Notas de Aula de Compiladores

Luiz Eduardo da Silva

4 de novembro de 2022

Sumário

Capítulo 1

Introdução

Com os computadores surgiu a necessidade de linguagens de programação para quebrar a barreira de comunicação entre o homem e a máquina. As primeiras linguagens de programação foram chamadas linguagens de montagem (assembly languages). [Aho et al. 2008, Ziviani 1999, Price 2008, Louden 2004, Kowaltowski 1983]

Na década de 50 surgiram as linguagens de alto nível FORTRAN e Algol 60. Desde então surgiram centenas de linguagens para facilitar a especificação de tarefas para o computador.

Com a introdução das linguagens de alto nível, surgiu a necessidade de programas tradutores, ou seja, sistemas para converter programas fonte (escritos em linguagem que pode ser entendida pelo usuário humano) para programa objeto (em linguagem que pode ser entendida pela máquina.

Figura 1.1: Processo de tradução

Um processo típico de tradução está ilustrado na figura 1.1. Vale observar que o compilador faz uma tradução "offline", i. e., o compilador traduz todo o programa e só depois dessa tradução o programa pode ser executado. O interpretador faz a tradução "online", onde partes do programa são traduzidas para serem executada imediatamente.

1.1 Aspectos básicos da compilação

Existem três aspectos fundamentais com relação a implementação de linguagens de programação:

- **Aspecto sintático** – identificação das seqüências de caracteres que correspondem a construções permitidas da linguagem.
- **Aspecto semântico** – o significado de cada construção válida.
- **Aspecto pragmático** – corresponde ao problema de integração do compilador para um sistema hospedeiro.

Os problemas sintáticos estão formalizados e praticamente resolvidos. Os problemas semânticos não estão formalizados mas dependendo da linguagem não são difíceis de resolver. O aspecto pragmático é o que apresenta mais variabilidade nos diversos sistemas de computação. Há grande diferença nas arquiteturas básicas.

Por não poder tratar com generalidade os problemas de compilação, usaremos um subconjunto da linguagem pascal para centralizar nossa discussão.

A compilação pode ser dividida nas seguintes fases:

- **Fase Análise Léxica:** Transforma seqüência de caracteres do programa em códigos internos (tokens)
- **Fase Análise Sintática:** Verifica se uma frase no programa fonte é uma construção permitida da linguagem.
- **Fase Análise Semântica:** Extrai da estrutura sintática informações que serão necessárias na fase de síntese.
- **Otimização Global:** Introduz melhorias que independem da linguagem de máquina.
- **Otimização Local:** Introduz melhorias no programa objeto, aproveitando o repertório de instruções de um dado computador.

