

**UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS**

COMPILADORES

PROFA. JULIANA HERBERT

1. COMPILADORES - CONCEITOS BÁSICOS

Um compilador é um programa que traduz um outro programa de uma linguagem (de mais alto nível) para outra (de mais baixo nível). A linguagem original é chamada de fonte e a final de destino. Durante o processo de compilação, vários tipos de erros podem ser identificados, devendo ser notificados ao usuário do compilador. Os compiladores constituem-se em uma categoria de programas tradutores. As outras categorias são apresentadas a seguir:

- montadores (*assemblers*): traduzem programas codificados em linguagem simbólica (*Assembly*) para instruções em linguagem de máquina. Normalmente, a relação de instruções é de uma para uma.
- macro-*assemblers*: traduzem instruções macro, codificadas em linguagem simbólica para um conjunto de instruções na mesma linguagem.
- processadores ou filtros: traduzem instruções codificadas em uma linguagem de alto nível estendida para instruções da linguagem de programação original. Ou seja, a tradução é feita entre duas linguagens de alto nível.
- decompiladores e desmontadores (*disassemblers*): programas que realizam o processo inverso ao normalmente realizado pelos outros tipos de tradutores. A partir do código objeto, é obtido o código simbólico.
- interpretadores: programas que traduzem programas codificados em linguagens de programação de alto nível para um código intermediário, que realiza a execução do algoritmo original, sem traduzi-lo para a linguagem de máquina. Alguns interpretadores analisam um comando fonte cada vez que este deve ser executado, o que consome muito tempo e é raramente utilizado. A forma mais utilizada de interpretação envolve os seguintes processos:
 - o programa fonte é submetido ao tradutor, que gera o código intermediário;
 - o programa intermediário é executado, com os dados de entrada, gerando os resultados.

Os interpretadores geralmente são mais lentos do que os compiladores, já que a execução do código intermediário tem embutido o custo do processamento de uma tradução virtual para código de máquina, cada vez que uma instrução deve ser operada. Entretanto, os interpretadores têm as seguintes vantagens:

- facilitam a implementação de construções complexas de linguagens de programação;
- são implementados de maneira amigável (*user friendly*), já que estão mais próximos do código fonte do que os compiladores;
- são mais adequados para a implementação de novas linguagens de programação para diferentes equipamentos de computação.

A execução de um programa escrito em uma linguagem de alto nível é feita em dois passos:

1. o programa fonte é compilado, sendo gerado o programa objeto;
2. os dados de entrada são submetidos ao programa objeto, que produz os resultados.

Os primeiros compiladores a serem codificados, por volta de 1950, foram considerados programas extremamente complexos de serem construídos. Atualmente, um compilador pode ser construído em poucos dias, caso a linguagem fonte não seja muito complexa. Várias fases da compilação podem ser geradas automaticamente através da utilização de programas geradores (por exemplo, LEX e YACC, para a análise léxica e sintática, respectivamente).

A compilação é formada por duas etapas gerais: análise, onde o programa na linguagem fonte é dividido em partes e uma representação intermediária é criada e síntese, onde o programa na linguagem destino é construído, a partir da representação intermediária. Um exemplo de representação intermediária comumente utilizada é a árvore de sintaxe, exemplificada na Figura 1.1.

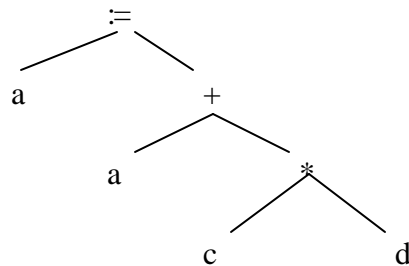


Figura 1.1. Árvore de sintaxe da expressão $a := b + c * d$.

A fase de análise é constituída pelas seguintes subfases (Figura 1.2):

- ANÁLISE LÉXICA: os caracteres que constituem o programa fonte são lidos da esquerda para a direita e agrupados em *tokens* (seqüências de caracteres com um significado coletivo).
- ANÁLISE SINTÁTICA: caracteres e *tokens* são agrupados hierarquicamente em seqüências aninhadas, com significado coletivo.
- ANÁLISE SEMÂNTICA: verificações são realizadas para assegurar que a organização dos componentes do programa é feita de forma adequada e significativa.

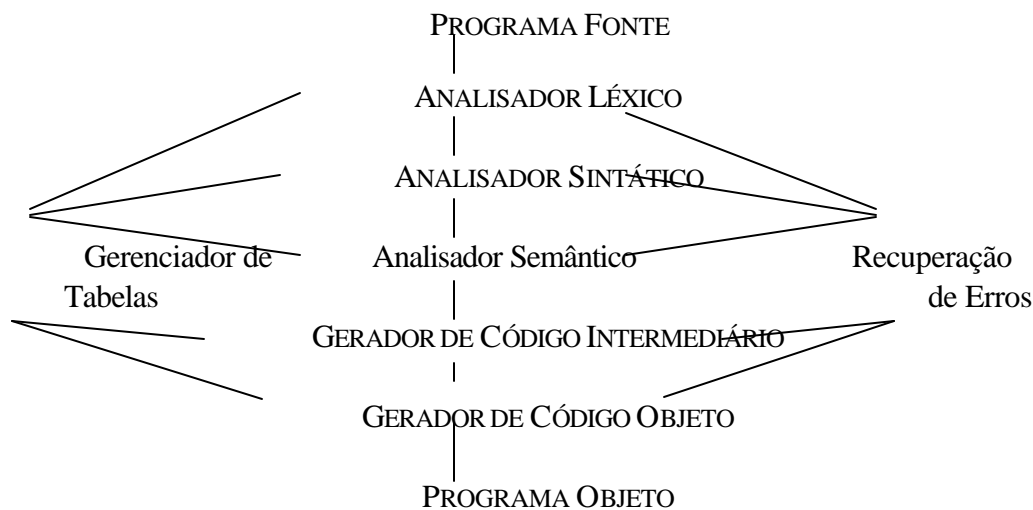


Figura 1.2. Fases de um compilador.

1.1. Análise Léxica

O objetivo da análise léxica é identificar seqüências de caracteres que se constituem em *tokens*. Os *tokens* constituem-se em classes de símbolos, tais como palavras reservadas, delimitadores e operadores, podendo ser representados internamente pelo próprio símbolos. Principalmente no caso de identificadores, que podem ser variáveis declaradas e utilizadas no programa, os *tokens* são representados por pares ordenados, onde o primeiro indica a classe do símbolo (identificador) e o segundo é um apontador para uma entrada na tabela de símbolos.

A análise léxica da expressão "a := b + c * d + e" gera como saída os seguintes *tokens*:

1. o identificador *a*, sendo representado pelo par ordenado (id, 5¹);
2. o operador de atribuição :=;
3. o identificador *b*, (id,6);
4. o operador de adição +;
5. o identificador *c* (id,7);
6. o operador de multiplicação *;
7. o identificador *d* (id,8);
8. o operador de adição +;
9. o identificador *e* (id,9).

Os espaços em branco que separam os *tokens* são geralmente desconsiderados durante a análise léxica.

Normalmente o analisador léxico é uma subrotina do analisador sintático. Durante o reconhecimento das produções sintáticas, cada vez que um *token* deve ser reconhecido, o analisador léxico é chamado, retornando o próprio *token* ou o apontador para a tabela de símbolos.

1.2. Análise Sintática

A etapa de análise sintática é também conhecida como *parsing*. Os *tokens* do programa são agrupados em sentenças da gramática que especifica a linguagem de programação. Uma estrutura hierárquica é normalmente representada através de regras recursivas, tais como:

1. qualquer *identificador* é uma *expressão*;
2. qualquer *número* é uma *expressão*;
3. se *expressão*₁ e *expressão*₂ são expressões, então também o são *expressão*₁+*expressão*₂, *expressão*₁**expressão*₂ e (*expressão*₁).

Erros de sintaxe são detectados nesta fase, podendo-se identificar objetivamente a posição do erro ocorrido e o seu tipo. Os analisadores sintáticos geralmente incluem rotinas de recuperação de erros, que permitem que, mesmo após encontrado um erro sintático, a análise do texto restante continue.

¹ Número hipotético, apontando para uma entrada na tabela de símbolos.

As regras gramaticais que definem as construções da linguagem podem ser descritas através de produções, cujos elementos incluem símbolos terminais (que fazem parte do código fonte) e símbolos não-terminais (que geram outras regras). As seguintes regras definem o comando WHILE, da linguagem Pascal, nas quais as palavras em maiúsculo representam terminais e as palavras em minúsculo os não-terminais.

```
comando      ® comandoWhile
              / comandoIf
              / comandoAtrib,
              / ...
comandoWhile ® WHILE expr_bool DO comando;
expr_bool   ® expr_arit < expr_arit
              / expr_arit > expr_arit
              / ...
expr_arit   ® expr_arit * termo
              / termo
              / ...
termo       ® expr_arit
              / número
              / variável.
```

1.3. Análise Semântica

A análise semântica tem o objetivo de determinar se as estruturas sintáticas fazem sentido. Por exemplo, o seguinte comando IF, sintaticamente correto, pode existir em um programa:

```
if a>7 then b:=5 else b:=10;
```

Entretanto, caso a variável *a* seja do tipo *string*, a comparação realizada com um número não está semanticamente correta. Tal verificação pode ser feita pelo analisador semântico, consultando as informações que constam na tabela de símbolos. Por isso, o analisador semântico geralmente atua juntamente com o analisador sintático.

As próximas fases a serem descritas atuam de maneira sintética: geração de código intermediário, otimização e geração de código objeto.

1.4. Geração de Código Intermediário

A geração do código intermediário é feita a partir da representação interna do programa, produzida pelo analisador sintático, gerando, como saída, uma seqüência de código. Este código é, posteriormente, traduzido para o código objeto. Esta representação intermediária tem as seguintes vantagens:

- possibilita a otimização do código intermediário, a fim de obter código objeto final mais eficiente;
- resolve, de maneira gradual, problemas da passagem do código fonte para objeto.

A geração de código intermediário geralmente é feita juntamente com as fases de análise sintática e semântica. A maior diferença entre o código intermediário e o código objeto é que o intermediário não especifica detalhes de baixo nível de implementação, tais como endereços de memória e registradores, entre outros.

Para o comando:

```
while i < 100 do
  i := j * i;
```

o seguinte código de três endereços (um dos tipos de código intermediário utilizado, no qual cada instrução deve ter, no máximo, três operandos) é gerado:

```
L0: if i < 100 goto L1
      goto L2
L1: temp := j * i
      i := temp
      goto L0
L2: ...
```

1.5. Otimização de Código

O objetivo desta fase é de otimizar o código intermediário em termos de velocidade e de utilização da memória. Por exemplo, o código apresentado na Seção 1.4 poderia ser otimizado da seguinte forma:

```
L0: if i < 100 goto L1
      i := j * i
      goto L0
L1 ...
```

1.6. Geração de Código

Os objetivos desta fase da compilação são: produção de código objeto, alocação de memória para dados e variáveis, geração de código para acessar as posições de memória e seleção de registradores, entre outros. O seguinte código, em linguagem simbólica para o processador 8086, seria gerado a partir do código intermediário da Seção 1.5:

```
L0    MOV AX,I
      CMP AX, 100
      JGE L1
      MOV AX,J
      MOV BX,I
```

```
    IMUL BX
    MOV I,AX
    JMP L0
L1    ...
```

1.7. Gerência de Tabelas

Este processo não é uma fase da compilação, mas compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do compilador. A tabela de símbolos, normalmente utilizada no processo de compilação, armazena as seguintes informações: declarações de variáveis, declarações dos procedimentos ou subrotinas e parâmetros destas subrotinas. A cada ocorrência de um identificador no programa fonte, a tabela é acessada e o identificador é pesquisado. Quando encontrado, as informações associadas a ele são comparadas com as informações obtidas no programa fonte. Caso houverem novas informações, estas são inseridas na tabela. Os seguintes atributos são armazenados, para cada categoria de informações:

- variáveis: classe, tipo, endereço no texto, precisão e tamanho;
- parâmetros formais: classe, tipo, mecanismo de passagem;
- procedimentos e subrotinas: classe e número de parâmetros.

2. ANÁLISE LÉXICA

A análise léxica é a primeira fase da compilação. Nesta fase, o programa fonte é lido, caracter a caracter, de forma a traduzi-lo para uma seqüência de unidades léxicas denominadas *tokens*.

Exemplos de tokens são palavras reservadas (*while, if, for*, da linguagem C, por exemplo), identificadores, operadores aritméticos e lógicos. Estes tokens são separados por delimitadores, que podem ser espaços em branco, ";" e caracteres de nova linha, por exemplo.

Já durante a fase de análise léxica, a tabela de símbolos começa a ser preenchida. As seguintes informações já são conhecidas neste momento: categoria do token, seu lexeme e sua posição no código fonte (linha e coluna).

O fato de o analisador léxico poder ser implementado a partir de uma gramática regular que descreva os tokens, torna-o menos complexo do que o analisador sintático, que é implementado a partir de uma gramática livre do contexto. Desta forma, é mais vantajoso que o compilador possua bem delimitadas as fases de análise léxica e sintática, permitindo também que a automação seja feita de forma mais simples e direta.

Portanto, como resultado da análise léxica, tem-se os tokens com os seguintes atributos associados:

- CLASSE: representa o tipo do token. Por exemplo, palavra reservada.
- VALOR: depende da classe, sendo dividido nas seguintes categorias:
 - **simples**: que não têm valor associado, pois a classe já o descreve completamente. Ex.: palavras reservada, delimitadores, operadores.

- **com argumento:** têm valor associado, como por exemplo, identificadores e constantes.
- **POSIÇÃO:** indica a posição (linha e coluna) do token no programa fonte.

A especificação de um analisador léxico pode ser feita através de uma gramática regular. Uma gramática regular pode ser representada através de um autômato finito, representação esta muito utilizada para a especificação de um analisador léxico.

A implementação de analisadores léxicos é feita, em geral, através de uma tabela de transição, a qual indica a passagem de um estado a outro pela leitura de um determinado caracter. Esta tabela e o correspondente programa de controle podem ser gerados automaticamente com o uso de um gerador de analisadores léxicos, como o LEX, por exemplo.

Outra alternativa para a implementação de um analisador léxico é a construção de um programa que simula o funcionamento do autômato correspondente. Esta alternativa não é aconselhável na construção de compiladores, pois uma linguagem tem um grande número de tokens associados, havendo a necessidade de desenvolvimento de várias rotinas para o seu reconhecimento. Aconselha-se utilizar um gerador de analisadores léxicos ou então o desenvolvimento de rotinas gerais para a simulação genérica de autômatos finitos.

2.1. LEX - Gerador de Analisadores Léxicos

Um gerador de analisadores léxicos como o LEX não é apenas utilizado na construção de compiladores. LEX pode também ser utilizado para resolver problemas relacionados às seguintes categorias, por exemplo:

- verificação e correção ortográfica em editores de texto;
- em rotinas de criptografias, na tradução de códigos.

A especificação do analisador léxico é submetida ao programa LEX, que gera um analisador na linguagem selecionada (geralmente em C, C++ ou Pascal). Este analisador é compilado, gerando o código executável, para o qual é submetida a entrada, que é transformada em uma seqüência de tokens adequadamente identificados e classificados.

2.1.1. Especificação no LEX

Uma especificação no LEX é constituída das seguintes partes:

```
definição                               /* opcional */  
%%  
regras  
%%  
subrotinas                             /* opcional */
```

Exemplo de especificação completa:

```
%{
int cont;
%}
%%
mosca          { cont++; barulho(); }
cachorro       { printf("Au!\n"); }
pessoa         { printf("Hello world! \n"); }
%%
void barulho()
{
    printf("Bzzzzzzz!\n");
}
```

⇒ REGRAS:

Cada regra consiste em um padrão que será pesquisado na entrada, seguido por uma ação que deverá ser feita caso este padrão seja reconhecido. Estes padrões são expressões regulares, que utilizam os seguintes operadores (Tabela 2.1):

Tabela 2.1. Operadores utilizados em regras de uma especificação LEX.

EXEMPLO	EXPLICAÇÃO
<i>palavra</i>	Uma seqüência de caracteres, sem operadores, que reconhecem a palavra " <i>palavra</i> ".
" <i>nova palavra</i> "	Seqüências de caracteres que incluem espaço em branco devem ser especificadas entre aspas.
<i>p</i> *	O operador "*" especifica 0 ou mais ocorrências da expressão (no caso, da letra " <i>p</i> ").
<i>p</i> +	O operador "+" especifica 1 ou mais ocorrências da expressão (no caso, da letra " <i>p</i> ").
<i>p</i> ?	O operador "?" especifica 0 ou 1 ocorrências da expressão.
<i>p...a</i>	O operador "." especifica um único caracter. No exemplo, especifica-se seqüências de caracteres que comecem com "p", contenham quaisquer três caracteres e terminem com "a".
<i>um one</i>	Alternância. No exemplo, uma das seqüências determina o padrão encontrado: "um" ou "one".
<i>(a/b)*c</i>	Os parênteses agrupam expressões. No exemplo, o padrão pesquisa por qualquer número de letras "a" ou "b", seguidas por uma letra "c".
<i>^Primeiro</i>	O operador "^" especifica que o padrão deve ser reconhecido no início da linha.
<i>Último\$</i>	O operador "\$" especifica que o padrão deve ser reconhecido no final da linha.
<i>(cd){3}</i>	O número que segue a expressão, entre chaves, indica o número de vezes que a expressão deve ser repetida no código fonte. Por exemplo, "cdcdcd".
<i>(cd){3,5}</i>	Os números entre chaves especificam um intervalo, no qual o limite inferior é o número

	mínimo de vezes que a expressão deve ser repetida, enquanto que o limite superior é o número máximo.
<code>[abcde]</code>	O padrão é reconhecido por qualquer um dos caracteres entre colchetes.
<code>[^abcde]</code>	O padrão é reconhecido por qualquer caracter, exceto os que aparecem entre colchetes.
<code>[0-9]</code>	O padrão constitui-se em qualquer dígito entre 0 e 9.
<code>[0-9][a-z]</code>	Expressões podem ser concatenadas, através da seqüência. No exemplo, o padrão constitui-se em qualquer dígito entre 0 e 9, seguido de qualquer letra de "a" a "z".

Para o reconhecimento de caracteres especiais, deve-se precedê-los com uma barra invertida ("\"), ou representá-lo entre aspas. Por exemplo: `*` ou `"*"`; `\\` ou `"\"`.

⇒ AÇÕES:

Uma ação é um bloco de código na linguagem de configuração do LEX (por exemplo, na linguagem C), que é executado sempre que o padrão correspondente é reconhecido. Normalmente, estas ações especificam operações tais como transformações do padrão reconhecido, retorno de um token ou coleta de estatísticas.

