessa tabela.

A expansão da macro continua pela leitura de linhas de código da definição, a partir da tabela de definição de macros. Caso a linha contenha no campo de operação a pseudo-instrução `ENDM`, o processo de expansão

está concluído e continua a leitura do arquivo de entrada. Caso contrário, caso o campo de operando contenha algum nome iniciado pelo símbolo `&`, então esse nome é buscado na tabela da lista de argumentos para ser substituído pela *string* correspondente nesta expansão. Após essa substituição (se houver), a linha resultante é passada para o arquivo de saída.

O processamento de macros conclui quando a pseudo-instrução `END` é copiada para o arquivo de saída, o qual então conterá apenas linhas com instruções do processador ou pseudo-instruções, já sem comentários ou macro-instruções.

O processamento de macros pode ocorrer em um único passo caso se restrinja que todas as invocações a uma macro só podem ocorrer no código-fonte após sua definição, quando então os dois passos podem ser combinados.

Criação da tabela de símbolos

Criado o código *assembly* contendo apenas instruções de processador e pseudo-instruções, o passo seguinte é analisar as referências simbólicas contidas no código de forma a permitir a criação do módulo objeto. Nessa etapa do processamento, a atividade principal é a criação da **Tabela de Símbolos** do montador.

Na sua forma mais simples, a tabela de símbolos tem como chaves as *strings* com os nomes simbólicos definidos no programa *assembly*. Símbolos podem ser definidos como resultado de duas situações:

1. Como um rótulo em uma pseudo-instrução `EQU`; neste caso, o valor do símbolo está definido no campo do operando.

2. Como um rótulo em uma outra instrução; neste caso, o valor do símbolo está relacionado à posição de memória dentro do segmento onde ocorre a definição do símbolo.

Para poder criar a tabela de símbolos, o montador deve obter a informação sobre o espaço ocupado pelo código de máquina gerado por cada instrução do processador ou pseudo-instrução que tenha impacto na alocação de memória. Para tanto, o montador faz uso de duas estruturas auxiliares, a tabela de instruções de máquina e a tabela de pseudo-instruções.

A **Tabela de Instruções da Máquina** (ou MOT, de *machine operations table)* contém toda a informação necessária para permitir a tradução de um mnemônico para o código de máquina correspondente. A chave dessa tabela é o código de operação da instrução. Os valores incluem as regras para a geração do código de máquina e o espaço de memória que será ocupado pela instrução, em bytes. O conteúdo dessa tabela é determinado pelo processador para o qual o código está sendo gerado.

A **Tabela de Pseudo-Instruções** (ou POT, de *pseudo-operations table*) tem seu conteúdo é definido pelos projetistas do montador. Assim como a MOT, é uma tabela com conteúdo exclusivamente para consulta, não sendo modificado durante a execução do programa. Tem como um dos valores as regras para obter o espaço de memória que deve ser alocado em função da pseudo-instrução. Enquanto muitas das pseudo-instruções, tais como `EQU` ou `END`, não têm impacto sobre a ocupação de memória, `DS` e `DC` têm como efeito a necessidade de reservar e/ou modificar o conteúdo de posições de memória associadas ao programa. Assim, tais pseudo-instruções deverão gerar informação que irá fazer parte do módulo de carregamento gerado pelo montador.

A partir da informação derivada das tabelas do montador é possível saber quanto espaço de memória cada linha de instrução do programa fonte irá ocupar no módulo de carregamento gerado. Será a partir dessa informação que o montador poderá definir qual a posição a ser alocada para cada instrução do programa. Esta informação será mantida em uma variável **contador de localização** (LC, de *location counter*). Essa informação é transiente, ou seja, ela apenas existe durante a execução do montador.

4.1.3 Formato do módulo objeto

Com o processo de montagem, os segmentos do programa *assembly* são convertidos em arquivos no formato de módulo objeto, que serão posteriormente carregados para execução na memória. Tipicamente, um

arquivo objeto contém os seguintes itens de informação:

Cabeçalho: contém a identificação do tipo de arquivo e dados sobre o tamanho do código e eventualmente o arquivo que deu origem ao arquivo objeto;

Código gerado: contém as instruções e dados em formato binário, apropriado ao carregamento;

Relocação: contém as posições no código onde deverá ocorrer mudanças de conteúdo quando for definida a posição de carregamento;

Símbolos: contém os símbolos globais definidos no módulo e símbolos cujas definições virão de outros módulos;

Depuração: contém referências para o código fonte, tais como o número de linha, nomes originais dos símbolos locais e estruturas de dados definidas.

Nem sempre todas essas informações precisam estar presentes no módulo objeto. Por exemplo, um arquivo em formato COM no sistema operacional DOS contém apenas o código gerado. Neste caso, algumas restrições são impostas para garantir essa simplicidade. A posição de carregamento é pré-definida no endereço `0x100` de algum segmento livre e o tamanho do código não deve exceder a capacidade de endereçamento interno a um segmento (64 KBytes). Caso o arquivo exceda esse tamanho, o programador será responsável por garantir a operação correta do programa executável.

4.2 Montagem e carregamento combinados

O esquema mais simples que incorpora a montagem e o carregamento como atividades separadas na execução de programas *assembly* é o esquema **absoluto**. Nesse esquema, o montador gera um arquivo (módulo de carregamento) contendo, além do código de máquina, a informação necessária para que o programa carregador possa carregar o código de máquina nas posições corretas de memória e transferir a execução para o programa carregado.

Esse tipo de esquema é na prática bastante limitado. No entanto, apresenta diversas funcionalidades que permitem introduzir detalhes importantes da operação de carregadores e montadores.

4.2.1 Montadores em dois passos

Uma vez que o código fonte *assembly* já tenha suas macro-instruções expandidas, a etapa de montagem propriamente dita pode ser iniciada. Para essa descrição, considera-se também que na etapa de pré-processamento os comentários foram eliminados do código fonte. Assim, todas as linhas devem conter instruções *assembly* ou pseudo-instruções do montador.

Para introduzir os conceitos relacionados à montagem em dois passos, será inicialmente apresentado um exemplo motivador, apresentando um pequeno código *assembly* e o que seria gerado pelo montador a partir dele. Posteriormente, cada um dos passos do montador será detalhado.

Motivação

Considere como exemplo uma subrotina *assembly* que deverá transferir um valor armazenado em uma posição para outra posição de memória, usando o registrador `D0` como armazenador temporário do dado. A subrotina terá o nome `PGM`, a posição de memória que tem o dado original será rotulada `VALUE` e a posição de memória destino será rotulada `RESULT`.

A listagem a seguir apresenta o código fonte, que é simplesmente uma seqüência de caracteres armazenada em um arquivo texto. Como descrito na Seção 4.1.1, *strings* na primeira coluna denotam rótulos para as posições de memória associadas; a segunda coluna contém *strings* que representam as instruções (mnemônicos), e a terceira coluna contém *strings* representando os argumentos das instruções.

```
1    DATA     EQU      $6000
2    PROGRAM  EQU      $4000
3             ORG      DATA
4    VALUE    DS.W     1
5    RESULT   DS.W     1
6             ORG      PROGRAM
7    PGM      MOVE.W   VALUE,D0
8             MOVE.W   D0,RESULT
9             RTS
10            END      PGM
```

No primeiro passo de execução, o montador deve gerar a Tabela de Símbolos. Para o exemplo acima, a Tabela de Símbolos associada deve conter a seguinte informação:

Símbolo	**Valor**
DATA	$6000
PGM	$4000
PROGRAM	$4000
RESULT	$6002
VALUE	$6000

Há duas situações que devem ser consideradas para possibilitar a geração dessa tabela. A primeira situação, a mais simples, é quando a definição do símbolo é derivada de uma pseudo-instrução `EQU`. Esse caso é o mais simples porque toda a informação necessária para compor a entrada da tabela de símbolos está explícita na instrução. Assim foram definidos os símbolos `DATA` e `PROGRAM` na Tabela de Símbolos.

A segunda situação envolve a definição de símbolos usados como rótulos em outras instruções — no exemplo, o caso de `VALUE`, `RESULT` e `PGM`. Essa situação é mais complexa por necessitar conhecimento da posição no segmento ou na memória que a instrução estará ocupando para poder efetivamente definir seu valor. Para tanto, o contador de localização (LC) deve ser atualizado durante o primeiro passo de acordo com o espaço reservado para cada instrução, informação que deve ser derivada a partir das tabelas de instruções de máquina (MOT) e de pseudo-instruções (POT).

No segundo passo do montador, o código deve ser efetivamente gerado. No caso desse exemplo, dois segmentos serão produzidos.

O primeiro segmento corresponde a uma área de dados, gerada a partir da interpretação das pseudo-instruções `DS` — eventualmente, a interpretação de pseudo-instruções `DC` também poderiam gerar informação para esse segmento, se estivessem presente. O segmento gerado contém a seguinte informação:

Posição	$6000	$6002
Conteúdo	$0000	$0000

O segundo segmento corresponde à área de instruções, contendo o código de máquina associado a cada instrução. Para gerar esse código, o montador teve que *(i)* obter a codificação de máquina para cada instrução e *(ii)* resolver as referências simbólicas presentes nos operandos das instruções.

O conteúdo do segundo segmento, assumindo que endereços absolutos são representados como *long word*, contém:

Posição	$4000	$4002	$4004	$4006	$4008	$400A	$400C
Conteúdo	$3038	$0000	$6000	$31C0	$0000	$6002	$4E75

Na seqüência, serão analisados os procedimentos que o montador deve realizar para possibilitar essa geração de código.

Primeiro passo

Considerando a montagem em dois passos, o primeiro passo pode ser descrito pelo Algoritmo 4.1. Neste primeiro passo, a principal atividade é a manipulação de rótulos de forma a descobrir o símbolo que está sendo criado em cada linha, se for o caso, e manter o controle sobre qual a posição (valor) de definição do símbolo. Esta última informação é obtida a partir da atualização do contador de localização LC a partir da avaliação do tamanho das instruções anteriores.

Algoritmo 4.1 Passo 1 do montador.

```
ASSEMBLER1(source)
 1  LC ← 0
 2  file ← OPENFILE(source)
 3  while ¬ENDOFFILE(file)
 4  do instruction ← READLINE(file)
 5     label ← GETLABEL(instruction)
 6     opcode ← GETOPERATION(instruction)
 7     operand ← GETOPERAND(instruction)
 8     entry ← FINDTABLE(POT, opcode)
 9     if entry ≠ NIL
10       then switch opcode
11               case ORG :
12                    LC ← GETOPERANDVALUE(operand)
13               case END :
14                    ASSEMBLER2(file)
15                    return
16               case default :
17                    if opcode = EQU
18                      then value ← GETOPERANDVALUE(operand)
19                           length ← 0
20                      else value ← LC
21                           length ← GETINSTRUCTIONSIZE(entry, operand)
22                    if label ≠ NIL
23                      then INSERTTABLE(ST, label, value)
24                    LC ← LC + length
25       else entry ← FINDTABLE(MOT, opcode)
26            if entry ≠ NIL
27              then length ← GETINSTRUCTIONSIZE(entry, operand)
28                   if label ≠ NIL
29                     then INSERTTABLE(ST, label, LC)
30                   LC ← LC + length
31              else ERRORMESSAGE( "Invalid opcode" , instruction)
32                   CLOSEFILE(file)
33                   return
```

Nesse procedimento do primeiro passo do montador, o arquivo fonte é manipulado linha a linha. Para

cada linha, o código de operação é analisado para descobrir se a instrução é uma pseudo-instrução (código de operação encontrado na POT) ou uma instrução *assembly* (código de operação encontrado na MOT). Caso o código não esteja em nenhuma das duas tabelas, uma mensagem de erro deve ser apresentada indicando que a instrução não foi reconhecida e o processo deve ser abortado.

Observe que duas buscas devem ser realizadas para cada linha de instrução obtida do arquivo fonte. Como a Tabela de Pseudo-Instruções é bem menor que a Tabela de Instruções de Máquina, a busca é inicialmente realizada na primeira tabela. Apenas se o código buscado não for encontrado na POT a busca será realizada na tabela maior. A eficiência de implementação destas buscas irá se refletir diretamente na eficiência do montador. Por este motivo, é importante que bons algoritmos de busca e de manutenção da informação em tabelas sejam adotados na implementação do montador.

A atualização da Tabela de Símbolos (ST) é efetivada durante o processamento das pseudo-instruções com código de operação `EQU`, `DC` ou `DS` (estas duas tratadas na condição *else* no caso *default*) ou durante o processamento de instruções *assembly* com campo de rótulo não-nulo. Para a pseudo-instrução `EQU`, o rótulo é o nome do símbolo cujo valor deve ser obtido do operando. No caso das pseudo-instruções `DC` e `DS`, o rótulo é o nome do símbolo cujo valor é a posição corrente da instrução.

No processamento das demais pseudo-instruções, nenhum símbolo é criado. Se a pseudo-instrução é `ORG`, apenas o contador de localização deve ser atualizado. A pseudo-instrução `END` deve encerrar o primeiro passo do montador, invocando o segundo passo.

O processamento das pseudo-instruções `ORG` e `EQU` requerem, já nesse passo, uma avaliação do valor do operando. Esse operando pode ser um literal ou um símbolo previamente definido. Caso seja um literal, o valor do operando é obtido a partir da conversão da *string* que representa o valor em alguma base — binária, octal, decimal ou hexadecimal. Caso seja um símbolo previamente definido, o valor é obtido a partir da busca da Tabela de Símbolos. Esse processamento auxiliar é realizado pela rotina GETOPERANDVALUE.

O procedimento do montador deve ainda avaliar o espaço ocupado pela instrução sendo processada, com o fim de atualizar corretamente o contador de localização. Para tanto, utiliza-se informação contida na tabela correspondente *(entry)* e o campo do operando, conforme indicado nas invocações à rotina GETINSTRUCTIONSIZE.

Usando a rotina do exemplo motivador (Seção 4.2.1), é possível acompanhar o processo de atribuição de valores a símbolos que ocorre durante o primeiro passo da montagem. À medida que as linhas de códigos forem lidos, o contador de posição LC vai assumir os valores apresentados na primeira coluna, em hexadecimal:

LC	**Instrução**		
0000	DATA	EQU	$6000
0000	PROGRAM	EQU	$4000
0000		ORG	DATA
6000	VALUE	DS.W	1
6002	RESULT	DS.W	1
6004		ORG	PROGRAM
4000	PGM	MOVE.W	VALUE,D0
4006		MOVE.W	D0,RESULT
400C		RTS	
400E		END	PGM

O valor inicial do LC é 0. A pseudo-instrução `EQU` não ocupa espaço no código gerado e portanto não altera o valor do LC. O efeito das duas primeiras instruções é registrar na Tabela de Símbolos a informação que os símbolos `DATA` e `PROGRAM` têm os valores $6000 e $4000, respectivamente.

A pseudo-instrução `ORG` da terceira linha também não ocupa espaço de código, mas tem efeito sobre o LC — ele passará a registrar a posição de memória para a próxima linha do programa. Assim, LC passará a $6000, que é o valor de `DATA` obtido da Tabela de Símbolos.

A linha seguinte reserva espaço para a variável VALUE. Na Tabela de Símbolos será registrado para o rótulo (o símbolo VALUE) o valor de LC ($6000). Como o sufixo do tamanho para a pseudo-instrução DS é .W, será reservado espaço para uma *word* (dois bytes). Portanto, o LC é incrementado para o valor $6002.

Da mesma forma, na linha seguinte, ao símbolo RESULT será associado o valor $6002 na Tabela de Símbolos e o LC será incrementado para $6004.

Com a pseudo-instrução ORG da sexta linha o LC será alterado para $4000. Deste modo, ao rótulo definido na instrução seguinte, PGM, é associado na Tabela de símbolos o valor $4000.

Para incrementar corretamente o LC, é preciso saber quantos bytes serão ocupados pela instrução da sétima linha. A partir do tratamento dos operandos e assumindo que endereços absolutos são representados em quatro bytes *(long word)*, encontra-se a informação que seis bytes serão usados para esta instrução, de forma que o LC será incrementado para $4006. Da mesma forma, encontra-se que a instrução seguinte também ocupa seis bytes, e o LC é incrementado para $400C.

Finalmente, no processamento da instrução RTS encontra-se que ela ocupa dois bytes, sendo que o valor do LC passa a $400E. A pseudo-instrução END simplesmente indica ao montador o fim do programa fonte, tendo como argumento um endereço da primeira instrução executável.

Segundo passo

Uma vez que todos os símbolos que podem vir a ser usados como operandos das instruções no código-fonte já foram avaliados no primeiro passo do montador, é possível concluir a montagem. A atividade do montador no segundo passo é a geração do código de máquina.

A estrutura básica do segundo passo do montador é similar àquela do primeiro passo. O procedimento deve ler todas as instruções do programa fonte e, para cada linha, gerar o código de máquina correspondente. No processamento de operandos de cada instrução pode ser necessário realizar consultas à Tabela de Símbolos para obter os valores dos operandos simbólicos.

O procedimento preciso para o segundo passo do montador depende do tipo de carregador associado. Nesta seção será apresentada uma estrutura genérica desse algoritmo (Algoritmo 4.2) para descrever as tarefas realizadas nesse passo; detalhes adicionais serão apresentados na seqüência, quando forem apresentadas algumas opções de carregamento.

Nesse procedimento são introduzidas novas rotinas auxiliares de processamento das instruções. A rotina GENERATEMACHINECODE produz o código de máquina associado ao código de operação da instrução e aos seus operandos. Tipicamente, na entrada da tabela correspondente à instrução sendo processada há uma referência para uma função que é capaz de realizar esse processamento. Por exemplo, o "código" gerado para a pseudo-instrução DS pode ser simplesmente uma seqüência de zeros no tamanho reservado pela instrução. Para a pseudo-instrução DC, o código gerado deve corresponder ao processamento dos literais e símbolos do operando. Para qualquer instrução *assembly*, essa rotina utiliza informação da MOT e o processamento dos literais e símbolos do operando para gerar o código de máquina correspondente.

Outra rotina auxiliar é ASSEMBLEMACHINECODE, que posiciona esse código gerado no módulo de carregamento. ASSEMBLECLOSINGCODE tem a função de fechar o módulo de carregamento, podendo adicionalmente realizar outras tarefas associadas ao encerramento do processo de montagem.

4.2.2 Montagem e carregamento *assemble and go*

A forma mais elementar para executar o código de máquina gerado pelo montador é através do esquema *assemble and go*, no qual um único programa de sistema combina a realização das tarefas associadas a um montador e a um carregador.

O esquema *assemble and go*, como o nome sugere ("monta e executa"), combina as etapas de montagem e carregamento em um único programa. Neste caso, não há a criação de um arquivo com o módulo objeto. Quando o código de máquina é gerado pelo montador ele é colocado diretamente na posição de memória

Algoritmo 4.2 Passo 2 do montador.

```
ASSEMBLER2(file)
 1  LC ← 0
 2  REWINDFILE(file)
 3  while ¬ENDOFFILE(file)
 4  do instruction ← READLINE(file)
 5     opcode ← GETOPERATION(instruction)
 6     operand ← GETOPERAND(instruction)
 7     switch opcode
 8        case EQU :
 9             /* nothing */
10        case ORG :
11             LC ← GETOPERANDVALUE(operand)
12        case END :
13             startAddress ← GETOPERANDVALUE(operand)
14             ASSEMBLECLOSINGCODE(startAddress)
15             CLOSEFILE(file)
16             return
17        case default :
18             if opcode = DC ∨ opcode = DS
19               then entry ← FINDTABLE(POT, opcode)
20               else entry ← FINDTABLE(MOT, opcode)
21             length ← GETINSTRUCTIONSIZE(entry, operand)
22             code ← GENERATEMACHINECODE(entry, operand)
23             ASSEMBLEMACHINECODE(LC, code)
24             LC ← LC + length
```

indicada pelo contador de localização. Assim, ao final da montagem o programa executável já está em memória e o montador simplesmente transfere o controle de execução para a primeira instrução executável do código de máquina gerado.

Nesse tipo de esquema, a rotina ASSEMBLEMACHINECODE no passo 2 do montador simplesmente copia o código gerado para a posição de memória indicado pelo contador de localização. A rotina ASSEMBLECLOSINGCODE simplesmente transfere a execução (através de uma instrução de desvio incondicional) para o início do programa montado.

A grande desvantagem do esquema *assemble and go* está no fato de que cada execução do programa requer uma nova montagem, mesmo que o programa não tenha sido alterado. Outra desvantagem está no fato de que dois programas devem obrigatoriamente ocupar a memória principal, o montador e o programa montado. Assim, a utilização desse esquema está restrita a sistemas muito simples, não sendo de utilidade na prática.

4.3 Carregamento absoluto

Uma outra forma simples para contornar as desvantagens do esquema *assemble and go* consiste em separar o processo de montagem do processo de execução do código montado. Nesse caso, o montador gera um módulo de carregamento que não precisa ser regenerado a cada execução. Adicionalmente, as funcionalidades do montador não são necessárias para a execução — assim, o espaço de memória ocupado pelo programa montador pode ser liberado durante a execução do programa montado.

Para o carregamento absoluto o módulo de carregamento contém, além do código objeto, a informação sobre as posições de memória para as quais as linhas de código devem ser carregadas. Uma possível estratégia é associar um registro do arquivo objeto a cada segmento, sendo que o início do registro indica a posição de carregamento e o tamanho do segmento em bytes.

O módulo de carregamento para o carregador absoluto é composto por dois tipos de registro. Todos os registros, exceto um, contêm informação que deve ser transferida para a memória na posição indicada (registro tipo 0). O outro tipo de registro deve ter apenas uma ocorrência no fim do módulo de carregamento, correspondendo à informação do endereço para início da execução do programa (registro tipo 1). Registros do tipo 0 são gerados no segundo passo do montador pela rotina ASSEMBLEMACHINECODE, enquanto o registro do tipo 1 é gerado pela rotina ASSEMBLECLOSINGCODE.

No exemplo do código gerado na Seção 4.2.1, a organização do módulo de carregamento segundo esse esquema seria composto por três registros:

```
0  00006000  4  00000000
0  00004000  E  30380000600031C0000060024E75
1  00004000
```

O primeiro campo de cada registro indica o tipo do registro. O valor `0` neste campo indica que o conteúdo a seguir (quarto campo), de dimensão quatro bytes (informação no terceiro campo) deverá ser transferido à memória a partir da posição $6000 (informação do segundo campo). Similarmente, a informação do segundo registro indica a transferência dos 14 bytes do quarto campo a partir da posição $4000. Finalmente, o último registro tem no primeiro campo o valor `1`, indicando que a execução deverá ser transferida para a posição indicada no segundo campo ($4000).

Para esse esquema de carregamento, o algoritmo do carregador absoluto é apresentado no Algoritmo 4.3.

Algoritmo 4.3 Carregador absoluto.

```
ABSOLUTELOADER(module)
 1  file ← OPENFILE(module)
 2  while ¬ENDOFFILE(file)
 3  do register ← READLINE(file)
 4     type ← GETREGISTERTYPE(register)
 5     address ← GETREGISTERADDRESS(register)
 6     if type = 0
 7       then size ← GETREGISTERLENGTH(register)
 8            code ← GETREGISTERCONTENT(register, size)
 9            MOVETOMEMORY(address, code, size)
10       else GOTO(address)
```

As rotinas auxiliares aqui utilizadas são de funcionalidade simples, servindo para extrair os diversos campos do registro do módulo de carregamento (GETREGISTERTYPE, GETREGISTERADDRESS, GETREGISTERLENGTH e GETREGISTERCONTENT), para transferir código do programa para a posição especificada de memória (MOVETOMEMORY) ou para transferir a execução para o endereço especificado (GOTO).

4.4 Relocação e Ligação

Os esquemas de montagem e carregamento absolutos, por sua simplicidade, não apresentam a flexibilidade necessária ao uso em sistemas operacionais modernos. Uma forte limitação está no fato de que o programador

deve ter acesso direto a posições de memória, especificando exatamente em que região da memória o programa e seus dados serão carregados através da pseudo-instrução `ORG`.

Em sistemas operacionais modernos, tal limitação inviabiliza o uso daqueles esquemas. A memória é um recurso controlado pelo sistema, sendo que o programador não deve estar amarrado a conhecer posições da memória física para que o seu programa funcione corretamente. Por outro lado, desenvolver um programa completamente independente de sua localização é uma atividade complexa, embora possível. A solução é deixar que o software de sistema resolva problemas relacionados com posicionamento do código através da *relocação*.