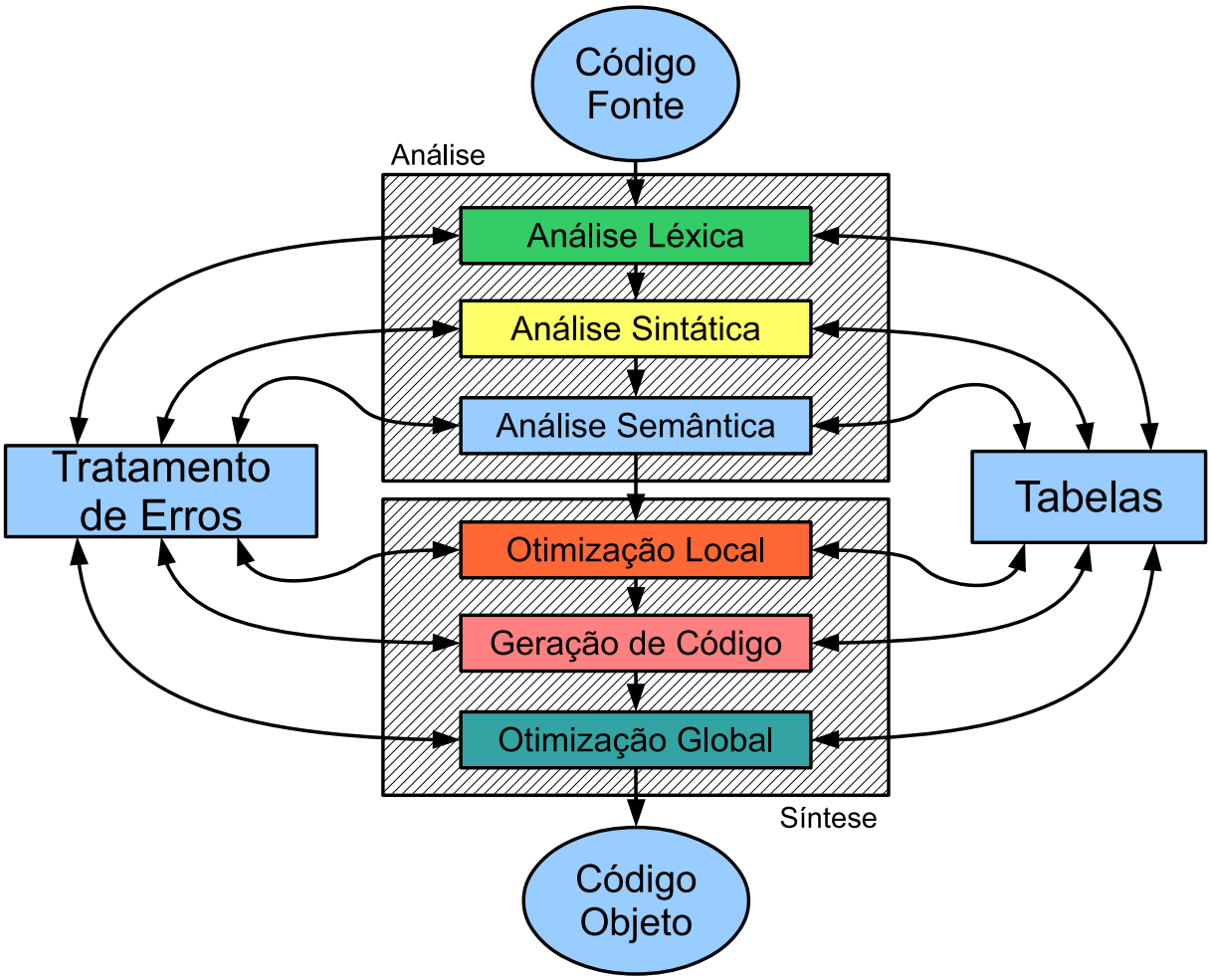

Figura 1.2: As fases da compilação

O funcionamento básico do Compilador, com suas diversas fases de compilação, está ilustrado na Figura 1.2.

Durante a análise do programa são encontrados erros (léxicos, sintáticos, semânticos), que devem ser tratados pelo compilador. As várias fases de compilação podem ser executadas em sequência (compilação em vários passos) ou ter a sua execução combinada (compilação num único passo). Na compilação em vários passos a execução de uma fase termina antes da execução das fases seguintes. Numa compilação em um passo, o programa objeto é gerado à medida que o programa fonte é processado.

Para compilação numa única passada pode-se utilizar o esquema de tradução dirigido por sintaxe. Nesse modelo, o código principal do compilador está no analisador sintático. No meio das regras de produção da gramática do analisador sintático são incluídos comandos (ações semânticas) que são executados concomitantes com as verificações sintáticas. Essas ações servem tanto para as verificações semânticas, como para geração do código objeto correspondente as estruturas sintáticas compiladas.

No projeto do compilador proposto nesse curso usaremos esta última alternativa de compilação com auxílio de algumas ferramentas (*flex* e *bison*) para automatizar a construção dos analisadores léxico e sintático.

Para a construção do analisador léxico usaremos a ferramenta *flex*, que transforma os padrões léxicos especificados usando expressões regulares, no código fonte de uma autômato finito que reconhece esses padrões.

Para construção do analisador sintático usaremos a ferramenta *bison*, que transforma o arquivo fonte com a gramática livre de contexto da linguagem, no autômato de pilha que reconhece essa linguagem.