O comportamento padrão (*default*) das ações é a listagem (impressão no meio de saída especificado) do token reconhecido. Caso deseje-se que um determinado padrão não seja listado, deve-se especificar uma ação vazia, que consiste de alguns espaços em branco seguidos de ponto e vírgula. Por exemplo:

```
%%
palavra      ;
```

Caso queira-se imprimir uma mensagem notificando o usuário que determinada expressão foi encontrada, pode-se utilizar o comando da linguagem C `printf`, por exemplo:

```
%%
"Disciplina de Compiladores I"      printf("Encontrada disciplina!!\n");
```

Para substituir o padrão de entrada por outro, na saída, basta imprimir outra mensagem, na ação:

```
%%
"Disciplina de Compiladores I"      printf("Compiladores I\n");
```

Para contar quantas vezes determinado caracter ou token apareceu no texto de entrada, pode-se utilizar uma variável contadora, como no exemplo a seguir:

```
%{
int cont=0;
}%
%%
```

```
\n    cont++;
```

É importante observar que se mais de um comando deve ser especificado como ação, deve-se representar estes comandos entre chaves.

Quando um padrão é reconhecido, este é armazenado em um array denominado *yytext*, no qual cada posição contém um caracter. O conteúdo deste array pode ser referenciado nas ações, se necessário. A variável *yylen* contém o número de caracteres armazenado em *yytext*.

Exemplo:

```
%{  
int contadorNumeros = 0;  
%}  
%%  
[-+]?[0-9]+          { contadorNumeros++;  
                      printf("%d %s\n", contadorNumeros, yytext); }
```

EXERCÍCIO: Analise a especificação LEX acima e explique o padrão utilizado.

Outro exemplo que utiliza vários conceitos apresentados é listado a seguir:

```
%{  
int contSubrotinas = 0;  
int contPalavrasG = 0;  
%}  
%%  
-[0-9]+          printf("Numero inteiros negativos\n");  
"+"?[0-9]+      printf("Numero inteiro positivo\n");  
-0\.[0-9]+      printf("Numero real negativo, sem parte fracionaria\n");  
inter[ ]+nacional printf("internacional é uma palavra\n");  
function        contSubrotinas++;  
G[a-zA-Z]*      { printf("A seguinte palavra começa com G: %s\n",yytext);  
                contPalavrasG++;}
```

⇒ DEFINIÇÕES:

A seção de definições pode conter as seguintes informações:

- definições externas de variáveis globais;
- comandos do tipo *#include*. Este comando é utilizado quando o gerador de *parsers* YACC é utilizado. O arquivo *ytab.h* é gerado pelo YACC, contendo definições dos tokens;
- abreviaturas para expressões regulares utilizadas na seção das regras, evitando repetições desnecessárias e aumentando a clareza da especificação. Por exemplo, poder-se-ia definir os tokens *identificador* e *digito*:

```
dig          [0-9]
identificador [a-zA-Z][a-zA-Z0-9]*
%%
-{dig}+      printf("Numero inteiro negativo\n");
"+"?{dig}+   printf("Numero inteiro positivo\n");
-0\.{dig}+   printf("Numero real negativo, sem parte fracionaria\n");
{identificador} printf("%s",yytext);
```

⇒ SUBROTINAS:

Subrotinas são utilizadas quando código utilizado em ações pode ser escrito uma vez e utilizado várias vezes.

2.1.2. Tópicos Avançados

- **Regras com ambigüidade:** Nos seguintes casos, duas regras são consideradas ambíguas:

- quando uma seqüência de caracteres de entrada corresponde ao padrão reconhecido por duas regras. Neste caso, a primeira regra será seguida, cuja ação deverá ser executada. Exemplo: a palavra reservada *START* será reconhecida pela primeira regra, e não pela quarta:

```
%{
#include "ytab.h"
%}
%%
START          return(STARTTOK);
BREAK         return(BREAKTOK);
END           return(ENDTOK);
[a-zA-Z][a-zA-Z0-9]* return(yytext);
```

- quando uma seqüência de caracteres de entrada corresponde ao padrão reconhecido por uma regra, mas uma seqüência mais longa que esta também tem seu início que corresponda a esta padrão, então a

seqüência mais longa será reconhecida. Exemplo: quando o comando de entrada "i++" é lido, o operador será reconhecido pela segunda regra, e não pela primeira:

```
%%  
"+"          return(SOMA);  
"++"        return(INCREMENTO);
```

- **Rotinas de entrada e saída:** LEX possui três rotinas para lidar com a entrada e saída de caracteres lidos: *input()*, *unput()* e *output()*. A primeira rotina não tem argumento, enquanto que as outras duas têm como argumento um caracter.

No exemplo a seguir, os comentários em C são ignorados. Depois que o token "/"* é lido, procura-se por um "*". Se o caracter que o segue não é "/", ele é colocado novamente na fita de entrada e a pesquisa continua. Se não, a subrotina é finalizada.

```
%%  
"/*"          skipcmts();  
...  
%%  
skipcmts()  
{  
  for (;;)   
  {  
    while (input() != '*') ;  
    if (input() != '/')  
      { unput(yytext[yytextleng-1]); }  
    else  
      return;  
  }  
}
```

- **Uso do comando REJECT:** O comando REJECT permite que o token sendo lido seja comparado com os outros padrões da especificação. Por exemplo, deseja-se contar o número de ocorrências da palavra "guarda-roupa" e da palavra "roupa":

```
%%  
guarda-roupa { contMoveis++; REJECT; }  
roupa        contRoupas++;
```

- **Processamento do final de arquivo:** A rotina "yywrap()" é utilizada para o processamento de final de arquivo. Normalmente, a rotina retorna 1 se o final de arquivo foi detectado e 0 caso contrário. Pode-se substituí-la por uma rotina definida pelo usuário, implementando alguma função específica para o final de arquivo.

- **Rotina yylex():** O analisador léxico gerado por LEX pode ser chamado pela função "yylex()", que retorna um número inteiro. O gerador de *parsers* YACC faz chamadas a esta função, automaticamente.

Caso o analisador léxico seja utilizado com um *parser*, cada ação do LEX deve terminar com um comando "return", que deve retornar um número inteiro correspondente ao token reconhecido. Caso esteja se utilizando o YACC junto, deve-se incluir o arquivo "ytab.h" na seção das declarações. Caso contrário, recomenda-se que os valores dos tokens seja definido em um outro arquivo, o qual deve ser incluído na seção mencionada. Por exemplo, no arquivo "tokens.h" haveriam as seguintes definições:

```
#define BEGIN 1
#define END 2
#define WHILE 3
...
#define RELOP 12
```

E o arquivo de especificação do LEX seria o seguinte:

```
%{
#include "tokens.h"
extern int tokval;
}%
%%
begin          return(BEGIN);
end            return(END);
while         return(WHILE);
"("           return("(");
")"           return(")");
[a-zA-Z][a-zA-Z0-9]*
               {tokval = put_in_tab();
return(IDENTIFIER); }
>             {tokval = GREATER;
return(RELOP); }

%%
put_in_tab()
{
...
}
```

A função "put_in_tab()" gerencia a tabela de símbolos, devendo ser definida na seção de subrotinas. As seguintes tarefas são por ela realizadas:

- quando um token é reconhecido, "put_in_tab()" recupera seu valor em "yytext" e o compara com os outros tokens armazenados na tabela;
- se o valor é encontrado, o índice correspondente da tabela é retornado; caso contrário, uma nova entrada deve ser criada para ele e um índice é gerado.

3. ANÁLISE SINTÁTICA

A estrutura sintática dos programas é definida através de regras, fornecidas pela linguagem de programação. Por exemplo, em Pascal, um programa é composto por blocos, um bloco por comandos, um comando por expressões e uma expressão por tokens. A sintaxe de uma linguagem de programação é geralmente descrita através de uma gramática livre do contexto ou uma BNF (*Backus-Naur Form*). As seguintes características constituem-se em vantagens da utilização de gramáticas:

- especificação precisa e facilmente compreendida de uma linguagem de programação;
- a partir de determinadas classes de gramática, pode-se construir um analisador sintático eficiente que determina se um programa fonte é sintaticamente correto ou não, revelando também ambigüidades sintáticas;
- uma gramática bem projetada fornece uma estrutura a uma linguagem de programação que pode ser útil para a tradução do programa fonte no código objeto e para a detecção de erros;
- eventuais evoluções que ocorram na linguagem de programação podem ser adicionadas à gramática, adicionando-se regras.

De acordo com o Capítulo 1, o analisador sintático coordena o processo de compilação de um programa. O código fonte é analisado, sendo separado em tokens. O analisador léxico envia tokens ao analisador sintático, que, por sua vez, faz solicitações referentes aos próximos tokens ao analisador léxico, de acordo com o reconhecimento das regras. Um código intermediário é gerado, como resultado da análise sintática (geralmente de ações semânticas a ela associadas). Ambos analisadores acessam periodicamente a tabela de símbolos, fornecendo informações, quando estas são obtidas ou consultando a tabela, conforme necessário.

Os métodos utilizados na análise sintática são classificados como:

- *top-down*: constroem árvores de derivação começando pela raiz, indo depois para as folhas;
- *bottom-up*: constroem árvores de derivação começando pelas folhas, indo depois para a raiz.

3.1. Gerenciamento de Erros

Os erros encontrados em um programa são divididos nas seguintes classes:

- *léxicos*, tais como falta de um caracter em uma palavra reservada ou operador;
- *sintáticos*, tais como expressões aritméticas com número não coincidente de parênteses abertos e fechados;
- *semânticos*, tais como operadores aplicados a operandos incompatíveis;
- *lógicos*, tais como chamadas recursivas infinitas.

Grande parte do processo de detecção e recuperação de erros do compilador está concentrada na fase de análise sintática, já que esta fase analisa uma seqüência de tokens de acordo com regras gramaticais precisas e utiliza métodos precisos de análise. Os seguintes objetivos devem ser seguidos no processo de gerenciamento de erros:

- deve ser reportada a presença de erros claramente e precisamente;
- cada erro deve ser recuperado rapidamente para detectar erros posteriores;
- o desempenho no processamento de programas corretos não deve ser prejudicado.

Vários trabalhos estatísticos já foram realizados, pesquisando ocorrências e tipos de erros sintáticos. Um dos trabalhos, realizado em 1978, por Ripley e Druseikis, com programas desenvolvidos por alunos de graduação, em Pascal, teve os seguintes resultados:

- 60% dos programas compilados estavam sintaticamente e semanticamente corretos;
- 80% dos programas com erros continham apenas um erro;
- 13% dos mesmos programas continham dois erros;
- a maior parte dos erros era trivial de ser descoberto: 60% relacionados à pontuação, 20% à utilização de operadores e operandos, 15% a palavras reservadas e os outros 5% de outros tipos.

O gerenciamento de erros deve ser feito de tal forma que o local aonde o erro foi detectado seja acusado como local do erro, já que há grandes chances que estes dois locais coincidam ou estejam próximos.

Quanto à recuperação do erro, não é aconselhável (nem prático) que o analisador pare quando um erro for detectado. Normalmente, o analisador sintático tenta recuperar-se para um estado aonde a entrada possa continuar sendo analisada. Um problema desta recuperação é o "encobrimento" de outros erros, já que parte do programa acaba não sendo analisada.

A estratégia conservadora faz com que o compilador requeira o processamento de vários tokens com sucesso, antes que outra mensagem de erro seja fornecida. Quando há muitos erros (por exemplo, um programa em C submetido a um compilador Pascal), o analisador tende a parar seu reconhecimento.

Outras estratégias de recuperação de erros são abordadas a seguir:

- **RECUPERAÇÃO PANIC-MODE:** é a estratégia mais simples para ser implementada e pode ser utilizada pela maioria dos métodos de análise sintática. Quando um erro é descoberto, o analisador descarta tokens até que um token de um determinado conjunto de sincronização é encontrado. Os tokens deste conjunto são normalmente delimitadores, ponto e vírgula e *end*. A desvantagem é a desconsideração de vários tokens, e a maior vantagem é a simplicidade de implementação. Em situações aonde erros múltiplos em um comando são raros, o método é bastante adequado.
- **RECUPERAÇÃO BASEADA EM EXPRESSÕES:** quando um erro é descoberto, o analisador realiza uma correção local no resto da entrada, substituindo um prefixo da entrada (que constitui o erro) por outro que permita com que o analisador continue o reconhecimento. Por exemplo, um ponto é substituído por um ponto e vírgula, um ponto e vírgula é inserido, etc. Substituições errôneas podem levar a laços infinitos. Entretanto, esta abordagem é mais "inteligente" do que a abordagem anterior. Esta estratégia apresenta problemas quando o erro ocorreu antes do ponto de detecção.
- **PRODUÇÃO DE ERROS:** se há uma previsão dos erros mais comuns, pode-se aumentar a gramática com a adição de produções que gerem construções errôneas. Quando uma destas produções é utilizada

pelo analisador, diagnósticos do erro e mensagens relacionadas podem ser geradas pelo analisador, quando a entrada é reconhecida.

- **CORREÇÃO GLOBAL:** há algoritmos que selecionam uma seqüência mínima de modificações que devam ser feitas para a recuperação do erro. Dado uma entrada incorreto x e uma gramática G , estes algoritmos determinam uma árvore de derivação para uma entrada relacionada y , tal que o número de inserções, retiradas e modificações de tokens necessário para transformar x em y é tão pequeno quanto possível. Infelizmente, estes métodos são bastante caros de serem implementados.

3.2. Gramáticas Livres do Contexto

Muitas linguagens de programação possuem estruturas que são inerentemente recursivas, podendo ser definidas por gramáticas livres do contexto. Por exemplo:

$$\text{comando} \rightarrow \text{if expressão then comando else comando}$$

Os seguintes conceitos serão utilizados, nas seções subseqüentes deste capítulo:

- **TERMINAIS:** símbolos básicos, tokens. Por exemplo: *if, then, else*.
- **NÃO-TERMINAIS:** variáveis sintáticas que denotam conjuntos de símbolos. Por exemplo, *comando*.
- **SÍMBOLO INICIAL:** símbolo não-terminal pertencente à gramática, que denota a linguagem por ela definida.
- **PRODUÇÕES:** especificam a forma em que os terminais e não-terminais devem ser combinados para formar comandos. Cada produção consiste em um não-terminal, seguido por uma flecha (ou outro símbolo pré-definido), seguida por uma seqüência de terminais e não-terminais.

Exemplo de gramática:

$$\text{expr} \rightarrow \text{expr op expr}$$
$$\text{expr} \rightarrow (\text{expr})$$
$$\text{expr} \rightarrow - \text{expr}$$
$$\text{expr} \rightarrow \text{id}$$
$$\text{op} \rightarrow +$$
$$\text{op} \rightarrow -$$
$$\text{op} \rightarrow *$$
$$\text{op} \rightarrow /$$

As seguintes convenções serão utilizadas:

- símbolos terminais são: letras em minúsculo do alfabeto sozinhas, símbolos de operadores, sinais de pontuação, dígitos e palavras que não representem nenhuma cabeça de produção;
- símbolos não-terminais: letras em maiúsculo, palavras que representem cabeças de produções. O símbolo S é, por convenção, o símbolo inicial da gramática;
- letras gregas em minúsculo representam seqüências gerais da gramática. Por exemplo, uma produção genérica poderia ser escrita como $A \rightarrow a$;

- se $A \rightarrow a_1$, $A \rightarrow a_2, \dots$, $A \rightarrow a_k$ são todas as produções com o não-terminal A na parte esquerda, então pode-se escrever $A \rightarrow a_1/a_2/\dots/a_k$, representando todas as alternativas de A .

Assim sendo, o exemplo apresentado anteriormente pode ser representado também da seguinte forma:

$$E \rightarrow E A E / (E) / - E / id$$

$$A \rightarrow + / - / * / /$$

EXERCÍCIO: Escreva a seqüência de derivação para o reconhecimento de $-(id+id)$, utilizando a gramática apresentada acima, a partir do não-terminal E .

O processo de derivação utilizado no exercício acima é classificado como *top-down*. A árvore de derivação apresenta graficamente a seqüência de derivações seguida.

EXERCÍCIO: Desenhe a árvore de derivação para a derivação realizada no exercício anterior.

Considere agora as duas possíveis seqüências de derivação para a expressão $id+id*id$:

$$E \rightarrow E + E \rightarrow id + E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$$

$$E \rightarrow E * E \rightarrow E + E * E \rightarrow id + E * E \rightarrow id + id * E \rightarrow id + id * id$$

EXERCÍCIO: Desenhe as árvores de derivação correspondentes.

Uma gramática que produz mais de uma árvore de derivação para o reconhecimento de uma mesma entrada é chamada de **ambígua**. Para alguns tipos de analisadores, é interessante que a gramática seja projetada de forma não ambígua, a fim de determinar que produção será seguida para o reconhecimento de determinada entrada. Em outros casos, gramáticas ambíguas podem ser utilizadas, juntamente com regras que retiram esta ambigüidade.

3.2.1. Projeto da Gramática

Nesta seção serão apresentada algumas considerações para que a gramática utilizada seja projetada de forma mais adequada à submissão a um gerador de analisadores sintáticos.

3.2.1.1. Eliminação da Ambigüidade

Como exemplo de gramática ambígua, será utilizada a seguinte, que possui ambigüidade no tratamento da parte *else* do comando *if*:

$$\text{comando} \rightarrow \text{if expr then comando}$$

$$\quad \quad \quad | \text{if expr then comando else comando}$$

$$\quad \quad \quad | \text{outro}$$

A palavra "outro" representa aqui qualquer outro comando. De acordo com a gramática, o comando condicional composto:

if E1 then C1 else if E2 then C2 else C3

tem a árvore de derivação apresentada a seguir:

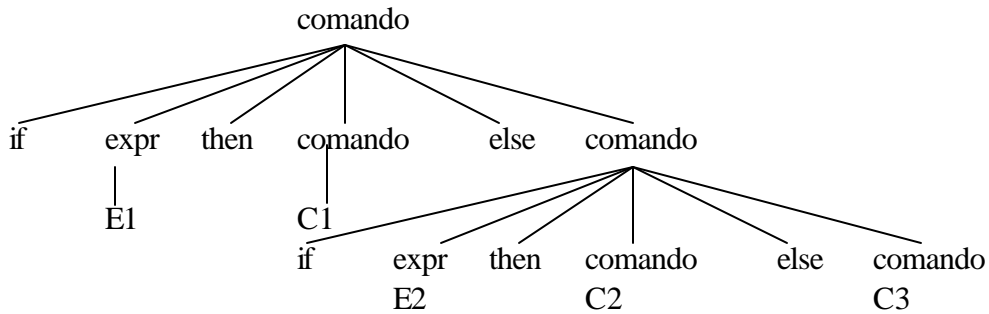
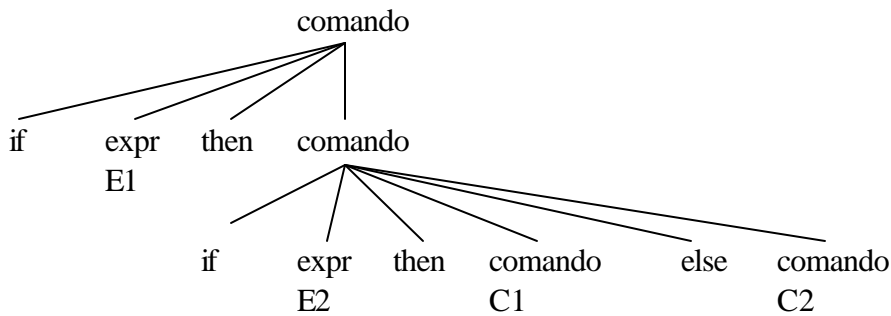


Figura 3.1. Árvore de derivação para o comando condicional.