Outro recurso que também requer a colaboração do montador e do carregador para seu funcionamento é a combinação, ou *ligação*, de módulos interdependentes mas montados independentemente. Neste caso, deve ser possível a partir de um módulo fazer uma referência a um símbolo definido em outro módulo. No esquema de montador absoluto apresentado, tal situação geraria uma condição de erro pelo símbolo não estar definido, ou seja, não ter um endereço associado. Qualquer referência a símbolos externos deveria ser resolvida manualmente pelo programador. Com esquema de montagem e carregamento ajustáveis, o montador recebe a informação de que um símbolo está definido em outro módulo ou de que um símbolo estará sendo referenciado por outro módulo. Esta informação é registrada junto ao módulo objeto para uso pelo carregador, que realiza a resolução destes símbolos entre os módulos envolvidos.

4.4.1 Estruturas de dados adicionais

Os dois tipos de ajustes que podem ocorrer no conteúdo do módulo objeto são:

relocação: ajuste interno ao segmento;

ligação: ajuste entre segmentos distintos.

A atividade de relocação é realizada conjuntamente por montadores e carregadores. Montadores são encarregados de marcar as posições no código objeto passíveis de alteração devido à relocação do código. Carregadores devem reservar um espaço na memória de tamanho suficiente para receber o código de máquina e atualizar suas posições alteráveis a partir da informação sobre sua localização na memória.

No exemplo da Seção 4.2.1, as palavras que começam nas posições $4002 e $4008 do código objeto contêm endereços relocáveis. Verificando o código gerado, observa-se que a posição $4002–$4003 contém uma referência ao endereço $6000, e a posição $4008–$4009 contém uma referência ao endereço $6002. Se o início do segmento de dados for alocado a outro endereço de memória que não $6000, o conteúdo destas posições de memória deverá ser ajustado de acordo com esta mudança. O programa carregador é o encarregado de realizar estes ajustes. Para tanto, o módulo objeto deverá conter informação adicional que permita a realização dos ajustes.

Outro tipo de informação que deverá ser mantida no módulo objeto refere-se a referências aos símbolos externos. Neste caso, há duas situações que podem ser tratadas:

1. o símbolo é referenciado neste segmento, mas é definido em outro segmento; e
2. o símbolo é definido neste segmento e poderá ser referenciado em outro segmento.

A primeira situação é usualmente descrita como uma *referência externa* (ER), enquanto que a segunda situação será descrita como uma *definição local* (LD) de um símbolo externamente referenciável. A informação sobre estes dois tipos de símbolos deverá estar presente no módulo objeto.

Na seqüência, analisaremos o esquema usual de resolução de relocação e referências externas — através de carregadores de *ligação direta*. Para este tipo de carregadores, o montador deverá incluir no módulo objeto estruturas de dados adicionais que incluam a informação necessária. São elas:

Dicionário de Símbolos Externos (ESD): contém todos os símbolos que podem estar envolvidos no processo de resolução de referências entre segmentos: símbolos associados a referências externas (ER), a definições locais (LD) ou a definições de segmentos (SD);

Diretório de Relocação e Ligação (RLD): para cada segmento indica que posições deverão ter seus conteúdos atualizados de acordo com o posicionamento deste e de outros segmentos na memória.

Estas duas estruturas de informação deverão estar presentes no módulo objeto. A partir deles, o carregador de ligação direta deve ser capaz de definir os valores para todos os símbolos com referências entre segmentos e reajustar o conteúdo das posições afetadas pela relocação.

O montador absoluto oferecia como resultado um módulo objeto com dois tipos de registros, registro com código de máquina (tipo 0) e um registro de fim (tipo 1). Um montador trabalhando no esquema de ligação direta deve fornecer dois tipos adicionais de registros além destes, um tipo para ESD e outro para RLD. Uma estrutura simplificada destes tipos de registros é indicada a seguir.

Registros do tipo ESD contêm todos os símbolos definidos no segmento que podem ser referenciados por outros segmentos, além de símbolos referenciados mas não definidos no segmento. Os símbolos locais que podem ser referenciados externamente podem ainda ser de dois tipos, definição do segmento ou definição local. Nos exemplos a seguir, um registro deste tipo apresentará a seguinte estrutura:

1. Tipo do registro (0)
2. Símbolo
3. Tipo de definição (SD — segmento, ou LD — local)
4. Endereço relativo no segmento
5. Comprimento em bytes

Neste modelo simplificado de montagem e carregamento por ligação direta apresentado aqui, definições do tipo ER não receberão tratamento diferenciado.

Registros do tipo TXT contêm o código de máquina, com a informação do endereço relativo incorporada. O formato deste registro é:

1. Tipo do registro (1)
2. Endereço relativo
3. Comprimento em bytes
4. Código de máquina

Registros do tipo RLD indicam quais posições no segmento deverão ter conteúdo alterado de acordo com os endereços alocados aos segmentos, indicando também a partir de que símbolo o conteúdo deverá ser corrigido. O formato deste registro adotado neste texto é:

1. Tipo de registro (2)
2. Posição relativa
3. Comprimento em bytes
4. Símbolo (base de ajuste)

Finalmente, um registro do tipo END especifica o endereço de início de execução para o segmento que contém a "rotina principal", sendo vazio para os demais segmentos:

1. Tipo de registro (3)
2. Endereço de execução

4.5 Carregamento e ligação combinados

Assim como esquemas primitivos de montagem já incorporavam o processo de carregamento (Seção 4.2), estratégias iniciais de ligação eram combinadas ao processo de carregamento.

Uma das estratégias primitivas adotada em alguns sistemas era o esquema de ligação por **vetor de transferência**, que reservava ao início da área de carregamento um espaço onde os endereços efetivos de rotinas seriam definidos. No código montado, as referências às rotinas eram "desviadas" para posições no vetor de transferência; na posição correspondente, encontrava-se uma instrução de desvio para a posição efetiva de memória onde a rotina havia sido carregada. A limitação desse tipo de programa é que apenas referências externas a rotinas podiam ser resolvidas automaticamente.

Nesta seção apresentaremos as atividades desempenhadas por um **carregador de ligação direta**, outro esquema simples que combina carregamento e ligação em um único programa. Carregadores de ligação direta permitem a resolução a rotinas e dados externos, representando um avanço em relação ao esquema de vetor de transferência.

A operação do carregador de ligação direta será apresentada a partir de um exemplo simples. Considere o seguinte programa, que faz referência a um símbolo externo `DIGIT`:

```
1     MAIN    MOVE.B    DIGIT,D0
2             CMPI.B    #10,D0
3             BLT       ADD_0
4             ADDQ.B    #('A'-'0'-10),D0
5     ADD_0   ADDI.B    #'0',D0
6             MOVE.B    D0,CHAR
7             RTS
8     CHAR    DS.W      1
9             END       MAIN
```

Este programa obtém um valor inteiro entre 0 e 15 de `DIGIT` e irá colocar na variável `CHAR` sua representação ASCII, entre '0' e 'F'.

No segmento onde `DIGIT` é definido, é preciso indicar que este símbolo poderá ser referenciado externamente. Para tanto, a pseudo-instrução `GLOB` é utilizada.

O trecho a seguir ilustra a definição de `DIGIT` em outro segmento:

```
1             GLOB      DIGIT
2     PGM     MOVE.W    VALUE,D0
3             MOVE.W    D0,DIGIT
4             RTS
5     VALUE   DS.W      1
6     DIGIT   DS.W      1
7             END
```

O efeito da pseudo-instrução `GLOB` será a criação de um registro do tipo `ESD` com tipo de definição `LD` — quando a posição relativa do símbolo for definida na tabela de símbolos locais, a informação do registro deverá ser complementada.

O montador deverá gerar o seguinte módulo objeto (com campos separados por pontos) para o segmento `MAIN`:

```
0.'MAIN'.'SD'.00.1C
1.00.6.103900000000
1.06.4.0C00000A
1.0A.2.6D02
```

```
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
2.02.4.'DIGIT'
2.14.4.'MAIN'
3.00
```

Neste exemplo, valores numéricos são apresentados em hexadecimal e símbolos na forma de seqüências ASCII — na realidade, o módulo objeto teria apenas a seqüência de bits associada a cada uma destas representações.

O início do módulo objeto contém o diretório de símbolos externos (ESD, registros com primeiro campo com valor 0), o código de máquina gerado (TXT, registros com primeiro campo 1), o diretório de relocação e ligação (RLD, registros com primeiro campo 2) e o registro de fim de segmento (END, com primeiro campo 3). Para o registro de fim de segmento, a posição relativa de execução (posição 00) é especificada.

Similarmente, para o segmento `PGM` o seguinte módulo é gerado:

```
0.'PGM'.'SD'.00.12
0.'DIGIT'.'LD'.10.2
1.00.6.30390000000E
1.06.6.33C000000010
1.0C.2.4E75
1.0E.2.0000
1.10.2.0000
2.02.4.'PGM'
2.08.4.'PGM'
3.
```

4.5.1 Algoritmos do carregador de ligação direta

O carregador de ligação direta recebe como argumentos a lista de módulos a carregar, trabalhando usualmente em vários passos — tipicamente dois.

O carregador apresentado aqui irá realizar três passos. No primeiro passo, ele deve alocar espaço contíguo de memória suficiente para os segmentos. Para saber quanto espaço é necessário, a informação sobre o comprimento de cada segmento — presente em registros tipo ESD, com tipo de definição SD — é obtida. O Algoritmo 4.5.1 apresenta um algoritmo para realizar esta etapa do primeiro passo.

Deve-se observar que, caso a restrição de que o espaço alocado aos módulos não precise ser contíguo, esse primeiro passo é dispensável.

Uma vez determinado qual o endereço inicial de carregamento (IPLA — *Initial Program Load Address*), o carregador inicia a criação de uma **Tabela de Símbolos Externos Globais** (GEST). Para tanto, apenas a informação presente em registros do tipo ESD, com tipos de definição SD e LD, é utilizada. Este segundo passo é apresentado no Algoritmo 4.5.

Nestas apresentações simplificadas do algoritmo de carregamento, o tratamento de erros não é indicado. Na fase de definição da GEST, um possível erro que poderia ser detectado e indicado ao usuário é a duplicação na definição de símbolos na tabela, ou seja, um mesmo símbolo sendo redefinido em segmentos distintos.

No último passo sobre os arquivos de entrada, o carregador irá realizar a transferência do código de máquina para a memória e transferir o controle da execução do programa para o endereço inicial do programa recém-carregado. Este passo é apresentado no Algoritmo 4.6.

Neste passo, o carregador volta a tomar como endereço inicial de carregamento o valor *ipla*, lendo novamente cada módulo objeto na seqüência original. A variável *execPoint* irá registrar a posição de início de execução para o segmento que definir um registro do tipo END com argumento.

Algoritmo 4.4 Primeiro passo do carregador de ligação direta: alocação de memória.

```
DLLOADER1(moduleList)
 1  total ← 0
 2  for each module in moduleList
 3  do file ← OPENFILE(module)
 4     found ← false
 5     repeat
 6           record ← READLINE(file)
 7           type ← GETRECORDTYPE(record)
 8           if type = 'ESD'
 9             then deftype ← GETDEFINITIONFIELD(record)
10                  if deftype = 'SD'
11                    then length ← GETLENGTHFIELD(record)
12                         total ← total + length
13                         found ← true
14       until found
15     CLOSEFILE(file)
16  ipla ← ALLOCMEMORY(total)
```

Algoritmo 4.5 Segundo passo do carregador de ligação direta: definição da GEST.

```
LDLOADER2(moduleList, ipla)
 1  GEST ← CREATETABLE()
 2  segLength ← 0
 3  segStart ← ipla
 4  for each module in moduleList
 5  do file ← OPENFILE(module)
 6     while ¬ENDOFFILE(file)
 7     do record ← READLINE(file)
 8        type ← GETRECORDTYPE(record)
 9        if type = 'ESD'
10          then deftype ← GETDEFINITIONFIELD(record)
11               if deftype = 'S'
12                 then value ← segStart
13                      segLength ← GETLENGTHFIELD(record)
14                 else value ← segStart + GETPOSITIONFIELD(record)
15               symbol ← GETSYMBOLFIELD(record)
16               INSERTTABLE(GEST, symbol, value)
17          else if type = 'END'
18                 then segStart ← segStart + segLength
19                      CLOSEFILE(file)
```

Algoritmo 4.6 Terceiro passo do carregador de ligação direta: transferência de código e início de execução.

```
LDLOADER3(moduleList, ipla, GEST)
 1  execPoint ← ipla
 2  segStart ← ipla
 3  segLength ← 0
 4  for each module in moduleList
 5  do file ← OPENFILE(module)
 6     while ¬ENDOFFILE(file)
 7     do record ← READLINE(file)
 8        type ← GETRECORDTYPE(record)
 9        switch type
10          case 'ESD' :
11               deftype ← GETDEFINITIONFIELD(record)
12               if deftype = 'S'
13                 then segLength ← GETLENGTHFIELD(record)
14                      segSymbol ← GETSYMBOLFIELD(record)
15          case 'TXT' :
16               position ← segStart + GETPOSITIONFIELD(record)
17               length ← GETLENGTHFIELD(record)
18               code ← GETCODEFIELD(record)
19               MOVETOMEMORY(position, code, length)
20          case 'RLD' :
21               position ← segStart + GETPOSITIONFIELD(record)
22               length ← GETLENGTHFIELD(record)
23               defType ← GETDEFINITIONFIELD(record)
24               if deftype = 'X'
25                 then symbol ← GETSYMBOLFIELD(record)
26                      newValue ← FINDTABLE(GEST, symbol)
27                 else base ← FINDTABLE(GEST, segSymbol)
28                      MOVEFROMMEMORY(position, oldValue, length)
29                      newValue ← oldValue + base
30               MOVETOMEMORY(position, newValue, length)
31          case 'END' :
32               position ← GETPOSITIONFIELD(record)
33               if position ≠ NIL
34                 then execPoint ← segStart + position
35               segStart ← segStart + segLength
36               CLOSEFILE(file)
37  GOTO(execPoint)
```

Quando o registro lido é do tipo ESD, o único processamento envolvido é obter o comprimento do segmento de forma a permitir a atualização correta da variável que indica a posição inicial de carga de cada segmento, *segStart*. Esta informação está contida no registro ESD cujo tipo de definição é SD (*Segment Definition*).

Os registros do tipo TXT têm seu conteúdo transferido para a memória principal. Cada campo do registro — posição relativa ao início do segmento, tamanho e conteúdo — é obtido, sendo que o endereço de destino é resolvido tomando por base o valor de *segStart*.

Ao final da transferência, as posições indicadas em registros do tipo RLD têm seu conteúdo ajustado a partir da informação registrada na GEST. Os registros do tipo RLD têm a indicação da posição relativa que deve ser corrigida, sendo que a posição de memória cujo conteúdo será alterado, *position*, é obtida a partir da combinação desta informação com o endereço de início do segmento, *segStart*. O valor pelo qual o conteúdo deverá ser alterado é especificado pelo campo de símbolo presente neste registro — o símbolo é lido do registro e seu valor é obtido a partir de uma busca na GEST. Neste ponto, pode se detectar um erro caso algum símbolo tenha sido referenciado e não definido em nenhum módulo — é onde se processará a informação para referências do tipo ER, não tratadas explicitamente pelo algoritmo.

4.5.2 Exemplo de aplicação

Considere a aplicação desses algoritmos de carregamento com ligação direta ao exemplo apresentado acima. Dois arquivos de módulo objeto são passados como argumentos ao carregador, um para o segmento `MAIN` e outro para o segmento `PGM`.

Na fase de alocação de memória, o carregador obtém a informação que o segmento `MAIN` tem 28 bytes (1C em hexadecimal), enquanto que o segmento `PGM` tem 18 bytes. Assim, o carregador deve requisitar a alocação de uma área de 46 bytes ao sistema operacional.

Considere que o sistema operacional alocou esta área a partir da posição de memória $1000, que será o valor de *ipla*. Na fase de criação da `GEST`, o primeiro registro lido é a definição de `MAIN`, que receberá o valor de *segStart*, inicializado com $1000. Assim, o par `(MAIN,$1000)` é inserido na tabela. A variável *segLength* receberá o valor $1C. Os demais registros deste módulo, de tipo 1 e 2, são ignorados nesta fase. Quando o último registro é lido, de tipo 3 (`END`), o valor de *segLength* é atualizado para $101C.

Quando o segundo módulo é processado neste segundo passo, o primeiro registro é também a definição de um segmento, `PGM`. O par `(PGM,$101C)` é então inserido na `GEST`, sendo *segLength* atualizado para $12. O segundo registro é a definição de um símbolo local. Neste caso, a variável *position* recebe o valor $10 e a variável *newValue* recebe $102C. Assim, a `GEST` receberá a definição do par `(DIGIT,$102C)`. Como para o primeiro módulo, os demais registros são ignorados, até a leitura do registro do tipo 3. Como não há outros módulos a varrer, o segundo passo é encerrado, com o valor de *segStart* sendo atualizado para $102E.

Ao final deste passo, a `GEST` apresentará o seguinte conteúdo:

Símbolo	**Valor**
MAIN	$1000
PGM	$101C
DIGIT	$102C

No terceiro passo, o valor das variáveis *segStart* e *execPoint* são reinicializados para $1000. No processamento do primeiro módulo, o efeito do registro tipo 0 (`ESD`) é simplesmente atribuir a *segLength* o valor $1C. Os registros do tipo 1 são então processados, sendo que para o primeiro deles o conteúdo `103900000000`, de dimensão 6 bytes, é transferido para a posição $1000+$00 de memória. O segundo registro indica a transferência de `0C00000A` (4 bytes) para a posição $1000+$06 de memória, e assim consecutivamente. Ao final destas transferências, a memória apresenta o seguinte conteúdo (em hexadecimal):

Posição	00	02	04	06	08	0A	0C	0E
$1000	1039	0000	0000	0C00	000A	6D02	5E00	0600
$1010	0030	13C0	0000	001A	4E75	0000		

Os registros do tipo 2 (`RLD`) são então lidos. O primeiro deles indica que os 4 bytes a partir da posição relativa $02, ou seja, a *long word* na posição de memória *segStart*+$02 ou $1002, deve ser atualizada pelo valor do símbolo `DIGIT`. A pesquisa na `GEST` indica que este símbolo tem o valor $102C, que somado ao conteúdo anterior da posição ($00000000) resulta em

Posição	*Conteúdo*
$1002	0000
$1004	102C

Observe que este é um endereço externo ao segmento sendo processado, ou seja, este procedimento corresponde a uma tarefa de ligação.

O segundo registro deste tipo no primeiro segmento indica que 4 bytes na posição relativa $14, ou seja, a *long word* na posição de memória $1014, deve ser atualizada pelo valor do símbolo `MAIN`. O conteúdo inicial desta posição é $0000001A, e o valor de `MAIN`, $1000, é obtido da `GEST`. Assim, estas posições de memória são atualizadas para

Posição	*Conteúdo*
$1014	0000
$1016	101A

Observe que este é o endereço de uma variável definida neste próprio segmento, que teve de ser reajustada por relocação.

O processamento do primeiro módulo se encerra com a definição da variável `EXE` para o valor $1000 após a leitura do registro de tipo 3.

Para o segundo módulo, o procedimento se repete, agora com *segStart* com valor $101C. Após a transferência de código a partir desta posição de memória, o processamento de registros de tipo 2 realizarão ajustes de relocação apenas no código deste segmento — a *long word* na posição $101E (*segStart*+2) receberá o valor $0000102A ($0000000E+`PGM`) e a *long word* na posição $1024 (*segStart*+8) receberá o valor $0000102C ($00000010+*segStart*).

Deste modo, o conteúdo completo da memória após o processamento do segundo módulo será

Posição	00	02	04	06	08	0A	0C	0E
$1000	1039	0000	102C	0C00	000A	6D02	5E00	0600
$1010	0030	13C0	0000	101A	4E75	0000	3039	0000
$1020	102A	33C0	0000	102C	4E75	0000	0000	

O carregador encerra seu processamento transferindo o controle (através de instrução JUMP) para a posição *execPoint* ($1000) de memória.

4.6 Ligadores

A estratégia de ligação direta apresentada na Seção 4.5 ilustra bem o princípio de resolução de endereços entre módulos, mas ainda apresenta limitações. Uma das limitações é que o programa carregador é mais complexo que o carregador absoluto, ocupando mais espaço em memória. Como o carregador compartilha memória com o programa sendo executado, menos memória é deixada para a aplicação.

Uma estratégia alternativa é isolar os procedimentos de ligação e de carregamento em programas separados. O programa *ligador* recebe como entrada os diversos módulos a conectar, gerando como saída um único *módulo de carga*. O programa carregador recebe o módulo de carga como entrada, transfere seu código para a memória e realiza apenas os ajustes de relocação de acordo com o endereço base de memória.

Separando as funções entre dois programas, para o exemplo acima um ligador poderia criar o seguinte módulo de carga:

```
0.2E
1.00.6.10390000002C
1.06.4.0C00000A
1.0A.2.6D02
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
1.1C.6.30390000002A
1.22.6.33C00000002C
1.28.2.4E75
1.2A.2.0000
1.2C.2.0000
2.02.4
2.14.4
2.1E.4
2.24.4
3.00
```

Este resultado foi obtido essencialmente através das seguintes modificações com relação ao algoritmo do carregador com ligação direta:

1. O registro de início de segmento indica o tamanho total do código de máquina.
2. O endereço inicial de carga, *ipla*, é considerado como sendo 0. Assim, todos os endereços passam a ser relativos ao início do módulo de carga.
3. A saída é enviada a um arquivo (o módulo de carga) ao invés de colocada na memória. A informação de relocação (quais posições de memória deverão ser atualizadas após alocação) deve ser preservada, mas o símbolo de referência não é necessário — será o início do segmento para todos.

O carregador obtém a informação de quanto espaço deve ser alocado do registro inicial (tipo 0), transfere o código (registros tipo 1) para a área de memória alocada e usa a informação do diretório de relocação (registros tipo 2) para ajustar o endereço nas posições indicadas.

Um ligador que produz módulos de carga relocáveis é usualmente denominado um *link-editor*, sendo que os mais elaborados permitem definir diversas seções e a inclusão de comandos específicos para a ligação. Um exemplo é o comando **ld** do Unix. A implementação deste comando (GNU) em Linux incorpora a seguinte informação na sua documentação:

> **ld** combina um número de arquivos objeto e de bibliotecas, reloca seus dados e amarra referências simbólicas. Usualmente o último passo na compilação de um programa é rodar **ld**.
>
> **ld** aceita arquivos *Linker Command Language* escritos em um superconjunto da sintaxe da *AT&T's Link Editor Command Language*, para fornecer controle total e explícito sobre o processo de ligação.

A linguagem de comandos suportada por **ld** controla os seguintes aspectos:

- arquivos de entrada,
- formatos de arquivos,
- *layout* do arquivo de saída,

- endereços de seções,
- posicionamento de blocos comuns.

4.6.1 Bibliotecas

Quando um programador usa em seus programas funções oferecidas pelo sistema, estas funções estão usualmente já montadas, disponibilizadas em formato objeto. Entretanto, ao invés de ter um arquivo objeto para cada função (o que tornaria o número de objetos excessivo) estas funções estão usualmente organizadas na forma de arquivos do tipo *biblioteca*.

Bibliotecas são arquivos que contêm um conjunto de módulos objetos, normalmente agrupados de acordo com sua funcionalidade. A origem do termo "biblioteca" vêm da época dos computadores de grande porte, para os quais as rotinas auxiliares eram mantidas em fitas ou cartões armazenados em salas com prateleiras.

Nesta seção, **bibliotecas estáticas** serão descritas — bibliotecas dinâmicas serão vistas na Seção 4.7. Uma biblioteca estática fornece código objeto que deve ser integrado ao módulo executável antes do momento de execução, durante o processo de ligação.

Em geral, o sistema operacional apresenta utilitários para manipular arquivos tipo biblioteca. Em Unix (Linux), o utilitário `ar` é utilizado para criar, manter e extrair módulos de arquivos de bibliotecas estáticas. Por exemplo, se um módulo objeto `arqmat.o` tiver sido criado com a linha de comando

```
> gcc -c arqmat.c
```

esse módulo pode ser incluído em uma biblioteca `libmy.a`, criada pelo usuário, com a linha de comando

```
> ar -r libmy.a arqmat.o
```

A chave `-r` indica que ocorrerá uma troca *(replacement)* do módulo, caso houvesse uma versão anterior já armazenada na biblioteca; caso contrário, o módulo é acrescentado à biblioteca.

Se essa for a primeira operação com essa biblioteca, ela será criada pelo programa `ar`. Nesse caso, uma solicitação de listar o conteúdo (com a opção `-t`) mostrará que apenas esse módulo está presente:

```
> ar -t libmy.a
arqmat.o
```

Novos módulos podem ser similarmente incluídos:

```
> ar -r libmy.a convexp.o
> ar -t libmy.a
arqmat.o
convexp.o
```

A estrutura típica de um arquivo do tipo biblioteca nesse tipo de sistema operacional é composta por uma identificação do tipo de arquivo (em geral, a *string* `!<arch>\n`), um diretório de membros do arquivo e por uma seção com o conteúdo de cada membro do arquivo.