1.2 Análise Léxica

A análise léxica é executada pelo Analisador Léxico ou *scanner*, e tem como objetivo principal identificar e classificar, dentro do arquivo fonte, os símbolos pertencentes a linguagem. Suas funções são:

1. Fazer uma varredura no texto em todas as linhas do texto, de cima para baixo e da esquerda para direita.

2. Agrupar os caracteres consecutivos que compõe o mesmo símbolo léxico e que tem uma significado único no programa, como as palavras reservadas, constantes, identificadores, símbolos especiais, operadores.

3. Atribuir um código numérico *token* para os símbolos léxicos encontrados. A codificação numérica simplifica a comparação dos símbolos nas outras fases da compilação.

4. Excluir os espaços em branco e os comentários do programa fonte.

5. Detectar erros léxicos, i. e., símbolos que não fazem parte do vocabulário da linguagem:

   - caracteres inválidos: ex: $a := 2\#3$
   - tamanhos dos identificadores, literais e constantes (overflow)

Exemplo: Para o Código 1.1 o resultado obtido pelo analisador léxico é apresentado na Tabela 1.1.

Item	Token	Descrição
1	program	palavra reservada
2	exemplo	identificador
3	;	símbolo ponto e vírgula
4	var	palavra reservada
5	A	identificador
6	,	símbolo vírgula
7	B	identificador
8	:	símbolo dois pontos
9	byte	identificador
10	;	símbolo ponto e vírgula
11	begin	palavra reservada
12	A	identificador
13	:=	símbolo atribuição
14	B	identificador
15	+	símbolo mais
16	0.5	número
17	;	símbolo ponto e vírgula
18	end	palavra reservada
19	.	símbolo ponto final

Tabela 1.1: Resultado da análise léxica do Código 1.1

Listing 1.1: Programa Exemplo em Pascal

```
1 program exemplo ;
2 var     A , B : byte ;
3 begin
4     A := B + 0.5 ;
5 end .
```

Para o exemplo anterior, temos os *tokens* identificados por uma analisador léxico, conforme a Tabela 1.1. Observe que um símbolo léxico pode ser simples como uma vírgula (,) ou composto como uma palavra (program) ou uma sequência de símbolos (:=).

1.3 Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados fundamental para a eficiência do compilador. Na tabela de símbolos ficam armazenados as informações relacionadas aos identificadores (nomes das entidades do programa), como variáveis, rotinas, constantes, tipos. Durante todas as fases da compilação a tabela de símbolos é atualizada e consultada. O acesso dos símbolos do compilador deve ser o mais eficiente possível para garantir a eficiência do compilador. Uma estrutura comumente usada para manutenção da tabela de símbolos é a Tabela Hash.

A Tabela 1.2 é um exemplo de uma tabela de símbolos simples, com as informações relacionadas as variáveis de um programa, por exemplo.

Nome	Tipo	Endereço
A	int	0
B	int	1

Tabela 1.2: Exemplo de tabela de símbolos

1.4 Análise Sintática

A análise sintática é executada pelo *parser* e sua função principal é agrupar os *tokens*, retornados do analisar léxico, em estruturas sintáticas (comando, bloco, expressão, identicador, número, etc). Uma estrutura que pode ser empregada é a árvore sintática. A árvore representa a aplicação das regras sintáticas da linguagem e definem, de certa forma, um significado para a estrutura do programa compilado. O programa está sintaticamente correto se for possível construir uma única árvore sintática, no qual a raiz é o símbolo inicial da gramática da linguagem (símbolo de partida) e as folhas são os tokens retornados do analisado léxico.

A Figura 1.3 representa a árvore sintática para o código 1.2.