Enquanto que o comando:

if E1 then if E2 then C1 else C2

tem duas árvores de derivação associadas:



ou:

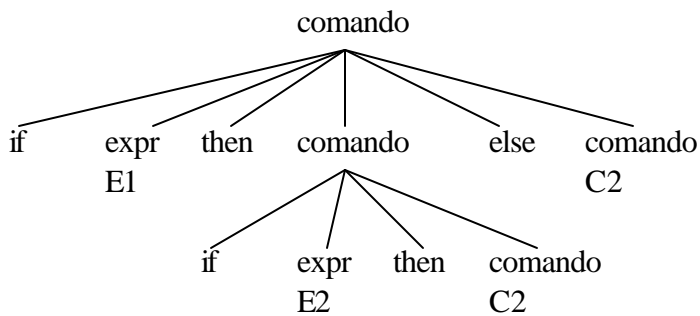


Figura 3.2. Árvores de derivação para gramática ambígua.

Para todas as linguagens de programação existentes, a primeira árvore é utilizada, já que cada "else" deve ser marcado com seu mais próximo "then" anterior. Esta regra pode ser incorporada à gramática, gerando a seguinte outra gramática:

$$\text{comando} \rightarrow \text{comando_n\~{a}o_marcado} \\ | \text{comando_marcado}$$

$$\text{comando_marcado} \rightarrow \text{if expr then comando_marcado else comando_marcado} \\ | \text{outro}$$

$$\text{comando_n\~{a}o_marcado} \rightarrow \text{if expr then comando} \\ | \text{if expr then comando_marcado else comando_n\~{a}o_marcado}$$

3.2.1.2. Eliminação da Recursividade à Esquerda

Uma gramática é recursiva à esquerda quando possui produções com o seguinte formato geral: $A \rightarrow A a$. Alguns métodos *top-down* não aceitam gramáticas deste tipo. Portanto, algumas vezes é necessário eliminar esta recursividade. Por exemplo, no seguinte caso:

$$A \rightarrow A a \\ | b$$

pode-se substituir as produções pelas seguintes, sem recursividade e sem perda de significado:

$$A \rightarrow b A' \\ A' \rightarrow a A' \\ | e$$

Considerando a seguinte gramática para expressões aritméticas:

$$E \rightarrow E + T / T \\ T \rightarrow T * F / F \\ F \rightarrow (E) / id$$

Eliminando-se a recursividade à esquerda, tem-se as seguintes produções:

$$E \rightarrow TE' \\ E' \rightarrow +TE' / e \\ T \rightarrow FT' \\ T' \rightarrow *FT' / e \\ F \rightarrow (E) / id$$

3.2.1.3. Fatoração à Esquerda

Uma gramática fatorada à esquerda deve ser projetada quando deseja-se utilizar um analisador sintático preditivo. A idéia é de que, quando não está claro quais de duas produções alternativas devem ser utilizadas para expandir um não-terminal A, deve-se reescrever as produções para facilitar esta escolha. Por exemplo:

comando \rightarrow *if expr then comando else comando*
 / *if expr then comando*

Nesta regra, o terminal 'if' inicia cada uma das produções, podendo ocasionar dúvida em qual delas deve ser seguida. Generalizando, se $A \rightarrow \mathbf{ab1} \mid \mathbf{ab2}$ são duas produções a partir de A, e a entrada inicia com o token que inicia α , não sabe-se a priori se A será expandido pela primeira ou segunda produção. Fatorada à esquerda, a gramática transforma-se em:

$A \rightarrow \mathbf{aA'}$
 $A' \rightarrow \mathbf{b1} \mid \mathbf{b2}$

3.3. Análise Top-down

Analisadores sintáticos *top-down* constroem árvores de derivação para as entradas, iniciando pela raiz e criando os nodos da árvore em pré-ordem. Dois algoritmos principais de análise são apresentados: o segundo algoritmo a ser apresentado é preditivo, sem processamento *backtracking*, enquanto que o primeiro envolve *backtracking*, fazendo a análise em várias passadas pela entrada.

O *backtracking* é necessário no seguinte tipo de situação: considere a gramática:

$S \rightarrow cAd$
 $A \rightarrow ab$
 / a

e a entrada *cad*. Inicialmente, é construída uma árvore de sintaxe tendo o símbolo S como raiz (Figura 3.3 - a). S é expandido para *cAd*. O primeiro caracter da entrada (c) é reconhecido, sendo necessário então expandir o não-terminal A (Figura 3.3 - b). Com a expansão de A, é reconhecido o segundo caracter a. O ponteiro da entrada é movido para o terceiro caracter (d). No reconhecimento deste caracter, há uma falha, já que tenta-se marcar d com b (da derivação de A). Neste momento, é necessário que se retorne para A, a fim de tentar realizar o reconhecimento através da segunda alternativa de derivação de A (Figura 3.3 - c).

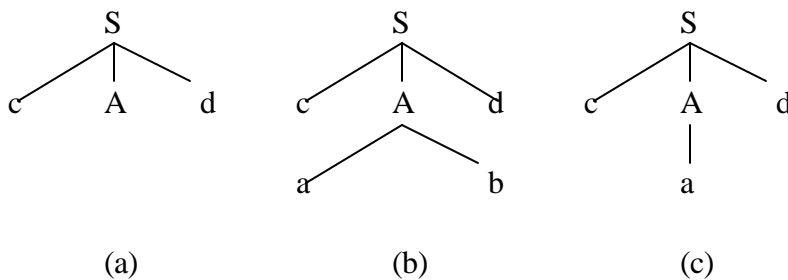


Figura 3.3. Passos de reconhecimento *top-down*.

3.3.1. Analisador Recursivo com *Backtracking*

A gramática G, apresentada abaixo, gera listas:

$$\begin{aligned} S &\textcircled{R} a \\ &| [L] \\ L &\textcircled{R} S ; L \\ &| S \end{aligned}$$

Para o reconhecimento da entrada $[a]$, são apresentadas as etapas de análise descendente na Figura 3.4, representadas na construção da árvore de derivação. Como o primeiro caracter a ser lido é $[$, a primeira produção a ser aplicada é $S \rightarrow [L]$. O reconhecimento de $[$ é bem sucedido e a análise prossegue pela derivação de L , que pode ser feita usando um dos lados direitos alternativos: $S;L$ ou S . Supondo que a primeira alternativa seja seguida e que S seja expandido novamente, obtém-se sucesso. Porém, a comparação seguinte ($] com ;$) falha e o analisador deve retroceder aplicando produções alternativas, na tentativa de encontrar uma derivação mais à esquerda para reproduzir a sentença da fita de entrada. Esta derivação é finalmente obtida, aplicando-se as produções $L \rightarrow S$ e $S \rightarrow a$.

EXERCÍCIO: construir as etapas de construção da árvore de derivação, que devem constar na Figura 3.4.

O esboço de um analisador recursivo com retrocesso é apresentado a seguir. Para cada símbolo não-terminal da gramática foi desenvolvida uma rotina que implementa o reconhecimento das produções alternativas que definem este símbolo. A função *lêToken* retorna um token lido da entrada; *marcaPonto* marca, na sentença de entrada, um ponto de possível reinício da análise; e *retrocede* volta o ponteiro de leitura para o último ponto marcado.

```
Function S;
begin
if token = 'a'
then token = lêToken
return true
else if token = '['
then token = lêToken
if L then if token = ']'
then token = lêToken
return true
else return false;
else return false;
end;
```

```
function L;
begin
```

```

if S then marcaPonto;
    if token = ';'
    then token = lêToken;
        if L then return true
        else return false
    else retrocede;
        return true
    else return false;
end;

```

O processo de voltar atrás no reconhecimento e de tentar produções alternativas denomina-se *backtracking*. Tal processo é ineficiente, pois leva à repetição da leitura de partes da sentença de entrada, não sendo muito utilizado no reconhecimento de linguagens de programação. Outra desvantagem deste tipo de analisador é que, quando ocorre um erro, não é possível determinar o ponto exato onde este ocorreu, devido à tentativa de produções alternativas.

Uma gramática recursiva à esquerda pode gerar um ciclo infinito de expansão de não-terminais. A seguinte gramática, que gera a linguagem {b, ba, baa, baaa, ...} é fornecida como exemplo:

```

A ® Aa
  | b.

```

Um analisador descendente para esta linguagem pode entrar em um ciclo infinito, expandindo repetidamente o não-terminal A.

3.3.2. Analisador Preditivo

Pode-se, muitas vezes, escrever uma gramática sem recursão à esquerda e fatorada à esquerda que gere um analisador que não precisa de *backtracking*, ou seja, que seja um analisador preditivo. Para construir este tipo de analisador, é necessário se ter uma gramática cujas produções derivadas a partir de um mesmo não-terminal tenham seus lados direitos iniciados por diferentes seqüências de caracteres, permitindo que o analisador "saiba" qual das alternativas a seguir, antes mesmo de começar a derivação. Por exemplo, os terminais *if*, *while* e *begin* permitem que a escolha da alternativa seja feita antecipadamente, nas seguintes produções:

```

comando ® if expr then comando else comando
         | while expr do comando
         | begin listaComandos end.

```

DIAGRAMAS DE TRANSIÇÃO

É construído um diagrama de transição para cada não-terminal. Os títulos dos arcos são tokens e não-terminais. Uma transição em um token significa que deve-se seguir tal alternativa caso o token seja

também encontrado na palavra de entrada. Uma transição em um não-terminal significa que este não-terminal deve ser expandido.

Para construir um diagrama de transição de um analisador preditivo, a partir de uma gramática, deve-se eliminar a recursão à esquerda, fatorá-la a direita e para cada não-terminal A , executar os seguintes passos:

1. criar um estado inicial e um estado final;
2. para cada produção $A \rightarrow X_1X_2\dots X_n$, criar um caminho do estado inicial ao estado final, com arcos denominados X_1, X_2, \dots, X_n .

O analisador preditivo trabalha iniciando com o estado inicial do símbolo de entrada. Se, depois de algumas ações, ele está no estado s , com um arco denominado pelo terminal a ao estado t , e se o próximo símbolo de entrada é a , então o analisador move o ponteiro de entrada uma posição à direita e vai para o estado t . Caso contrário, caso o arco seja denominado por um não-terminal A , o analisador vai para o símbolo de entrada de A , sem mover o ponteiro de entrada. Se ele chega ao estado final de A , ele imediatamente vai para o estado t .

3.3.2.1. Analisador Preditivo Recursivo

As seguintes regras definem a função $FIRST(\beta)$, que identifica o conjunto de símbolos terminais que iniciam sentenças deriváveis a partir da forma sentencial β :

1. Se $\beta \Rightarrow \epsilon$, então ϵ é um elemento de $FIRST(\beta)$.
2. Se $\beta \Rightarrow a\delta$, então a é um elemento de $FIRST(\beta)$.

sendo a um símbolo terminal e δ uma forma sentencial qualquer, podendo ser vazia.

Dado um símbolo não-terminal A , definido por várias alternativas que não iniciam por terminais, a implementação de um analisador preditivo recursivo para A exige que os conjuntos $FIRST$ para os não-terminais que iniciam as várias alternativas de produção sejam disjuntos.

A partir da seguinte gramática:

comando \rightarrow *condicional*
 | *iterativo*
 | *atribuição*

condicional \rightarrow *if expr then comando*

iterativo \rightarrow *repeat lista until expr*
 | *while expr do comando*

atribuição \rightarrow *id := expr*

tem-se a seguinte implementação de um analisador preditivo recursivo para *comando*:

```

procedure COMANDO;
begin
  if token = 'if'
    then CONDICIONAL
  else if token = 'while' or 'repeat'
    then ITERATIVO
  else if token = 'id'
    then ATRIBUIÇÃO
  else ERRO
end;

```

EXERCÍCIO: escreva as rotinas CONDICIONAL, ITERATIVO, ATRIBUIÇÃO e ERRO.

Para as produções que derivam a palavra vazia, não é escrito código. Entretanto deve-se observar que esta produção não consiste em um erro.

3.3.2.2. Analisador Preditivo Não Recursivo

Pode-se construir um analisador preditivo sem recursão, mantendo-se uma pilha explicitamente, e não implicitamente, via chamadas não-recursivas. O problema principal desta classe de análise é a determinação da produção a ser aplicada por um não-terminal. Tal decisão é feita a partir da tabela de derivação.

O analisador é constituído por um *buffer* de entrada, uma pilha, uma tabela de derivação e uma saída. O *buffer* de entrada contém a palavra a ser analisada, seguida pelo símbolo \$, utilizado como um marcador que indica o final da entrada. A pilha contém a seqüência de símbolos da gramática com \$ na base, indicando a base da pilha. A tabela de derivação é um vetor bidimensional $M[A,a]$, onde A é um não-terminal e a é um terminal ou o símbolo \$.

O analisador é controlado por um programa que considera X , o símbolo no topo da pilha, e a , o símbolo atual de entrada. Estes dois símbolos determinam a ação do analisador. Há as seguintes três possibilidades:

1. Se $X = a = \$$, o analisador encerra o reconhecimento, com sucesso.
2. Se $X = a \neq \$$, o analisador desempilha X da pilha e avança o ponteiro de entrada ao próximo símbolo de entrada.
3. Se X é um não-terminal, o programa consulta a entrada $M[X,a]$ da tabela de derivação M . Esta entrada pode ser uma produção de X ou um erro. Se, por exemplo, $M[X,a] = \{X @UVW\}$, o analisador substitui X no topo da pilha por WVU (com U no topo). Como saída, assume-se que o analisador apenas imprime a produção utilizada. Se $M[X,a] = erro$, o analisador chama a rotina de recuperação de erro.

O comportamento do analisador pode ser descrito por configurações, que indicam o conteúdo da pilha e o resto da entrada a ser reconhecido.

Considere a gramática já apresentada anteriormente para o reconhecimento de expressões aritméticas:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' / \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' / \epsilon$$

$$F \rightarrow (E) / id$$

Uma tabela de derivação preditiva para esta gramática é apresentada na Tabela 3.1. As células em branco representam erros. Outras células indicam uma produção que deve ser usada para a expansão do não-terminal do topo da pilha. Posteriormente será explicado como estas produções foram selecionadas e classificadas.

Tabela 3.1. Tabela de derivação preditiva para gramática de expressões aritméticas.

NÃO-TERMINAL SÍMBOLO DE ENTRADA	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Para a entrada $id+id*id$, o analisador preditivo realiza a seqüência de passos apresentada na Tabela 3.2. O ponteiro de entrada aponta, inicialmente, para o símbolo mais à esquerda da coluna de entrada. A coluna da saída indica quais produções foram seguidas para a expansão.

Tabela 3.2. Passos realizados pelo analisador preditivo para o reconhecimento da entrada $id+id*id$.

PILHA	ENTRADA	SAÍDA
\$E	id+id*id\$	
\$E'T	id+id*id\$	$E \rightarrow TE'$
\$E'T'F	id+id*id\$	$T \rightarrow FT'$
\$E'T'id	id+id*id\$	$F \rightarrow id$
\$E'T'	+id*id\$	
\$E'	+id*id\$	$T' \rightarrow \epsilon$
\$E'T+	+id*id\$	$E' \rightarrow +TE'$
\$E'T	id*id\$	
\$E'T'F	id*id\$	$T \rightarrow FT'$
\$E'T'id	id*id\$	$F \rightarrow id$
\$E'T'	*id\$	

$\$E'T'F*$	$*id\$$	$T' \rightarrow *FT'$
$\$E'T'F$	$id\$$	
$\$E'T'id$	$id\$$	$F \rightarrow id$
$\$E'T'$	$\$$	
$\$E'$	$\$$	$T' \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

CONSTRUÇÃO DA TABELA DE DERIVAÇÃO PREDITIVA

A tabela de derivação preditiva é feita utilizando duas funções associadas à gramática G : FIRST e FOLLOW. Conjuntos de tokens determinados pela função FOLLOW podem ser utilizados também como tokens de sincronização, durante a recuperação de erros *panic-mode* (Seção 3.1).

Se α é um símbolo da gramática, $FIRST(\alpha)$ é o conjunto dos terminais que iniciam as palavras derivadas por α . Já a função $FOLLOW(A)$, para um não-terminal A , é o conjunto dos terminais a que aparecem imediatamente no lado direito de A , em alguma forma de derivação, mesmo que indiretamente (passando por várias produções). Se A for o símbolo mais à direita de alguma derivação, então $\$$ pertence ao $FOLLOW(A)$.

Para determinar o conjunto $FIRST(X)$, para todos os símbolos X da gramática, devem ser aplicadas as seguintes regras até que não existam mais terminais ou produções vazias possam ser adicionadas a qualquer conjunto FIRST:

1. Se X é um terminal, então $FIRST(X)$ é $\{X\}$.
2. Se $X \rightarrow \epsilon$ é uma produção, então acrescenta ϵ ao $FIRST(X)$.
3. Se X é um não-terminal e $X \rightarrow Y_1Y_2\dots Y_k$ é uma produção, então insira a em $FIRST(X)$ se para algum i , a está em $FIRST(Y_i)$, e insira ϵ , caso este esteja em todos os conjuntos $FIRST(Y_1), \dots, FIRST(Y_{i-1})$; ou seja, Y_1, \dots, Y_{i-1} derivam, mesmo que indiretamente, ϵ . Se ϵ está no conjunto $FIRST(Y_j)$, então ϵ deve ser acrescentado ao conjunto $FIRST(X)$.

Para determinar o conjunto $FOLLOW(A)$, para todos os não-terminais A , as seguintes regras devem ser aplicadas até que não seja possível adicionar mais símbolos ao conjunto:

1. $\$$ pertence ao conjunto $FOLLOW(S)$, onde S é o símbolo inicial da gramática e $\$$ é o marcador de final de entrada.
2. Se há uma produção $A \rightarrow \alpha B \beta$, então todos os terminais de $FIRST(\beta)$, com exceção de ϵ , fazem parte de $FOLLOW(B)$.
3. Se há uma produção $A \rightarrow \alpha B$, ou uma produção $A \rightarrow \alpha B \beta$, onde $FIRST(\beta)$ contém ϵ , então todos os terminais que pertencerem a $FOLLOW(A)$ pertencem também a $FOLLOW(B)$.

Por exemplo, com a gramática de reconhecimento de expressões aritméticas:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid e$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' / \epsilon$$

$$F \rightarrow (E) / id$$

tem-se o estabelecimento das seguintes relações:

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, id \}$$

$$\text{FIRST}(E') = \{ +, \epsilon \}$$

$$\text{FIRST}(T) = \text{FIRST}(F) = \{ (, id \}$$

$$\text{FIRST}(T') = \{ *, \epsilon \}$$

$$\text{FIRST}(F) = \{ (, id \}$$

$$\text{FOLLOW}(E) = \text{FOLLOW}(E') = \{), \$ \}$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ +,), \$ \}$$

$$\text{FOLLOW}(F) = \{ +, *,), \$ \}$$

$\text{FOLLOW}(E)$ contém $\$$ pela regra 1 e $)$ pela regra 2, porque este símbolo segue E na produção $F \rightarrow (E)$.