O diretório de membros do arquivo é composto por uma série de cabeçalhos de membro, sendo que cada cabeçalho de membro contém:

- o nome do membro;
- a data de modificação;
- a identificação do usuário e grupo criador do módulo;
- as permissões de acesso para o módulo;

- o tamanho do módulo em bytes; e
- um terminador de cabeçalho, usualmente a seqüência com os dois caracteres `\` e `n` para indicar um "fim de linha".

A estrutura de um arquivo do tipo biblioteca pode ser usado para agregar qualquer tipo de conteúdo, mas usualmente apenas módulos objetos são agrupados em bibliotecas.

No processo de ligação, além dos módulos objetos gerados a partir dos arquivos fontes originais, o programador pode especificar arquivos do tipo biblioteca. Inicialmente, o ligador irá resolver as referências que puderem ser estabelecidas a partir dos módulos objetos fornecidos. Se, ao final dessa etapa, ainda houver referências não-resolvidas, o ligador procura pela definição dos símbolos dentro das bibliotecas. Ao encontrar o cabeçalho do módulo especificado, o ligador obtém dali toda a informação necessária para extrair apenas o módulo desejado e assim integrá-lo ao módulo de carga executável.

Arquivos de biblioteca são extensamente utilizados, embora nem sempre de forma explícita. Por exemplo, na implementação GNU para o compilador C a biblioteca `libc.a`, armazenada no diretório `/usr/lib`, contém as funções da biblioteca padrão da linguagem. Como essas funções são amplamente utilizadas (tal como a rotina `printf`), o programador não precisa explicitar para o ligador que essa biblioteca deverá ser utilizada para a resolução de símbolos — o próprio compilador `gcc` irá integrar essa biblioteca ao processo de ligação.

Quando uma outra biblioteca tiver de ser utilizada, contendo por exemplo rotinas matemáticas (em Unix, na biblioteca `libm.a`) ou rotinas associadas a outros pacotes ou aplicativos (bancos de dados, interfaces gráficas), é preciso passar essa informação ao ligador. No caso do ligador `ld`, há duas chaves relacionadas ao fornecimento dessa informação — essas chaves podem ser especificadas para o compilador, que as repassa ao programa ligador. A chave `-l`*xxx* indica que a biblioteca cujo nome é `lib`*xxx*`.a` deve ser incorporada ao processo de resolução de referências.

Por exemplo, considere o seguinte programa que incorpora uma rotina matemática — no caso, `cos` para o cálculo do cosseno de um valor real especificado na linha de comando:

```
1   // calccos.c: calcula o cosseno do valor especificado
2   #include <math.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   int main(int argc, char *argv[]) {
7     double valor, result;
8
9     if (argc != 2) {
10      fprintf(stderr, "%s requer um argumento numérico\n", argv[0]);
11      return 1;
12    }
13    valor = atof(argv[1]);
14    result = cos(valor);
15    printf("Cosseno de %lf é %lf\n", valor, result);
16    return 0;
17  }
```

Se o correspondente arquivo de biblioteca com rotinas matemáticas, que contém o código objeto para a rotina `cos`, não for especificado, um erro de ligação será gerado:

```
> gcc calccos.c
/tmp/ccKOiW4b.o: In function ‘main’:
```

```
/tmp/ccKOiW4b.o(.text+0x54): undefined reference to ‘cos’
collect2: ld returned 1 exit status
```

Para incluir a ligação das rotinas matemáticas, o código deve ser compilado e ligado com a inclusão da chave `-lm`, como em

```
> gcc calccos.c -lm
```

A outra chave associada à especificação de bibliotecas é `-L`, que especifica um diretório onde arquivos do tipo biblioteca estarão armazenados, caso seja necessário fazer essa busca em um diretório distinto dos usados por padrão pelo sistema operacional — tipicamente, os diretórios `/lib`, `/usr/lib` e `/usr/local/lib`.

4.7 Carregamento e Ligação Dinâmicos

Os esquemas de ligação e carregamento apresentados até o momento assumem que o módulo executável, uma vez carregado a uma área da memória principal, será o “proprietário” desta área até o fim de sua execução. Em sistemas multiusuários mais recentes, não é isto o que ocorre. Programas em execução podem ser retirados da memória *(swaped-out)* e depois retornar à memória *(swap-in)* em outra posição diferente daquela na qual estava executando inicialmente.

Para atender a este tipo de necessidade, utilizam-se esquemas de ligação e carregamento dinâmico. O princípio básico destes esquemas é que referências a endereços (de dados ou de instruções) são mantidos na forma relativa até o momento em que eles são realmente necessários, ou seja, até o momento de execução da instrução que contém esta referência. Usualmente, esta funcionalidade deve ter parte suportada em hardware de modo a não haver degradações sensíveis de desempenho.

Há dois esquemas básicos de ligação dinâmica, em tempo de carregamento *(load-time)* ou em tempo de execução *(run-time)*. Na ligação dinâmica em tempo de carregamento, o módulo de carga primário (módulo da aplicação) é inicialmente transferido para a memória. Qualquer referência neste módulo para módulos externos (módulos alvos) faz com que o carregador procure cada módulo alvo, carregue-o para a memória e altere as referências para um endereço relativo em memória a partir do início do módulo da aplicação.

Entre as vantagens neste esquema de carregamento pode-se destacar:

- facilidade de atualização de versões de módulo alvo sem alterar a aplicação;
- facilidade no suporte ao compartilhamento de módulos alvo entre aplicações distintas — se o sistema operacional detectar que um módulo alvo já está em memória, uma única cópia pode ser mantida em memória (contanto que o conteúdo do módulo alvo não seja alterado pela aplicação).

A diferença para a ligação dinâmica em tempo de execução está no fato de que o carregamento e a resolução de referências são retardados até o momento em que a instrução com a referência ao módulo externo é executada. As referências a módulos externos continuam presentes na aplicação, mas se em alguma execução a lógica de fluxo de controle fizer com que aquela referência não seja executada, então o módulo alvo não será carregado à memória por aquela aplicação. As vantagens descritas para o esquema de ligação dinâmica em tempo de carregamento continuam válidas também neste caso.

Em versões mais recentes do sistema operacional Linux os módulos objeto e de carga estão principalmente em formato ELF *(Executable and Linking Format)*. Bibliotecas para este tipo de arquivos são denominadas bibliotecas dinâmicas ainda ou arquivos de objetos compartilhados, que são diferenciadas das bibliotecas estáticas por sua extensão — estáticas têm extensão `.a` (archive) e dinâmicas têm extensão `.so` *(shared objects)*.

Sob o ponto de vista do programador usuário, não há diferença no procedimento para uso de rotinas em bibliotecas estáticas ou dinâmicas — da mesma forma, rotinas em bibliotecas padrões são automaticamente buscadas e outras bibliotecas deverão ser especificadas através da chave `-l`. A diferença está na forma de

operação interna do ligador e carregador, que utiliza a interface de programação (API) associado ao formato ELF.

ELF apresenta uma API que pode ser utilizada em programas do sistema desenvolvidos em C para manipular objetos em biblioteca compartilhadas durante a execução de um programa. Essas rotinas, disponibilizadas através da biblioteca `libdl.so`, são:

```
#include <dlfcn.h>
void       * dlopen  (const char *filename, int flag);
const void * dlsym   (void *handle, const char *symbol);
int          dlclose (void *handle);
const char * dlerror (void);
```

A rotina `dlopen` disponibiliza uma biblioteca dinâmica para o programa em execução — em outros termos, as rotinas no arquivo especificados são mapeadas para o espaço de endereçamento do processo em execução. O seu valor de retorno é um ponteiro (*handle*), utilizado nas chamadas posteriores de manipulação da biblioteca. O argumento `flag` indica quando deverá se dar o carregamento. Se tiver o valor `RTLD_NOW` (uma constante definida no arquivo `dlfcn.h`), o carregamento deverá ser imediato, ou seja, ao retornar dessa rotina a biblioteca já estará carregada na memória. Caso o valor especificado seja `RTLD_LAZY`, o carregamento será postergado até o momento em que houver (e se houver) a um símbolo dessa biblioteca. Em caso de erro, o apontador nulo será retornado.

A rotina `dlsym` retorna o endereço do símbolo especificado (variável ou função) que está disponível na biblioteca compartilhada que foi aberta. Em caso de erro, o apontador nulo será retornado.

Quando a biblioteca compartilhada não é mais necessária, ela é liberada através da invocação da rotina `dlclose`, que retorna 0 em caso de sucesso.

Em qualquer situação de erro, a rotina `dlerror` pode ser invocada para obter uma *string* com o diagnóstico do erro.

O exemplo a seguir ilustra como a função `cos` pode ser dinamicamente carregada da biblioteca `libm.so` usando ELF em linux:

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3  int main(int argc, char *argv[]) {
4    void *handle;
5    double (*cosine)(double);
6    char *error;
7    handle = dlopen("/lib/libm.so.5", RTLD_LAZY);
8    if (!handle) {
9      fputs (dlerror(),stderr);
10     return 1;
11   }
12   cosine = dlsym(handle, "cos");
13   if ((error = dlerror()) != 0) {
14     fputs(error, stderr);
15     return 1;
16   }
17   printf("%f\n", (*cosine)(2.0));
18   dlclose(handle);
19 }
20
```

Para criar uma biblioteca compartilhada, inicialmente é preciso gerar um módulo objeto que possa ser carregado dinamicamente. Para tanto, o código gerado deve ser independente de posição. O compilador `gcc`

permite a criação desse tipo de código de forma automática, através do uso da chave `-fPIC` (de *position-independent code*):

```
> gcc -fPIC -c arqmat.c
> gcc -fPIC -c convexp.c
```

Para criar a biblioteca dinâmica contendo os objetos compartilhados, o próprio compilador é utilizado:

```
> gcc -shared -o libmyd.so arqmat.o convexp.o
```

A chave `-shared` indica para o programa `gcc` que o programa que está sendo gerado (indicado pela opção `-o`) é uma biblioteca compartilhada cujos módulos podem ser carregados e ligados dinamicamente. Caso algum desses módulos faça referências a arquivos em outras bibliotecas dinâmicas, é possível tornar transparente para o usuário a necessidade de se carregar essa outra biblioteca especificando-a no momento da criação. Por exemplo, se `arqmat.o` faz uso de rotinas em `libm.so`, a linha de comando

```
> gcc -shared -o libmyd.so arqmat.o convexp.o -lm
```

fará com que `libm.so` seja automaticamente carregada quando `libmyd.so` for especificada.

O padrão em sistemas operacionais modernos é utilizar arquivos compartilhados com carregamento e ligação dinâmica. O compilador `gcc` inclui a opção `-static` caso seja necessário criar um executável ligado estaticamente, mudando assim o comportamento padrão.

4.8 Exercícios

4.1 Das instruções em Assembly 68K abaixo, indique quais são válidas e quais não são, justificando suas resposta. Para aquelas que forem válidas, apresente o código de máquina em hexadecimal. A descrição da instrução `MOVE` encontra-se no Apêndice B.

(a) `MOVE.L D1,#10`

(b) `MOVE.W 16(A4),D2`

(c) `MOVE.B #2056,D3`

4.2 Explique qual a diferença entre

(a) o resultado gerado por um montador absoluto e um montador de ligação direta

(b) ajuste de relocação e ajuste de ligação

(c) ligação dinâmica em tempo de carga e em tempo de execução

4.3 Justifique se as afirmações abaixo são verdadeiras ou falsas:

(a) A Tabela de Símbolos gerada por um montador é uma estrutura de dados usada internamente pelo montador, não sendo nunca incorporada ao módulo objeto gerado.

(b) Uma das vantagens do esquema de ligação dinâmica é a redução de tamanho de módulos objetos, uma vez que nestes a incorporação de códigos de outras rotinas é substituída por referências a estas rotinas.

4.4 Um montador de ligação direta aplicado a dois arquivos em linguagem simbólica do 68K gerou os seguintes módulos objetos:

Módulo 1	**Módulo 2**
0.’MAIN’.’S’.0000.001A	0.’CALC’.’S’.0000.0006
0.’RESULT’.’L’.0018.0002	1.0000.02.2200
1.0000.06.203900000014	1.0002.02.9081
1.0006.06.4EB900000000	1.0004.02.4E75
1.000C.06.33C000000018	3.00
1.0012.02.4E75	
1.0014.04.00004E75	
2.0002.04.’MAIN’	
2.0008.04.’CALC’	
2.000E.04.’MAIN’	
3.02.0000	

Passados como argumentos nessa ordem (módulo 1 seguido de módulo 2) para um carregador de ligação direta, obteve-se o endereço inicial de carga (IPLA) $0200.

(a) Qual o conteúdo da Tabela de Símbolos Externos Globais (GEST) gerada pelo carregador?

(b) O diagrama abaixo é um mapa de conteúdo da memória após o carregamento *sem* os ajustes de ligação e relocação. Indique neste mapa quais posições são ajustadas pelo carregador e qual o novo conteúdo destas posições.

	0	2	4	6	8	A	C	E
020-	2039	0000	0014	4EB9	0000	0000	33C0	0000
021-	0018	4E75	0000	4E75	0000	2200	9081	4E75

Apêndices

Apêndice A

Representação numérica binária

Inteiros sem sinal têm uma representação computacional (em números binários) equivalente à representação usual para números decimais, ou seja, através da atribuição de pesos associados à posição de cada bit. Grande parte dos computadores atuais utilizam 32 bits para representar números inteiros, o que permite representar $4.924.967.296$ valores distintos. (A geração mais recente de computadores suporta também inteiros com 64 bits.) Uma seqüência binária

$$s_{n-1}s_{n-2}s_{n-3}\ldots s_2s_1s_0$$

está associada ao valor inteiro

$$\sum_{i=0}^{n-1} s_i \cdot 2^i$$

onde $s_i \in \{0,1\}$. O bit s_{n-1} é chamado *bit mais significativo* (MSB), enquanto que s_0 é o *bit menos significativo* (LSB).

A representação de inteiros com sinal pode usar outros formatos. A forma mais básica é a representação em sinal e magnitude, onde o bit mais significativo denota o sinal associado ao restante da seqüência ($s_{n-1} = 1$ indicaria que o número é negativo). Este formato tem a desvantagem de ter duas representações diferentes para o valor zero, além de ter circuitos complicados para suportar operações básicas, diferenciando adição de subtração, por exemplo.

Outra formato suportado para representar inteiros com sinal é a representação em complemento de um. A representação para um número negativo neste formato pode ser obtida facilmente a partir da representação do número positivo correspondente simplesmente complementando cada bit da seqüência, ou seja, trocando 0's por 1's e 1's por 0's. Apesar de simplificar circuitos para operações básicas, este formato ainda mantém duas representações distintas para o valor zero.

O formato mais aceito para inteiros com sinal é a representação em complemento de dois. Para obter a representação de um número negativo neste formato, computa-se inicialmente a representação em complemento de um e adiciona-se 1 ao bit menos significativo. Neste caso, o valor inteiro associado à seqüência $s_{n-1}\ldots s_0$ é

$$\sum_{i=0}^{n-2} s_i \cdot 2^i - s_{n-1} \cdot 2^n.$$

Este formato mantém a simplicidade dos circuitos aritméticos e tem apenas uma representação para o valor zero. Uma característica que lhe é peculiar é o fato de que a faixa de valores representáveis não é simétrica em

torno de 0, havendo um valor negativo a mais que a quantidade de valores positivos distintos. Por exemplo, seqüências de cinco bits podem representar valores entre -16 (`10000`) e +15 (`01111`).

No formato de representação para números reais, associado ao conceito de notação científica, cada valor (pertencente ao domínio dos reais) é representado por um sinal, uma mantissa e um expoente. Entre as inúmeras combinações possíveis de formatos de representação que seguem esta filosofia básica, o padrão IEEE-754 tem sido o mais aceito e usualmente suportado em hardware (através das unidades de ponto flutuante em co-processadores ou incorporados a CPUs). Este formato suporta representações de números reais em *precisão simples* (32 bits, dos quais 8 para a representação do expoente e 23 para a representação da mantissa), em *precisão dupla* (64 bits, sendo 11 para o expoente e 53 para a mantissa) e em precisão estendida (80 bits). Há também representações especiais para os valores $-\infty$, $+\infty$ e `NaN` (*Not a Number*, associado ao resultado de operações sem significado matemático, tal como a divisão de zero por zero).

Parece evidente que a representação binária, apesar de ideal para o processador, é de difícil manipulação por humanos. Por este motivo, adota-se usualmente a representação hexadecimal para denotar seqüências binárias.

A vantagem da representação hexadecimal sobre a decimal, que usamos no dia a dia, é a fácil associação com seqüências binárias. A tradução é direta: cada seqüência de quatro bits corresponde a um símbolo hexadecimal. A tabela a seguir define este mapeamento:

binário	**hexa**	**binário**	**hexa**
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

A representação octal também permite um mapeamento similar, de três bits para um dígito entre 0 e 7. Entretanto, a representação hexadecimal também apresenta a vantagem de alinhamento com um byte (8 bits, dois dígitos hexadecimais) e palavras de 16 bits (quatro dígitos).

Apêndice B

Assembly do 68000

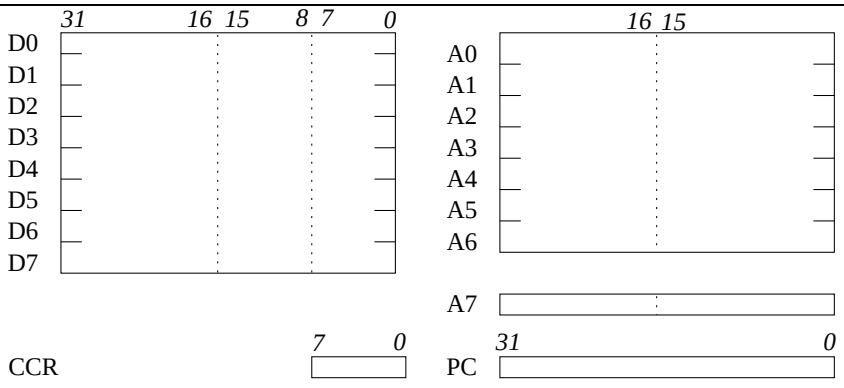
Uma instrução em linguagem *assembly* corresponde a uma representação simbólica de uma instrução de um processador. Cada processador apresenta um repertório de tais instruções, que são utilizadas para compor os programas em linguagem de máquina. Assim, para cada processador a programação *assembly* será diferente, embora a estrutura básica da programação deste nível seja equivalente para muitos procesadores.

Neste capítulo, o *assembly* de processadores da família 68000 será utilizado como referência para nossos exemplos, sem perda de generalidade sobre a forma como montadores trabalham para gerar um módulo objeto a partir de um código fonte em *assembly*. Para que a compreensão dos programas em *assembly* não seja prejudicada para quem não conhece esta família de processadores, a estrutura e organização do 68000 serão inicialmente apresentadas.

B.1 Organização dos dados

O 68000 têm oito registradores de dados de 32 bits cada, D0 a D7, sendo que cada um deles pode ser manipulado com operandos de tamanho *byte* (bits 0 a 7), *word* (bits 0 a 15) ou *long word* (todos os 32 bits). A Figura B.1 ilustra a visão que um programador usuário[1] tem dos registradores do 68000.

Figura B.1 Modelo de registradores do 68000.



Há sete registradores de endereço de uso geral, A0 a A6, cada um deles de 32 bits. Estes registradores

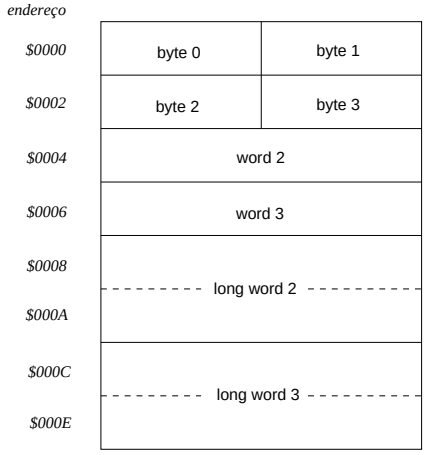
[1] O 68000 apresenta dois modos de operação, supervisor e usuário. Neste texto, nos restringiremos ao modo usuário.

podem ser manipulados com operandos de tamanho *word* e *long word*. O registrador de endereço A7 é usado como o apontador de pilha.

Há ainda registradores contador de programa (PC), de 32 bits, e de códigos de condição (CCR — *condition codes register*), de 8 bits. O registrador CCR inclui bits para sinalizar condições de *carry* (bit C, posição 0), *overflow* (bit V, posição 1), *zero* (Z, 2), *negativo* (N, 3) e *extend* (X, 4).

Na memória, operandos estão organizados como apresentado na Figura B.2. Operandos de tamanho *byte* podem ser acessados individualmente. Operandos de tamanho *word* estão sempre localizados em endereços pares, enquanto que operandos *long word* estão sempre em endereços múltiplos de quatro.

Figura B.2 Organização da Memória.



Nesta figura, observe que a *word* 0 é composta pelos bytes 0 e 1, enquanto que a *word* 1 é composta pelos bytes 2 e 3. Da mesma forma, a *long word* 0 é composta pelos bytes 0, 1, 2 e 3 ou, similarmente, pelas *words* 0 e 1; enquanto que a *long word* 1 é composta pelos bytes 4, 5, 6 e 7 ou, similarmente, pelas *words* 2 e 3.

B.2 Instruções *assembly*

Neste texto, o microprocessador 68000 da Motorola será utilizado como exemplo. Seu amplo repertório inclui instruções para:

aritmética inteira: adição (ADD, ADDI, ADDQ), subtração (SUB, SUBI, SUBQ), divisão com ou sem sinal (DIVS, DIVU), multiplicação com ou sem sinal (MULS, MULU), negação (NEG)

aritmética em BCD [2]: adição (ABCD), subtração (SBCD), negação (NBCD)

manipulação de endereços: adição (ADDA), comparação (CMPA), subtração (SUBA), carregar (LEA), mover (MOVEA), *push* (PEA)

operações lógicas E (AND, ANDI), deslocamento (ASL, ASR, LSL, LSR), comparação (CMP, CMPI), OU-exclusivo (EOR, EORI), complemento (NOT), OU (OR, ORI), *test-and-set* (TAS)

movimento e alteração de valores: zerar conteúdo (CLR), trocar conteúdo de dois registradores (EXG), mover dados (MOVE, MOVEQ), mover conteúdo de registros (MOVEM, SWAP)

[2] *Binary Coded Decimal.*

desvio: incondicional (BRA, JMP), condicional (Bcc, DBcc), chamada de subrotina (BSR, JSR), retorno de subrotina (RTS)

manipulação de bits: testa valor (BTST), testa e complementa (BCHG), testa e zera (BCLR), testa e seta (BSET)

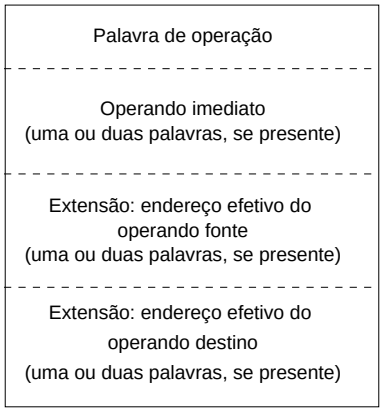
manipulação da pilha: ligar e alocar (LINK), desligar (UNLK)

Dentre estas instruções, há várias instruções similares que apresentam pequenas modificações de comportamento. Por exemplo, a instrução de soma ADD permite somar o conteúdo de dois registradores ou de um registrador e de um operando em memória. A instrução ADDI recebe um dos operandos em modo imediato (valor especificado no corpo da instrução). A instrução ADDQ também recebe o valor de um operando em modo imediato, sendo que neste caso este valor está limitado entre 1 e 8, permitindo uma representação mais compacta da instrução binária.

Várias destas instruções envolvem a especificação de operandos. O 68000 trabalha com operandos de três tamanhos: *byte*, *word* (dois bytes) e *long word* (quatro bytes). O tamanho do operando para a instrução é especificado através de sufixos **.B**, **.W** e **.L**, respectivamente, ao código mnemônico da instrução. Quando não especificado, *word* é o tamanho assumido.

O formato de uma instrução na memória está estruturado da seguinte forma (Figura B.3): a primeira palavra, sempre presente, determina a operação e os modos de endereçamento dos operandos. Caso haja algum operando imediato, este ocupará uma ou duas palavras seguintes. O endereço de um operando fonte, se presente, pode ocupar uma ou duas palavras seguintes. Finalmente, o endereço de destino da operação, se presente, ocupará uma ou duas palavras seguintes. A necessidade ou não destas palavras adicionais é determinada pelos modos de endereçamento, descritos a seguir.

Figura B.3 Formato de uma instrução.



B.3 Modos de endereçamento

O 68000 apresenta quatorze modos de endereçamento distintos, agrupados em seis modos básicos. Estes modos básicos são absoluto, imediato, direto a registrador, indireto a registrador, relativo a PC e implícito.

No modo de endereçamento *absoluto* o operando está no endereço especificado na própria instrução. Por exemplo, a instrução