Listing 1.2: Programa calcula maior

```
1  programa maior
2    inteiro a b
3  inicio
4    leia a
5    leia b
6    se a > b
7       entao escreva a
8       senao escreva b
9    fimse
10 fimprograma
```

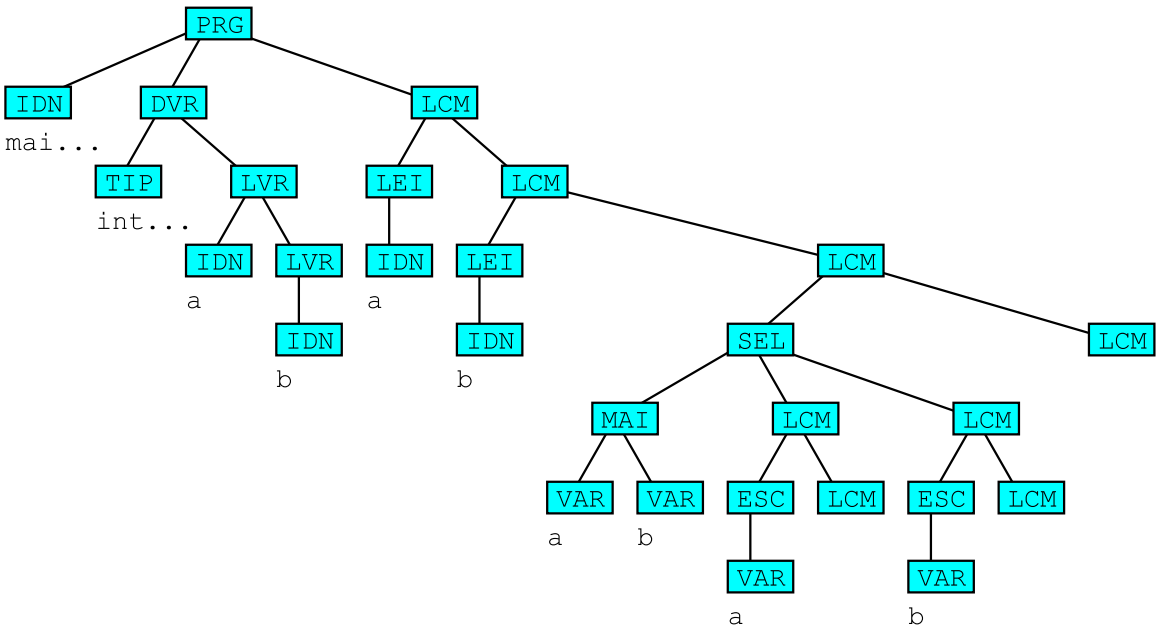

Figura 1.3: Árvore sintática do programa calcula maior

1.5 Análise Semântica

O análise semântica é a fase da compilação em que é extraído o significado das estruturas sintáticas. É a partir dessa significação é que é possível a tradução (geração) e otimização de código. Na análise semântica são verificados também aspectos semânticos do código com:

1. Compatibilidade de tipos. Por exemplo, para algumas linguagens, o tipo da expressão no lado direito de uma atribuição deve ser compatível com o tipo da variável no lado esquerdo da atribuição.

2. Verificação do aspecto de visibilidade (escopo) de uma variável. Por uma questão lógica, não é permitido duas variáveis com o mesmo nome, no mesmo escopo do código.

3. Compatibilidades entre declarações e uso de entidades. Por exemplo, não faz sentido, declarar um vetor e tentar usar como se fosse um registro.

4. Referências não resolvidas. Todo nome de entidade deve ter uma correspondência no código. Por exemplo, o nome de um rótulo de desvio deve marcar alguma posição no código.

1.6 Tratamento de erros

Identificado um erro em qualquer fase da compilação, o módulo de tratamento de erro do compilador é invocado. O tratador de erro deve diagnosticar o erro e emitir uma mensagem o coerente possível para explicar a situação de erro. A partir dessa mensagem duas estratégias podem ser adotadas:

- Pânico: abortar a compilação. É a estratégia mais simples de implementar.
- Recuperação: tentar recuperar a situação de erro, permitindo que todo o programa fonte possa ser analisado, mesmo na ocorrência do erro.

1.7 Geração de código intermediário

Para simplificar o processo de tradução, pode-se empregar uma codificação intermediária. Por exemplo, para tradução de expressões aritméticas pode-se gerar tuplas. Exemplo:

Listing 1.3: Atribuição

```
1   A := (B - C) * (D + E)
```

gera-se quádruplas:

```
(+ , B , C , T1)
(- , D , E , T2)
(* , T1 , T2 , A)
```

1.8 Otimização geral

A principal função da otimização do código é melhorar o código a fim de que ocupe menos espaço (otimização de espaço) ou que execute mais rápido (otimização de execução).