$\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{ \$,) \}$, pois E' é o último símbolo do lado direito da produção $E \rightarrow TE'$ (regra 3).

A partir da análise da produção $E' \rightarrow +TE'$, $\text{FOLLOW}(T)$ é obtido pela união de $\text{FIRST}(E')$, pela regra 2, com $\text{FOLLOW}(E')$, pela regra 3, pois $E' \rightarrow \epsilon$.

$\text{FOLLOW}(T') = \text{FOLLOW}(T)$, pois T' é o último na produção $T \rightarrow FT'$.

$\text{FOLLOW}(F) = \{ +, *,), \$ \}$ é obtido pela união de $\text{FIRST}(T')$, pela regra 2, com $\text{FOLLOW}(T')$, pela regra 3.

O seguinte algoritmo pode ser seguido para a construção da tabela de análise preditiva para a gramática G . A lógica do algoritmo é a seguinte: suponha que $A \rightarrow \alpha$ é uma produção com a no conjunto $\text{FIRST}(\alpha)$. Então, o analisador expandirá A por α quando o símbolo de entrada for a . Quando $\alpha = \epsilon$, ou α deriva, mesmo que indiretamente, o símbolo ϵ , deve-se expandir novamente A por α se o símbolo de entrada está em $\text{FOLLOW}(A)$, ou se o símbolo $\$$ da entrada for alcançado e $\$$ está no conjunto $\text{FOLLOW}(A)$. M representa a tabela sendo construída.

1. Para cada produção $A \rightarrow \alpha$ da gramática, siga os passos 2 e 3.
2. Para cada terminal a pertencente ao conjunto $\text{FIRST}(\alpha)$, acrescente $A \rightarrow \alpha$ a $M[A,a]$.
3. Se ϵ pertence a $\text{FIRST}(\alpha)$, acrescente $A \rightarrow \alpha$ a $M[A,b]$ para cada terminal b em $\text{FOLLOW}(A)$. Se ϵ pertence a $\text{FIRST}(\alpha)$ e $\$$ pertence a $\text{FOLLOW}(A)$, acrescente $A \rightarrow \alpha$ a $M[A,\$]$.
4. Cada entrada não definida determina um estado de erro.

Aplicando-se o algoritmo à gramática das expressões aritméticas, tem-se as seguintes entradas na tabela de análise preditiva:

$E \rightarrow TE'$	obtem-se	$FIRST(TE') = \{ (, id \}$	$M[E, (] = M[E, id] = E \rightarrow TE'$
$E' \rightarrow +TE'$	obtem-se	$FIRST(+TE') = \{ + \}$	$M[E', +] = E \rightarrow +TE'$
$E' \rightarrow \epsilon$	obtem-se	$FOLLOW(E') = \{ \$,) \}$	$M[E', \$] = M[E',)] = E' \rightarrow \epsilon$
$T \rightarrow FT'$	obtem-se	$FIRST(FT') = \{ (, id \}$	$M[T, (] = M[T, id] = T \rightarrow FT'$
$T' \rightarrow *FT'$	obtem-se	$FIRST(*FT') = \{ * \}$	$M[T', *] = T' \rightarrow *FT'$
$T' \rightarrow \epsilon$	obtem-se	$FOLLOW(T') = \{ +,), \$ \}$	$M[T', +] = M[T',)] = M[T', \$] = T' \rightarrow \epsilon$
$F \rightarrow (E)$	obtem-se	$FIRST((E)) = \{ (\}$	$M[F, (] = F \rightarrow (E)$
$F \rightarrow id$	obtem-se	$FIRST(id) = \{ id \}$	$M[F, id] = F \rightarrow id$

Se em cada entrada da tabela de derivação houver apenas uma produção, então a gramática G que derivou a tabela é do tipo **LL(1)**. Uma gramática LL(1) gera sentenças que podem ser analisadas da esquerda para a direita, produzindo uma derivação mais à esquerda.

Gramáticas ambíguas e recursivas à esquerda têm mais de uma produção para algumas entradas, não pertencendo, portanto, à classe LL(1). A gramática a seguir apresentada é ambígua, pois para a entrada *if b then if b then a else a*, gera duas árvores de derivação distintas. A tabela LL(1) para esta gramática é apresentada na Tabela 3.3.

$S \text{ @ } \textit{if C then S S'}$
 | *a*
 $S \text{ @ } \textit{else S}$
 | *e*
 $C \text{ @ } \textit{b}$

Tabela 3.3. Tabela LL(1) para gramática que reconhece comando de seleção.

	a	b	else	if	then	\$
S	$S \rightarrow a$			$S \rightarrow \textit{if C then S S'}$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow \textit{else S}$			$S' \rightarrow \epsilon$
C		$C \rightarrow b$				

Quando S' estiver no topo da pilha, e *else* sob o ponteiro de leitura, o analisador terá duas opções: desempilhar S' apenas ou desempilhar S' e empilhar *else S*. A primeira significa o reconhecimento do comando *if-then* e a segunda o reconhecimento de *if-then-else*.

Como a maioria das linguagens de programação que implementam comandos de seleção optam por associar a cláusula *else* à cláusula *then* mais próxima, pode-se eliminar a produção $S' \rightarrow \epsilon$ da entrada [S',else] da tabela. Os passos apresentados na Tabela 3.4 são seguidos pelo analisador para o reconhecimento da entrada *if b then if b then a else a*.

Uma gramática G é LL(1) se e somente se, sempre que $A \rightarrow \alpha$ e $A \rightarrow \beta$, que são produções de G, ocorre que:

1. A interseção dos conjuntos $FIRST(\alpha)$ e $FIRST(\beta)$ é vazia.

2. Se $\beta \rightarrow \epsilon$, então a interseção de $FIRST(\alpha)$ com $FOLLOW(A)$ é vazia.

Tabela 3.4. Reconhecimento do comando de seleção.

PILHA	ENTRADA	DERIVAÇÃO
$\$S$	if b then if b then a else a\$	$S \rightarrow \text{if } C \text{ then } S S'$
$\$S' S \text{ then } C \text{ if}$	if b then if b then a else a\$	
$\$S' S \text{ then } C$	b then if b then a else a\$	$C \rightarrow b$
$\$S' S \text{ then } b$	b then if b then a else a\$	
$\$S' S \text{ then}$	then if b then a else a\$	
$\$S' S$	if b then a else a\$	$S \rightarrow \text{if } C \text{ then } S S'$
$\$S' S \text{ then } C \text{ if}$	if b then a else a\$	
$\$S' S \text{ then } C$	b then a else a\$	$C \rightarrow b$
$\$S' S \text{ then } b$	b then a else a\$	
$\$S' S \text{ then}$	then a else a\$	
$\$S' S'$	a else a\$	$S \rightarrow a$
$\$S' S' a$	a else a\$	
$\$S' S'$	else a\$	$S' \rightarrow \text{else } a$
$\$S' S' \text{ else}$	else a\$	
$\$S' S'$	a\$	$S \rightarrow a$
$\$S' a$	a\$	
$\$S'$	\$	$S' \rightarrow \epsilon$
$\$$	\$	

3.4. Análise *Bottom-up*

A análise *bottom-up* é também conhecida com análise sintática *shift-reduce*. O objetivo deste tipo de análise é construir uma árvore de derivação para uma entrada, começando pelas folhas e chegando à raiz. Ou seja, uma entrada é reduzida ao símbolo inicial da gramática. A cada passo de redução, uma parte da entrada que esteja de acordo com a parte direita de uma produção da gramática é substituída pela parte esquerda, da mesma produção. Por exemplo, com a gramática:

$S \rightarrow aABe$

$A \rightarrow Abc$

$\quad \quad \quad | b$

$B \rightarrow d$

A entrada *abcde* pode ser reduzida para *S* de acordo com os seguintes passos:

abcde

aAbcde

aAde

aABe

S

De forma reversa, a seguinte derivação é equivalente à redução realizada:

$S \text{ @ } aABe \text{ @ } aAde \text{ @ } aAbcde \text{ @ } abcde$

Com a gramática:

$E \text{ @ } E + E$

$E \text{ @ } E * E$

$E \text{ @ } (E)$

$E \text{ @ } id$

e a entrada $id_1 + id_2 * id_3$, tem-se a seqüência de reduções apresentada na Tabela 3.5, a fim de transformar a entrada no símbolo inicial E da gramática.

Tabela 3.5. Reduções feitas por um analisador *shift-reduce*.

FORMA SENTENCIAL À DIREITA	HANDLE	PRODUÇÃO PARA REDUÇÃO
$id_1 + id_2 * id_3$	id_1	$E \rightarrow id$
$E + id_2 * id_3$	id_2	$E \rightarrow id$
$E + E * id_3$	id_3	$E \rightarrow id$
$E + E * E$	$E * E$	$E \rightarrow E * E$
$E + E$	$E + E$	$E \rightarrow E + E$
E		

Há dois problemas principais que devem ser resolvidos no processamento *shift-reduce*:

- localização da parte da palavra de entrada que deve ser reduzida;
- determinação de que produção deve ser escolhida, caso haja mais do que uma produção com a parte da palavra de entrada do lado direito.

Uma maneira de implementar um analisador *shift-reduce* é usar uma pilha que armazene os símbolos de gramática e um *buffer* de entrada que armazene a palavra w de entrada a ser reconhecida. Usa-se o símbolo \$ para marcar a base da pilha e o lado direito da entrada. Inicialmente, a pilha está vazia e a palavra w está na entrada. O analisador funciona *shifting* zero ou mais símbolos na pilha, até que o *handle* β esteja no topo. O analisador reduz β para o lado esquerdo da produção, repetindo este ciclo até que seja detectado um erro ou até que a pilha contenha o símbolo inicial da gramática e a entrada esteja vazia. Um exemplo é apresentado na Tabela 3.6.

Tabela 3.6. Configurações de um analisador *shift-reduce* para o reconhecimento da sentença $id_1 + id_2 * id_3$.

PILHA	ENTRADA	AÇÃO
-------	---------	------

(1)	\$	$id_1 + id_2 * id_3 \$$	shift
(2)	$\$id_1$	$+ id_2 * id_3 \$$	reduce por $E \rightarrow id$
(3)	$\$E$	$+ id_2 * id_3 \$$	shift
(4)	$\$E +$	$id_2 * id_3 \$$	shift
(5)	$\$E + id_2$	$* id_3 \$$	reduce por $E \rightarrow id$
(6)	$\$E + E$	$* id_3 \$$	shift
(7)	$\$E + E *$	$id_3 \$$	shift
(8)	$\$E + E * id_3$	$\$$	reduce por $E \rightarrow id$
(9)	$\$E + E * E$	$\$$	reduce por $E \rightarrow E * E$
(10)	$\$E + E$	$\$$	reduce por $E \rightarrow E + E$
(11)	$\$E$	$\$$	accept

A seguir são apresentadas os quatro tipos de ações que podem ser realizadas por um analisador *shift-reduce*:

1. *shift*: o próximo símbolo de entrada é colocado no topo da pilha.
2. *reduce*: o analisador reconhece o lado direito do *handle* que está no topo da pilha, devendo então pesquisar o lado esquerdo e decidir que não-terminal será utilizado para substituí-lo.
3. *accept*: o analisador identifica um estado de final de análise, com sucesso.
4. *error*: o analisador identifica um erro de sintaxe e chama (se houver) uma rotina de recuperação de erro.

3.4.1. Conflitos na Análise *Shift-Reduce*

Em algumas gramáticas livres do contexto podem ocorrer situações em que o analisador "entra em conflito". Estas situações são classificadas em dois tipos:

- conflito *shift-reduce*, , no qual o analisador não sabe se deve realizar uma ação *shift* ou *reduce*;
- conflito *reduce-reduce*, no qual o analisador não consegue decidir que redução deve ser feita, já que várias possibilidades são adequadas.

Por exemplo, considerando a seguinte gramática:

comando \textcircled{R} *if expr then comando*
 | *if expr then comando else comando*
 | *outro*

tem-se o conflito *shift-reduce* na configuração:

Pilha: ... *if expr then comando*
 Entrada: *else ...*\$

Normalmente, em um conflito *shift-reduce*, o analisador opta pela ação *shift*.

A seguir é apresentado outro exemplo de conflito, no qual tem-se um analisador léxico que retorna o token *id* para todos os identificadores, independentemente de seu uso. Os identificadores, nesta linguagem, podem ser utilizados tanto como nomes de rotinas, seguidos por parênteses, como nomes de *arrays*, seguidos por parênteses também. Na gramática, há duas regras distintas para estes casos:

comando @ *id*(*listaParâmetros*)
| *expr* := *expr*

listaParâmetros @ *listaParâmetros*, *parâmetro*
| *parâmetro*

parâmetro @ *id*

expr @ *id*(*listaExpr*)
| *id*

listaExpr @ *listaExpr*, *expr*
| *expr*

Supondo que um comando iniciando com *A(I,J)* apareça na entrada, representado por *id(id,id)*. Após ser acionada a ação *shift* três vezes, a seguinte configuração é atingida:

Pilha: ...*id*(*id*
Entrada: ,*id*)...

O símbolo *id* no topo da pilha deve ser reduzido, mas há duas possibilidades. Caso *A* seja um nome de rotina, deve-se reduzir pela produção *parâmetro*. Caso seja um nome de *array*, deve-se reduzir pela produção *expr*. Desta forma, tem-se um conflito do tipo *reduce-reduce*. A solução mais simples para este conflito é mudar o analisador léxico, configurando-o para retornar dois tipos de identificadores diferentes, dependendo do contexto em que este é analisado. Para tal solução, deve-se utilizar a tabela de símbolos, na qual constará informações sobre os tipos das variáveis.

3.4.2. YACC - Gerador de Analisadores Sintáticos

YACC (*Yet Another Compiler Compiler*) é um gerador de analisadores sintáticos, cuja primeira versão faz parte do conjunto de ferramentas de programação do sistema operacional Unix. Esta primeira versão gera código na linguagem C, havendo outros geradores desenvolvidos a partir desta versão, tais como o MKS Lex & Yacc, para DOS e Windows, que gera código nas linguagens C, C++ e Pascal. Os exemplos apresentados nesta seção da apostila serão apresentados na linguagem C, sendo compatíveis tanto para a versão para Unix quanto para DOS e Windows.

Como entradas, Yacc recebe:

- uma gramática, descrevendo a estrutura sintática da linguagem a ser reconhecida;
- ações semânticas, associadas às regras da gramática;
- declarações auxiliares e subrotinas.

Yacc faz chamadas à rotina *yylex()*, que pode ser gerada pelo gerador Lex, apresentado na Seção 2.1 desta apostila. *yylex()* retorna códigos de tokens reconhecidos, que são combinados em seqüências que podem ou não ser reconhecidas por regras da gramática. O seguinte exemplo apresenta uma regra para o reconhecimento de datas. O sinal ":" separa o não-terminal gerador (lado esquerdo) da produção gerada (lado direito). O sinal ";" marca o final da regra.

```
data : dia '/' mês '/' ano;
```

Um arquivo de especificação submetido ao Yacc tem o seguinte formato:

```
declarações
%%
regras
%%
subrotinas
```

O seguinte exemplo lê datas no formato utilizado normalmente no Brasil (dia/mês/ano), convertendo-as para o formato americano (mês/dia/ano):

```
%union
{
char *texto;
int valor;
}
%token DIA
%token MÊS
%token ANO
%%
data : DIA '/' MÊS '/' ANO          { escreveData($3,$1,$5); } ;
%%
void escreveData(m,d,a);
char *m;
int d, a;
{
printf("%m/%d/%a", m,d,a);
}
```

Na seção de declarações, é declarada uma estrutura *union*, utilizada para armazenar os valores associados aos tokens, além de nomes associados a estes tokens, nas regras da gramática.

Espaços em branco, tabulações e caracteres de formatação são ignorados em especificações Yacc, não podendo aparecer em nomes ou em palavras reservadas. Comentários podem ser acrescentados, desde que se obedeça a sintaxe definida pela linguagem na qual o analisador será gerado (por exemplo, na linguagem C, comentários devem ser apresentados entre `/*` e `*/`).

⇒ REGRAS:

Várias alternativas geradas a partir de um mesmo não-terminal podem ser representadas pela barra vertical (|), como no seguinte exemplo. É importante observar também a notação de produção vazia, apresentada na última alternativa do exemplo que segue:

```
A : BCD
   / EF
   / ;
```

Uma constante alfanumérica deve ser apresentada entre aspas simples ('). A barra invertida (\) é utilizada como um caracter de *escape*, como na linguagem C. Os seguintes *escapes* são reconhecidos pelo Yacc (Tabela 3.7):

Tabela 3.7. Caracteres de *escape* reconhecidos pelo Yacc.

CARACTER	SIGNIFICADO
\n	Nova linha.
\r	<i>Return.</i>
'	Aspa simples.
\\	Barra invertida.
\t	Tabulação.
\b	<i>Backspace.</i>
\f	<i>Form feed.</i>
\nmn	Um caracter em notação octal.

⇒ AÇÕES:

As ações são codificadas na linguagem na qual o gerador será gerado. Em C, por exemplo, são delimitadas por "{" e "}". Quando uma regra é reconhecida, a ação a ela associada é executada. Tais ações podem ser inseridas em qualquer lugar da regra, incluindo antes do primeiro símbolo. Entretanto, na maioria dos casos, elas apareçam após o último símbolo.

Cada símbolo em uma regra tem um valor que pode ser referenciado nas ações. O símbolo do lado esquerdo é denominado "\$\$", representando o valor retornado pela ação, enquanto que os símbolos

do lado direito iniciam em "\$1", sendo incrementados seqüencialmente. No seguinte exemplo, a ação faz com que o valor retornado pela ação seja o valor da expressão, ignorando-se os parênteses. Como ação *default*, \$\$ retorna o valor correspondente ao símbolo \$1.

```
Expr : '(' expr ')' { $$ = $2 } ;
```

O exemplo a seguir ilustra o uso de ações no meio das regras. Ele atribui o valor 1 à variável *x* e o valor retornado por *C* à variável *y*:

```
A : B { $$ = 1; }  
  C { x = $2; y = $3; };
```

Uma ação associada ao símbolo gerador (lado esquerdo) pode referenciar valores associados com símbolos que ocorreram antes deste não-terminal. Estes valores são referenciados como valores relativos ao contexto da esquerda, pois são associados a símbolos que ocorrem à esquerda do não-terminal considerado, em uma produção na qual ele apareça no lado direito. Por exemplo:

```
%token ANO DIA MÊS  
%union  
{  
  char *text;  
  int ival;  
};  
%%  
data : ano dia mês;  
  
mês : MÊS { if (!strcmp($1, "Fevereiro"))  
           if ($0==29) && ($-1)%4!=0)  
           printf("Muitos dias para fevereiro! Ano não é bissexto! \n"); };  
  
dia : DIA { $$ = $1; };  
  
ano : ANO { $$ = $1; };
```

A ação associada ao símbolo “mês” verifica quando uma data contém 29 de fevereiro, em um ano que não é bissexto. Para isso, além do mês, é necessário conhecer o dia e o ano associados à data. Como estes valores aparecem no lado esquerdo de “mês” na regra “data”, pode-se acessar seus valores através dos símbolos \$0 e \$-1 (formato geral: \$-*n*).