```
MOVE.W  1000,D1
```

copia o conteúdo da palavra (dois bytes) na posição de memória 1000_{10} para o registrador de dados `D1`. Na codificação desta instrução há duas possíveis variantes. Se o endereço absoluto pode ser representado em dois bytes, então o modo *absolute short* pode ser utilizado. Caso contrário, quatro bytes deverão ser utilizados (modo *absolute long*). Observe que está informação não está contida na instrução — o `.W` na instrução refere-se ao tamanho do operando, e não de seu endereço. Neste texto, quando não explicitado de outra forma serão utilizados quatro bytes para endereços absolutos.

No modo de endereçamento *imediato* o valor do operando está incorporado à instrução, sendo precedido pelo símbolo #. Por exemplo, a instrução

```
ADDQ.L  #2,A0
```

soma o valor 2 ao conteúdo do registrador de endereço `A0`.

No modo de endereçamento *direto a registrador* o operando é o conteúdo de um registrador especificado na instrução. Por exemplo, para mover o conteúdo do registrador de dados `D0` para o registrador de dados `D1`

```
MOVE.W D0,D1
```

Há duas variantes deste modo de endereçamento, direto a registrador de dados `Dn` ou direto a registrador de endereços `An`.

No modo de endereçamento *indireto a registrador* o operando está no endereço contido em um registrador de endereços especificado na instrução. Por exemplo, a instrução

```
MOVE.W (A0),D1
```

move o conteúdo da palavra na posição de memória cujo endereço está no registrador `A0` para o registrador de dados `D1`.

Este modo de endereçamento indireto oferece variantes com pós-incremento, pré-decremento, com deslocamento e indexado. O modo de endereçamento indireto a registrador com *pós-incremento* permite incrementar o valor do registrador de endereços após o acesso ao valor corrente do registrador. Por exemplo, a instrução

```
MOVE.W (A0)+,D1
```

move o conteúdo da palavra cujo endereço é o conteúdo do registrador de endereços `A0` para o registrador de dados `D1` e incrementa o valor de `A0` de 2 (o tamanho de uma *word*). Assim, esta única instrução equivale à seqüência de instruções

```
MOVE.W (A0),D1
ADDQ.L #2,A0
```

O modo de endereçamento indireto a registrador com *pré-decremento* é similar a este modo, sendo que neste caso o conteúdo do registrador de endereços é decrementado *antes* do acesso ao seu conteúdo. Assim, a instrução

```
MOVE.W -(A0),D1
```

equivaleria a

```
SUBQ.L #2,A0
MOVE.W (A0),D1
```

No modo de endereçamento indireto a registrador com *deslocamento*, é possível especificar na instrução um valor constante que deve ser adicionado ao conteúdo corrente do registrador de endereços. O deslocamento é uma constante representável em 16 bits com sinal. Assim, a instrução

```
MOVE.W 32(A0),D1
```

irá adicionar 32 ao conteúdo de `A0` para obter o endereço da palavra que deverá ser carregada no registrador `D1`, sendo que o conteúdo de `A0` permanecerá inalterado. A título de ilustração, a implementação desta instrução caso este modo de endereçamento não fosse suportado iria requerer a utilização de outro registrador para armazenagem temporária. Supondo que o registrador `A6` fosse utilizado para este fim, então a instrução acima equivaleria a

```
MOVEA.L A0,A6
ADDI.L  #32,A6
MOVE.W  (A6),D1
```

O modo de endereçamento indireto a registrador pode ser também *indexado*. Este é similar ao modo com deslocamento, sendo que além de se adicionar uma constante ao registrador de endereços é possível especificar um registrador de dados cujo conteúdo também será acrescentado ao conteúdo do registrador de endereços para a obtenção do endereço efetivo do operando. Por exemplo, a instrução

```
MOVE.W  16(A0,D0.W),D1
```

equivaleria a

```
MOVEA.L A0,A6
ADDA.L  D0,A6
ADDI.L  #16,A6
MOVE.W  (A6),D1
```

novamente utilizando `A6` como um registrador temporário.

No modo de endereçamento *relativo ao PC* o operando é especificado tendo por base o conteúdo corrente do registrador contador de programa. Há duas variantes possíveis para este modo de endereçamento, com deslocamento ou indexado, ambos similares aos modos equivalentes descritos no modo de endereçamento indireto a registrador de endereços. Por exemplo,

```
MOVE.W  16(PC),D1
```

adiciona 16 ao conteúdo corrente do registrador `PC` para obter o endereço efetivo da palavra fonte cujo conteúdo será copiado para `D1`. É importante observar que o valor corrente do PC não corresponde ao endereço da palavra da instrução que contém o código de operação, mas sim à palavra de extensão (que contém o deslocamento). Esta forma de endereçamento é muito importante na definição de códigos independentes de posição.

No modo de endereçamento *implícito* o operando não está especificado na instrução. Por exemplo, todas instruções de desvio da forma *branch* referem-se implicitamente ao valor corrente do registrador PC.

B.4 Codificação binária

A cada instrução está associado um código binário que será diretamente interpretado pelo processador durante a execução do programa (o código de máquina). Para simplificar a tradução de todas as possíveis combinações de operações e operandos para o código de máquina, cada instrução é dividida em *campos*. Para o *assembly* do 68000, estes campos são:

opcode: o nome da operação (ADD, MOVE)

size: byte, word, long word

address: modo e endereço efetivo do(s) operando(s)

Nem todas as instruções apresentam todos estes campos, uma vez que em alguns casos eles não fariam sentido. Na seção acima, foram apresentados diversos exemplos de instruções que seguem este formato geral.

Para obter o código de máquina correspondente a uma instrução *assembly*, é preciso consultar tabelas que identificam os códigos binários correspondentes a cada código de operação, tamanho de operando e endereços efetivos de operandos.

O código binário para especificar o endereço efetivo de operandos segue um padrão, embora nem todos os casos sejam aplicáveis a todas as instruções. Em geral, cada endereço efetivo é representado por seis bits, sendo 3 bits para o *modo* e 3 bits para o *registrador*. A tabela B.1 a seguir identifica os códigos associados aos diversos modos de endereçamento suportados pelo 68000 e a sintaxe destes operandos quando presente em uma instrução *assembly*.

Tabela B.1 Codificação para modos de endereçamento do 68000.

Modo de endereçamento	modo	registrador	sintaxe
reg dados direto	000	no. reg.	Dn
reg endereços direto	001	no. reg.	An
reg endereços indireto	010	no. reg.	(An)
reg ender indir pos-inc	011	no. reg.	(An)+
reg ender indir pre-dec	100	no. reg.	-(An)
reg ender indir desloc	101	no. reg.	d(An)
reg ender indir index	110	no. reg.	d(An,Ri)
absoluto, short	111	000	Abs.W
absoluto, long	111	001	Abs.L
PC deslocamento	111	010	d(PC)
PC indexado	111	011	d(PC,Ri)
imediato	111	100	Imm

Na seqüência, serão apresentados detalhes sobre a codificação de algumas das principais instruções de nível usuário do 68000, agrupadas de acordo com o tipo da instrução.

B.4.1 Instrução sem efeito

As instruções de codificação mais simples são aquelas que não requerem a especificação de nenhum operando. Neste caso, a instrução de máquina a ser introduzida no código objeto é fixa e conhecida de antemão.

Um exemplo deste tipo de instrução é `NOP` *(No Operation)*, que não apresenta nenhum impacto na execução a não a atualização do registrador PC. A instrução de máquina correspondente é `$4E71` (o símbolo `$` denota um valor hexadecimal), ou em binário

```
0100111001110001
```

B.4.2 Instruções lógicas e aritméticas

Instruções de um operando

As instruções que serão apresentadas aqui são `CLR`, `NEG` e `NOT`. Cada uma destas instruções requer um único argumento, que indicará o endereço efetivo do operando, indicado por `<ea>`:

Formato da Instrução	Efeito
`CLR <ea>`	`<ea>` $\leftarrow 0$
`NEG <ea>`	`<ea>` $\leftarrow 0 -$ `(<ea>)`
`NOT <ea>`	`<ea>` $\leftarrow$ `~(<ea>)`

O formato de máquina destas instruções é:

```
cccccccc.ss.mmm.rrr
```

onde

cccccccc é o campo do código da operação,

Operação	Código
CLR	01000010
NEG	01000100
NOT	01000110

ss é o campo de tamanho do operando,

Tamanho	Código
byte	00
word	01
long	10

mmmrrr são os campos modo (`mmm`) e registrador (`rrr`) que especificam o valor do endereço efetivo do operando, conforme a Tabela B.1. Para estas operações, os modos válidos são *Dn*, *(An)*, *(An)+*, *-(An)*, *d(An)*, *d(An,Ri)*, *Abs.W* e *Abs.L* — os chamados modos de endereçamento de dados alteráveis.

Por exemplo, a instrução de máquina para `CLR.L D5` terá a seqüência `01000010` para o código da operação, a seqüência `10` para indicar o comprimento *long* (.L), a seqüência `000` para o modo de acesso direto a um registrador de dados, e finalmente a seqüência `101` para indicar o número (5) do registrador de dados a ser alterado. Assim, seu código de máquina deverá ser

```
0100001010000101
```

ou `$4285`.

Instruções envolvendo um registrador de dados e outro operando

As instruções aqui apresentadas têm por característica o fato de que um dos operandos envolvidos deve ser sempre um registrador de dados acessado no modo direto, enquanto que o outro operando apresenta flexibilidade de endereçamento.

O primeiro grupo de instruções deste tipo que serão aqui aprsentadas inclui `ADD`, `AND`, `OR` e `SUB`. Estas instruções podem ser utilizadas como em

```
ADD <ea>, Dn
```

onde o registrador de dados é também o destino do resultado, ou como em

```
ADD Dn, <ea>
```

onde o outro operando será o destino do resultado da operação.

A forma genérica destas instruções de máquina é

```
cccc.ddd.ooo.mmm.rrr
```

onde

cccc é o campo de código da instrução,

Instrução	Código
ADD	1101
AND	1100
OR	1000
SUB	1001

ddd é o número do registrador de dados envolvido na operação;

ooo é o modo de operação, que pode assumir os valores

byte	word	long	operação
000	001	010	`Dn` ← `(Dn) op (<ea>)`
100	101	110	`<ea>` ← `(Dn) op (<ea>)`

mmmrrr indicam os campos de modo e de registrador do endereço efetivo do operando, que podem ser do modo *(An), (An)+, -(An), d(An), d(An,Ri), Abs.W* e *Abs.L* se o endereço efetivo refere-se ao operando de destino. Se o endereço efetivo for do operando fonte, os modos *Dn, An, d(PC), d(PC,Ri)* e *Imm* também são válidos, sendo que o modo *An* pode apenas ser utilizado com operandos de tamanho *word* ou *long*.

Considere por exemplo a instrução ADD.L D0,D1. A codificação começa com a seqüência do código de operação `1101`, seguida de `001` (registrador D1), `010` (long com destino sendo o registrador especificado, D1), `000` (outro operando está em registrador) e `000` (o outro registrador é o D0). Portanto, o código binário para esta instrução é

```
1101001010000000
```

ou `$D280`.

A instrução ADD.L #1,D1 terá a primeira palavra da instrução codificada pela seqüência de bits `1101` (instrução ADD), `001` (D1), 010 (resultado em D1), `111100` (operando imediato). A primeira palavra da instrução terá portanto o valor `$D2BC`. Seguirão então mais duas palavras de extensão (uma vez que a operação é sobre operandos do tipo *long*) contendo o valor imediato a ser adicionado, sendo que a primeira palavra será `$0000` e a segunda palavra será `$0001`. A instrução ocupa portanto três palavras de memória, com conteúdo `$D2BC00000001`.

Outro grupo de instruções com esta mesma estrutura básica incluem `DIVS` (divisão com sinal), `DIVU` (divisão sem sinal), `MULS` (multiplicação com sinal) e `MULU` (multiplicação sem sinal). Estas instruções tem apenas uma forma de uso, onde o destino deve ser o registrador de dados, e apenas um tamanho de operando (*word*), sendo o resultado armazenado em 32 bits. No caso da divisão, este resultado de 32 bits é dividido em dois resultados de 16 bits cada, sendo a parte menos significativa o quociente da operação e a parte mais significativa o resto da divisão. As formas de uso destas instruções são:

```
DIVS <ea>,Dn
DIVU <ea>,Dn
MULS <ea>,Dn
MULU <ea>,Dn
```

O formato da instrução de máquina é similar ao das instruções do grupo acima, sendo:

Instrução	Código
DIVS	1000.ddd.111.mmm.rrr
DIVU	1000.ddd.011.mmm.rrr
MULS	1100.ddd.111.mmm.rrr
MULU	1100.ddd.011.mmm.rrr

onde o único modo de endereçamento inválido é o registrador de endereços direto.

Há ainda mais duas instruções adicionais com formato similar, `CMP` (comparação) e `EOR` (ou exclusivo). A forma de uso destas instruções é

```
CMP <ea>,Dn
```

e

```
EOR Dn,<ea>
```

O código de máquina para estas duas instruções tem o mesmo campo de operação,

```
1011.ddd.ooo.mmm.rrr
```

sendo que as duas instruções são diferenciadas pelo campo `ooo`:

Instrução	byte	word	long	Operação
CMP	000	001	010	`(Dn) - (<ea>)`
EOR	100	101	110	`<ea>` $\leftarrow$ `(<ea>)` $\oplus$ `Dn`

Para a instrução `CMP` todos os modos de endereçamento são válidos, embora o modo *An* apenas para operandos do tipo *word* ou *long*. A instrução `EOR` é restrita aos modos de endereçamento de dados alteráveis.

Instruções com operando em registrador de endereços

Quando um dos operandos na instrução de soma, subtração ou comparação corresponde a um endereço, instruções diferenciadas são oferecidas. São elas:

Instrução	Codificação
`ADDA <ea>,An`	1101.eee.ooo.mmm.rrr
`CMPA <ea>,An`	1011.eee.ooo.mmm.rrr
`SUBA <ea>,An`	1001.eee.ooo.mmm.rrr

onde

eee indica um dos oito registradores de endereço, o destino do resultado das operações no caso de soma ou subtração;

ooo é o modo de operação, sendo

- **011** operação em *word*: o operando fonte é estendido com sinal de 16 para 32 bits para então ser operado com o endereço especificado; ou
- **111** operando *long*;

mmm.rrr são os campos modo e registrador do endereço efetivo, sendo válidos todos os modos de endereçamento.

Instruções com operando imediato

Quando a instrução aritmética ou lógica envolve um operando imediato e um outro operando que não um registrador de dados é preciso usar as formas imediatas destas instruções,

```
OPER.S  #<data>,<ea>
```

onde `S` é B, W ou L, enquanto que `OPER` pode ser:

Instrução	Codificação (palavra de operação)
ADDI	00000110.ss.mmm.rrr
ANDI	00000010.ss.mmm.rrr
CMPI	00001100.ss.mmm.rrr
EORI	00001010.ss.mmm.rrr
ORI	00000000.ss.mmm.rrr
SUBI	00000100.ss.mmm.rrr

sendo

ss o campo de tamanho do operando:

byte	word	long
00	01	10

mmm.rrr os campos de modo e registrador do endereço efetivo do operando, que deve ser um dos modos de endereçamento de dados alteráveis.

O valor do operando imediato ocupa uma (no caso de byte ou word) ou duas (no caso de long) palavras após a palavra de operação. Se o endereço efetivo também precisar de palavras adicionais — como um endereço em modo de endereçamento absoluto — estas virão codificadas após o valor imediato.

Para as instruções de soma e subtração envolvendo pequenos valores imediatos — entre 1 e 8 — há instruções adicionais que permitem uma codificação mais compacta:

Instrução	Codificação
ADDQ #<data>,<ea>	0101.vvv.0.ss.mmm.rrr
SUBQ #<data>,<ea>	0101.vvv.1.ss.mmm.rrr

onde

vvv é o campo do valor imediato, sendo 000 interpretado como 8 e 001 a 111 como 1 a 7, respectivamente,

e os demais campos são como descritos acima para as outras formas de instruções com operandos imediatos.

B.4.3 Instruções de transferência

A forma primordial de transferência de dados ocorre através da instrução MOVE, que recebe dois operandos, respectivamente o endereço efetivo do operando fonte e o endereço efetivo do destino. Três formas básicas desta instrução são

Instrução	Codificação
MOVE <ea>,<ea>	00.ss.ttt.nnn.mmm.rrr
MOVEA <ea>,An	00.ss.ttt.001.mmm.rrr
MOVEQ #<data>,Dn	0111.rrr.0.vvvvvvvv

Para a forma MOVE, os campos são:

ss campo de tamanho do operando, sendo

byte	word	long
01	11	10

ttt.nnn o registrador *(ttt)* e o modo *(nnn)* de endereçamento do operando destino, sendo permitidos apenas os modos de endereçamento de dados alteráveis;

mmm.rrr o modo e registrador do operando fonte, sendo permitidos todos os modos de endereçamento.

No exemplo da instrução MOVE.W D5,-(A7), a seqüência de bits da instrução codificada será `00` (código da operação), `11` (comprimento word). O destino é especificado como o registrador de endereço A7 (`111`) no modo indireto pré-decrementado (`100`) — portanto, a seqüência de bits para o endereço efetivo do destino será `111100`. O operando fonte é o registrador de dados D5 (`101`) acessado no modo direto (`000`). Portanto, a seqüência de bits para o operando fonte será `000101`. Portanto, a instrução é codificada pela palavra `$3F05`.

Para a forma `MOVEA`, o campo de tamanho *(ss)* pode assumir apenas os valores 11 *(word)* ou 10 *(long)*. Os demais campos equivalem à descrição de `MOVE`.

A forma `MOVEQ` tem apenas um tamanho de operando, *long*. O campo *rrr* identifica o registrador de dados (destino) e *vvvvvvvv* é o campo de valor imediato de oito bits que são estendidos para um operando *long*.

B.4.4 Instruções de desvios

Há duas instruções que permitem a especificação de desvios incondicionais, `JMP` e `BRA`. A instrução `JMP` tem como operando um endereço, enquanto que `BRA` recebe como operando um deslocamento relativo à posição corrente do programa.

A instrução de desvio incondicional `JMP` tem o formato

```
0100111011.mmm.rrr
```

onde `mmm.rrr` são respectivamente os campos de modo e registrador do operando. Apenas os modos *(An), d(An), d(An,Ri), Abs.W, Abs.L, d(PC)* e *d(PC,Ri)* — denominados modos de endereçamento de controle — são válidos.

A codificação da instrução `BRA` pode ocupar uma ou duas palavras:

```
01100000.ssssssss
llllllll llllllll
```

Caso o deslocamento especificado possa ser representado em 8 bits (em complemento de 2), apenas o campo *ssssssss* estará presente, e a segunda palavra é omitida. Caso o deslocamento requeira 16 bits para sua representação, então o campo *ssssssss* deverá ser zerado, e o deslocamento será representado na segunda palavra da instrução.

A instrução de desvio condicional, B*cc*, especifica que o desvio só será tomado quando a condição *cc* — avaliada a partir dos *flags* de condição no registrador de estado CCR — for verdadeira. A codificação desta instrução tem o formato

```
0110.ffff.ssssssss
llll llll llllllll
```

onde *tttt* é a codificação binária para cada uma das condições *(cc)*, enquanto que os campos restantes são equivalentes à instrução `BRA`. Os códigos de condição e codificação correspondentes são apresentados na Tabela B.2.

O mapeamento entre estas condições e testes lógicos "convencionais" (apresentados aqui na sintaxe de C) é apresentado nas Tabelas B.3 para números com sinal e B.4 para números sem sinal.

Observe que o deslocamento em `BRA` e B*cc* é relativo ao conteúdo corrente do registrador PC (a posição da instrução mais 2), de modo que não é possível estabelecer um desvio para a próxima instrução usando o campo de 8 bits — o valor do deslocamento seria 0, o que forçaria a leitura da palavra seguinte para a obtenção do deslocamento de 16 bits.

Uma outra instrução, DB*cc*, permite agregar um decremento em um registrador de dados e a especificação de um desvio condicional. Esta instrução tem por objetivo facilitar a implementação de desvios associados a laços de iteração (equivalente à forma *until* presente em algumas linguagens de alto nível). Sua sintaxe de uso é

Tabela B.2 Códigos de condição.

cc	**condição**	**expressão**	`tttt`
CC	carry clear	$\overline{C}$	0100
CS	carry set	C	0101
EQ	equal	Z	0111
GE	greater or equal	$N \cdot V + \overline{N} \cdot \overline{V}$	1100
GT	greater than	$N \cdot V \cdot \overline{Z} + \overline{N} \cdot \overline{V} \cdot \overline{Z}$	1110
HI	high	$\overline{C} \cdot \overline{Z}$	0010
LE	less or equal	$Z + N \cdot \overline{V} + \overline{N} \cdot V$	1111
LS	low or same	$C + Z$	0011
LT	less than	$N \cdot \overline{V} + \overline{N} \cdot V$	1101
MI	minus	N	1011
NE	not equal	$\overline{Z}$	0110
PL	plus	$\overline{N}$	1010
VC	overflow clear	$\overline{V}$	1000
VS	overflow set	V	1001

Tabela B.3 Relacionamento entre testes e condições para números com sinal.

Teste C	**Instrução 68K**
<	BLT
<=	BLE
==	BEQ
>=	BGE
>	BGT
!=	BNE

Tabela B.4 Relacionamento entre testes e condições para números sem sinal.

Teste C	**Instrução 68K**
<	BCS
<=	BLS
==	BEQ
>=	BCC
>	BHI
!=	BNE

```
DBcc Dn,<label>
```

onde `<label>` identifica um rótulo da posição de destino, que deve ser traduzido para um deslocamento de 16 bits relativo ao PC na codificação da instrução. Esta instrução primeiro testa a condição para verificar se a condição de término do laço foi alcançada. Caso sim, então passa-se à próxima instrução. Caso não tenha sido alcançada a condição de término, então a palavra no registrador especificado é decrementada. Quando o resultado for -1, então o contador foi esgotado e passa-se à próxima instrução. Se o resultado não for -1, então o desvio é tomado.

A codificação desta instrução é

```
0101.tttt.11001.rrr
llll llll lllll lll
```

onde *tttt* é o código da condição, *rrr* identifica o registrador de dados que estará atuando como contador e *l . . . l* é o deslocamento de 16 bits relativo ao registrador PC. Além das formas de condição especificadas para B*cc*, as condições `F` (*false*, código 0001) e `T` (*true*, código 0000) podem ser especificadas para esta instrução.

B.4.5 Instruções para subrotinas

A chamada de subrotinas em *assembly* é essencialmente uma instrução de desvio, onde o endereço da instrução seguinte é salvo na pilha para permitir o retorno após a execução da subrotina. Assim, duas formas de desvio para subrotina são suportados:

```
JSR <ea>
BSR <label>
```

A instrução `JSR` é codificada por

```
0100111010.mmm.rrr
```

onde *mmm.rrr* são os campos de modo e registrador do endereço efetivo da subrotina, sendo que apenas os modos de endereçamento de controle são válidos.

A instrução `BSR` é codificada por

```
01100001.vvvvvvvv
llllllll llllllll
```

onde *v . . . v* é o campo de deslocamento de 8 bits e *l . . . l* é o campo de deslocamento de 16 bits, só presente quando *v . . . v* for 0.

O retorno de uma subrotina sempre ocorre pela instrução `RTS`, que não recebe nenhum operando e tem codificação

```
0100111001110101
```

ou `$4E75`.

B.5 Exercícios

B.1 Para cada instrução do *assembly* do 68K abaixo, apresente a instrução de máquina correspondente usando a representação em hexadecimal. Explique o efeito de cada instrução.

(a) CLR.W D5

(b) CLR.L (A7)

(c) ADDA.W D2,A1

(d) ADD.W #1,D1

(e) ADD.L #-1,D1

(f) MOVE.W #4,D5

(g) MOVEQ.W #4,D5

(h) DBF.W D0,-12 (onde -12 é o deslocamento que deve estar presente no campo da instrução)

B.2 Apresente o código de máquina para o programa abaixo. Para cada instrução de máquina, indique também a posição de memória onde a instrução deverá ser carregada (assumindo que a primeira instrução será carregada na posição 0 de memória). O que este programa faz?