As principais otimização realizadas são:

1. Agrupamento das subexpressões comuns numa expressão aritmética. Exemplo, considerando as quádruplas que representam operações aritméticas, o cálculo A + B na expressão C := (A+B) * (A+B), pode ser calculada uma única vez e utilizado duas vezes na expressão.

2. Eliminação de saltos (jumps) desnecessários no código. Na tradução de estruturas de repetição e de seleção, acontecem situações de jumps encadeados no código. Essa fase pode eliminar esse problema.

3. Realocação de comandos invariantes dentro de uma repetição. Exemplo: a atribuição J := X pode ser colocada fora de uma repetição sem alterar o programa:

Listing 1.4: Atribuição invariante

```
1          i := 1;
2          repeat
3              J := X;
4              i := i + 1;
5          until i >= 100;
```

1.9 Otimização local

Essa otimização leva em consideração o conjunto específico de instruções da máquina alvo que pode deixar o código mais rápido. É uma fase fortemente dependente da máquina para qual está sendo gerado o código traduzido.

1.10 Geração de código objeto

Nessa fase é gerado o código objeto correspondente a cada estrutura sintática identificada no programa.

Capítulo 2

Análise Léxica

A análise léxica é a fase do compilador responsável por fazer a leitura do texto (programa fonte) caracter por caracter e traduzir os símbolos léxicos em *tokens*.

Os símbolos léxicos são as palavras reservadas, os operadores, os identificadores, as constantes literais, as constantes numéricas, os comentários, os espaços em branco e as tabulações. Os comentários, os espaços em branco e as tabulações ajudam na documentação e identação do código fonte, mas são excluídos do programa, pelo analisador léxico. A identação aparece na sintaxe de algumas linguagens, mas os comentários não são utilizados nas outras fases da compilação.

Os identificadores reconhecidos nessa fase da compilação devem ser cadastrados e consultados da tabela de símbolos. Essa verificação e cadastramento é realizado a partir do reconhecimento desses símbolos léxicos.

Os *tokens* do analisador léxico são utilizados com entrada para a fase da análise sintática, quando então esses *tokens* são agrupados em estruturas sintáticas, seguindo as regras de produção da gramática da linguagem de programação.

Para a fase de análise léxica, especificamos os tokens a serem reconhecidos usando uma gramática regular. Especificamente para as ferramentas que geram automaticamente os analisadores léxicos, como a ferramenta *flex*, os tokens são definidos através de expressões regulares. A partir dessas expressões regulares a ferramenta gera o autômato finito para o seu reconhecimento, na forma de uma programa de reconhecimento.

Um processo similar é empregado para construção do analisador sintático, que veremos no próximo capítulo. A sintaxe da linguagem de programação é especificada, para as ferramentas que geram automaticamente analisadores sintáticos, através de gramáticas livres de contexto (GLC). O utilitário *bison*, traduz GLC no autômato de pilha que é o analisador sintático da linguagem.

Na próxima seção serão revistos alguns conceitos relacionados a gramáticas, linguagens, autômatos apresentados na disciplina de Linguagens Formais e Autômatos e que serão utilizados nas fases de análise léxica e sintática do projeto do compilador.

2.1 Revisão de teoria de linguagens

Apresentamos nessa seção uma série de conceitos relacionados a linguagens formais e autômatos e que são úteis para o melhor entendimento do projeto do compilador. Um detalhamento desses temas pode ser encontrado em diversos títulos que tratam especificamente sobre o formalismo de linguagens e autômatos [Sipser 2012, Vieira 2006, J.L.Gersting 2004, Lewis e Papadimitriou 2004].

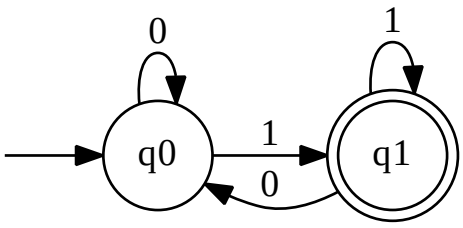

Figura 2.1: Autômato de Estados Finitos Determinístico.