⇒ DECLARAÇÕES:

Cada token referenciado nas regras deve ser declarado. A maneira mais comum de declarar um token é utilizando a palavra reservada *%token*. Cada nome que aparece após é declarado como um token.

%token nome1 nome2 ...

Tokens podem ser declarados também através das palavras reservadas *%left*, *%right* e *%nonassoc*, caso se queira definir sua precedência.

O símbolo não-terminal considerado como o símbolo inicial da gramática deve ser declarado com a palavra reservada *%start*.

Código em C pode ser utilizado para declarações de variáveis, que têm então escopo global. Comandos do tipo *#include* são também comumente utilizados. Deve-se evitar a declaração de variáveis iniciando com “yy”, já que estes nomes são geralmente utilizados pelo Yacc.

⇒ SUBROTINAS:

Esta seção contém rotinas definidas pelo usuário, normalmente utilizadas nas ações da seção das regras. Outra alternativa para incluir subrotinas é utilizar a diretiva *#include*.

⇒ ANÁLISE LÉXICA:

O analisador sintático gerado faz chamadas à rotina *yylex()*, que retorna os tokens lidos da entrada. Tais tokens são identificados através de códigos, que devem ser conhecidos tanto nas fases da análise léxica quanto da análise sintática. A rotina *yylex()* retorna um valor inteiro, que representa o tipo de token lido. Este valor é armazenado na variável *yylval*, sendo modificado a medida em que novos tokens vão sendo lidos.

Os códigos numéricos associados aos tokens são definidos pelo Yacc, por *default*. Estas definições constam no arquivo “*ytab.h*”, que deve ser incluído através do comando *#include* no arquivo que contém a especificação léxica. O valor “0” ou um número negativo é retornado quando é identificado o final de arquivo.

Deve-se evitar usar como nomes de tokens palavras reservadas da linguagem C, ou a palavra *error*, que é utilizada pelo Yacc para gerenciamento de erros.

⇒ EXECUÇÃO DO ANALISADOR SINTÁTICO GERADO:

O arquivo gerado pelo Yacc é denominado “*ytab.c*”, contendo a função denominada *yyparse()*, que retorna um valor inteiro. Quando *yyparse()* é chamado, a função *yylex()* é chamada repetidas vezes, a

fim de reconhecer os tokens da entrada. Quando o analisador reconhece a entrada, retorna o valor 0. Caso contrário, quando um erro ocorre, retorna o valor 1.

A função *main()* deve chamar *yyparse()*, e deve ser definida uma outra função denominada *yyerror()*, que imprime mensagens de erro quando erros sintáticos são detectados. Exemplos:

```
main()
{
    return(yyparse());
}

#include <stdio.h>

yyerror(s)
char *s;
{
    (void) fprintf(stderr, "%s\n", s);
}
```

O argumento da função *yyerror(s)* é uma mensagem de erro, normalmente com o valor “*syntax error!*”.

⇒ OPERAÇÕES DO ANALISADOR:

O analisador gerado pelo Yacc consiste de uma máquina de estados finitos com uma pilha de estados. O analisador tem a capacidade de ler e lembrar o próximo token de entrada, denominado *look-ahead token*. O estado atual está sempre no topo da pilha, sendo representado por um número inteiro. Além da pilha de estados, uma pilha de valores armazena os valores retornados pelo analisador léxico e as ações. Inicialmente a máquina está no estado 0 e não existe nenhum token *look-ahead*.

A máquina tem disponíveis cinco tipos de ações: *shift*, *reduce*, *accept*, *error* e *goto*. A ação *goto* é sempre realizada como um componente da ação *reduce*. O analisador opera da seguinte forma:

- baseado no estado atual, o analisador decide se precisa conhecer o token *look-ahead* para escolher a ação a ser realizada. Se necessário, a função *yylex()* é chamada para obter o próximo token;
- usando o estado atual e o token *look-ahead* se necessário, o analisador decide sua próxima ação. Esta ação pode empilhar e/ou desempilhar estados da pilha.

Quando Yacc é invocado com a opção *-v*, um arquivo denominado *y.out* é gerado com uma descrição da operação do analisador. A seguir cada ação é explicada.

Ação SHIFT:

Ocorre sempre que um token é reconhecido, sempre havendo um token *look-ahead*. No arquivo “*y.out*”, uma ação deste tipo é representada da seguinte forma:

LOOP shift 34

Ou seja, supondo que o estado atual seja 56, o token *look-ahead* seja LOOP, o estado atual (56) é empilhado e o estado 34 passa a ser o estado atual, sendo também empilhado. O token *look-ahead* é inicializado e a variável *yyval* é empilhada na pilha de valores.

Ação REDUCE:

Uma ação *reduce* ocorre quando o analisador reconhece determinado lado direito de uma produção. Neste momento, todos os estados empilhados durante o reconhecimento deste lado direito da produção são desempilhados. No topo da pilha fica o símbolo que estava sendo analisado anteriormente, para o qual é executada a ação *goto*. Quando o arquivo "y.out" é analisado, o número que vem depois da ação *reduce* refere-se a uma regra da gramática, enquanto que o número que segue a ação *shift* refere-se a um estado.

Ação ACCEPT:

A ação *accept* indica que o reconhecimento da entrada foi realizado com sucesso.

Ação ERROR:

Uma ação *error* indica que houve um erro sintático no reconhecimento da entrada. A função *yyerror()* é então chamada, onde pode ser implementada uma rotina de recuperação de erros.

EXEMPLO DE ARQUIVO Y.OUT:

A partir da seguinte especificação:

```
%token DING DONG DELL
%%
rhyme : sound place;
sound : DING DONG;
place : DELL;
```

o seguinte arquivo foi gerado:

```
state 0
  $accept : _rhyme $end
  DING shift 3
  . error
  rhyme goto 1
  sound goto 2
state 1
```

```

    $accept : rhyme_$end
    $end accept
    . error
state 2
    rhyme : sound_place
    DELL shift 5
    . error
    place goto 4
state 3
    sound : DING_DONG
    DONG shift 6
    . error
state 4
    rhyme : sound place_           (1)
    . reduce 1
state 5
    place : DELL_                 (3)
    . reduce 3
state 6
    sound : DING DONG_           (2)
    . reduce 2

```

O símbolo "_" é utilizado para separar os símbolos que já foram reconhecidos dos que não foram. A seguinte entrada é utilizada: *DING DONG DELL*.

A entrada é processada através dos seguintes passos:

1. Inicialmente, o estado é 0. O analisador precisa analisar a entrada para decidir que ação, entre as disponíveis no estado 0, deve ser realizada. O primeiro token é lido (*DING*) e se torna o token *look-ahead*. A ação no estado 0 para este token é *shift 3*, o que faz com que o estado 3 seja colocado na pilha e o token *look-ahead* seja inicializado.
2. O próximo token (*DONG*) é lido e torna-se o token *look-ahead*. A ação no estado 3 para este token é *shift 6*. A pilha de estados contém, neste momento: 0, 3, 6.
3. No estado 6, o analisador reduz pela regra 2, desempilhando os estados 6 e 3.
4. *sound*, do lado esquerdo da regra 2 foi reconhecido. Consultando o estado 0, há uma ação *goto 2* para este símbolo, empilhando o estado 2 e tornando-o o estado atual.
5. No estado 2, o próximo token (*DELL*) é lido. A ação é *shift 5*, empilhando o estado 5 e inicializando o token *look-ahead*.
6. No estado 5, há uma redução pela regra 3. Assim, o estado 5 é desempilhado.
7. Há uma ação *goto* em *place*, no estado 2, fazendo com que o estado atual seja o 4. Neste momento, a pilha contém os estados 0, 2 e 4.
8. No estado 4, há uma redução pela regra 1, desempilhando os estados 4 e 2.
9. No estado 0, há uma ação *goto* em *rhyme*, fazendo com que o analisador vá para o estado 1.
10. No estado 1, o final de entrada, indicado por *\$end* é obtido quando a entrada é lida. A ação *accept* no estado 1 termina com sucesso o reconhecimento.

Quando conflitos de ambigüidade são encontrados pelo Yacc, as seguintes regras são utilizadas:

- em conflitos *shift-reduce*, a ação realizada é o *shift*;
- em conflitos *reduce-reduce*, é realizada a redução pela regra que aparece antes na especificação da gramática.

Outros conflitos relacionados ao reconhecimento de expressões aritméticas, principalmente, são minimizados através do estabelecimento de regras de precedência e de associatividade de operadores.

Regras de precedência determinam que operador deve ser utilizado primeiro. Por exemplo: $3 + 4 * 5$. Pela regra matemática, deve-se primeiro resolver a multiplicação, para então resolver a soma. Já as regras de associatividade determinam que lado da expressão, envolvendo um determinado operador, deve ser avaliado antes. Por exemplo: $3 + 4 + 5$. Se o operador é associativo à esquerda, então a expressão será avaliada como: $(3 + 4) + 5$. Caso seja associativo à direita: $3 + (4 + 5)$.

Ambos os tipos de regras são especificados no Yacc através das palavras reservadas *%left*, *%right* e *%nonassoc*, associadas aos tokens. Todos os tokens na mesma linha têm o mesmo nível de precedência e associatividade. As linhas estão dispostas na ordem de precedência. Por exemplo:

```
%left '+' '-'  
%left '*' '/'
```

indicam que os operadores * e / têm mais alta precedência do que os operadores + e -.

Considerando a seguinte especificação Yacc:

```
%right '='  
%left '+' '-'  
%left '*' '/'  
%%  
expr : expr '=' expr  
      / expr '+' expr  
      / expr '-' expr  
      / expr '*' expr  
      / expr '/' expr  
      / NAME;
```

indique a ordem em que as operações são realizadas, considerando a seguinte expressão como entrada: $a = b = c * d - e - f * g$.

Pode-se também associar níveis de precedência em regras da gramática. Por exemplo, para especificar que o operador unário '-' tem maior precedência do que o operador binário '-', pode-se utilizar a seguinte especificação:

```
%right '='
```

```

%left '+' '-'
%left '*' '/'
%%
expr  : expr '=' expr
      / expr '+' expr
      / expr '-' expr
      / expr '*' expr
      / expr '/' expr
      / '-' expr    %prec '*'
      / NAME;

```

⇒ DICAS PARA A CONSTRUÇÃO DA ESPECIFICAÇÃO:

As seguintes dicas são sugeridas, para a construção de especificações para o Yacc:

- utilizar letras maiúsculas para nomes de tokens e letras minúsculas para não-terminais;
- iniciar os nomes dos tokens com algum prefixo comum, como "t_", por exemplo, a fim de evitar conflitos com palavras reservadas da linguagem C, por exemplo;
- colocar as regras e ações em linhas separadas, auxiliando na leitura da especificação;
- colocar todas as regras com mesmo lado esquerdo juntas, escrevendo este lado esquerdo apenas uma vez;
- criar funções para ações muito complicadas;
- evitar regras recursivas à direita, dando preferência a regras recursivas à esquerda.

⇒ EXEMPLO DE ESPECIFICAÇÃO:

A seguir é apresentado um exemplo de especificação Yacc, para a construção de uma calculadora. Esta calculadora tem 26 registradores, com identificação de *a* a *z* e aceita operações aritméticas feitas com os operadores +, -, *, /, % (mod), & (e binário), | (ou binário) e atribuições.

Este exemplo apresenta formas de utilização das regras de precedência e ambigüidade.

```

%{
#include <stdio.h>
#include <ctype.h>

int regs[26];
int base;
%}

%start list

%token DIGIT LETTER

```

```

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS      /* Determina a precedência para o '-' unário. */
%%
list      :      /* vazia */
          |      list stat '\n';

stat      :      expr          { (void) printf("%d\n", $1); }
          |      LETTER '=' expr { regs[$1] = $3; };

expr      :      (' expr ')    { $$ = $2; }
          |      expr '+' expr { $$ = $1 + $3; }
          |      expr '-' expr { $$ = $1 - $3; }
          |      expr '*' expr { $$ = $1 * $3; }
          |      expr '/' expr { $$ = $1 / $3; }
          |      expr '%' expr { $$ = $1 % $3; }
          |      expr '&' expr { $$ = $1 & $3; }
          |      expr '|' expr { $$ = $1 | $3; }
          |      '-' expr %prec UMINUS { $$ = -$2; }
          |      LETTER        { $$ = regs[$1]; }
          |      number;

number : DIGIT          { $$ = $1; base = ($1 = 0)?8:10; }
       | number DIGIT  { $$ = base * $1 + $2; };

%%
int yylex()
/* Rotina de análise léxica que retorna LETTER, para letras minúsculas, sendo yyval = 0 a 25;
retorna DIGIT para dígitos, sendo yyval = 0 a 9. Os outros caracteres são retornados
imediatamente. */
int c;
while ((c = getchar()) == ' ');
if (islower(c))
{
    yyval = c - 'a';
    return(LETTER);
}
if (isdigit(c))
{
    yyval = c - '0';
    return(DIGIT);
}
return(c);
}

```

3.5. Analisadores de Precedência de Operadores

Os analisadores de precedência de operadores operam sobre a classe das gramáticas de operadores. Tal tipo de gramática tem as seguintes características:

- não apresentam dois não-terminais adjacentes do lado direito de uma produção;
- não apresentam produções que derivam a palavra vazia.

A análise de precedência é bastante deficiente, sendo aplicada apenas no reconhecimento de expressões. Os analisadores baseiam-se em relações de precedência disjuntas entre terminais para identificar o *handle*. São três as relações de precedência entre terminais da gramática: $<$, $>$ e $=$. Desta forma, sejam a e b símbolos terminais:

- $a < b$ significa que b tem precedência sobre a ;
- $a = b$ significa que a e b têm a mesma precedência;
- $a > b$ significa que a tem precedência sobre b .

A análise de precedência é baseada em uma tabela de precedência, cujas relações definem o movimento que o analisador deve fazer: empilhar, reduzir, aceitar ou chamar uma rotina de atendimento a erro. Por exemplo, para uma gramática que reconheça expressões aritméticas, tem-se a seguinte tabela:

Tabela 3.7. Tabela de precedência de operadores para gramática que reconhece expressões aritméticas.

	id	+	*	()	\$
id		$>$	$>$		$>$	$>$
+	$<$	$>$	$<$	$<$	$>$	$>$
*	$<$	$>$	$>$	$<$	$>$	$>$
($<$	$<$	$<$	$<$	$=$	
)		$>$	$>$		$>$	$>$
\$	$<$	$<$	$<$	$<$		

Um analisador de precedência funciona da seguinte maneira:

- Se $a < b$ ou $a = b$, então empilha.
- Se $a > b$, procura o *handle* da pilha (o qual deve estar delimitado pelas relações $<$ e $>$) e o substitui pelo não-terminal correspondente.

Por exemplo, para reconhecer a sentença $id+id*id$, o analisador realiza os seguintes movimentos (Tabela 3.8).

Tabela 3.8. Movimentos do analisador de precedência para reconhecimento da sentença $id+id*id$.

Pilha	Relação	Entrada	Ação	Handle
\$	<	id+id*id\$	empilha id	
\$id	>	+id*id\$	reduz	id
\$E	<	+id*id\$	empilha +	
\$E+	<	id*id\$	empilha id	
\$E+id	>	*id\$	reduz	id
\$E+E	<	*id\$	empilha *	
\$E+E*	<	id\$	empilha id	
\$E+E*id	>	\$	reduz	id
\$E+E*E	>	\$	reduz	E*E
\$E+E	>	\$	reduz	E+E
\$E		\$	aceita	

Existem dois métodos para a construção da tabela de precedências:

- MÉTODO INTUITIVO: baseado no conhecimento da precedência e associatividade dos operadores. Por exemplo, se * tem precedência sobre +, então: $+ < * \text{ e } * > +$.
- MÉTODO MECÂNICO: a partir das produções da gramática de operadores, não-ambígua, que reflita a associatividade e precedência de operadores em suas produções.

MÉTODO INTUITIVO

Considerando os operadores θ_1 e θ_2 , tem-se que:

- se o operador θ_1 tem maior precedência sobre o operador θ_2 , então $\theta_1 > \theta_2$ e $\theta_2 > \theta_1$.
- se θ_1 e θ_2 têm igual precedência (ou são iguais) e são associativos à esquerda, então $\theta_1 > \theta_2$ e $\theta_2 > \theta_1$. Se são associativos à direita, então $\theta_1 < \theta_2$ e $\theta_2 < \theta_1$. Por exemplo, os operadores * e / têm a mesma precedência e são associativos à esquerda, portanto $* > /$ e $/ > *$. Em Pascal, o operador de exponenciação é associativo à direita, sendo $** < **$.
- para todos os operadores θ , tem-se:

$\theta < id$	$\theta < ($	$) > \theta$	$\theta > \$$
$id > \theta$	$(< \theta$	$\theta >)$	$\$ < \theta$
- as seguintes relações são igualmente válidas:

$) >)$	$\$ < ($	$id >)$	$(< id$	$(=)$
$(< ($	$\$ < id$	$) > \$$	$id > \$$	

Por exemplo, a partir da seguinte gramática:

$$E \rightarrow E + E \mid E * E \mid E ** E \mid (E) \mid id$$

deseja-se obter a tabela de precedência de operadores. Os seguintes níveis de precedência e associatividade existem entre os operadores:

- ** tem maior precedência e é associativo à esquerda;
- * é associativo à esquerda;
- + tem menor precedência e é associativo à esquerda.

Obtém-se, então, a seguinte tabela:

Tabela 3.9. Tabela de precedência para gramática exemplo.

	+	*	**	()	id	\$
+	>	<	<	<	>	<	>
*	>	>	<	<	>	<	>
**	>	>	<	<	>	<	>
(<	<	<	<	=	<	
)	>	>	>		>		>
id	>	>	>		>		>
\$	<	<	<	<		<	Ac

3.6. Recuperação de Erros

Quando o compilador detecta um erro de sintaxe, é desejável que ele tente continuar com o processo de análise de modo a detectar erros adicionais (como ocorre com a maior parte dos compiladores para a linguagem C). Para que tal característica ocorra, é necessário que seja feito um procedimento denominado “recuperação de erros”.

A qualidade de uma rotina de recuperação de erros consiste na precisão com que ela resincroniza a análise do restante da sentença, sendo inversamente proporcional à sua capacidade de gerar falsos erros e induzir a erros em cascata (efeitos colaterais).

A seguir são apresentadas algumas estratégias.

3.6.1. Recuperação de Erros na Análise LL

Na tabela LL, as posições em branco representam situações de erro, devendo ser utilizadas para chamadas a rotinas de recuperação. Pode-se alterar a tabela de análise para recuperar erros de acordo com o seguinte método: *modo pânico*, no qual, na ocorrência de um erro, o analisador despreza símbolos da entrada até encontrar um token de sincronização;

Modo Pânico

O conjunto de sincronização par um símbolo não-terminal A é formado pelos terminais em FOLLOW(A). Assim, quando A estiver no topo da pilha e o símbolo da entrada for um terminal de FOLLOW(A), a ação do analisador deverá ser “desempilha A”.