```
CLR.W D0
MOVEQ.W #4,D1
ADD.W (A0)+,D0
DBF.W D1,-6
MOVE.W D0,(A0)
RTS
```

Apêndice C

Programação C

A linguagem de programação C foi desenvolvida no início dos anos 70 nos Laboratórios AT&T Bell, nos Estados Unidos. A motivação para que o autor de C, Dennis Ritchie, criasse uma nova linguagem de programação foi o desenvolvimento do sistema operacional Unix. C é uma ferramenta tão básica que praticamente todas as ferramentas suportadas por Unix e o próprio sistema operacional foram desenvolvidas em C.

C acompanhou o ritmo da distribuição do sistema operacional Unix, que foi amplamente divulgado e livremente distribuído na década de 70. Apesar de haver compiladores para linguagens mais "tradicionais" na distribuição Unix, aos poucos C foi ganhando simpatizantes e adeptos. Atualmente, não há dúvidas de que C é uma das linguagens de programação de maior aceitação para uma ampla classe de aplicações.

Um dos grandes atrativos da linguagem C é o balanço atingido entre características próximas da arquitetura de computadores e características de linguagens de programação com alto nível de abstração. O ascendente mais remoto de C, Algol 60, desenvolvida por um comitê internacional, foi uma linguagem que buscava um alto grau de abstração, com estruturas modulares e sintaxe regular. Por Algol ser "abstrata demais", variantes surgiram que buscavam aproximar aquela linguagem um pouco mais da máquina, tais como CPL *(Combined Programming Language)*, desenvolvida na Inglaterra. Esta linguagem era ainda muito complexa, o que dificultava seu aprendizado e a implementação de bons compiladores. BCPL *(Basic CPL)* buscava capturar apenas as características principais de CPL, e B (desenvolvida por Ken Thompson nos Laboratórios Bell, em 1970) levava este objetivo ainda mais adiante. Entretanto, estas linguagens ficaram tão "básicas" que tinham pouca aplicação direta. Ritchie reincorporou algumas características de alto nível à B, tais como suporte a tipos de dados, para criar a linguagem C.

A simplicidade de C não restringe, no entanto, a potencialidade de suas aplicações. Blocos desempenhando tarefas muito complexas podem ser criados a partir da combinação de blocos elementares, e este mecanismo de combinação de partes pode se estender por diversos níveis. Esta habilidade de construir aplicações complexas a partir de elementos simples é um dos principais atrativos da linguagem.

O sucesso de C foi tão grande que diversas implementações de compiladores surgiram, sendo que nem todos apresentavam o memso comportamento em pontos específicos, devido a características distintas arquiteturas de computadores ou a "extensões" que se incorporavam à linguagem. Para compatibilizar o desenvolvimento de programas em C, o Instituto Norte-Americano de Padrões (ANSI) criou em 1983 um comitê com o objetivo de padronizar a linguagem. O resultado deste trabalho foi publicado em 1990, e foi prontamente adotado como padrão internacional. Além de padronizar aspectos básicos da linguagem, ANSI-C também define um conjunto de rotinas de suporte que, apesar de não ser parte integrante da linguagem, deve ser sempre fornecido pelo compilador.

C.1 Organização básica de programas C

Na linguagem C, todo algoritmo deve ser traduzido para uma *função*. Uma função nada mais é do que um conjunto de expressões da linguagem C (possivelmente incluindo invocações ou *chamadas* para outras funções) com um nome e argumentos associados. Em geral, a definição de uma função C tem a forma

```
tipo nome(lista de argumentos) {
   declaracoes;
   comandos;
}
```

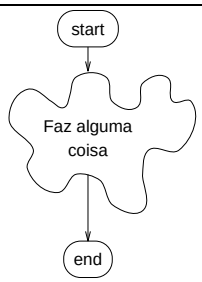
O *tipo* indica o valor de retorno de uma função, podendo assumir qualquer valor válido da linguagem C (que serão vistos adiante). O tipo da função pode ser omitido, sendo que neste caso o compilador irá assumir que o tipo `int` (inteiro) será retornado.

Nome é o rótulo dado à função, que em geral deve expressar de alguma forma o que a função realiza. Nos compiladores mais antigos, o número de caracteres em um nome era limitado (em geral, a 6 ou 8 caracteres). Atualmente, não há restrições ao comprimento de um nome, de forma que nomes significativos devem ser preferencialmente utilizados. Em C, todo nome que estiver seguido por parênteses será reconhecido como o nome de uma função. A *lista de argumentos* que fica no interior dos parênteses indica que valores a função precisa para realizar suas tarefas. Quando nenhum valor é necessário para a função, a lista será vazia, como em `( )`.

O que se segue na definição da função, delimitado entre chaves { e }, é o corpo da função. *Declarações* correspondem às variáveis internas que serão utilizadas pela função, e *comandos* implementam o algoritmo associado à função.

Todo algoritmo (e consequentemente todo programa) deve ter um ponto de início e um ponto de fim de execução. Na Figura C.1, esta estrutura básica de um algoritmo é ilustrada — onde há uma "nuvem" *faz alguma coisa*, deve ser inserido o corpo do algoritmo que descreve a função a ser realizada.

Figura C.1 Estrutura básica de um algoritmo.



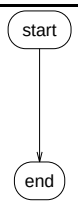
Um programa C é basicamente um conjunto de funções. O ponto de início e término de execução de um programa C está associado com uma função com um nome especial: a função **main** (principal). O menor programa C que pode ser compilado corretamente é um programa que nada faz (Figura C.2). Este programa C é:

```
main( ) {
}
```

Neste caso, a função de nome `main` é definida sem nenhum comando. Neste ponto, algumas observações devem ser feitas:

- Todo programa C tem que ter pelo menos uma função.

Figura C.2 Algoritmo que faz nada.



- Pelo menos uma função do programa C tem o nome `main` — esta função indica o ponto onde se iniciará a execução do programa, e após executado seu último comando o programa finaliza sua execução.

Ao contrário do que ocorre em Pascal ou FORTRAN, que diferenciam procedimentos (subrotinas) de funções, em C há apenas funções; mesmo que elas não retornem nenhum valor, o valor de retorno especial `void` é utilizado.

C.1.1 Declarações de variáveis

Variáveis representam uma forma de identificar por um nome simbólico uma região da memória que armazena um valor sendo utilizado por uma função. Em C, uma variável deve estar associada a um dos tipos de dados descritos na Seção 2.1.

Toda variável que for utilizada em uma função C deve ser previamente declarada. A forma geral de uma declaração de variável é:

```
tipo  nome_variavel;
```

ou

```
tipo  nome_var1, nome_var2, ... ;
```

onde `nome_var1, nome_var2, ...` são variáveis de um mesmo tipo de dado. Exemplos válidos de declaração de variáveis em C são:

```
int um_inteiro;
unsigned int outro_inteiro;
char c1, c2;
float SalarioMedio;
double x,
       y;
```

Nomes de variáveis podem ser de qualquer tamanho, sendo que usualmente nomes significativos devem ser utilizados. C faz distinção entre caracteres maiúsculos e caracteres minúsculos, de forma que `Salario-medio` é diferente de `SalarioMedio`.

Há restrições aos nomes de variáveis. Palavras associadas a comandos e definições da linguagem (tais como `if`, `for` e `int`) são *reservadas*, não podendo ser utilizadas para o nome de variáveis. A lista de palavras reservadas em C são apresentadas no Apêndice C.9. O nome de uma variável pode conter letras e números, mas deve começar com uma letra.

Como pode ser observado no exemplo acima, diversas variáveis de um mesmo tipo podem ser declaradas em um mesmo comando, sendo que o nome de cada variável neste caso estaria separado por vírgulas. Além disto, variáveis podem ser também inicializadas enquanto declaradas, como em

```
int a = 0,
    b = 20;
char c = 'X';
long int d = 12345678L;
```

Na última linha deste exemplo (inicialização da variável `d`), o sufixo `L` indica que a constante é do tipo `long`.

Uma variável cujo valor não será alterado pelo programa pode ser qualificada como `const`, como em

```
const int NotaMaxima = 100;
```

Neste caso, a variável `NotaMaxima` não poderá ter seu valor alterado. Evidentemente, variáveis deste tipo devem ser inicializadas no momento de sua declaração.

C.1.2 Expressões

Após a declaração das variáveis, o corpo de uma função é definido através dos comandos que serão executados pela função. Estes comandos devem ser expressos sob a forma de uma seqüência de expressões válidas da linguagem C.

Antes de mais nada, é interessante que se apresente a forma de se expressar comentários em um programa C. Comentários em C são indicados pelos terminadores `/*` (início de comentário) e `*/` (fim de comentário). Quaisquer caracteres entre estes dois pares de símbolos são ignorados pelo compilador. Comentários em C não podem ser aninhados, mas podem se estender por diversas linhas e podem começar em qualquer coluna. Por exemplo,

```
/*  Exemplo de
 *  comentario
 */
main( ) {
    /* esta funcao nao faz coisa alguma */
}
```

As expressões na linguagem C são sempre terminadas pelo símbolo `;` (ponto e vírgula). Uma *expressão nula* é constituída simplesmente pelo símbolo terminador. Assim, o exemplo acima é equivalente a

```
/*  Exemplo de
 *  comentario
 */
main( ) {
    /* esta funcao nao faz coisa alguma */
    ;
}
```

Expressões aritméticas

O comando de atribuição em C é indicado pelo símbolo `=`, como em

```
main() {
   int a, b, c;

   a = 10;      /* a recebe valor 10 */
   b = c = a;   /* b e c recebem o valor de a (10) */
}
```

Observe neste exemplo que a atribuição pode ser encadeada — na última linha da função acima, `c` recebe inicialmente o valor da variável `a`, e então o valor de `c` será atribuído à variável `b`.

Expressões aritméticas em C podem envolver os operadores binários (isto é, operadores que tomam dois argumentos) de *soma* (`+`), *subtração* (`-`), *multiplicação* (`*`), *divisão* (`/`). Valores negativos são indicados pelo operador unário `-`. Adicionalmente, para operações envolvendo valores inteiros são definidos os operadores de resto da divisão inteira ou *módulo* (`%`), incremento (`++`) e decremento (`--`). Por exemplo,

```
main() {
  int a=10, b, c, d;

  b = 2*a;       /* b = 20 */
  a++;           /* a = a+1 (11) */
  c = b/a;       /* divisao inteira:  c = 1 */
  d = b%a;       /* resto da divisao: d = 9 */
}
```

Cada um dos operadores de incremento e decremento tem duas formas de uso, dependendo se eles ocorrem antes do nome da variável (pré-incremento ou pré-decremento) ou depois do nome da variável (pós-incremento ou pós-decremento). No caso do exemplo acima, onde o operador de incremento ocorre de forma isolada em uma expressão (sozinho na linha), as duas formas possíveis são equivalentes. A diferença entre eles ocorre quando estes operadores são combinados com outras operações. No exemplo acima, as linhas de atribuição à `b` e incremento de `a` poderiam ser combinados em uma única expressão,

```
b = 2*(a++);    /* b recebe 2*a e entao a recebe a+1 */
```

Observe como esta expressão é diferente de

```
b = 2*(++a);    /* a recebe a+1 e entao b recebe 2*a */
```

Na prática, os parênteses nas duas expressões acima poderiam ser omitidos uma vez que a precedência do operador de incremento é maior que da multiplicação — ou seja, o incremento será avaliado primeiro. O Apêndice C.10 apresenta a ordem de avaliação para todos os operadores da linguagem C.

C tem também uma forma compacta de representar expressões na forma

```
var = var op (expr);
```

onde uma mesma variável `var` aparece nos dois lados de um comando de atribuição. A forma compacta é

```
var op= expr;
```

Por exemplo,

```
a += b;      /* equivale a a = a+b */
c *= 2;      /* equivale a c = c*2 */
```

Expressões condicionais

Um tipo muito importante de expressão em C é a *expressão condicional*, cujo resultado é um valor que será interpretado como *falso* ou *verdadeiro*. Como a linguagem C não suporta diretamente o tipo de dado *booleano*, ela trabalha com representações inteiras para denotar estes valores — o resultado de uma expressão condicional é um valor inteiro que será interpretado como *falso* quando o valor resultante da expressão é igual a 0, e como *verdadeiro* quando o valor resultante é diferente de 0.

Assim, qualquer expressão inteira pode ser interpretada como uma expressão condicional. A situação mais comum, entretanto, é ter uma expressão condicional comparando valores através dos operadores relacionais. Os operadores relacionais em C são:

`>`	maior que
`>=`	maior que ou igual a
`<`	menor que
`<=`	menor que ou igual a
`==`	igual a
`!=`	diferente de

Observe que o operador de igualdade é `==`, e não `=` como em Pascal! Esta é uma causa comum de erros para programadores que estão acostumados com outras linguagens onde `=` é um operador relacional.

Expressões condicionais elementares (comparando duas variáveis ou uma variável e uma constante) podem ser combinadas para formar expressões complexas através do uso de operadores booleanos. Estes operadores são

`&&`	AND
`||`	OR
`!`	NOT

O operador `&&` *(and)* resulta verdadeiro quando as duas expressões envolvidas sãão verdadeiras (ou diferente de 0). O operador `||` *(or)* resulta verdadeiro quando pelo menos uma das duas expressões envolvidas é verdadeira. Além destes dois conectores binários, há também o operador unário de negação, `!`, que resulta falso quando a expressão envolvida é verdadeira (diferente de 0) ou resulta verdadeiro quando a expressão envolvida é falsa (igual a 0).

Expressões lógicas complexas, envolvendo diversos conectores, são avaliadas da esquerda para a direita. Além disto, `&&` tem precedência maior que `||`, e ambos têm precedência menor que os operadores lógicos relacionais e de igualdade. Entretanto, recomenda-se sempre a utilização de parênteses em expressões para tornar claro quais operações são desejadas. A exceção a esta regra ocorre quando um número excessivo de parênteses pode dificultar ainda mais a compreensão da expressão; em tais casos, o uso das regras de precedência da linguagem pode facilitar o entedimento da expressão.

Expressão para manipulação de bits

A linguagem C oferece também operadores que trabalham sobre a representação binária de valores inteiros e caracteres. Estes operadores são:

`&`	*AND* bit-a-bit
`|`	*OR* bit-a-bit
`^`	*XOR* bit-a-bit
`<<`	deslocamento de bits à esquerda
`>>`	deslocamento de bits à direita
`~`	complemento de um (inverte cada bit)

Expressões envolvendo esses operadores tomam dois argumentos — exceto pelo operador `~`, que é unário. Por exemplo,

```
a = x & 0177;    /* a recebe 7 bits menos signif. de x */
b &= ~0xFF;      /* zera os 8 bits menos signif. de b */
c >>= 4;         /* desloca bits de 4 pos. a direita */
```

C.1.3 Controle do fluxo de execução

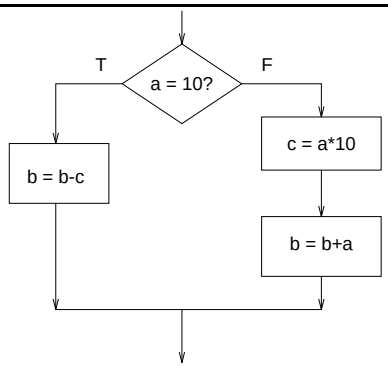
C é uma linguagem que suporta a programação estruturada, ou seja, permite agrupar comandos na forma de seqüência, seleção e repetição.

Uma seqüência de comandos em uma função C é denotada simplesmente como uma seqüência de expressões, como já exemplificado em diversos exemplos anteriores. Além de uma expressão ser composta por um único comando, é possível ter uma única expressão com a seqüência de comandos separados pelo operador `,` (vírgula), como em

```
for (i=0, j=0; i<k; ++i, j+=2) {
 ...
}
```

A construção de seleção `IF-THEN-ELSE` é expressa em C com as palavras-chaves `if ... else` (Figura C.3). Após a palavra-chave `if` deve haver uma expressão condicional entre parênteses. Se a expressão for avaliada como verdadeira, então a expressão sob `if` será realizada; se for falsa, a expressão sob `else` será executada.

Figura C.3 Seleção com if . . . else em C: (a) fluxograma; (b) equivalente em C.



(a)

```
if (a == 10)
    b = b-c;
else {
    c = a*10;
    b = b+a;
}
```

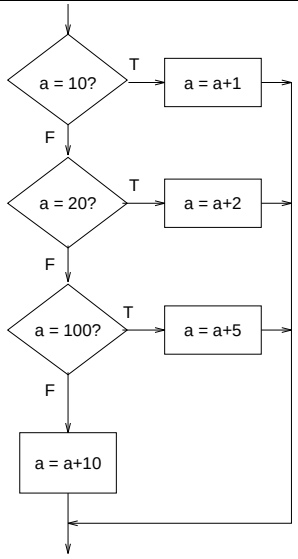
(b)

Nesta figura, introduz-se o conceito de *expressão composta*, ou seja, a expressão da parte *else* deste exemplo é na verdade um bloco contendo diversas expressões. Neste caso, o bloco de comandos que deve ser executado nesta condição deve ser delimitado por chaves { e }. Algumas observações adicionais relevantes com relação a este comando são:

1. Em C, há diferenças entre letras minúsculas e maiúsculas. Como todos os comandos em C, as palavras chaves deste comando estão em letras minúsculas. Assim, as formas *IF* (ou *If* ou *iF*) não são formas válidas em C para denotar o comando `if`.
2. Ao contrário do que ocorre em Pascal ou FORTRAN, a palavra *then* não faz parte da sintaxe deste comando em C.
3. A cláusula *else* pode ser omitida quando a expressão a executar na condição falsa for nula.
4. No caso de haver mais de um `if` que possa ser associado a uma cláusula `else`, esta será associada ao comando `if` precedente mais próximo.

A construção estruturada de seleção `SWITCH` é suportada em C pelo comando *switch . . . case* (Figura C.4). Neste caso, após a palavra-chave `switch` deve haver uma *variável* do tipo inteiro ou caráter entre parênteses. Após a variável, deve haver uma lista de casos que devem ser considerados, cada caso iniciando com a palavra-chave `case` seguida por um valor ou uma expressão inteira.

Figura C.4 Seleção em C usando a forma switch . . . case. Observe que o conjunto de ações associado a cada caso encerra-se com a palavra-chave *break*.



(a)

```
switch (a) {
  case 10:  a = a+1;
            break;
  case 20:  a = a+2;
            break;
  case 100: a = a+5;
            break;
  default:  a = a+10;
}
```

(b)

Neste exemplo, a variável `a` pode ser do tipo `int` ou `char`. A palavra-chave especial `default` indica que ação deve ser tomada quando a variável assume um valor que não foi previsto em nenhum dos casos. Assim como a condição `else` no comando `if` é opcional, a condição `default` também é opcional para o `switch-case`. Observe também a importância da palavra-chave `break` para delimitar o escopo de ação de cada caso — fossem omitidas as ocorrências de `break` no exemplo, a semântica associada ao comando seria essencialmente diferente (Figura C.5).

Comandos de repetição em C são suportados em três formas distintas. A primeira forma é `while`, cuja construção equivale ao comando estruturado `WHILE-DO` (Figura C.6(a) e (b)), enquanto que a segunda forma equivale ao comando estruturado `DO-WHILE` (Figura C.6(c) e (d)).

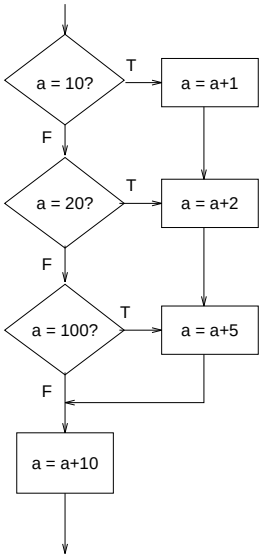
A terceira forma associada ao comando de repetição em C, `for`, facilita a expressão de iterações associadas a contadores. Um exemplo de uso deste comando é apresentado na Figura C.7.

Neste exemplo, `a` é uma variável que tem a função de contador, assumindo valores 0, 1, . . . , `MAX-1`. Enquanto o valor de `a` for menor que `MAX` (a condição de término da iteração), a expressão (simples ou composta) no corpo da iteração será repetidamente avaliada.

Qualquer que seja forma usada para indicar o comando de repetição — `while`, `do while` ou `for` — há duas formas de se desviar a seqüência de execução do padrão do comando. A primeira forma, `continue`, serve para indicar o fim prematuro de *uma* iteração. A outra forma de interrupção de um comando de repetição é o comando `break`, que indica o fim prematuro de *todo* o comando de iteração. Por exemplo, em

```
for (a=0; a<MAX; ++a) {
    if (b == 0) {
        b = a;
        continue;
    }
    c = c/b;
    b = b-1;
```

Figura C.5 Seleção em C usando a forma switch . . . case onde se omitiu a palavra-chave *break* do bloco de comandos.

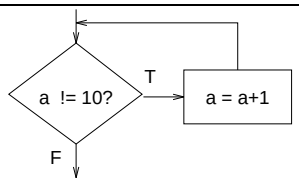


(a)

```
switch (a) {
  case 10:  a = a+1;
  case 20:  a = a+2;
  case 100: a = a+5;
  default:  a = a+10;
}
```

(b)

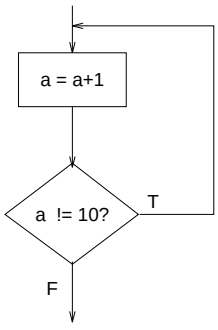
Figura C.6 Repetição em C: (a) forma *while*, fluxograma; (b) forma *while*, código C; (c) forma *do* . . . *while*, fluxograma; (d) forma *do* . . . *while*, código C.



(a)

```
while (a != 10)
   a = a+1;
```

(b)

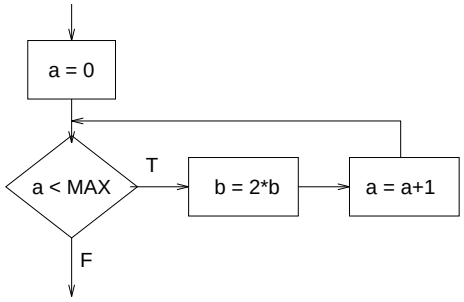


(c)

```
do
   a = a+1;
while (a != 10);
```

(d)

Figura C.7 Repetição em C usando a forma *for*: (a) fluxograma; (b) código C.



(a)

```
for (a=0; a<MAX; ++a)
   b = 2*b;
```

(b)

```
}
```

se a linha com o comando `continue` for executada, o valor de `a` será incrementado e então o teste da iteração será reavaliado para definir a continuidade ou não do laço de repetição. Já no exemplo abaixo,

```
for (a=0; a<MAX; ++a) {
    if (b == 0)
        break;
    c = c/b;
    b = b-1;
}
```

quando (se) `b` assumir o valor 0, o laço será simplesmente interrompido.

C.1.4 Invocação de funções

Já foi visto na Seção C.1 como uma função é definida. Agora será visto como usar uma função.

Uma função agrupa sob um nome simbólico um conjunto de expressões que realizam uma ação que, potencialmente, pode se repetir diversas vezes. Por exemplo, considere uma aplicação em instrumentação onde os 8 bits menos significativos de uma variável inteira devem ser frequentemente lidos. Seria possível fazer uma função associada à leitura de cada bit, como em

```
int get_bit1 (int var) {
    return (var & 01);
}

int get_bit2 (int var) {
    return ((var & 02) >> 1);
}
```

```
...

int get_bit8 (int var) {
    return ((var & 0200) >> 7);
}
```

o que já auxiliaria no sentido de evitar a repetição das operações de manipulação de bits, reduzindo assim a possibilidade de erro e facilitando o entendimento do código. Estas funções poderiam estar sendo utilizadas pela aplicação como em

```
main() {
    int leitura, bit;  /* declaracao de variavel */
    int faz_medida(void),  /* declaracao de funcoes  */
        get_bit1(int),
        get_bit2(int),
        ...
        get_bit8(int);

    do {
        /* alguma funcao de medida */
        leitura = faz_medida();
        bit = get_bit1(leitura);
        ... /* trata o bit 1 */
        bit = get_bit2(leitura);
        ... /* trata o bit 2 */
        ...
    } while (1);
}
```

Este exemplo ilustra alguns pontos interessantes. O primeiro deles é a declaração de funções, onde se expressa quais funções serão utilizadas, quais os tipos de seus argumentos e quais os seus valores de retorno. Um dos "tipos" indicados é `void`, o que significa *vazio* — ou seja, a função `faz_medida` deste exemplo não toma nenhum argumento e retorna um valor inteiro.

Outro ponto interessante é o uso da forma `while(1)`, o que equivale a dizer "faça para sempre." Esta forma é frequentemente utilizada, sendo em geral interrompida por um comando `break`. Usando o comando `for`, a forma equivalente seria `for(;;)`.

Outra observação que deve ser feita é que o valor da variável `leitura` não se altera com as chamadas das diversas funções. Esta é uma característica da *passagem por valor* dos argumentos da função, que é o padrão em C. O que cada função manipula — cada variável `var` — é na verdade uma cópia do valor da variável passada como argumento (`leitura`), e não diretamente a variável.

A função `printf`

A função `printf` é parte de um conjunto de funções pré-definidas armazenadas em uma biblioteca padrão de rotinas da linguagem C. Ela permite apresentar na tela os valores de qualquer tipo de dado. Para tanto, `printf` utiliza o mecanismo de *formatação*, que permite traduzir a representação interna de variáveis para a representação ASCII que pode ser apresentada na tela.

O primeiro argumento de `printf` é um *string de controle*, uma seqüência de caracteres entre aspas. Esta *string*, que sempre deve estar presente, pode especificar através de caracteres especiais (as *seqüências de conversão*) quantos outros argumentos estarão presentes nesta invocação da função. Estes outros argumentos

serão variáveis cujos valores serão formatados e apresentados na tela. Por exemplo, se o valor de uma variável inteira `x` é 12, então a execução da função