Por exemplo, com a seguinte tabela:

	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	sinc	sinc
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	sinc		$T \rightarrow FT'$	sinc	sinc
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	sinc	sinc	$F \rightarrow (E)$	sinc	sinc

tem-se as seguintes situações:

- se a entrada estiver vazia, lê-se o próximo token;
- se a entrada é *sinc*, desempilha o não-terminal do topo;
- se o token do topo não é igual ao símbolo da entrada, desempilha.

3.6.2. Recuperação de Erros na Análise de Precedência de Operadores

Existem dois pontos nos quais o analisador pode descobrir erros sintáticos:

- na consulta à matriz de precedência, quando não existe relação de precedência entre o terminal do topo da pilha e o símbolo de entrada;
- quando o analisador supõe a existência de um *handle* no topo da pilha, mas não existe produção com o lado direito correspondente.

Erros detectados no empilhamento

As lacunas na tabela de precedência evidenciam condições de erro. As classes de erro são identificadas, para a implementação de uma rotina de atendimento para cada classe. Por exemplo, na tabela abaixo, para quatro casos de erro foi possível aplicar a mesma rotina E2 de recuperação:

	id	+	*	()	\$
id	E2	>	>	E2	>	>
+	<	>	<	<	>	>
*	<	>	>	<	>	>
(<	<	<	<	=	E1
)	E2	>	>	E2	>	>
\$	<	>	>	>	E3	Ac

E1: desempilha e avisa: “parêntese esquerdo a mais”.

E2: insere + na entrada e avisa: “falta operador”

E3: apaga) da entrada e avisa: “parêntese direito ilegal”.

Erros durante reduções

Como os símbolos não-terminais são transparentes na identificação do *handle* do topo da pilha, a redução somente deve acontecer se, realmente houver no topo um lado direito de produção. Ações de recuperação devem ser definidas a partir do exame dos lados direitos de produção, a fim de identificar a existência ou não de não-terminais adjacentes a símbolos terminais. Por exemplo, para a seguinte gramática:

$$E \rightarrow E+E \mid E^*E \mid (E) \mid \text{id}$$

se + ou * forem identificados como *handle*, verificar se existem não-terminais em ambos os lados. Caso negativo, executar a redução e emitir a mensagem: “falta expressão”.

4. TRADUÇÃO DIRIGIDA POR SINTAXE

Uma definição dirigida por sintaxe é uma generalização de uma gramática livre do contexto, na qual cada símbolo da gramática tem um conjunto de atributos associados, particionados em dois subconjuntos denominados atributos sintetizados e herdados.

Um atributo pode representar uma palavra, um número, um tipo ou uma posição na memória, por exemplo. O valor de um atributo na árvore de derivação é definido por uma regra semântica associada com a produção utilizada naquele nodo. O valor de um atributo sintetizado é computado a partir dos atributos dos filhos daquele nodo, enquanto que um atributo herdado é computado a partir dos atributos dos irmãos ou dos pais daquele nodo.

Regras semânticas estabelecem dependências entre atributos que são representadas por um grafo. A partir do *grafo de dependências*, é derivada uma ordem de avaliação para as regras semânticas. A avaliação das regras semânticas define os valores dos atributos dos nodos, para a sentença de entrada considerada.

Uma árvore de derivação que apresenta os valores dos atributos de cada nodo é denominada árvore de derivação “decorada”.

4.1. Formato de uma Definição Dirigida por Sintaxe

Em uma definição dirigida por sintaxe, cada produção da gramática $A \rightarrow \alpha$ tem associada um conjunto de regras semânticas no formato $b := f(c_1, c_2, \dots, c_k)$, onde f é uma função e:

1. b é um atributo sintetizado de A e c_1, c_2, \dots, c_k são atributos pertencentes aos símbolos da gramática da produção, ou
2. b é um atributo herdado de um dos símbolos da gramática do lado direito da produção, e c_1, c_2, \dots, c_k são atributos pertencentes aos símbolos da gramática da produção.

Em ambos os casos b depende dos atributos c_1, c_2, \dots, c_k . Uma gramática de atributos é uma definição dirigida por sintaxe na qual as funções nas regras semânticas não têm efeitos colaterais.

Assume-se que terminais tenham apenas atributos sintetizados, assim como o símbolo inicial da gramática. Um exemplo de definição dirigida por sintaxe é apresentado na Tabela 4.1:

Tabela 4.1. Definição dirigida por sintaxe de uma calculadora simples.

PRODUÇÃO	REGRAS SEMÂNTICAS
$L \rightarrow E \ n$	<code>printf(“%i”, E.val);</code>
$E \rightarrow E_1 + T$	<code>E.val = E₁.val + T.val;</code>
$E \rightarrow T$	<code>E.val = T.val;</code>
$T \rightarrow T_1 * F$	<code>T.val = T₁.val * F.val;</code>

$T \rightarrow F$	$T.val = F.val;$
$F \rightarrow (E)$	$F.val = E.val;$
$F \rightarrow \text{dígito}$	$F.val = \text{dígito.lexval};$

O token *dígito* tem um atributo sintetizado *lexval* cujo valor é fornecido pelo analisador léxico.

4.2. Atributos Sintetizados

Uma definição que utiliza apenas atributos sintetizados é denominada de *definição-S de atributos*. Uma árvore de sintaxe para uma definição deste tipo pode ser *anotada* através da avaliação das regras semânticas para os atributos em cada nodo, de forma *bottom-up*, das folhas para a raiz.

A Figura 4.1 apresenta um exemplo de árvore de derivação anotada para o reconhecimento e cálculo da sentença de entrada $3*5+4n^2$.

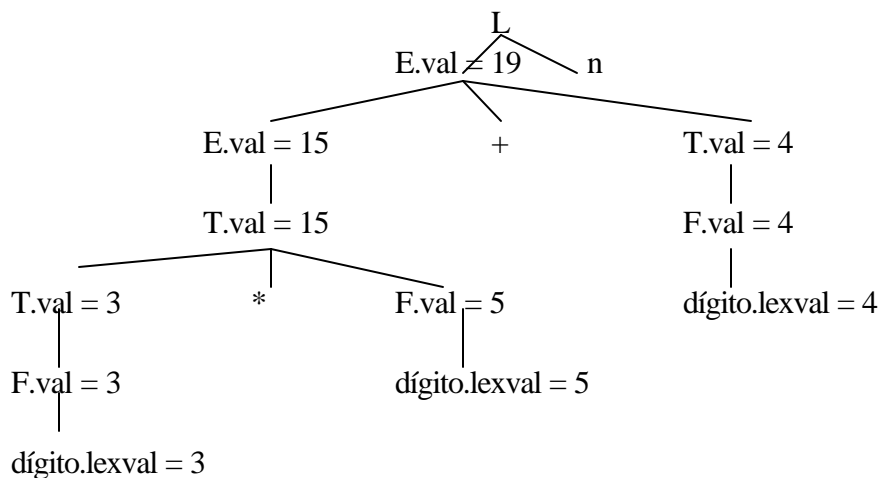


Figura 4.1. Árvore de derivação anotada para o reconhecimento da sentença $3*5+4n$.

4.3. Atributos Herdados

Um atributo herdado é definido a partir dos atributos dos nodos irmãos ou pais. Este tipo de atributo é adequado para expressar a dependência de uma estrutura da linguagem de programação no contexto em que ela aparece. Um exemplo de definição dirigida por sintaxe utilizando atributos herdados é apresentada na Tabela 4.2:

Tabela 4.2. Definição dirigida por sintaxe com o atributo herdado *L.in*.

PRODUÇÃO	REGRAS SEMÂNTICAS
$D \rightarrow T L$	$L.in = T.type;$
$T \rightarrow \text{int}$	$T.type = \text{integer};$
$T \rightarrow \text{real}$	$T.type = \text{real};$

² A letra *n* no final da expressão representa o caracter de nova linha, que causa a impressão do resultado da expressão.

$L \rightarrow L_1, id$	$L_1.in = L.in;$ $addType(id.entry, L.in);$
$L \rightarrow id$	$addType(id.entry, L.in);$

A Figura 4.2 apresenta uma árvore de derivação anotada para a sentença *real id1, id2, id3*.

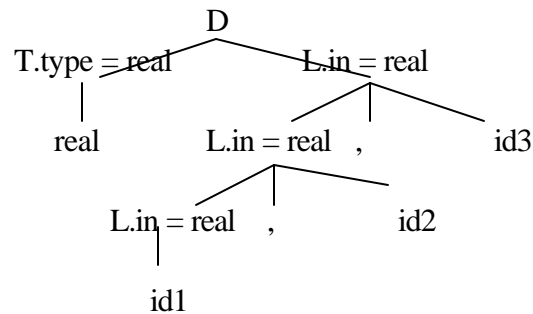


Figura 4.2. Árvore de derivação com o atributo herdado *in* em cada nodo *L*.

4.4. Grafo de Dependências

Se um atributo *b* em um nodo da árvore de derivação depende do atributo *c*, então a regra semântica para *b* naquele nodo deve ser avaliada depois da regra semântica que define *c*. As interdependências entre os atributos herdados e sintetizados nos nodos da árvore de derivação pode ser apresentados por um grafo direcionado denominado *grafo de dependências*.

Antes de construir um grafo de dependência para uma árvore de sintaxe, deve-se expressar cada regra semântica no formato $b = f(c_1, c_2, \dots, c_k)$, introduzindo um atributo sintetizado *b* para cada regra semântica, que consiste em uma chamada de procedimento. O grafo tem um nodo para cada atributo e um arco do nodo *b* para o nodo *c*, se o atributo *b* depende do atributo *c*. O seguinte algoritmo pode ser seguido para a construção do grafo de dependências:

Para cada nodo n da árvore de derivação faça

Para cada atributo a do símbolo da gramática em n faça

construa um nodo no grafo de dependências para a;

Para cada nodo n da árvore de derivação faça

Para cada regra semântica $b = f(c_1, c_2, \dots, c_k)$ associada a uma produção usada em n faça

Para $i = 1$ até k faça

construa um arco do nodo c_i para o nodo b .

Por exemplo, considerando que a regra semântica $A.a = f(X.x, Y.y)$ esteja associada à produção $A \rightarrow XY$, de forma que o atributo sintetizado *A.a* dependa dos atributos *X.x* e *Y.y*. Se esta produção é usada em uma árvore de derivação, haverá três nodos *A.a*, *X.x* e *Y.y* no grafo de dependências com um arco de *X.x* para *A.a* e de *Y.y* para *A.a*.

Se a produção $A \rightarrow XY$ tivesse uma regra semântica $X.i = g(A.a, Y.y)$ associada, haveria um arco de $A.a$ para $X.i$ e de $Y.y$ para $X.i$. A seguir é apresentado um exemplo de grafo de dependência para a produção $E \rightarrow E_1 + E_2$, que tem a seguinte regra semântica associada: $E.val = E_1.val + E_2.val$.

O grafo de dependência resultante tem três nodos, representando os atributos sintetizados $E.val$, $E_1.val$ e $E_2.val$. O arco de $E.val$ para $E_1.val$ mostra que $E.val$ depende de $E_1.val$. As linhas pontilhadas representam a árvore de derivação e não fazem parte do grafo de dependências (Figura 4.3).

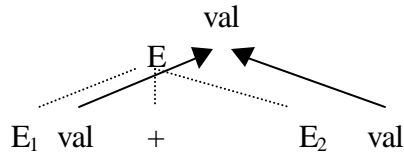


Figura 4.3. $E.val$ é sintetizado a partir de $E_1.val$ e $E_2.val$.

A Figura 4.4 apresenta o grafo de dependência para a árvore de derivação apresentada na Figura 4.2. Os nodos no grafo são marcados por números, que serão utilizados a seguir. Há um arco do nó 4 para o nó 5, porque o atributo herdado $L.in$ depende do atributo $T.type$, pela regra semântica $L.in = T.type$, da produção $D \rightarrow TL$. O arco entre os nodos 7 e 9 é representado porque $L_1.in$ depende de $L.in$, de acordo com a regra semântica $L_1.in = L.in$ para a produção $L \rightarrow L_1, id$. Cada uma das regras semânticas $addType(id.entry, L.in)$ associada com as produções de L leva à criação de um atributo temporário. Nodos 6, 8 e 10 são construídos para este tipo de atributo.

Figura 4.4. Grafo de dependência para árvore de derivação da Figura 4.2.

4.5. Ordem de Avaliação

Uma ordenação topológica e um grafo acíclico direcionado é qualquer ordenação m_1, m_2, \dots, m_k de nodos do grafo, tal que os arcos que saem de nodos anteriores (numeração menor), na ordenação, chegam em nodos posteriores; ou seja, se $m_i \rightarrow m_j$ é um arco de m_i para m_j , então m_i aparece antes de m_j na ordenação.

Qualquer ordenação topológica de um grafo de dependências fornece uma ordem válida na qual as regras semânticas associadas aos nodos na árvore de derivação podem ser avaliados. Ou seja, na ordem topológica, os atributos aparecem após aos atributos de que eles dependem.

Inicialmente a árvore de sintaxe é construída (pelo analisador sintático). Depois o grafo de dependências. A partir da ordenação topológica, obtém-se a ordem de avaliação para as regras semânticas. Por exemplo, considerando a Figura 4.4, verifica-se que a ordenação topológica é obtida escrevendo-se os números dos nodos em ordem crescente. Desta ordenação, obtém-se o seguinte programa, onde a_n é o atributo associado com o nodo numerado n , no grafo de dependências:

```

a4 = real;
a5 = a4;
addType(id3.entry, a5);
a7 = a5;
addType(id2.entry, a7);
a9 = a7;
addType(id1.entry, a9);

```

A avaliação destas regras semânticas armazena o tipo *real* na entrada da tabela de símbolos para cada identificador.

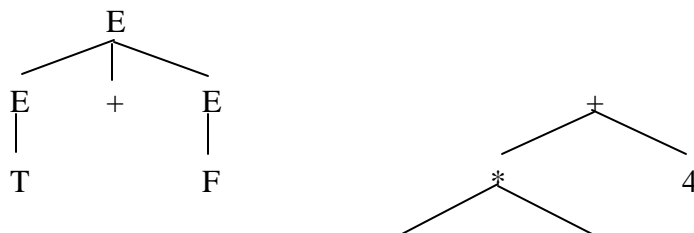
Os seguintes métodos foram propostos para a avaliação das regras semânticas:

1. BASEADOS NA ANÁLISE SINTÁTICA: em tempo de compilação, estes métodos obtêm uma ordem de avaliação a partir da ordem topológica do grafo de dependências. Estes métodos falham na obtenção da ordem da avaliação somente se o grafo considerado possui um ciclo.
2. BASEADOS EM REGRAS: em tempo de construção do compilador, as regras semânticas associadas com as produções são analisadas, manualmente, ou por uma ferramenta específica. Para cada produção, a ordem na qual os atributos associados são avaliados é então determinada.
3. ÓBVIOS: uma ordem de avaliação é escolhida sem considerar as regras semânticas. Por exemplo, se a tradução é feita durante a análise sintática, a ordem de avaliação é forçada pelo método sintático, independentemente das regras semânticas. Uma avaliação óbvia restringe a classe de definições baseadas na sintaxe que podem ser implementadas.

4.6. Construção de Árvores de Sintaxe

Árvores de sintaxe constituem-se em um tipo de representação intermediária utilizado entre a fase de análise do código fonte e da geração do código intermediário. Esquemas de tradução podem ser utilizados para especificar a construção de árvores de sintaxe. Uma árvore de sintaxe é uma forma condensada da árvore de derivação, na qual somente os operandos aparecem nas folhas, enquanto que os operadores aparecem em nodos interiores da árvore. Cadeias simples de produções, tais como $A \rightarrow B$ e $B \rightarrow C$, por exemplo.

A Figura 4.5 apresenta um exemplo de árvore de derivação e de árvore de sintaxe para a sentença $3*5+4$.



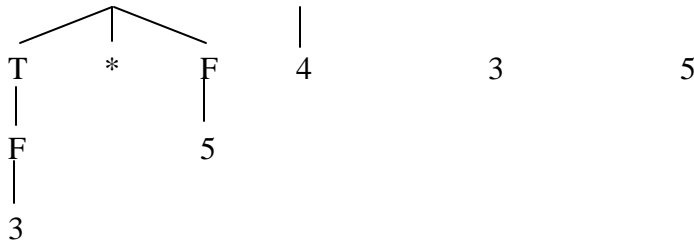


Figura 4.5. Árvores de derivação e de sintaxe para a sentença $3*5+4$.

O seguinte esquema de tradução pode ser utilizado para a construção de árvores de sintaxe para expressões aritméticas simplificadas (Tabela 4.3):

Tabela 4.3. Esquema de tradução para construção de árvores de sintaxe para expressões aritméticas simplificadas.

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E1 + T$	$E.ptr = \text{geranodo}("+", E1.ptr, T.ptr);$
$E \rightarrow E1 - T$	$E.ptr = \text{geranodo}("-", E1.ptr, T.ptr);$
$E \rightarrow T$	$E.ptr = T.ptr;$
$T \rightarrow (E)$	$T.ptr = E.ptr;$
$T \rightarrow id$	$T.ptr = \text{gerafolha}(id, id.índice);$
$T \rightarrow num$	$T.ptr = \text{gerafolha}(num, num.val);$

O atributo sintetizado *ptr* conserva os ponteiros retornados pelas chamadas das funções. As funções que geram os nodos de operadores binários e folhas de operandos são as seguintes:

- *geranodo(op,esq,dir)*: cria um nodo de operador com rótulo *op* e dois campos contendo ponteiros para *esq* e *dir*.
- *gerafolha(id,índice)*: cria um nodo de identificador com rótulo *id* e o campo *índice* contendo um ponteiro para a entrada do identificador na tabela de símbolos.
- *gerafolha(num,val)*: cria um nodo de número com rótulo *num* e campo *val* contendo o valor do número.

EXERCÍCIO: Construa uma árvore de derivação anotada, mostrando a construção de uma árvore de sintaxe, para a expressão $a - 4 + c$. Faça a construção no sentido *bottom-up*.

A árvore de sintaxe é realmente implementada, enquanto que a árvore de derivação pode não ser explicitamente construída.

4.7. Avaliação Bottom-up de Esquemas de Tradução S-atribuídos

Atributos sintetizados podem ser avaliados por um analisador redutivo a medida em que a cadeia de entrada é reconhecida. Os valores dos atributos sintetizados são conservados, associados aos símbolos da gramática, em sua própria pilha. Sempre que ocorre uma redução, são computados os valores dos novos atributos sintetizados, a partir dos atributos que estão na pilha, associados aos símbolos do lado direito da produção que está sendo reduzida.

São usados campos adicionais na pilha do analisador para armazenar os valores dos atributos sintetizados. Por exemplo, considerando o seguinte esquema de tradução:

$$A \rightarrow X Y Z \qquad A.a = f(X.x, Y.y, Z.z);$$

tem-se o seguinte conteúdo na pilha (Tabela 4.4):

Tabela 4.4. Pilha para análise sintática de gramática com esquema de tradução.

PILHA	
ANÁLISE	ATRIBUTOS
Z	Z.z
Y	Y.y
X	X.x
...	...

Antes que XYZ seja reduzido para A, os valores dos atributos Z.z, Y.y e X.x estão armazenados na pilha, na coluna *Atributos*, nas posições [topo], [topo-1] e [topo-2], respectivamente. Se um símbolo não tem atributos, então a entrada correspondente na pilha é vazia. Após a redução, topo é decrementado de 2, A é armazenado em *Análise[topo]* e o valor do atributo sintetizado A.a é colocado em *Atributos[topo]*.

A gramática definida na Tabela 4.1 pode ter seus atributos sintetizados avaliados por um analisador empilha-reduz durante uma análise *bottom-up*. Para isso, deve-se modificar o analisador para executar as instruções abaixo ao fazer as reduções apropriadas. O código foi obtido a partir das regras semânticas substituindo-se cada atributo por uma posição no vetor *Atrib*.

```

L → E =          printf("%i",Atrib[topo]);
E → E1 + T       Atrib[ntopo] = Atrib[topo-2] + Atrib[topo];
E → T
T → T1 * F       Atrib[ntopo] = Atrib[topo-2] * Atrib[topo];
T → F
F → (E)          Atrib[ntopo] = Atrib[topo-1];
F → dígito       Atrib[topo] = dígito.lexval;

```

Quando o analisador reconhece um dígito, o token *dígito* é empilhado em *Análise[topo]* e seu atributo é armazenado em *Atrib[topo]*. O código não mostra como as variáveis *topo* e *ntopo* são

operadas. Quando uma produção com n símbolos do lado direito é reduzida, o valor de $ntopo$ é atualizado para $topo-n+1$. Após a execução de cada instrução, $topo$ é atualizado para $ntopo$. A tabela abaixo mostra a seqüência de movimentos realizados com a sentença $3*5+4=$.

Tabela 4.5. Seqüência de movimentos realizados com a sentença $3*5+4$.

ENTRADA	ANÁLISE	ATRIBUTOS	PRODUÇÃO
$3*5+4=$	-	-	
$*5+4=$	3	3	
$*5+4=$	F	3	F \rightarrow dígito
$*5+4=$	T	3	T \rightarrow F
$5+4=$	T*	3 -	
$+4=$	T*5	3 - 5	
$+4=$	T*F	3 - 5	F \rightarrow dígito
$+4=$	T	15	T \rightarrow T*F
$+4=$	E	15	E \rightarrow T
$4=$	E+	15 -	
$=$	E+4	15 - 4	
$=$	E+T	15 - 4	F \rightarrow dígito
$=$	E+T	15 - 4	T \rightarrow F
$=$	E	19	E \rightarrow E + T
	E=	19	
	L	19	L \rightarrow E=

Na implementação, as instruções são executadas sempre antes de uma redução. Os esquemas de tradução considerados na próxima seção provêm uma notação para intercalar ações com análise sintática.

4.8. Esquemas de Tradução L-atribuídos

Quando a tradução ocorre durante a análise, a ordem de avaliação de atributos está amarrada à ordem pela qual os nodos da árvore de derivação são criados pelo método de análise. Uma ordem

natural, que caracteriza vários métodos de tradução *top-down* e *bottom-up* é chamada *depth-first order*, definida pelo seguinte algoritmo:

```
DEPTH_FIRST(n: nodo);
Para cada filho m de n, da esquerda para a direita,
    avaliar os atributos herdados de m;
    DEPTH_FIRST(m);
Avaliar os atributos sintetizados de n.
```

Os atributos dos esquemas L-atribuídos devem ser avaliados sempre na ordem *depth-first*. Este tipo de esquema baseia-se nas gramáticas LL(1). Um esquema é L-atribuído se cada atributo herdado de X_j , $1 < j < n$, no lado direito de $A \rightarrow X_1 X_2 \dots X_n$, depende somente:

- a) dos atributos dos símbolos X_1, X_2, \dots, X_{j-1} à esquerda de X_j na produção e
- b) dos atributos herdados de A .

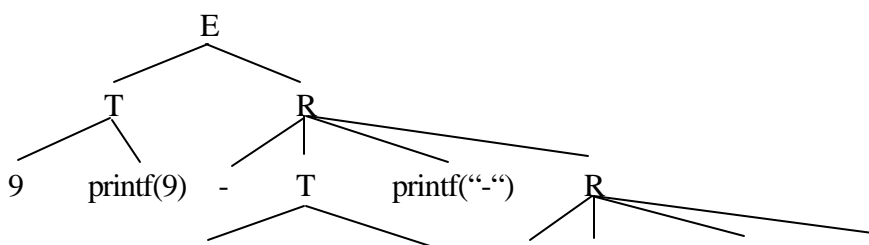
Todo esquema S-atribuído é também L-atribuído, há que as restrições *a* e *b* aplicam-se somente a atributos herdados. O esquema apresentado abaixo não é L-atribuído, já que o atributo herdado $Q.i$ depende do atributo $R.s$, cujo não-terminal associado R está a direita de Q .

$A \rightarrow LM$	{ $L.i = I(A.i);$ $M.i = m(L.s);$ $A.s = f(M.s);$
$A \rightarrow QR$	$R.i = r(A.i);$ $Q.i = q(R.s);$ $A.s = f(Q.s);$

O exemplo apresentado a seguir é de um esquema de tradução que mapeia expressões infixadas em pós-fixadas:

```
E → TR
R → op T {printf("%c",op.val); } R
    | ε
T → num {printf("%i",num.val); }
```

A Figura 4.6 apresenta a árvore de derivação para a expressão $9 - 5 + 2$ com as ações semânticas agregadas aos nodos apropriados. Ações semânticas são tratadas como símbolos terminais. Quando executadas na ordem *depth-first*, imprimem a expressão $9 5 - 2 +$.



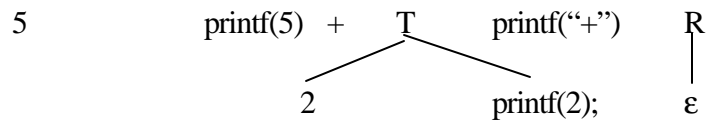


Figura 4.6. Árvore de derivação para a expressão $9 - 5 + 2$.

Quando um esquema de tradução é definido, deve-se assegurar que os valores dos atributos estejam disponíveis sempre que ações se refiram a eles. No caso mais simples, quando são usados apenas atributos sintetizados, pode-se construir o esquema de tradução colocando-se as ações que computam os valores dos atributos no fim do lado direito da produção associada. Se existem ambos, atributos herdados e sintetizados, deve-se proceder da seguinte forma:

1. Um atributo herdado para um símbolo no lado direito de uma produção deve ser computado em uma ação antes deste símbolo.
2. Uma ação não deve se referir a um atributo sintetizado de um símbolo à direita desta ação.
3. Um atributo sintetizado associado ao não-terminal do lado esquerdo da produção deve ser computado somente depois que todos os atributos que ele referencia tenham sido computados. A ação que computa tais atributos pode, em geral, ser especificada no final da produção.

O seguinte esquema de tradução não satisfaz o primeiro dos requisitos acima:

$S \rightarrow A1 A2$ { A1.h = 1; A2.h = 2; }
 $A \rightarrow a$ { printf(“%c”,A.h); }

O atributo herdado A.h na segunda produção ainda não está definido quando uma tentativa de imprimir seu valor é feita durante uma travessia *depth-first* da árvore de derivação para a entrada *aa*. A travessia inicia em S e visita as sub-árvores A1 e A2 antes que os valores A1.h e A2.h sejam atribuídos.

5. GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

A geração de código intermediário é a transformação da árvore de derivação (representação interna produzida pelo analisador sintático) em uma seqüência de código mais próximo do código objeto. As seguintes vantagens estão associadas à geração de código intermediário:

- A portabilidade torna-se mais fácil de ser obtida, já que a primeira parte do compilador (análise léxica, sintática e código intermediário) não precisa ser mudada. Apenas a geração de código objeto deve ser específica para a máquina;
- Um otimizador de código intermediário pode ser aplicado à representação intermediária.

As representações mais utilizadas de código intermediário são:

- Árvores de sintaxe;
- Notação pós-fixada;
- Código de três endereços.

As primeiras duas representações já foram abordadas no capítulo anterior (Capítulo 4). Por isso, maior atenção será dispensada ao terceiro tipo de representação: o código de três endereços.

5.1. Código de Três Endereços

O código de três endereços (CTE) é formado por uma seqüência de comandos com o seguinte formato geral:

$$X = y \text{ op } z,$$

onde x , y e z são nomes, constantes ou variáveis temporárias geradas pelo compilador e op é qualquer operador. Uma expressão do tipo $x + y * z$ é traduzida para o seguinte código:

```
t1 = y * z;  
t2 = x + t1;
```

onde $t1$ e $t2$ são variáveis temporárias geradas pelo compilador.

O CDE é assim chamado porque utiliza, para cada comando, no máximo três posições de memória: duas para operandos e uma para o resultado. Os seguintes tipos de comandos são utilizados no CDE:

1. Comandos de atribuição do tipo $x = y \text{ op } z$, onde op é um operador binário ou um operador lógico.
2. Comandos de atribuição do tipo $x = op y$, onde op é um operador unário.
3. Comandos de cópia do tipo $x = y$, onde o valor de y é atribuído a x .
4. Desvio incondicional do tipo $goto L$, onde L é o label do comando a ser executado.
5. Desvios condicionais, tais como $if x \text{ relop } y \text{ goto } L$, onde $relop$ é um operador relacional.

6. Chamadas de rotinas *param x* e *call p,n* e retornos de valores de rotinas, do tipo *return y*. Um exemplo de uso é apresentado a seguir, com o CDE gerador a partir de uma chamada de rotina $p(x_1, x_2, \dots, x_n)$. *n* representa o número de parâmetros reais em *call p,n*.

```

param x1
param x2
...
param xn
call p,n

```

7. Comandos de atribuição com variáveis indexadas, do tipo $x = y[i]$ e $x[i] = y$.
8. Comandos de atribuição com variáveis como endereços e ponteiros, do tipo $x = \&y$, $x = *y$ e $*x = y$.

A escolha de um conjunto de comandos é bastante importante na geração do CDE. Um conjunto pequeno de comandos é mais simples para a tradução para o código objeto, enquanto que apresenta a desvantagem de gerar longas seqüências de código que são mais árduas de serem otimizadas.

O esquema de tradução dirigida por sintaxe da Figura 5.1 gera CDE para comandos de atribuição. O atributo sintetizado *S.code* representa o CDE para a atribuição *S*. O não-terminal *E* tem os seguintes dois atributos: *E.place*, que aponta para o nome que armazenará o valor de *E* e *E.code*, que aponta para a seqüência de CDE's resultante da avaliação de *E*. O símbolo “||” representa a operação de concatenação. A função *newtemp* retorna uma seqüência de nomes distintos t_1, t_2, \dots a cada chamada. A função *gen*($x = y + z$) representa o CDE do comando $x = y + z$. Expressões que apareçam no lugar das variáveis *x*, *y* e *z* são avaliadas quando passadas ao *gen* e operadores entre aspas são considerados literalmente. Na prática, CDE's poderiam ser enviados a um arquivo de saída, que constrói os atributos do código.

PRODUÇÃO	REGRA SEMÂNTICA
$S \rightarrow id = E$	$S.code = E.code \parallel gen(id.place = E.place);$
$E \rightarrow E_1 + E_2$	$E.place = newtemp;$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.place = E_1.place + E_2.place);$
$E \rightarrow E_1 * E_2$	$E.place = newtemp;$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.place = E_1.place * E_2.place);$
$E \rightarrow -E_1$	$E.place = newtemp;$ $E.code = E_1.code \parallel gen(E.place = - E_1.place);$
$E \rightarrow (E_1)$	$E.place = E_1.place;$ $E.code = E_1.code;$
$E \rightarrow id$	$E.place = id.place;$ $E.code = "";$

Figura 5.1. Esquema de tradução dirigida por sintaxe para geração de CDE's para comandos de atribuição.

A partir da entrada $a = b * -c + b * -c$, o seguinte CDE é obtido:

$t1 = -c;$

```

t2 = b * t1;
t3 = -c;
t4 = b * t3;
t5 = t2 + t4;
a = t5;

```

5.1.1. Código de Três Endereços para Declarações

Para cada variável local, uma entrada na tabela de símbolos é criada, com o tipo e seu endereço relativo de armazenamento. O endereço relativo é calculado através de um deslocamento a partir da área de dados estáticos ou dos dados no registro de ativação.

No esquema de tradução apresentado na Figura 5.2, o não-terminal P gera uma seqüência de declarações no formato $id: T$. Antes que a primeira declaração é considerada, ao $offset$ é atribuído o valor 0. A medida em que é encontrada cada variável, seu nome é armazenado na tabela com o $offset$ igual ao valor atual de $offset$, sendo incrementado pelo tamanho do dado que consta no tipo daquela variável.

A rotina $enter(name, type, offset)$ cria uma entrada na tabela de símbolos para $name$, com o tipo $type$ e o endereço relativo $offset$. Na Figura 5.2, os inteiros têm tamanho de 4 bytes e os reais de 8 bytes. O tamanho de um $array$ é calculado multiplicando-se o tamanho de cada elemento pelo número de elementos. O tamanho de cada ponteiro é assumido como sendo de 4 bytes.

$P \rightarrow$	{ offset = 0; } D	
$D \rightarrow$	D ; D	
$D \rightarrow$	id : T	{ enter(id.name,T.type,offset); offset = offset + T.width; }
$T \rightarrow$	integer	{ T.type = integer; T.width = 4; }
$T \rightarrow$	real	{ T.type = real; T.width = 8; }
$T \rightarrow$	array [num] of T1	{ T.type = array(num.val,T1.type); T.width = num.val * T1.width; }
$T \rightarrow$	$\wedge T1$	{ T.type = pointer(T1.type); T.width = 4; }

Figura 5.2. Esquema de tradução para geração de CDE's para declarações.

O esquema de tradução da Figura 5.3 constrói uma árvore de tabelas de símbolos, sendo uma tabela para cada rotina, dada a possibilidade de geração de subrotinas embutidas (como na linguagem Pascal, por exemplo). São usadas duas pilhas, $tblPtr$ e $offset$, que contêm, respectivamente, ponteiros para as tabelas de símbolos e os próximos endereços de memória local disponíveis, relativos às áreas de dados das subrotinas em análise. Durante a compilação, o ponteiro do topo da pilha $tblPtr$ aponta para a tabela de símbolos da subrotina em análise. Cada tabela tem um apontador para a tabela da subrotina

envolvente. Os símbolos não-terminais M e N servem, respectivamente, para gerar a tabela raiz (tabela de símbolos relativa ao programa principal, que contém as variáveis globais), e as tabelas referentes às demais subrotinas. As seguintes funções constituem parte do esquema abaixo:

mktable(ptr): gera uma tabela de símbolos (filha da tabela apontada por ptr) e retorna um ponteiro para a tabela gerada;

addWidth(ptr,width): armazena na tabela de símbolos apontada por ptr, o tamanho da área de dados local da subrotina correspondente;

enterProc(p1,name,p2): armazena na tabela de símbolos apontada por p1 o nome da subrotina e o ponteiro para a tabela correspondente;

enter(ptr,name,type,offset): insere na tabela de símbolos apontada por ptr, um novo símbolo, seu tipo e endereço relativo à área de dados local correspondente.

$P \rightarrow MD$	{ addwidth(top(tblptr),top(offset)); pop(tblptr); pop(offset); }
$M \rightarrow \epsilon$	{ t = mktable(nil); push(t,tblptr); push(0,offset); }
$D \rightarrow D1 ; D2$	
$D \rightarrow \text{proc id}; N D1; S$	{ t = top(tblptr); addwidth(t,top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr),id.name,t); }
$D \rightarrow \text{id}: T$	{ enter(top(tblptr), id.name, T.type, top(offset)); top(offset) = top(offset) + T.width; }
$N \rightarrow \epsilon$	{ t = mktable(top(tblptr)); push(t,tblptr); push(0,offset); }

Figura 5.3. Esquema de tradução para geração de CDE's para rotinas embutidas.

5.1.2. Código de Três Endereços para Atribuições

A Figura 5.4 apresenta um esquema de tradução para geração de CDE's para atribuições. A função *lookup(id.name)* verifica se existe uma entrada para o identificador na tabela de símbolos. Caso exista, retorna um ponteiro para esta entrada; caso contrário, retorna nil, indicando que o identificador não foi encontrado. A ação semântica *emit* grava CDE's em um arquivo de saída.

$S \rightarrow \text{id} = E$	{ p = lookup(id.name); if p <> nil then emit(p "=" E.place) else error; }
$E \rightarrow E1 + E2$	{ E.place = newtemp; emit(E.place "=" E1.place "+" E2.place); }
$E \rightarrow E1 * E2$	{ E.place = newtemp; emit(E.place "=" E1.place "*" E2.place); }
$E \rightarrow -E1$	{ E.place = newtemp; emit(E.place "=" "-" E1.place); }

$E \rightarrow \{E1\}$	{ E.place = E1.place; }
$E \rightarrow id$	{ p = lookup(id.name); if p \diamond nil then E.place = p else error; }

Figura 5.4. Esquema de tradução para geração de CDE's de atribuições.

5.1.3. Conversão de Tipos para Expressões Aritméticas

Na geração de código intermediário para expressões aritméticas, o tipo dos operandos determina a natureza da operação (por exemplo, adição de inteiros, ou de ponto flutuante) que deve ser aplicada. As ações semânticas da Figura 5.5, associadas à produção soma, fazem verificação de tipo de operandos para determinar o tipo de operação a ser aplicado.

```

E → E1 + E2
{ E.place = newtemp;
if E1.type = integer and E2.type = integer
  then begin
    emit(E.place "=" E1.place "int+" E1.place);
    E.type = integer;
  end
else if E1.type = real and E2.type = real
  then begin
    emit(E.place "=" E1.place "real+" E2.place);
    E.type = real;
  end
else if E1.type = integer and E2.type = real
  then begin
    u = newtemp;
    emit(u "=" "inttoreal" E1.place);
    emit(E.place "=" u "real+" E2.place);
    E.type = real;
  end
else if E1.type = real and E2.type = integer
  then begin
    u = newtemp;
    emit(u "=" "inttoreal" E2.place);
    emit(E.place "=" E1.place "real+" u);
    E.type = real;
  end
else E.type = typeError;}

```

Figura 5.5. Ações semânticas para $E \rightarrow E1 + E2$.

5.1.4. Endereçamento de Elementos de Matrizes

O endereço do i -ésimo elemento de um vetor A é obtido por:

$$base + (i - low) * w$$

onde w = tamanho de cada elemento;

low = limite inferior;

$base$ = endereço relativo da memória reservada para o vetor, ou seja, é o endereço $A[low]$.