```
printf("Valor de x = %d", x);
```

imprime na tela a frase `Valor de x = 12`. Se `y` é uma variável do tipo caráter com valor 'A', então a execução de

```
printf("x = %d e y = %c\n", x, y);
```

imprime na tela a frase `x = 12 e y = A` seguida pelo caráter de nova linha (`\n`), ou seja, a próxima saída para a tela aconteceria na linha seguinte. Observe que a seqüência de conversão pode ocorrer dentro de qualquer posição dentro do *string* de controle.

A função `printf` não tem um número fixo de argumentos. Em sua forma mais simples, pelo menos um argumento deve estar presente — a *string* de controle. Uma *string* de controle sem nenhuma seqüência de conversão será literalmente impressa na tela. Com variáveis adicionais, a única forma de saber qual o número de variáveis que será apresentado é por inspeção da *string* de controle. Desta forma, cuidado deve ser tomado para que o número de variáveis após a *string* de controle esteja de acordo com o número de seqüências de conversão presente na *string* de controle.

Além de ter o número correto de argumentos e seqüências de conversão, o tipo de cada variável deve estar de acordo com a seqüência de conversão especificada na *string* de controle. A seqüência de conversão pode ser reconhecida dentro da *string* de controle por iniciar sempre com o caráter `%`.

As principais seqüências de conversão para variáveis caracteres e inteiras são:

`%c` imprime o conteúdo da variável com representação ASCII;

`%d` imprime o conteúdo da variável com representação decimal com sinal;

`%u` imprime o conteúdo da variável com representação decimal sem sinal;

`%o` imprime o conteúdo da variável com representação octal sem sinal;

`%x` imprime o conteúdo da variável com representação hexadecimal sem sinal.

Uma largura de campo pode ser opcionalmente especificada logo após o caráter `%`, como em `%12d` para especificar que o número decimal terá reservado um espaço de doze caracteres para sua representação. Se a largura de campo for negativa, então o número será apresentado alinhado à esquerda ao invés do comportamento padrão de alinhamento à direita. Para a conversão de variáveis do tipo `long`, o caráter `l` também deve ser especificado, como em `%ld`.

Para converter variáveis em ponto flutuante, as seqüências são:

`%f` imprime o conteúdo da variável com representação com ponto decimal;

`%e` imprime o conteúdo da variável com representação em notação científica (exponencial);

`%g` formato geral, escolhe a representação mais curta entre `%f` e `%e`.

Como para a representação inteira, uma largura de campo pode ser especificada para números reais. Por exemplo, `%12.3f` especifica que a variável será apresentada em um campo de doze caracteres com uma precisão de três dígitos após o ponto decimal.

Finalmente, se a variável a ser apresentada é uma seqüência de caracteres (uma *string*), então o formato de conversão `%s` pode ser utilizado. Para apresentar o caráter `%`, a seqüência `%%` é utilizada.

C.2 Tipos agregados e derivados

Além dos tipos básicos já apresentados na Seção 2.1.1, C permite trabalhar com outros tipos agregados e/ou derivados, tais como arranjos, estruturas, uniões e enumerações. O mecanismo de definição de nomes de tipos permite ainda que tantos tipos básicos quanto derivados possam ser tratados de maneira uniforme.

C.2.1 Arranjos

Os princípos da definição e manipulação de arranjos em C foram descritos na Seção 2.1.3. Assim como para variáveis de tipos básicos, os elementos de um arranjo podem ser também inicializados durante a declaração da variável. Neste caso, os valores de cada um dos elementos são delimitados por chaves. Por exemplo,

```
main() {
   /* declara um arranjo de cinco inteiros */
   int elem[5] = {0,0,0,0,0};

   ...
}
```

O índice de um arranjo pode ser qualquer expressão inteira, incluindo-se variáveis e constantes inteiras. C não verifica se o valor do índice está dentro da faixa declarada — é responsabilidade do programador garantir que o acesso esteja dentro dos limites de um arranjo.

Arranjos podem ser multidimensionais. Por exemplo, um arranjo bidimensional pode ser declarado, inicializado e acessado como em

```
main() {
   /* declara um arranjo de duas linhas
      com tres inteiros por linha */
    int elem[2][3] = { {0,0,0}, {0,0,0}};
    int i, j;
    /* modifica conteudo do arranjo */
    for (i=0; i<2; ++i)
       for (j=0; j<3; ++j)
          elem[i][j] = i*3 + j;
}
```

Em C, um arranjo bidimensional é na verdade um arranjo unidimensional onde cada elemento é um arranjo. Por este motivo, dois operadores de indexação são utilizados ao invés da forma `[i,j]`. Elementos são armazenados por linha.

Em geral, arranjos com muitas dimensões não são utilizados em C visto que ocupam muito espaço e o acesso a seus elementos não ocorre de forma eficiente.

C.2.2 *Strings*

Um dos tipos de arranjos que mais ocorre em C é o arranjo de caracteres, ou *string*. C não suporta um tipo básico `string`; ao invés, há uma convenção para tratamento de arranjos de caracteres que permite o uso de diversas funções de manipulação de *strings* na linguagem.

Por convenção, C considera como uma *string* uma seqüência de caracteres armazenada sob a forma de um arranjo de tipo `char` cujo último elemento é o caráter NUL, tipicamente representado na forma de caráter, `'\0'`, ou simplesmente pelo seu valor, 0. Por exemplo, um *string* poderia ser declarado e inicializado como em

```
char exemplo[4] = {'a','b','c','\0'};
```

Observe que o espaço para o caráter `'\0'` deve ser previsto quando dimensionando o tamanho do arranjo de caracteres que será manipulado como *string*. No exemplo, o arranjo de quatro caracteres pode receber apenas três letras, já que o último caráter está reservado para o NUL.

C suporta uma forma alternativa de representação de um *string* constante, que é através do uso de aspas:

```
char exemplo[4] = "abc";
```

Este exemplo é equivalente ao anterior — a *string* `"abc"` contém quatro caracteres, sendo que o caráter `'\0'` é automaticamente anexado à *string* pelo compilador.

Funções que manipulam *strings* trabalham usualmente com a referência para o início da seqüência de caracteres, ou seja, com um ponteiro para a *string*. A manipulação de ponteiros é fonte usual de confusão em qualquer linguagem. Por exemplo, tendo duas variáveis ponteiros `char* s1` e `char* s2` indicando o início de duas *strings*, não seria possível copiar o conteúdo de `s2` para `s1` simplesmente por atribuição,

```
s1 = s2;        /* copia o endereco! */
```

ou comparar seus conteúdos diretamente,

```
if (s1 != s2)   /* compara os enderecos! */
   ...
```

Diversas rotinas são suportadas para manipular *strings* na biblioteca padrão de C, tais como:

```
char  *strcat(char *s1, const char *s2);
char  *strncat(char *s1, const char *s2, size_t n);
int    strcmp(const char *s1, const char *s2);
int    strncmp(const char *s1, const char *s2, size_t n);
char  *strcpy(char *s1, const char *s2);
char  *strncpy(char *s1, const char *s2, size_t n);
size_t strlen(const char *s);
char  *strchr(const char *s, int c);
```

Os protótipos para essas rotinas estão definidos no arquivo de cabeçalho `string.h`. Nesses protótipos, `size_t` é um nome de tipo (ver Seção C.2.6) para representar tamanhos correspondentes a algum tipo inteiro definido no arquivo de cabeçalho `stddef.h`.

A função `strcat` concatena a *string* apontada por `s2` à *string* apontada por `s1`. O arranjo associado ao endereço `s1` deve ter espaço suficiente para armazenar o resultado concatenado. A função `strncat` permite limitar a quantidade de caracteres concatenados, agregando no máximo `n` caracteres de `s2` a `s1`.

A função `strcmp` compara o conteúdo de duas *strings*. Se a *string* apontada por `s1` for igual àquela de `s2`, o valor de retorno da função é 0. Se forem diferentes, o valor de retorno será negativo quando a *string* em `s1` for lexicograficamente menor que aquela de `s2`, ou positivo caso contrário. A função `strncmp` compara no máximo até `n` caracteres das duas *strings*.

A função `strcpy` copia a *string* em `s2` (até a ocorrência do caráter `'\0'`) para o arranjo apontado por `s1`. A função `strncpy` copia no máximo até `n` caracteres. Observe que neste caso, se a *string* em `s2` for maior que `n` caracteres, a *string* resultante não será terminada pelo caráter `'\0'`.

A função `strlen` retorna o comprimento da *string* em `s`, sem incluir nessa contagem o caráter `'\0'`.

A função `strchr` retorna o apontador para a primeira ocorrência do caráter `c` na *string* em `s`, ou o apontador nulo se o caráter não está presente na *string*.

Além destas funções, é interessante destacar que existe uma função `sprintf` (declarada em `stdio.h` e parte da biblioteca padrão) que permite formatar valores seguindo o mesmo padrão utilizado em `printf`, com a diferença que a saída formatada é colocada em uma *string* ao invés de ser enviada para a tela. Seu protótipo é:

```
int sprintf (char *s, const char *format, ...);
```

É responsabilidade do programador garantir que o arranjo para a *string* apontado por `s` tenha espaço suficiente para armazenar o resultado.

C.2.3 Estruturas

Através do conceito de estrutura, C oferece um mecanismo uniforme para a definição de registros, unidades de informação organizadas em campos de tipos que podem ser não-homogêneos. A definição de estruturas foi descrita na Seção 2.1.3.

Como um exemplo, esse mecanismo pode ser aplicado tanto a uma linha de código *assembly*, com campos associados a rótulo, código de operação, operandos e comentários, como a tabelas, com campos símbolo e valor. Considere a definição de uma entidade `linha`, composta por quatro componentes — `rotulo`, `opcode`, `operand`, `comment`. Cada componente é um arranjo de caracteres de tamanho adequado à recepção de um dos campos das linhas de um programa *assembly*. Em uma estrutura C,

```
#define LINSIZE 80
#define LABSIZE 8
#define OPCSIZE 10
#define OPRSIZE 20
#define CMTSIZE (LINSIZE-(LABSIZE+OPCSIZE+OPRSIZE))
typedef struct linha {
    char rotulo[LABSIZE];
    char opcode[OPCSIZE];
    char operand[OPRSIZE];
    char comment[CMTSIZE];
} Linha;
```

Variáveis deste tipo de estrutura podem ser definidas como

```
Linha  l;
```

Apesar deste exemplo só ter como componentes seqüências de caracteres, qualquer tipo válido pode estar presente em uma estrutura — até mesmo outra estrutura, como em

```
struct linCode {
    int    posicao;
    Linha  linha;
};
```

Usando essa estrutura `Linha`, uma versão simplificada da função `getLabel` usada na Seção 4.2.1 seria:

```
char *getLabel(Linha linha) {
    char term;
    int pos;
    do {
      term = linha.rotulo[pos];
      if (term == ':' || term == ' ') {
         linha.rotulo[pos] = '\0';
         return(linha.rotulo);
      }
      ++pos;
```

```
        } while (pos < LABSIZ);
        linha.rotulo[pos] = '\0';
        return(linha.rotulo);
    }
```

Em aplicações sucessivas do operador `.` — quando for necessário acessar estruturas aninhadas — a associatividade é da esquerda para a direita.

Em situações onde há a necessidade de se minimizar o espaço ocupado por dados de um programa, pode ser preciso compartilhar uma palavra da máquina para armazenar diversas informações. Neste caso, é preciso trabalhar a nível de subseqüências de bits internas à representação de um número inteiro.

A forma básica de manipular bits em um inteiro seria através dos operadores bit-a-bit, trabalhando com máscaras e deslocamentos para isolar as informações individuais. Entretanto, a linguagem C suporta o conceito de *campos* de bits internos a um inteiro para facilitar este tipo de manipulação. Campos são baseados em estruturas.

Por exemplo, considere uma instrução de um dado dispositivo que está sendo programado em C. Cada instrução neste dispositivo têm um formato binário de codificação em quatro bits que especificam a operação, seguidos por dois campos de seis bits cada que especificam a fonte e o destino associados à operação dada. Com o uso de campos, uma variável poderia ser definida como

```
    struct {
        unsigned int opcode: 4;
        unsigned int fonte: 6;
        unsigned int dest: 6;
    } instrucao;
```

Campos são acessados exatamente da mesma forma que membros de estruturas, comportando-se como inteiros sem sinal.

C.2.4 Uniões

Uma *união* permite que uma dada área de memória seja tratada como variáveis de tipos diferentes em instantes de tempos diferentes. Uniões são definidas e acessadas de forma similar a estruturas, usando a palavra chave `union` ao invés de `struct`.

A título de exemplo, considere a situação onde se queira obter a representação interna de um número em ponto flutuante especificado na linha de comando. O programa a seguir pode ser usado para imprimir esta representação em octal:

```
#include <stdlib.h>

int main(int argc, char *argv[]) {
  union {
    float f;
    unsigned int i;
  } num;

  if (argc == 2) {
    num.f = atof(argv[1]);
    printf("%f tem representacao octal %o\n",
           num.f, num.i);
  }
```

```
  return 0;
}
```

O compilador se encarrega de reservar espaço para armazenar dados de tamanho do maior dos membros da união.

C.2.5 Enumerações

Uma outra forma de tipo composto em C é a *enumeração*. Usualmente, faz parte do processo de desenvolvimento de um programa associar códigos númericos a variáveis que podem assumir um único valor dentre um conjunto finito de opções. O tipo enumeração permite associar nomes descritivos a tais conjuntos de valores numéricos.

Considere uma extensão da estrutura `dados_pessoais` apresentada acima que incorporasse também o sexo da pessoa. Há dois estados possíveis para uma variável deste tipo: ela pode assumir o valor `masculino` ou o valor `feminino`. Uma enumeração que poderia representar este tipo de informação seria

```
enum sex { masculino, feminino };
```

Uma variável deste tipo de enumeração poderia ser então incorporada na estrutura apresentada a seguir, `dados_pessoais`,

```
struct dados_pessoais {
    char nome[40];
    struct data nascimento;
    enum sex genero;
};
```

O seguinte trecho de programa ilustra como os nomes descritivos definidos em enumerações são utilizados como valores:

```
int calc_idade(struct dados_pessoais pessoa,
               struct data hoje) {
    int idade;

    idade = hoje.ano - pessoa.nascimento.ano;
    if (pessoa.genero == feminino)
        idade -= 10;

    return(idade);
}
```

Internamente, o compilador designa o valor 0 para o primeiro símbolo da enumeração, e incrementa de um o valor associado a cada símbolo na seqüência. Isto pode ser modificado se o programador quiser através de atribuição explícita de um valor inteiro a um símbolo, como em

```
enum cedula {
    beijaflor = 1,
    garca = 5,
    arara = 10 };
```

C.2.6 Definição de nomes de tipos

Embora C não permita a criação de novos tipos de dados, ela oferece uma facilidade para criar novos nomes para os tipos existentes, sejam eles básicos ou derivados. Este mecanismo, `typedef`, permite principalmente melhorar a facilidade de compreensão de programas.

A forma geral de uma definição de nome de tipo é

```
typedef  tipo  novo_nome;
```

Por exemplo, os tipos de estruturas `data` e `dados_pessoais` definidos anteriormente poderiam ser associados a nomes de tipos `Data` e `Pessoa` respectivamente pelas declarações

```
typedef struct data Data;
typedef struct dados_pessoais Pessoa;
```

Com estas definições, as declarações do programa que apresenta a idade de pessoas poderiam ser reescritas como

```
/*
 *  Exemplo calcula idade com definicao de nomes de tipos
 */
/* Define estruturas e nomes de tipos */
typedef enum sex {masculino, feminino} Sexo;

typedef struct data {
            int dia;
            int mes;
            int ano;
        } Data;

typedef struct dados_pessoais {
            char nome[40];
            Data nascimento;
            Sexo genero;
        } Pessoa;

int main() {
    Data hoje;
    Pessoa aluno_pt;
    int idade;
    /* prototipos: */
    int calc_idade(Pessoa, Data);
    Data le_hoje();
    Pessoa le_aluno();
    ...
}
```

Outros exemplos de uso de `typedef` são

```
typedef unsigned int Tamanho;
typedef enum {false=0, true} Boolean;
```

C.3 Ponteiros

Um ponteiro é uma variável que contém um endereço de outra variável. Este conceito é amplamente utilizado por diversos computadores, estando diretamente relacionado ao modo de endereçamento indireto (Seção B.3).

Este mecanismo deve ser utilizado com critério e disciplina. O uso descuidado de ponteiros pode levar a situações onde um endereço inválido é acessado, levando a erros de execução de programas.

C.3.1 Aritmética de ponteiros

Não apenas o conteúdo de ponteiros podem tomar parte em expressões aritméticas. C também suporta o conceito de operações sobre endereços, embora as operações que possam ser utilizadas neste caso sejam limitadas. Tais operações definem a *aritmética de ponteiros*.

Para apresentar o conceito de aritmética de ponteiros, considere o seguinte exemplo:

```
main() {
  int arr[10];     /* arr: arranjo com 10 inteiros */
  int *el;         /* el:  ponteiro para um inteiro */
  int i;

  el = &arr[0];  /* inicializa ponteiro */

  /* inicializa conteudo do arranjo via ponteiro */
  for (i=0; i<10; ++i)
     *(el + i) = 0;
}
```

O ponteiro `el` aponta inicialmente para o primeiro elemento do arranjo `arr`, ou seja, `arr[0]` — `&arr[0]` é o endereço deste elemento. Assim, para acessar este elemento através do ponteiro, a expressão `*el` poderia ser utilizada. No entanto, é possível também acessar outros elementos do arranjo através do ponteiro. Para acessar o elemento seguinte, a expressão `*(el+1)` retorna o endereço do próximo inteiro armazenado após o endereço `el`, ou seja, o endereço de `arr[1]`. Portanto, o que a instrução interna ao laço no exemplo está realizando é o acesso a cada elemento do arranjo através de um ponteiro.

Um aspecto fundamental da aritmética de ponteiros é que ela libera o programador de saber qual a dimensão alocada para cada tipo de dado. Quando um ponteiro é incrementado, este incremento irá refletir o tamanho do tipo da variável que está sendo apontada. Assim, o exemplo acima funcionará independentemente da máquina no qual ele for executado, ocupe a representação de um inteiro dois ou quatro bytes. Quando a expressão `el+i` é encontrada, o endereço do `i`-ésimo inteiro após o endereço `el` é obtido — ou seja, esta expressão aponta para o elemento `arr[i]`.

A aritmética de ponteiros está limitada a quatro operadores: soma, subtração, incremento e decremento. Outra limitação é que, no caso de soma, o outro operando além do ponteiro deve ser uma expressão (ou variável ou constante) inteira. A subtração de dois ponteiros para um mesmo tipo de dado é uma operação válida, retornando o número de elementos entre os dois ponteiros. A comparação entre dois ponteiros também é uma operação legal.

Há uma única exceção ao uso legal de aritmética de ponteiros: quando um ponteiro é definido para o tipo `void`. Neste caso, a variável ponteiro contém um endereço genérico, sobre um tipo que não pode ser determinado. Portanto, operações aritméticas sobre este tipo de ponteiro não são permitidas.

C.3.2 Ponteiros e arranjos

Como observado no exemplo acima, ponteiros e arranjos estão intimamente relacionados em C. Na verdade, qualquer referência a um arranjo é convertida internamente para uma referência do tipo ponteiro. Por este motivo, quando eficiência de tempo de acesso é uma preocupação, muitos programadores trabalham diretamente com ponteiros.

O nome de um arranjo é uma expressão do tipo ponteiro que corresponde ao endereço do primeiro elemento do arranjo. Assim, a inicialização do ponteiro no exemplo acima poderia ser reescrita como

```
el = arr;   /* arr equivale a &arr[0] */
```

Observe que, uma vez que `arr` equivale a um ponteiro, o elemento `arr[i]` poderia ser acessado da mesma forma como `*(arr+i)`. Na verdade, é isto que o compilador irá fazer internamente: qualquer expressão da forma `E1[E2]` será internamente traduzida para `*((E1)+(E2))`. Observe que isto implica que esta operação de indexação é comutativa, embora tal fato raramente seja utilizado em programação C.

Por outro lado, o inverso (usar o operador de indexação com uma variável ponteiro) também é possível. Assim, o laço de atribuição no exemplo acima poderia ter sido escrito como

```
for (i=0; i<10; ++i)
    el[i] = 0;
```

Apesar da forma usando o operador `*` ser mais eficiente, programadores iniciantes muitas vezes acham mais simples entender o acesso usando o operador de indexação, e acabam preferindo esta forma.

Uma diferença fundamental entre um ponteiro e o nome de um arranjo é que o ponteiro é uma *variável*, enquanto que o nome de um arranjo é uma *constante*. Assim, expressões como `arr++` ou `&arr` não fazem sentido.

Outra diferença que deve ser ressaltada é o fato de que a declaração de um arranjo efetivamente reserva o espaço para as variáveis, enquanto que a declaração de um ponteiro reserva apenas espaço para guardar um endereço. Considere o seguinte exemplo:

```
/*
 *  Exemplo do uso indevido de ponteiro
 */
main() {
  int *el;               /* el:  ponteiro para inteiro */
  int i;

  /* inicializa conteudo */
  for (i=0; i<10; ++i)
       el[i] = 0;        /* onde esta el[i]? */
}
```

Uma vez que o ponteiro `el` não foi inicializado, a expressão `el[i]` pode estar apontando para qualquer posição da área de memória — possivelmente, para alguma posição inválida. Observe que o fato de ter declarado o ponteiro não significa que esta variável possa ser utilizada como um arranjo. Para tal, o ponteiro deve estar com o endereço de alguma posição válida, seja através de uma atribuição envolvendo um arranjo, seja através do uso de rotinas de alocação dinâmica.

Uma vez que ponteiros são variáveis, nada impede que arranjos de ponteiros sejam definidos. De fato, uma declaração tal como

```
int *aa[10];
```

define uma variável `aa` que é um arranjo de dez ponteiros para variáveis inteiras. Cada elemento deste arranjo, desde `aa[0]` até `aa[9]`, é um ponteiro para inteiro(s) que tem a mesma propriedade que os ponteiros vistos até o momento.

Observe que esta forma suporta uma opção para trabalhar com arranjos multidimensionais, desde que respeitadas as diferenças entre ponteiros e arranjos. Arranjos de ponteiros trazem uma flexibilidade adicional pelo fato de que cada "linha" pode ter tamanho variável. No exemplo acima, cada ponteiro `aa[i]` pode estar apontando para um inteiro, para o primeiro elemento de um arranjo com diversos inteiros, ou mesmo para nenhum inteiro.

C.3.3 Ponteiro como argumento de funções

A passagem por valor, padrão em C, não permite que uma função manipule diretamente uma variável que lhe esteja sendo passada. Quando se deseja manipular diretamente a variável, ponteiros devem ser usados como o recurso de acesso.

Para ilustrar esta condição, imagine como implementar a rotina `swap` que recebe dois argumentos de um mesmo tipo e troca seus valores. (Esta função poderia fazer parte de uma rotina de ordenação de elementos.) Por exemplo, para trocar dois inteiros, algo similar à seguinte função seria desejado:

```
/*
 *  funcao (errada) de troca de inteiros
 */
void swap_err(int el1, int el2) {
    int temp;       /* variavel temporaria */

    temp = el1;
    el1 = el2;
    el2 = temp;
}
```

Entretanto, como observado no comentário inicial, esta função não funciona. Supondo que a função `main` de um programa tente acessar esta rotina, como em

```
main() {
    int a=10,
        b=20;
    void swap_err(int, int);

    printf("a=%d, b=%d\n", a, b);
    swap_err(a, b);
    printf("a=%d, b=%d\n", a, b);
}
```

A saída obtida seria:

```
a= 10, b= 20
a= 10, b= 20
```

Como se observa, a função `swap_err` não realiza a troca de valores das variáveis `a` e `b` de `main`, apesar de sua lógica interna estar correta. O que `swap_err` faz é trocar os valores das *cópias* destas variáveis, que são apenas suas variáveis locais.

A fim de se obter o efeito correto, ponteiros devem ser utilizados como argumentos. Assim, os elementos a serem trocados serão acessados por seus endereços, e seus conteúdos serão efetivamente alterados. Nesta nova versão, a função `swap` é definida como:

```
/*
 *  funcao de troca de inteiros
 */
void swap(int *el1, int *el2) {
    int temp;       /* variavel temporaria */

    temp = *el1;
    *el1 = *el2;
    *el2 = temp;
}
```

A chamada à função deve passar os endereços das variáveis, como em

```
main() {
    int a=10,
        b=20;
    void swap(int *, int *);

    printf("a=%d, b=%d\n", a, b);
    swap(&a, &b);
    printf("a=%d, b=%d\n", a, b);
}
```

A saída obtida neste caso seria:

```
a= 10, b= 20
a= 20, b= 10
```

como desejado.

Outros usos de ponteiros como argumentos incluem funções que devem retornar mais de um valor e a passagem de arranjos para funções. Quando um arranjo é passado para uma função, na verdade o que se passa é o endereço de seu primeiro elemento. Por este motivo, é possível omitir qual a dimensão do arranjo na declaração do tipo do argumento, como foi visto no caso de `argv` (Seção C.4). Quando o argumento é um arranjo multidimensional, apenas a dimensão do primeiro índice pode ser omitida — as demais devem ser fornecidas. Caso contrário, seria impossível saber como acessar corretamente os elementos do arranjo.

C.3.4 Ponteiros e estruturas

Uma vez que variáveis do tipo estrutura são tratadas exatamente da mesma forma que variáveis de tipos básicos, é possível definir variáveis do tipo ponteiro para estruturas, como em

```
struct dados_pessoais *pa;
```

Componentes de uma estrutura podem ser ponteiros para outros tipos de dados ou estruturas. Em algumas situações, pode haver a necessidade de ter como um dos componentes da estrutura um ponteiro para um tipo da própria estrutura. Um exemplo típico é a construção de listas ligadas, compostas por nós onde um dos dados armazenados em cada nó é um ponteiro para o próximo nó. Esta situação seria representada como

```
struct no_lista {
    /* conteudo do no: */
    ...
    /* ponteiro ao proximo no: */
    struct no_lista *proximo;
};
```

Uma forma básica de acesso aos membros de uma estrutura (através do operador `.`) já foi descrita neste capítulo. A outra forma de acesso a membros de estruturas facilita a notação quando ponteiros para estruturas estão envolvidos. Para ilustrar esta forma, suponha que uma função que lê os dados de um aluno no programa acima retorna na verdade um ponteiro para uma estrutura do tipo `dados_pessoais`. Por exemplo,

```
struct dados_pessoais *le_aluno();   /* prototipo */
struct dados_pessoais *aluno_pt;     /* ponteiro */

aluno_pt = le_aluno();
```

O acesso a membros da variável `aluno_pt` poderia ser feito da forma usual, ou seja, `*aluno_pt` é uma estrutura, então seria possível acessar seus membros como em

```
printf("%s nasceu em %2d/%2d/%4d\n",
       (*aluno_pt).nome,
       (*aluno_pt).nascimento.dia,
       (*aluno_pt).nascimento.mes,
       (*aluno_pt).nascimento.ano);
```

A notação simplificada utiliza o ponteiro `->` (seta) para substituir uma construção na forma `(*A).M` por `A->M`. O exemplo original poderia ser reapresentado integrando estes últimos aspectos como

```
int main() {
    struct data hoje;
    struct dados_pessoais *aluno_pt;
    int idade;
    /* prototipos: */
    int calc_idade(struct dados_pessoais, struct data);
    struct data le_hoje();
    struct dados_pessoais *le_aluno();

    /* obtem dados para hoje e aluno */
    hoje = le_hoje();
    aluno_pt = le_aluno();

    idade = calc_idade(*aluno_pt, hoje);

    /* apresenta resultado */
    printf("Idade de %s: %d\n",
           aluno_pt->nome,
           idade);

    return(0);
}
```

C.3.5 Ponteiros para funções

Como foi visto no Capítulo 1, um programa é um conjunto de instruções armazenado na memória, assim como seus dados. Por este motivo, é possível referenciar o *endereço de uma função*. Em C, o endereço de uma função é acessível ao programador através de uma variável do tipo ponteiro para função.

Ponteiros para funções podem ser passados como argumentos para outras funções, e a função apontada pode ser invocada a partir de seu ponteiro. Um exemplo prático desta capacidade é seu uso em uma rotina de

ordenação de elementos de um arranjo. Se o arranjo é de inteiros, então uma função de comparação de inteiros deverá ser suportada, tal como

```
/*
 *  compara dois inteiros, retornando:
 *      0 se os dois elementos forem iguais
 *      um inteiro negativo se o primeiro elemento for menor
 *      um inteiro positivo se o primeiro elemento for maior
 */
 int comp_int(int *e1, int *e2) {
    return(*e1 - *e2);
 }
```

O problema surge quando se deseja usar o mesmo algoritmo de ordenação para ordenar outros arranjos de tipos que não sejam inteiros. Por exemplo, se os elementos a comparar forem *strings*, então a rotina de comparação acima não mais serviria, apesar de todo o restante do algoritmo de ordenação ainda ser basicamente o mesmo.

A solução é passar qual função deve ser usada para a comparação como um dos argumentos para a rotina de ordenação genérica. Esta abordagem é adotada por rotinas usualmente supridas juntamente com o compilador C, tal como `qsort` para ordenação de arranjos e `bsearch` para a realização de busca binária em arranjos ordenados.

A forma de declarar uma variável do tipo ponteiro para função é ilustrada no seguinte exemplo, com uma referência à função `comp_int` definida acima:

```
main() {
    /* prototipo de comp_int: */
    int comp_int(int *, int *);
    /* ponteiro para uma funcao retornando inteiro */
    int (*apcmp)();
    int a, b;

    apcmp = comp_int;       /* inicializa ponteiro */
    ...
    (*apcmp)(a, b);         /* invoca funcao */
}
```

Algumas observações relativas a este exemplo são importantes. A primeira refere-se à declaração do ponteiro. A declaração de um ponteiro para a função deve incluir os parênteses em torno do nome da variável ponteiro. Uma definição na forma `int *apcmp();` seria interpretada como o protótipo de uma função retornando um ponteiro para um inteiro, o que não é o desejado neste caso.

A segunda observação refere-se à forma utilizada para definir o valor do ponteiro no comando de atribuição. Como o protótipo de `comp_int` já havia sido definido, então o compilador sabe que este identificador refere-se a uma função. Quando o identificador `comp_int` é encontrado novamente, desta vez sem parênteses, ele é identificado como o endereço desta função, podendo assim ser atribuído a um ponteiro para uma função com o mesmo tipo de retorno. Repare a semelhança com referências a nomes de arranjos.

Finalmente, a invocação da função através de seu ponteiro: a forma usando o operador de dereferência `(*apcmp)` indica o conteúdo do ponteiro `apcmp`, que é a função (neste caso, `comp_int`). Assim, a última linha no exemplo é apenas uma invocação para a rotina apontada por `apcmp`, e o que vem a seguir de `(*apcmp)` são simplesmente os argumentos para a função. O padrão ANSI também permite que a forma equivalente,

```
apcmp(a, b);
```

seja utilizada. Muitos programadores preferem a forma apresentada no exemplo original para tornar claro que um ponteiro para função está sendo usado, embora internamente não haja diferenças entre a ativação de uma função por seu nome ou através de um ponteiro.

Ponteiros para funções tornam-se interessantes quando o programador não pode determinar qual função deve ser executada em uma dada situação a não ser durante a execução do programa. Em tais casos, o trecho do programa referenciando esta "função variável" pode ser escrito em termos de ativação de uma função através de ponteiros para funções, os quais são corretamente inicializados em tempo de execução.

C.4 Argumentos na linha de comando

Até o momento, a definição da função `main` foi sempre utilizada em sua forma sem parâmetros. Há, no entanto, uma outra forma de definição de `main` que é utilizada para passar para o programa quais argumentos foram dados na linha de comando do sistema operacional. Nesta outra forma, `main` recebe dois parâmetros, como indicado abaixo:

```
int main(int argc, char *argv[]) {
    ...
}
```

O primeiro parâmetro, `argc`, indica o número de *tokens* presente na linha de comando. Por exemplo, uma chamada a um programa de nome `eco` com dois argumentos, como

```
eco um dois
```

faria com que o valor de `argc` passado para a função `main` fosse igual a três.

O segundo parâmetro, `argv`, é um arranjo de *strings*, onde cada elemento do arranjo representa um dos tokens da linha de comando. Assim, no exemplo acima a função `main` receberia as seguintes *strings* nesta variável:

- `argv[0]`: a *string* `"eco"`;
- `argv[1]`: a *string* `"um"`;
- `argv[2]`: a *string* `"dois"`.

Observe que `argv[0]` sempre armazenará o nome do programa sendo executado, enquanto que `argv[i]` armazenará o i-ésimo argumento passado para o programa, para i variando de `1` até `argc-1`.

Um exemplo simplificado de uma rotina que ecoe seus argumentos é apresentado abaixo, ilustrando o uso de `argc` e `argv`:

```
/*
 *  eco.c: repete os argumentos da linha de comando
 */
main(int argc, char *argv[]) {
    int i = 1;

    /* apresenta todos argumentos a partir de argv[1] */
    while (i < argc) {
        printf("%s ", argv[i]);
        ++i;
    }
    /* terminando, passa para proxima linha */
    printf("\n");
}
```

Este exemplo também ilustra uma outra convenção usual em programas C: a definição do valor de retorno de `main`. Em geral, um valor de retorno diferente de 0 servirá para indicar ao sistema operacional (ou a um outro processo que tenha ativado este programa) que alguma condição de erro ocorreu que impediu o programa de completar com sucesso sua execução; o valor de retorno 0 indica um a terminação com sucesso.

C.5 Rotinas para entrada e saída de dados

Rotinas de entrada e saída estão usualmente associadas ao arquivo de cabeçalho `stdio.h`. Em geral, este arquivo deve ser incluído no arquivo fonte usando estas rotinas. Em alguns casos, estas "rotinas" são na verdade macros que estão definidas neste arquivo de cabeçalho.

É importante que se observe que informações sobre estas e outras funções podem ser obtidas a partir da seção 3 do manual *on-line* de Unix (`man` em um ambiente Unix com interação texto, `xman` em um ambiente Xwindows ou opção `help` em um ambiente em computador pessoal). Estas informações incluem tipo de retorno, número e tipos de argumentos, e quais arquivos de cabeçalhos devem ser incluídos para que se possa usar a função.

C.5.1 Interação com dispositivos padrão

Os dispositivos padrão de interação com o usuário são o teclado (dispositivo de entrada padrão) e a tela do monitor (dispositivo de saída padrão). C suporta rotinas para acessar diretamente estes dispositivos.

A rotina básica de entrada de dados lê um caráter do teclado, retornando seu valor ASCII. Esta rotina é

```
#include <stdio.h>
int getchar();
```

A descrição acima deve ser lida da seguinte forma: o arquivo de cabeçalho `stdio.h` tem que ser incluído no arquivo fonte que for usar esta rotina `getchar`, e esta rotina não tem nenhum argumento e retorna um valor inteiro.

O seguinte exemplo ilustra o uso de `getchar`:

```
#include <stdio.h>

main() {
  int ch;

  ch=getchar();
  printf("valor ASCII de %c = %d (hexa %x)\n", ch, ch, ch);
}
```

Este programa, quando executado, irá aguardar que o usuário entre algum caráter via teclado. A saída será uma indicação de qual caráter foi obtido ao longo com seu valor em representação decimal e hexadecimal.

A rotina correspondente a `getchar` para a apresentação de um caráter na tela é `putchar`

```
#include <stdio.h>
int putchar(char);
```

Por exemplo, o programa anterior poderia ser estendido de forma a que um *prompt* (tal como >) indicasse ao usuário que uma entrada de dados é aguardada:

```
#include <stdio.h>
```

```
main() {
  int ch;

  putchar('>');
  ch=getchar();
  printf("valor ASCII de %c = %d (hexa %x)\n", ch, ch, ch);
}
```

É possível entrar uma *string* a partir do teclado usando a rotina `gets`,

```
#include <stdio.h>
char *gets(char *);
```

Esta rotina lê uma *string* do teclado armazenando-o no arranjo de caracteres apontado por seu argumento. A entrada da *string* é terminada pela tecla ENTER, que é substituído internamente pelo caráter NUL. O valor de retorno é o endereço do argumento se tudo correu bem ou o apontador nulo se houve alguma condição de erro. Por exemplo,

```
#include <stdio.h>

main() {
  char strch[60];
  char *pch;

  putchar('>');
  pch=gets(strch);
  if (pch != 0)
    printf("string aceita foi: %s\n", pch);
  else
    printf("Erro na obtencao da string!\n");
}
```

Este programa aceita uma *string* via teclado, armazenando-a no arranjo de caracteres `strch`. Observe que o programador deve reservar o espaço que será utilizado para guardar a *string*. Aceitando a *string*, esta será ecoado para a tela. Caso contrário (por exemplo, a entrada é o caráter de fim de arquivo `^D`), a mensagem de erro será apresentada.

A rotina de saída com formatação de valores, `printf`, já foi apresentada anteriormente. Há também uma rotina para a entrada de dados formatados, que é `scanf`:

```
#include <stdio.h>
int scanf(char *format, lista_enderecos);
```

O primeiro argumento de `scanf` é uma *string* contendo seqüências de conversão de formatos, seguindo o que já foi especificado em `printf`. Caracteres que não sejam parte de uma seqüência de conversão na *string* de formato indica entrada que deve ser ignorada. O valor de retorno é o número de conversões realizadas com sucesso.

Por exemplo, para ler um valor inteiro do teclado diretamente para uma variável inteira, o seguinte programa poderia ser usado:

```
#include <stdio.h>

main() {
```

```
  int  value;

  scanf("%d",&value);
  printf("numero aceito foi: %d\n", value);
}
```

Observe que o *endereço* da variável é o argumento de `scanf`.

C.5.2 Interação com arquivos

Os dispositivos padrão de entrada e saída são apenas casos especiais de *arquivos*.

Os arquivos referentes a teclado e tela são abertos automaticamente pelo sistema quando se inicia a execução de um programa. Eles podem ser acessados através dos manipuladores `stdin` (entrada padrão, teclado) e `stdout` (saída padrão, tela). Além destes dois arquivos, um terceiro arquivo padrão também é iniciado pelo sistema — o arquivo padrão de mensagens de erro, `stderr`, também associado à tela. Por exemplo, para enviar uma mensagem de erro para a saída padrão de erros, a função `fprintf` pode ser utilizada, como em

```
fprintf(stderr, "Mensagem de erro");
```

Essa função tem o mesmo comportamento que a função `printf`[1], recebendo como primeiro argumento uma variável do tipo `FILE *`.

Há rotinas de acesso a arquivos similares àquelas descritas para a interação com teclado e tela. (Na verdade, aquelas rotinas são criadas a partir destas.)

Por exemplo, para obter um caráter de um arquivo pode-se utilizar a rotina `getc`:

```
#include <stdio.h>
int getc(FILE *);
```

Assim, `getchar` é equivalente a `getc(stdin)`. Há também uma rotina `fgetc` que tem o mesmo formato. A diferença entre elas é que `getc` é uma macro enquanto que `fgetc` é uma função.

Observe que o valor de retorno de `getc` é um inteiro. Isto permite testar se o caráter obtido é `EOF` (definido em `stdio.h`), que é um valor que não é armazenável em um tipo `char`. Por exemplo, o seguinte programa para apresentar o conteúdo de um arquivo texto na tela pode não terminar propriamente:

```
/*
 *  Exemplo de programa "type" que pode nao funcionar
 */
#include <stdio.h>

main(int argc, char *argv[]) {
  char  ch;       /* deve ser int! */
  FILE *arq;

  /* testa se numero de argumentos correto */
  if (argc != 2) {
    printf("Uso: %s <nome arquivo>\n",argv[0]);
    return(1);
  }
  /* abre arquivo para leitura */
  arq = fopen(argv[1],"r");
```

[1] Veja a Seção C.1.4.

```
  if (arq == 0) {
    perror(argv[1]);
    return(1);
  }
  /* apresenta arquivo caracter a caracter */
  do {
    ch = getc(arq);
    putchar(ch);
  } while (ch != EOF);

  /* finalizacoes */
  fclose(arq);
  return(0);
}
```

O programa deste exemplo pode não terminar porque `ch`, sendo do tipo `char`, nunca será igual a `EOF`. Mudando-se a declaração de `ch` para `int` resolverá o problema.

Observe também neste exemplo a utilização da rotina de impressão de erros do sistema, `perror`:

```
void perror(char *);
```

Quando algum recurso do sistema operacional (neste caso, o sistema de arquivos) é utilizado e um erro ocorre, esta rotina pode ser utilizada para detectar que tipo de erro ocorreu. O argumento de `perror` é uma *string* que irá preceder a mensagem de erro do sistema. Por exemplo, suponha que o usuário tente acessar um arquivo chamado `abobora` que não existe. Então a abertura de arquivo para leitura irá falhar, e a seguinte mensagem será apresentada:

```
abobora: No such file or directory
```

Para inserir um caráter em um arquivo, a rotina `putc` pode ser usada:

```
#include <stdio.h>
int putc(char, FILE *);
```

Acesso a dados formatados em arquivos pode ser realizado através das rotinas `fprintf` e `fscanf`,

```
#include <stdio.h>
int fprintf(FILE *, char *, ...);
int fscanf(FILE *, char *, ...);
```

A entrada e saída de dados binários são suportadas pelas rotinas `fread` (leitura) e `fwrite` (escrita):

```
#include <stdio.h>
int fread(void *aptr, int tam, int qtde, FILE *arq);
int fwrite(void *aptr, int tam, int qtde, FILE *arq);
```

Estas rotinas suportam a transferência de blocos de `qtde` elementos de tamanho `tam` bytes cada elemento entre o arquivo com manipulador `arq` e a área de memória cujo endereço inicial é `aptr`. O valor retornado é o número de elementos transferidos de fato.

Por exemplo, supondo que um arquivo `teste.dat` contém cem números inteiros armazenados em formato binário (isto é, os padrões de bits da representação inteira de cada número são armazenados no arquivo). O seguinte programa lê estes cem inteiros do arquivo, incrementa cada inteiro caso seu conteúdo seja diferente de 0, e escreve de volta os valores atualizados para o arquivo:

```
    /*
     *  atualiza NELEM valores inteiros em um arquivo NOMARQ
     */
    #include <stdio.h>

    #define NELEM  100
    #define NOMARQ "teste.dat"

    int main() {
        FILE *fp;
        int buffer[NELEM];
        int i, qtde_lida;

        /* abrir arquivo */
        if ((fp = fopen(NOMARQ,"rw")) == 0) {
           perror(NOMARQ);
           return(1);
        }

        /* le dados */
        qtde_lida = fread(buffer, sizeof(int), NELEM, fp);

        /* atualiza */
        for (i=0; i<qtde_lida; ++i)
            if (buffer[i] != 0)
                buffer[i]++;

        /* escreve dados atualizados */
        fwrite(buffer, sizeof(int), qtde_lida, fp);

        /* finalizacoes */
        fclose(fp);
        return(0);
}
```

C.6 Rotinas para interação com o sistema operacional

Uma das formas de interação de um programa C com o sistema operacional é através da invocação da função `system`, também parte da biblioteca padrão:

```
    #include <stdlib.h>
    int system(const char *s);
```

O valor de retorno corresponde ao status de saída na execução do processo que executa o comando, sendo um valor distinto de 0 na ocorrência de problemas.

O exemplo abaixo ilustra o uso de `system`. Este exemplo é uma extensão do exemplo `type` anterior, onde além de apresentar o conteúdo do arquivo o programa também atualiza o atributo com a data do arquivo. Para atualizar este atributo, o comando `touch` do sistema operacional é utilizado. Este comando faz com que a data e hora associadas ao arquivo especificado como argumento sejam a data e hora correntes; se o arquivo

especificado não existe, ele é criado com conteúdo vazio. Neste exemplo, se o arquivo não existe o programa encerra execução sem criar o arquivo, pois o comando `touch` é executado após a tentativa de abrir o arquivo com a função `fopen`.

```
/*
 *   exemplo uso de system: leitura com atualizacao de data
 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  FILE *arq;
  int ch;
  char comando[60];

  /* testa se numero de argumentos correto */
  if (argc != 2) {
    printf("Uso: %s <nome arquivo>\n",argv[0]);
    return(1);
  }

  /* abre arquivo para leitura */
  arq = fopen(argv[1],"r");
  if (arq == 0) {
    perror(argv[1]);
    return(1);
  }

  /* atualiza data do arquivo */
  /*-- prepara comando */
  sprintf(comando, "touch %s", argv[1]);
  /*-- executa */
  system(comando);

  /* apresenta arquivo carater a carater */
  do {
    ch = getc(arq);
    putchar(ch);
  } while (ch != EOF);

  /* finalizacoes */
  fclose(arq);
  return(0);
}
```

A outra forma de interação de um programa C com o sistema operacional é através da invocação de chamadas do sistema *(system calls)*. O sistema operacional oferece um conjunto de serviços acessível através desta interface de rotinas que podem ser diretamente invocadas a partir de programas C.

Em geral, muito da funcionalidade das chamadas do sistema já está coberta em termos de rotinas equivalentes da biblioteca padrão. Por exemplo, é parte desta interface o conjunto de rotinas para manipulação de

arquivos — `open`, `creat`, `close`, `read` e `write`. Estas rotinas são acessadas pelas funções C da biblioteca padrão que suportam a funcionalidade de entrada e saída para terminais e arquivos.

As rotinas do conjunto de chamadas do sistema são em geral extremamente dependentes de cada sistema operacional. No contexto de Unix há um esforço de padronização deste nível de chamadas, que é o padrão POSIX 1003.1 (*Portable Operating System* — o "IX" é uma referência não explícita a Unix). Entretanto, o padrão cobre aspectos básicos, sendo que há diversos grupos — tais como OSF (*Open Software Foundation*, consórcio liderado por IBM, DEC e HP) e UI (*Unix International*, consórcio liderado pela AT&T) — que buscam oferecer suas próprias extensões ao padrão, de forma que não há ainda uma interface única para todos os serviços providos por um sistema operacional.

Se portabilidade é importante para sua aplicação, o uso de chamadas do sistema deve ser evitado e, se realmente necessário, as chamadas utilizadas deveriam ser restritas ao conjunto POSIX.

C.7 O pré-processador C

Há uma série de definições que podem coexistir em diversos módulos de um mesmo programa: valores constantes que devem ser compartilhados, definição de estruturas, definições de nomes de tipos, protótipos de funções, etc. Seria tedioso e muito sujeito a erros se estas definições tivessem de ser inseridas em cada módulo.

A linguagem C oferece mecanismos que permitem manter definições unificadas que são compartilhadas entre diversos arquivos. A base destes mecanismos é o pré-processamento de código, a primeira fase na compilação do programa. Nesta fase, por exemplo, comentários são substituídos por espaços antes do código ser passado para a fase de compilação. Essencialmente, o pré-processador é um processador de macros para uma linguagem de alto nível.

O programador se comunicar com o pré-processador inserindo **diretivas** em um código fonte de forma a facilitar a manutenção do programa. As diretivas para o pré-processador C podem ser reconhecidas pelo símbolo # na primeira coluna da linha onde ocorrem. Estas diretivas não são expressões C, de forma que as linhas onde elas ocorrem não são terminadas por ponto e vírgula.

A diretiva `#include` permite que um arquivo (geralmente com definições e declarações de protótipos) possa ser incluído em um módulo. Por convenção, tais arquivos — chamados de **arquivos de cabeçalho** — recebem a extensão `.h` (de *header*).

A linguagem C oferece um conjunto de rotinas de suporte. Para essas rotinas não é necessário que o usuário entre sempre com os protótipos de cada função que ele irá utilizar: esta informação já está contida em arquivos de cabeçalho usados pelo compilador. Por exemplo, protótipos para rotinas de ordenação e busca estão em um arquivo de cabeçalho do sistema de nome `stdlib.h`. Para usar essas deinições e outras que eventualmente estejam nesse arquivo, o programador inclui no início do arquivo a linha