A expressão acima pode ser parcialmente avaliada em tempo de compilação se reescrita como: $i * w + (base - low * w)$. A subexpressão $c = base - low * w$ pode ser avaliada quando a declaração do vetor é analisada. O valor dela pode ser armazenado na tabela de símbolos, na entrada para o vetor A , tal que o endereço relativo de $A[i]$ é obtido somando-se $(i * w)$ a c .

O mesmo tipo de cálculo pode ser aplicado a matrizes bidimensionais, tridimensionais, etc. Uma matriz bidimensional é, normalmente, armazenada de uma das duas seguintes maneiras: por linha ou por coluna.

Por exemplo, uma matriz A , 2×3 , pode ser armazenada por linha como:

$A[1,1]$	$A[1,2]$	$A[1,3]$	$A[2,1]$	$A[2,2]$	$A[2,3]$
----------	----------	----------	----------	----------	----------

Ou por coluna:

$A[1,1]$	$A[2,1]$	$A[1,2]$	$A[2,2]$	$A[1,3]$	$A[2,3]$
----------	----------	----------	----------	----------	----------

No caso de matriz armazenada por linha, o endereço relativo $A[i_1, i_2]$ pode ser calculado pela fórmula:

$$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w,$$

onde low_1 e low_2 são os limites inferiores;

n_2 é o número de elementos da segunda dimensão ($n_2 = high_2 - low_2 + 1$).

Assumindo que i_1 e i_2 são os únicos valores não conhecidos em tempo de compilação, a expressão acima pode ser reescrita da seguinte forma:

$$((i_1 * n_2) + i_2) * w + (base - ((low_1 * n_2) + low_2) * w)$$

cujo último termo pode ser calculado em tempo de compilação.

5.1.5. Geração de CDE's para Expressões Lógicas

Expressões lógicas são usadas, normalmente, em comandos de atribuição para avaliar variáveis lógicas e em expressões condicionais de comandos de controle de fluxo. Dois métodos de tradução de expressões lógicas são comumente utilizados:

1. codificação numérica dos valores *true* e *false* (por exemplo, $true \triangleleft 1$ e $false = 0$) e avaliar uma expressão lógica tal como uma expressão aritmética;
2. representação do valor de uma expressão lógica por um ponto a ser atingido no programa usando instruções *if-goto*. Tal método é mais conveniente para implementação de expressões lógicas em comandos de controle.

5.1.5.1. Representação Numérica de Expressões Lógicas

Assumindo $false = 0$ e $true = 1$, as expressões são, integralmente, avaliadas da esquerda para a direita. Por exemplo, a expressão *A or B and not C* é traduzida para:

```
t1 = not C;
t2 = B and t1;
t3 = A or t2;
```

A expressão relacional $A < B$ é equivalente ao comando *if A < B then 1 else 0*, sendo traduzida para:

```
100: if A < B goto 103
101: t1 = 0
102: goto 104
103: t1 = 1
104:...
```

O esquema de tradução abaixo inclui ações semânticas para geração de código para expressões lógicas conforme o exemplo acima:

$E \rightarrow E1 \text{ or } E2$	{ E.place = newtemp; emit(E.place "=" E1.place "or" E2.place); }
$E \rightarrow E1 \text{ and } E2$	{ E.place = newtemp; emit(E.place "=" E1.place "and" E2.place); }
$E \rightarrow \text{not } E1$	{ E.place = newtemp; emit(E.place "=" "not" E1.place); }
$E \rightarrow (E1)$	{ E.place = E1.place; }
$E \rightarrow id1 \text{ relop } id2$	{ E.place = newtemp; emit("if" id1.place relop.op id2.place "goto" nextstat + 3); emit(E.place "=" "0"); emit("goto" nextstat+2); emit(E.place "=" "1"); }

$E \rightarrow \text{true}$	{ E.place = newtemp; emit(E.place “=” “1”); }
$E \rightarrow \text{false}$	{ E.place = newtemp; emit(E.place “=” “0”); }

Figura 5.6. Esquema de tradução de expressões lógicas usando representação numérica.

Exercício: Gere o CDE para a expressão $A < B \text{ or } C < D \text{ and } E < F$, usando o esquema de tradução da Figura 5.6.

5.1.5.2. Expressões Lógicas Traduzidas como Fluxo de Controle

O esquema da Figura 5.7 inclui ações semânticas para traduzir expressões lógicas como instruções de fluxo de controle do tipo if-goto. No esquema, os atributos herdados *true* e *false* servem para armazenar rótulos que são gerados pela função *newlabel*. O atributo sintetizado *code* conterá, no final da tradução, o código gerado para a expressão analisada.

$E \rightarrow E1 \text{ or } E2$	{ E1.true = E.true; E1.false = newlabel; E2.true = E.true; E2.false = E.false; E.code = E1.code gen(E1.false “:”) E2.code; }
$E \rightarrow E1 \text{ and } E2$	{ E1.true = newlabel; E1.false = E.false; E2.true = E.true; E2.false = E.false; E.code = E1.code gen(E1.true “:”) E2.code; }
$E \rightarrow \text{not } E1$	{ E1.true = E.false; E1.false = E.true; E.code = E1.code; }
$E \rightarrow (E1)$	{ E1.true = E.true; E1.false = E.false; E.code = E1.code; }
$E \rightarrow \text{id1 relop id2}$	{ E.code = gen(“if” id1.place relop.op id2.place “goto” E.true) gen(“goto” E.false); }
$E \rightarrow \text{true}$	{ E.code = gen(“goto” E.true); }
$E \rightarrow \text{false}$	{ E.code = gen(“goto” E.false); }

Figura 5.7. Esquema de tradução para CDE’s de expressões booleanas.

Exercício: Gere o CDE para a expressão $A < B \text{ or } C < D \text{ and } E < F$, usando o esquema de tradução da Figura 5.7. Suponha que para os valores true e false da expressão tenham sido atribuídos os rótulos Ltrue e Lfalse.

5.1.6. Geração de CDE's para Comandos de Controle

O esquema a seguir inclui ações semânticas para traduzir comandos de fluxo de controle do tipo if-then-else, if-then e while-do.

```
S → if E then S1          { E.true = newlabel;
                           E.false = S.next;
                           S1.next = S.next;
                           S.code = E.code || gen(E.true “:”) || S1.code; }
S → if E then S1 else S2  { E.true = newlabel;
                           E.false = newlabel;
                           S1.next = S.next;
                           S2.next = S.next;
                           S.code = E.code || gen(E.true “:”) || S1.code ||
                               gen(“goto” S.next) || gen(E.false “:”) ||
                               S2.code; }
S → while E do S1        { S.begin = newlabel;
                           E.true = newlabel;
                           E.false = S.next;
                           S1.next = S.begin;
                           S.code = gen(S.begin “:”) || E.code || gen(E.true “:”) ||
                               S1.code || gen(“goto” S.begin); }
```

Para o comando

```
while A < B do
  if C < D
    then X = Y + Z
    else X = Y - Z;
```

a seguinte seqüência de comandos é gerada:

```
L1:  if A < B goto L2
      goto Lnext
L2:  if C < D goto L3
      goto L4
L3:  t1 = Y + Z
      X = t1
      goto L1
L4:  t2 = Y - Z
      X = t2
      goto L1
Lnext: ...
```

5.1.7. Backpatching

O maior problema na geração de código para expressões lógicas e comandos de controle em um único passo é o desconhecimento dos endereços no código gerado para os quais os o controle deve ser passado quando da geração de comandos de desvio condicional e incondicional. Este problema pode ser solucionado criando-se listas de comandos de desvio (cojos endereços não tenham sido resolvidos) que são devidamente completados quando o endereço de destino for conhecido (*backpatching*). São necessárias três funções para o processo de backpatching:

makelist(i): cria uma lista contendo i, índice do vetor de quádruplas, retornado um ponteiro para a lista criada.

merge(p1,p2): concatena as listas apontadas por p1 e p2, e retorna um ponteiro para a lista resultante.

backpatch(p,i): insere i como rótulo destino para cada um dos comandos da lista apontada por p.

O esquema de tradução da Figura 5.8 produz quádruplas a partir de expressões lógicas, durante a análise redutiva (*bottom-up*). Foi acrescentado à gramática de geração de expressões lógicas o símbolo não-terminal M, que tem como objetivo guardar o endereço da próxima quádrupla disponível no momento que M é empilhado.

$E \rightarrow E1 \text{ or } M E2$	{ backpatch(E1.falselist,M.quad); E.truelist = merge(E1.truelist,E2.truelist); E.falselist = E2.falselist; }
$E \rightarrow E1 \text{ and } M E2$	{ backpatch(E1.truelist,M.quad); E.truelist = E2.truelist; E.falselist = merge(E1.falselist,E2.falselist); }
$E \rightarrow \text{not } E1$	{ E.truelist = E1.falselist; E.falselist = E1.truelist; }
$E \rightarrow (E1)$	{ E.truelist = E1.truelist; E.falselist = E1.falselist; }
$E \rightarrow \text{id1 rel op id2}$	{ E.true = makelist(nextquad); E.false = makelist(nextquad + 1); emit("if" id1.place rel op id2.place "goto _"); emit("goto _"); }
$E \rightarrow \text{true}$	{ E.truelist = makelist(nextquad); emit("goto _"); }
$E \rightarrow \text{false}$	{ E.falselist = makelist(nextquad); emit("goto _"); }
$M \rightarrow \epsilon$	{ M.quad = nextquad; }

Figura 5.8. Backpatching para expressões lógicas.

A seguir é apresentado um esquema de tradução com backpatching em comandos de controle. Dois símbolos não-terminais M e N, que geram produções vazias, foram introduzidos a fim de gerar informação para a produção do código intermediário. M, através do atributo m.quad, registra o índice da próxima quádrupla disponível. N serve para marcar o fim do bloco *then* em comandos *if-then-else* e gera o *goto* sobre o comando *else*. O atributo *next*, associado a *S* e *L* é um ponteiro para uma lista de desvios para a quádrupla que segue o comando/lista de comandos representado por *S* ou *L*.

```

S → if E then M S1      { backpatch(E.truelist,M.quad);
                        S.nextlist = merge(E.falselist, S1.nextlist); }
S → if E then M1 S1 N else M2 S2
                        { backpatch(E.truelist,M1.quad);
                          backpatch(E.falselist,M2.quad);
                          S.nextlist = merge(S1.nextlist,
                                              merge(N.nextlist,S2.nextlist)); }
N → ε                  { N.nextlist = makelist(nextquad);
                        emit("goto _"); }
M → ε                  { M.quad = nextquad; }
S → while M1 E do M2 S1 { backpatch(S1.nextlist,M1.quad);
                          backpatch(E.truelist,M2.quad);
                          S.nextlist = E.falselist;
                          emit("goto" M1.quad); }
S → begin L end       { S.nextlist = L.nextlist; }
S → A                  { S.nextlist = nil; }
L → L1; M S           { backpatch(L1.nextlist,M.quad);
                        L.nextlist = S.nextlist; }
L → S                  { L.nextlist = S.nextlist; }

```

Geração de Código Intermediário para Expressões Booleanas – Fluxo de Controle

De acordo com a estratégia de geração pelo fluxo de controle, cada expressão é traduzida em uma seqüência de comandos de três endereços, que se constituem em comandos de desvio condicional ou incondicional a um de dois rótulos: *E.true*, aonde a expressão deve levar, caso seja verdadeira e *E.false*, caso a expressão seja falsa. A idéia básica é a seguinte: supondo que E seja $a < b$, o código gerado seria o seguinte:

```

if a < b goto E.true
goto E.false

```

Supondo que E seja *E1 or E2*. Se *E1* é verdadeiro, então sabe-se imediatamente que E é verdadeiro, então $E1.true = E.true$. Se *E1* é falso, então *E2* deve ser avaliado, usando o label *E1.false* par o primeiro comando do código de *E2*. As saídas *true* e *false* de *E2* podem ser apresentadas da mesma maneira das saídas *true* e *false* de E, respectivamente.

De forma análoga, pode-se considerar a tradução da expressão *E1 and E2*. Nenhum código é necessário para a expressão *not E1*, bastando apenas trocar entre si as saídas *E1.false* e *E2.false*.

Considerando a expressão $A < B \text{ or } C < D \text{ and } E < F$, e supondo que as saídas *true* e *false* da expressão estão identificadas pelos rótulos *Ltrue* e *Lfalse*. Utilizando a definição apresentada na apostila, tem-se o seguinte código:

```

if a < b goto Ltrue
goto L1
L1:  if c < d goto L2
      goto Lfalse
L2:  if e < f goto Ltrue
      goto Lfalse.

```

Para o seguinte código fonte:

```
while a < b do
  if c < d then
    x = y + z
  else
    x = y - z
```

tem-se o seguinte código intermediário gerado:

```
L1:  if a < b goto L2
      goto Lnext
L2:  if c < d goto L3
      goto L4
L3:  t1 = y + z
      x = t1
L4:  t2 = y - z
      x = t2
      goto L1
Lnext:
```

LISTA DE EXERCÍCIOS - GRAU A

1. Enumere vantagens e desvantagens de interpretadores em relação a compiladores:
2. Defina os seguintes termos: compilador, interpretador, montador e pré-processor:
3. Enumere e explique cada uma das fases do processo de compilação:
4. Defina um autômato finito que reconheça constantes numéricas em Pascal, cujo formato é o seguinte: [+|-] n [.n] [E [+|-] n], onde n é uma seqüência de um ou mais dígitos:
5. Identifique os tokens do seguinte programa em C, identificando sua classe, valor e posição no código fonte. Assuma que o código inicia na linha 1 e na coluna 1:

```
int max(i,j)
int i,j;
/* Retorna o maior dos inteiros i e j. */
{
return i>j?i:j;
}
```

6. Defina expressões regulares para descrever as linguagens explicadas a seguir, construindo um arquivo de especificação do LEX:
 - a) palavras formadas por letras, contendo as cinco vogais, na ordem alfabética (não necessariamente seqüencialmente);
 - b) Comentários delimitados por /* e */, sem a ocorrência dos símbolos */, a menos que estes apareçam entre aspas: "*/";
 - c) palavras formadas pelas letras "a" e "b", com um número par de letras "a" e ímpar de letras "b";

7. Cite exemplos de regras, que poderiam estar inseridas em uma especificação LEX, com ambigüidade. Que decisões são tomadas, pelos analisadores léxicos gerados pelo LEX?

8. Considere a seguinte gramática:

$$S \rightarrow (L) / a$$

$$L \rightarrow L, S / S$$

- a) Quais são os terminais, não-terminais e símbolo inicial da gramática?
 b) Construa árvores de derivação para as seguintes entradas, considerando a derivação mais à esquerda:
 (a,a)
 (a, (a,a))
 (a, ((a,a), (a,a)))

9. A gramática: $S \rightarrow aSa / aa$ gera palavras com números pares de letras "a", com exceção da palavra vazia. Construa um analisador descendente recursivo com *backtracking* para esta gramática, que tente a alternativa "aSa" antes de "aa". Explique porque o procedimento tem sucesso no reconhecimento de 2, 4 ou 8 letras "a", mas falha para o reconhecimento de 6 letras "a".

10. Pense na seguinte afirmação, corrija-a caso necessário e justifique-a: "*gramáticas sem produções vazias, na qual cada alternativa inicia com um terminal distinto, são sempre LL(1)*".

11. A partir da seguinte gramática, realize as seguintes tarefas:

$$S \rightarrow \sim S \mid [S] \mid S > S \mid p \mid q$$

- a) Elimine a recursividade à esquerda da gramática.
 b) Escreva um analisador descendente recursivo preditivo que reconheça L(G).
 c) A gramática é LL(1)? Justifique. Construa uma tabela LL(1) para reconhecer L(G).

12. Construa uma tabela LL(1) para a seguinte gramática:

$$E \rightarrow -E \mid (E) \mid V L$$

$$L \rightarrow -E \mid \epsilon$$

$$V \rightarrow id S$$

$$S \rightarrow (E) \mid \epsilon$$

13. Quais das gramáticas abaixo são LL(1)? Justifique:

$S \rightarrow A B c$	$S \rightarrow A b$	$S \rightarrow A B B A$	$S \rightarrow a S e \mid B$
$A \rightarrow a \mid \epsilon$	$A \rightarrow a \mid B \mid \epsilon$	$A \rightarrow a \mid \epsilon$	$B \rightarrow b B e \mid C$
$B \rightarrow b \mid \epsilon$	$B \rightarrow b \mid \epsilon$	$B \rightarrow b \mid \epsilon$	$C \rightarrow c B e \mid d$

14. Fatore à esquerda a gramática apresentada na questão 12, e escreva um analisador preditivo recursivo para o reconhecimento da linguagem por ela especificada:

15. Considerando a seguinte gramática:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

Apresente os passos realizados pelo analisador preditivo não-recursivo para o reconhecimento das entradas: $(id+id*(id+id) + (id*id))$ e $(id+id)*(id+id)$. Considere, para isto, a tabela de derivação preditiva apresentada na página 29 da apostila (Tabela 3.1).

16. Considerando a seguinte gramática:

$E \rightarrow E + E \mid E * E \mid (E) \mid id$

Apresente os passos realizados pelo analisador *bottom-up* (do tipo *shift-reduce*) para o reconhecimento das entradas: $(id+id*(id+id) + (id*id))$ e $(id+id)*(id+id)$.

17. Explique e exemplifique os conflitos dos tipos *shift-reduce* e *reduce-reduce*, que podem ocorrer na geração de analisadores sintáticos feita pelo YACC. Que tipo de modificações poderiam ser feitas na gramática para evitar tais tipos de conflitos?

18. Os exemplos (a) e (b) de especificação para LEX e YACC convertem números romanos para números arábicos. Entretanto, ambos utilizam processos diferentes para tal. Analise cada um dos exemplos, e explique os processos empregados por eles. Quais as vantagens de uma abordagem em relação à outra?

Obs.: Assuma que a constante GLYPH foi definida em outro arquivo, representando um token qualquer utilizado na representação de um número romano.

(a)

```
/****** ROM.L *****/
I      { yylval= 1; return GLYPH; }
IV     { yylval= 4; return GLYPH; }
V      { yylval= 5; return GLYPH; }
IX     { yylval= 9; return GLYPH; }
X      { yylval= 10; return GLYPH; }
XL     { yylval= 40; return GLYPH; }
L      { yylval= 50; return GLYPH; }
XC     { yylval= 90; return GLYPH; }
C      { yylval=100; return GLYPH; }
```

```
-----
/****** ROM.Y *****/
num : GLYPH      { $$=$1; }
    | num GLYPH  { $$=$1+$2; };
```

(b)

```
/****** ROM.L *****/
I      { yylval= 1; return GLYPH; }
V      { yylval= 5; return GLYPH; }
X      { yylval= 10; return GLYPH; }
L      { yylval= 50; return GLYPH; }
C      { yylval=100; return GLYPH; }
```

```
-----
/****** ROM.Y *****/
{% int last=0; %}
%%
num : GLYPH      { $$=last=$1; }
    | GLYPH num  { if ($1>=last)
                  $$=$2+(last=$1);
                  else
                  $$=$2-(last=$1); };
```