```
#include <stdlib.h>
```

O nome do arquivo de cabeçalho foi incluído entre `<...>`. Isto indica que este arquivo é um arquivo de cabeçalho fornecido pelo compilador, que será incluído a partir de um diretório padrão do sistema — geralmente o diretório `/usr/include` em um sistema Unix.

Existe uma forma alternativa de inclusão que permite que o programador crie seus próprios arquivos de cabeçalho e os inclua em seus módulos. Esses arquivos deverão ser localizados em algum diretório do usuário. Neste caso, o nome do arquivo de cabeçalho deve ser incluído entre aspas.

O uso da diretiva `#include` facilita a leitura do código C ao abstrair detalhes de definições para uma outra etapa e, principalmente, favorece a coerência entre módulos que devem compartilhar as mesmas definições.

Outra importante diretiva para o pré-processador C é a diretiva `#define`, que permite definir constantes simbólicas e macros que serão substituídas no código fonte durante a compilação. Com o uso desta diretiva torna-se mais simples manter o código envolvendo constantes correto. Ela também simplifica a compreensão do código ao usar nomes simbólicos que indicam o papel de constantes no módulo.

Essa diretiva foi usada na seção anterior para definir os tamanhos dos campos de uma linha de instrução:

```
#define LINSIZE 80
#define LABSIZ 8
#define OPCSIZE 10
#define OPRSIZE 20
#define CMTSIZE (LINSIZE-(LABSIZ+OPCSIZE+OPRSIZE))
```

A vantagem em se usar essas constantes simbólicas é que qualquer modificação nessas definições — por exemplo, mudar o tamanho da linha para 100 caracteres ao invés de 80 — pode ser realizada em um único local. O restante do código permanece inalterado.

Um outro uso da diretiva `#define` é a definição de macro-instruções. Uma macro tem sintaxe de uso similar a uma chamada de função; entretanto, o código da macro é substituído no código fonte durante o pré-processamento. Macros não geram chamadas de funções, não têm variáveis na pilha e não fazem verificação de tipos de argumentos. Em geral, são utilizadas para substituir expressões complexas de forma eficiente.

Considere uma função `max`, que retorna o maior de dois valores passados como argumentos. Esta função tem um código que poderia ser expresso em uma linha com o uso do operador condicional. Entretanto, como **função** ela está restrita ao uso com variáveis inteiras. Para obter o maior valor entre duas variáveis do tipo `double`, outra função deveria ser escrita tendo exatamente o mesmo corpo — apenas o tipo de retorno e tipo dos argumentos seriam modificados.

Uma definição de macro pode simplificar este problema. A forma geral de definição de macros é

```
#define nome_macro(lista_argum)  (corpo_macro)
```

O par de parênteses em torno do corpo da macro não é necessário, mas é usualmente incluído para evitar problemas de mudança de precedência de operadores após a expansão da macro no código fonte.

A macro para obter o máximo de dois valores poderia ser escrita como

```
#define max(a,b)   (a<b ? b : a)
```

e utilizada da mesma forma que funções:

```
int i, j, k;
double x, y, z;
...
k = max(i,j);
z = max(x,y);
```

Após a fase de pré-processamento, o código efetivamente repassado para a fase de compilação seria:

```
int i, j, k;
double x, y, z;
...
k = (i<j ? j : i);
z = (x<y ? y : x);
```

A definição de uma macro pode se estender por mais de uma linha. Nestes casos, cada linha a ser continuada deve ser terminada por uma contrabarra (`\`). Por exemplo, a mesma macro `max` poderia ter sido definida como

```
#define max(a,b) \
        ( a<b ? \
          b : \
          a )
```

O uso da diretiva `#define` também facilita a manutenção de código. Tais diretivas são usualmente incluídas como parte de arquivos de cabeçalho quando suas definições são compartilhadas entre diversos módulos. Por exemplo, a macro `max` exemplificada acima já é geralmente incluída em um arquivo padrão de cabeçalho, `macros.h`.

O pré-processador também entende a diretiva `#undef`, que permite eliminar a definição de um identificador. Por exemplo,

```
#undef TRUE        /* esquece qualquer definicao anterior */
#define TRUE 1     /* nova definicao */
```

Em algumas situações, pode ser interessante incluir ou excluir alguns trechos de código em um programa — por exemplo, para incluir testes e mensagens de depuração durante o desenvolvimento do programa e excluí-los na versão final. Para programas de porte razoável, a manutenção manual deste tipo de trechos de programa pode se tornar uma tarefa complexa. Um mecanismo que pode facilitar esta tarefa é a utilização de diretivas de compilação condicional.

A diretiva básica para a compilação condicional é `#if ... #endif`:

```
#if expr_constante
    /* codigo incluido quando expr_constante != 0 */
    ...
#else
    /* codigo incluido quando expr_constante == 0 */
    ...
#endif
```

O trecho `#else` é opcional, podendo ser omitido. Um exemplo de uso destas diretivas é a verificação se uma constante já foi definida, como em

```
...
x = malloc(n);
#if defined(DEBUG)
    printf("malloc: %d bytes alocados a partir de %p\n",
           n, x);
#endif
...
```

(A seqüência de conversão `%p` apresenta uma variável apontador.) A função `printf` acima será invocada apenas quando o identificador `DEBUG` tiver sido previamente definido, como em

```
#define DEBUG
```

Observe que nem é necessário que um valor seja associado ao identificador neste caso; basta que ele esteja definido. Assim, durante a fase de desenvolvimento a definição acima seria incluída em módulos sendo depurados, sendo posteriormente removida para a geração do programa final.

A forma `#if defined( ... )` ocorre tão frequentemente que há uma forma abreviada de diretiva, `#ifdef`. O exemplo acima poderia ser reescrito como

```
...
x = malloc(n);
#ifdef DEBUG
    printf("malloc: %d bytes alocados a partir de %p\n",
           n, x);
#endif
...
```

Um dos principais usos da compilação condicional é evitar a reinclusão de arquivos de cabeçalho. Em alguns casos, um arquivo de cabeçalho pode já incluir definições de outro arquivo de cabeçalho. A questão é: como evitar erros de redeclaração por causa de uma outra inclusão explícita de um arquivo já incluído implicitamente?

Por exemplo, suponha que a definição da estrutura `linha` estivesse em um arquivo de cabeçalho `montador.h`, por ser uma construção que será compartilhada por vários módulos:

```
/*
 *  montador.h
 */
#define LINSIZE 80
#define LABSIZ 8
#define OPCSIZE 10
#define OPRSIZE 20
#define CMTSIZE (LINSIZE-(LABSIZ+OPCSIZE+OPRSIZE))

typedef struct linha {
    char rotulo[LABSIZ];
    char opcode[OPCSIZ];
    char operand[OPRSIZ];
    char comment[CMTSIZ];
} Linha;
```

A compilação condicional traz a solução para o problema associado à reinclusão de arquivos, gerando erros de redefinição de estruturas. Quando um arquivo de cabeçalho é incluído pela primeira vez, ele pode definir um identificador associado apenas àquele arquivo. Quando se tenta incluir novamente o arquivo de cabeçalho, um teste é realizado — caso o identificador já esteja definido, então o conteúdo do arquivo não é incluído.

No caso acima, o símbolo poderia ser por exemplo `_H_MONTADOR`, e o conteúdo do arquivo seria

```
/*
 *  montador.h
 */
#if ! defined(_H_MONTADOR)
#define _H_MONTADOR

   /* conteudo original aqui */

#endif
```

Além de `#if`, `#ifdef`, `#else` e `#endif`, as diretivas `#ifndef` (se não definido) e `#elif` (else-if) são suportadas.

O pré-processador C oferece ainda diversos outros recursos. O operador # permite substituir na macro a grafia de um argumento. Por exemplo, o programa

```
#define path(prof,curso) \
              "/home/faculty/" #prof "/courses/" #curso

main() {
  printf ("path: %s\n",path(ricarte,progc));
}
```

iria resultar na seguinte saída quando executado:

```
path: /home/faculty/ricarte/courses/progc
```

No exemplo acima, o recurso de concatenação de *strings* adjacentes (outra tarefa desempenhada pelo pré-processador) é utilizado.

Concatenação é suportada através do operador ##. Quando este operador é usado em uma macro entre dois outros símbolos, os símbolos são inicialmente expandidos e então o símbolo do operador e quaisquer espaços em volta dele são eliminados. Por exemplo, a macro

```
#define sport(a)  a ## bol
```

poderia ser usada para criar identificadores em um programa, como

```
int sport(fute), sport(basquete);
 ...
futebol = 1;         /* criado por sport(fute) */
basquetebol = 2;     /* criado por sport(basquete) */
...
```

Há também macros que são pré-definidas e que podem ser usadas em qualquer programa C, que são:

__LINE__ uma constante decimal contendo o número da linha atual no arquivo fonte;

__FILE__ uma *string* com o nome do arquivo fonte que está sendo compilado;

__DATE__ uma *string* com a data da compilação;

__TIME__ uma *string* com a hora (`hh:mm:ss`) da compilação.

Por exemplo, considere o seguinte programa criado em um arquivo de nome `predefin.c`:

```
main() {
  printf ("%s:%d (%s %s)\n",
          __FILE__, __LINE__, __DATE__, __TIME__);
}
```

Após compilado e executado, este programa apresentaria o seguinte resultado:

```
predefin.c:2 (May 17 1995 13:27:29)
```

Outras diretivas do pré-processador incluem:

#error string causa a geração pelo compilador de uma mensagem de erro contendo `string`;

#line constante arquivo indica novos valores para a macro __LINE__ (passa a ter o valor `constante`, que deve ser um número decimal) e, caso `arquivo` esteja presente, para a macro __FILE__;

#pragma string é um mecanismo de comunicação com recursos não padronizados oferecidos pelo compilador, e cujo comportamento depende de cada implementação.

Por exemplo, considere que o arquivo `predefin.c` do exemplo anterior seja modificado como se segue:

```
main() {
#ifdef TST
#  line 100 "arq_teste"
#else
#  error Esqueceu de definir TST!
#endif
  printf ("%s:%d (%s %s)\n",
          __FILE__, __LINE__, __DATE__, __TIME__);
}
```

A linha de comando

```
cc predefin.c -o predefin
```

geraria a seguinte resposta do compilador:

```
"predefin.c", line 5.0: 1506-205 (S) Esqueceu de definir TST!
```

É possível definir um símbolo na linha de comando de compilação através do uso da chave `-D`. A linha de comando

```
cc -DTST predefin.c -o predefin
```

produziria um programa executável `predefin` que quando executado geraria a seguinte mensagem:

```
arq_teste:103 (May 17 1995 14:10:27)
```

Deve ser ressaltado que o uso do pré-processador C não está restrito exclusivamente a programas fonte C, uma vez que ele existe como um programa independente (`cpp`). Assim, outras aplicações podem usar essas mesmas funcionalidades. Por exemplo, o compilador `idltojava` da Sun, que mapeia especificações de interface expressas em *Interface Description Language* (padrão especificado pelo *Object Management Group* para a arquitetura CORBA) para programas na linguagem Java, requer o uso de um pré-processador C para sua operação.

C.8 Exemplo de aplicativo

Além das rotinas da biblioteca padrão e de suporte, há diversos aplicativos que são diretamente suportados pela linguagem C. Estes aplicativos são suportados em geral através de bibliotecas que podem ser ligadas ao código da aplicação, como ocorre com rotinas da biblioteca padrão — a diferença é que neste caso estas bibliotecas de aplicativos devem ser explicitamente indicadas para o compilador (por exemplo, através da chave `-l<lib>` para o comando `cc`, onde `<lib>` é uma indicação para a biblioteca do aplicativo.

Em Unix, por exemplo, há uma biblioteca de rotinas que suporta a funcionalidade básica de um gerenciador de base de dados. Esta biblioteca, `dbm`, inclui rotinas que permitem:

- iniciar (criar ou reabrir) e fechar uma base de dados;
- armazenar registros na base de dados, onde registros têm uma estrutura na forma `[chave, dados]`;
- buscar registros na base de dados, seja a partir de sua chave ou sequencialmente;
- remover um registro da base de dados.

Como pode se observar, `dbm` é uma biblioteca de funções de base de dados. Estas funções são disponíveis para o programador que necessita criar e manipular uma base de dados organizada através de uma função hash.

O uso convencional do `dbm` é através do armazenamento de pares chave/dado em um arquivo de dados. Cada chave deve ser única e associada a somente um item de dado.

Definições necessárias ao gerenciador estão incorporadas no arquivo de cabeçalho `dbm.h`. O tipo de dado `datum` define o componente básico de um registro, sendo este componente uma estrutura C na forma

```
typedef struct {
        char *dptr;
        int   dsize;
} datum;
```

Esta estrutura permite chaves e itens de dado de tamanhos arbitrários. Um registro da base de dados é composto por dois elementos do tipo datum `[chave, dados]`.

As rotinas incorporadas à biblioteca do gerenciador são as seguintes:

dbminit inicia a base de dados.
`Argumentos:` nome da base de dados `(char *)`.
`Retorno:` código de erro `(int)`.

store armazena registro.
`Argumentos:` registro `[chave, dados]` `(datum, datum)`.
`Retorno:` código de erro `(int)`.

fetch busca um registro.
`Argumentos:` elemento com a chave do registro a ser buscado `(datum)`.
`Retorno:` elemento com dados `(datum)`.

delete remove um registro.
`Argumentos:` elemento com a chave do registro a ser removido `(datum)`.
`Retorno:` código de erro `(int)`.

firstkey recupera a chave do primeiro registro armazenado.
`Argumentos:` nenhum.
`Retorno:` elemento com a chave do registro `(datum)`.

nextkey recupera a chave do registro seguinte ao registro dado.
`Argumentos:` elemento com a chave do registro conhecido `(datum)`.
`Retorno:` elemento com a chave do registro seguinte ao registro dado como argumento `(datum)`.

dbmclose fecha a base de dados.
`Argumentos:` nenhum.
`Retorno:` código de erro `(int)`.

Rotinas que retornam um inteiro indicam erro através de valores negativos; sucesso é indicado por um valor de retorno 0. Por outro lado, rotinas que retornam um `datum` indicam erro setando o valor de `dptr` igual a 0 na estrutura retornada.

Para ligar programas C com a biblioteca `dbm`, a chave de compilação `-ldbm` deve ser incluída. Para executar os programas, `dbm` requer que os arquivos correspondente à base de dados já existam. Para criar os arquivos para uma base de dados de nome *xxx*, pode-se utilizar o seguinte comando Unix:
`touch xxx.dir xxx.pag`.

O seguinte exemplo ilustra o uso desta biblioteca para armazenar dados em um arquivo com nomes e telefones:

```
/*
 *  Armazena nome e telefone em um arquivo dbm "agenda"
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dbm.h>           /* define datum (typedef) */

#define ARQDBM "agenda"    /* nome da base de dados */
```

```
int main(int argc, char *argv[])
{
  int erro;
  datum dbnome, dbfone;         /* estruturas para dbm */
  char nome[60], fone[20];      /* buffers para dados */
  char comando[60];             /* buffer para comando */

  /* ENTRADA DE DADOS */
  if (argc > 3) {
    /* numero invalido de argumentos */
    printf("Uso: %s [nome] [fone]\n",argv[0]);
    return(1);
  }
  else {
    if (argc == 1) {
      /* nenhum argumento fornecido */
      printf("\tNome: ");
      gets(nome);
      printf("\tFone: ");
      gets(fone);
    }
    else if (argc == 2) {
      /* nome foi fornecido em argv[1] */
      strcpy(nome,argv[1]);
      printf("\tFone: ");
      gets(fone);
    }
    else {
      /* nome e fone fornecidos */
      strcpy(nome,argv[1]);
      strcpy(fone,argv[2]);
    }
  }

  /* CONVERSAO DOS DADOS PARA ESTRUTURA DATUM */
  /* incluindo o ultimo '\0' nos strings */
  dbnome.dptr = nome;
  dbnome.dsize = strlen(nome)+1;
  dbfone.dptr = fone;
  dbfone.dsize = strlen(fone)+1;

  /* ARMAZENA NA BASE DE DADOS */
  /* -- prepara arquivos internos de dbm */
  sprintf(comando,"touch %s.pag %s.dir", ARQDBM, ARQDBM);
  system(comando);
  /* -- abre base de dados */
  erro = dbminit(ARQDBM);
  if (erro != 0) {
    /* imprime mensagem de erro */
```

```
    perror("Abre agenda");
    return(1);
  }
  /* -- armazena dados na agenda */
  erro = store(dbnome,dbfone);
  if (erro != 0) {
    /* imprime mensagem de erro */
    perror("Armazena em agenda");
    return(1);
  }
  /* -- fecha base de dados */
  erro = dbmclose();
  if (erro != 0) {
    /* imprime mensagem de erro */
    perror("Fecha agenda");
    return(1);
  }

  /* -- tudo certo, saida sem erro */
  return(0);
}
```

C.9 Palavras reservadas em C e C++

asm	auto	break	case
catch	char	class	const
continue	default	delete	do
double	else	enum	extern
float	for	friend	goto
if	inline	int	long
new	operator	private	protected
public	register	return	short
signed	sizeof	static	struct
switch	template	this	throw
try	typedef	union	unsigned
virtual	void	volatile	while

C.10 Precedência de operadores

Na tabela a seguir resume-se a precedência dos operadores da linguagem C, assim como sua associatividade. Operadores em uma mesma linha têm a mesma precedência, e as linhas estão ordenadas em ordem decrescente de precedência.

Operador	**Associatividade**
`( ) [ ] -> .`	esq-dir
`! ~ ++ -- -` (type) `* & sizeof`	dir-esq
`* / %`	esq-dir
`+ -`	esq-dir
`<< >>`	esq-dir
`< <= > >=`	esq-dir
`== !=`	esq-dir
`&`	esq-dir
`^`	esq-dir
`|`	esq-dir
`&&`	esq-dir
`||`	esq-dir
`?:`	dir-esq
`= += -=` etc.	dir-esq
`,`	esq-dir

C.11 Exercícios

C.1 Crie um programa C que imprima na tela a mensagem

```
"C ou nao C, eis a questao!"
```

Edite e compile o arquivo `ser.c` contendo este programa, criando um executável de nome `ser`.

C.2 Analise as mensagens de erro que o compilador C irá apresentar para cada uma das seguintes situações:

(a) tentar gerar um programa executável sem uma função `main`;

(b) incluir uma expressão dentro de uma função onde o terminador `;` tenha sido omitido;

(c) tentar alterar uma variável declarada como `const`;

(d) usar uma variável que não foi declarada;

(e) declarar uma variável que não é usada;

(f) a expressão `y = x;`, onde `x` e `y` são de um mesmo tipo, ocorre sem que o valor de `x` tenha sido definido;

(g) a expressão `y = x;` ocorre, porém `x` é uma variável do tipo `float` enquanto que `y` é do tipo `char`;

(h) uma variável é declarada com um nome inválido;

(i) um bloco em um dos comandos de controle estruturado não foi apropriadamente encerrado (um `}` foi esquecido).

Repita a experiência para distintos níveis de verificação do compilador *(Warning levels)*.

C.3 A função `int rand()` da biblioteca de rotinas padrão retorna um número inteiro aleatório. Use esta rotina para desenvolver um programa que gere e imprima na tela uma seqüência de 80 bits gerados aleatoriamente. Execute o programa diversas vezes e compare as seqüências geradas. O que você pode deduzir do resultado?

C.4 Uma função é *recursiva* quando ela invoca a si própria. Desenvolva uma função recursiva para calcular fatorial $n!$ de um número inteiro n, onde

$$n! = n \times (n-1)!$$

Use esta função em um programa que imprima o fatorial dos primeiros N inteiros, onde N é um parâmetro da sua função. Qual o máximo valor de N que você pode especificar?

C.5 Use a tabela ASCII (Tabela 2.1) para criar funções que recebem um argumento do tipo `char` e:

(a) retorne 1 se um caráter é um dígito decimal (algarismo entre 0 e 9), 0 em caso contrário;

(b) retorne 1 se um caráter é uma letra, 0 se não;

(c) retorne 1 se um caráter é uma letra ou dígito, 0 se não;

(d) retorne 1 se um caráter é uma imprimível (letra, dígito, símbolo, pontuação), 0 se não;

(e) retorne uma letra maiúscula se o argumento for uma letra minúscula, ou o caráter inalterado em caso contrário;

(f) retorne uma letra minúscula se o argumento for uma letra maiúscula, ou o caráter inalterado caso contrário;

(g) retorne o valor inteiro que corresponde a um argumento caráter que é um dígito decimal (por exemplo, recebendo o caráter '9' retorna o inteiro 9).

C.6 Desenvolva um programa `soma` que receba um número arbitrário de inteiros na linha de comando e imprima a soma destes números.

C.7 Desenvolva um programa `opera` que receba como argumentos um símbolo indicando uma operação (`+, -, *, /`) e dois números (inteiros ou reais), retornando o resultado da operação. Que cuidados especiais devem ser tomados na execução desse programa?

C.8 A rotina da biblioteca padrão `char *getenv(char *)` recebe como argumento o nome de uma variável do ambiente e retorna a *string* com o valor atual da variável. Defina um programa `ecoe` que aceita como argumentos variáveis de ambiente do sistema operacional e apresente na saída padrão o seu valor.

C.9 Desenvolva uma função que receba duas *strings* como argumento e retorne 1 se seus conteúdos forem iguais e 0 se forem diferentes. Essa função não pode fazer uso de nenhuma outra função.

C.10 Estenda a função `calcula_idade` de forma a melhorar o cálculo da idade, retornando uma `Data` dizendo qual a idade da pessoa em anos, meses e dias.

C.11 Desenvolva um program que receba três *strings* como argumento, sendo o primeiro um nome, o segundo uma data e o terceiro um caráter M ou F. Estes argumentos são usados para inicializar os dados do exemplo da Seção C.2.6 e imprimir a idade da pessoa no formato da atividade anterior.

C.12 Desenvolva um programa `gerarand` que recebe zero, um ou dois argumentos inteiros (n_1 e n_2) e apresente na saída uma seqüência de n_1 números aleatórios na faixa de 0 a $n_2 - 1$. Se apenas um argumento estiver presente, não há restrições na faixa de valores dos números gerados (o valor retornado por `rand()` pode ser diretamente utilizado). Se nenhum argumento estiver presente, 20 números aleatórios sem restrição de faixa de valores são gerados e apresentados.

C.13 Modifique o programa acima para que os números sejam apresentados em ordem. Utilize a rotina `qsort` do sistema para ordenar os números em ordem crescente.

C.14 Estenda o programa `gerarand` de forma que a saída, além de ordenada, apresente após cada número aleatório um número entre colchetes que indique a ordem em que o número foi gerado. Assim, o primeiro número aleatório gerado estaria associado à `[1]`, o segundo a `[2]` e assim por diante.

Bibliografia

Alfred V. Aho, John E. Hopcroft & Jeffrey D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983.

Alfred A. Aho, Ravi Sethi & Jeffrey D. Ullman: *Compiladores: Princípios, Técnicas e Ferramentas*. Tradução de Daniel de Ariosto Pinto. LTC Editora, 1995.

Alfred V. Aho & Jeffrey D. Ullman: *Principles of Compiler Design*. Addison Wesley, 1977.

Robert Corbett, Richard Stallman & Wilfred Hansen: *bison*. Manual on-line, versão 1.27, Sistema operacional linux, 1999.

Beatriz Mascia Daltrini, Mario Jino & Léo Pini Magalhães: *Introdução a Sistemas de Computação Digital*. Makron Books, 1999.

José Mário De Martino, Wu Shin Ting & Daniel Camilo: *Apostila de EA870 — Laboratório de Computação*. FEEC/UNICAMP, 1996.

John J. Donovan: *Systems Programming*. McGraw-Hill, 1972.

Ellis Horowitz & Sartaj Sahni: *Fundamentals of Data Structures*. Computer Science Press, 1983.

Robin Hunter: *The Design and Construction of Compilers*. John Wiley & Sons, 1981.

Brian W. Kernigham & Dennis M. Ritchie: *C: A Linguagem de Programação*. Tradução de Pedro Sérgio Nicoletti. EDISA-Campus, 1986.

James F. Korsh & Leonard J. Garrett: *Data Structures, Algorithms, and Program Style Using C*. PWS-Kent, 1988.

John R. Levine: *Linkers and Loaders*. Morgan Kaufmann, 2000.

Hongjiu Lu: *ELF: From the Programmer's Perspective*. NYNEX Science & Technology, 1997.

Motorola: *M68000 8-/16-/32-bit Microprocessors: User's Manual*. Prentice-Hall, 1989.

Vern Paxson: ***flex*** *— fast lexical analyzer generator*. Documentação de manual *on-line*, versão 2.5, Sistema operacional linux, 1995.

Thomas P. Skinner: *Assembly Language Programming for the 68000 Family*. John Wiley & Sons, 1988.

Brian C. Smith: *Assemblers, Linkers and Loaders*. Notas de aula, Carnegie-Melon University, 1995.

William Stallings: *Operating Systems, 2nd ed.* Prentice Hall, 1995.

Andrew S. Tanembaum: *Modern Operating Systems*. Prentice-Hall International Editions, 1992.

Wu Shin Ting: *Notas de aula*. FEEC/UNICAMP, 1995.

Jeffrey D. Ullman: *Fundamental Concepts of Programming Systems*. Addison-Wesley, 1976.