2.1.1 Alfabeto (ou vocabulário)

O alfabeto é um conjunto finito e não vazio de símbolos que podem ser simples como dígitos, letras e caracteres especiais, ou então compostos, como begin, :=, que serão tratados, em geral, como símbolos indivisíveis (*tokens*). Seja Σ um alfabeto. Uma palavra (cadeia ou sentença) sobre Σ é uma sequência finita de símbolos de Σ. Σ^* é o conjunto de todas as palavras sobre Σ.

2.1.2 Linguagens

Uma linguagem sobre um alfabeto é um subconjunto de Σ^*, formalmente, podemos escrever $L \subseteq \{\alpha \in \Sigma^*\}$. São exemplos de linguagem sobre o alfabeto $\{a, b\}$, $L_1 = \{\alpha \in \{a, b\}^*$ tal que $|\alpha| \leq 2\}$ e $L_2 = \{a^n b^n | n \geq 1\}$. Considerando todos os símbolos léxicos (identificadores, palavras reservadas, operadores, etc.) que podem ser combinados das mais diversas formas, como em: a var b integer : ;, a linguagem determina, através de regras de sintaxe, a forma correta como esses símbolos devem ser encadeados, como em: var a, b : integer;.

2.1.3 Autômatos Finitos

Autômatos finitos são máquinas abstratas usadas para representar algumas características de uma máquina concreta. São representadas usando estados e transições entre os estados. Como o número de estados é finito é denominada também como uma máquina estados finitos.

Uma representação de um autômato está na Figura 2.1. Onde os estados são os círculos e as transições são as setas rotuladas com 0 e 1. O estado marcado por uma seta é o estado inicial e o(s) estados(s) com dois círculos aninhados representa(m) o(s) estados(s) final(is).

Para representar a computação do autômato podemos usar duas definições: a) a configuração instantânea, representada pelo par $[e, w]$, onde e é o estado atual e w a palavra a ser computada num dado instante; b) a relação $\vdash$ que mostra a transformação da configuração instantânea durante a computação da palavra w. Então $[q_0, aw] \vdash [q_1, w]$ se existe uma transição de q_0 para q_1 para o símbolo a no autômato.

Considerando o autômato da Figura 2.1 e a palavra $w = 001$, tem-se:

$$[q_0, 001] \vdash [q_0, 01] \vdash [q_0, 1] \vdash [q_1, \lambda]$$

Os símbolos λ ou ϵ são usados para representar uma palavra vazia. Dizemos que uma palavra é reconhecida por uma autômato se, começando do estado inicial, processamos todos os símbolos da palavra e terminamos no estado final.

A linguagem L reconhecida por um autômato A pode ser definida como:

$$L(A) = \{w \in \Sigma^* | [i, w] \overset{*}{\vdash} [f, \lambda]\}$$

onde:

- w é a palavra da linguagem
- i é o estado inicial do autômato
- f é um estado final do autômato

2.1.4 Autômatos Finitos Determinísticos

O Autômato Finito Determinístico (AFD) é matematicamente definido como:

Definição 1 *Um AFD é uma quíntupla $(E, \Sigma, \delta, i, F)$, onde:*

- *E é o conjunto finito dos estados do autônomo;*
- *Σ é um alfabeto;*
- *$\delta : E \times \Sigma \to E$ é a função de transição;*
- *$i \in E$ é o estado inicial;*
- *$F \subseteq E$ é o conjunto de estados finais.*

Usando essa definição podemos descrever formalmente a máquina da Figura 2.1 como $A_1 = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$, onde δ é descrita pela Tabela 2.1.

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_1

Tabela 2.1: Função de Transição δ

Para descrever a computação da máquina de maneira formal, usamos a definição da função de transição estendida:

Definição 2 *Seja um AFD $A = (E, \Sigma, \delta, i, F)$. A função de transição estendida para A, $\hat{\delta} : E \times \Sigma^* \to E$, definida recursivamente como:*

- $\hat{\delta}(a, \lambda) = a$
- $\hat{\delta}(e, ay) = \hat{\delta}(\delta(e, a), y)$, *para todo $a \in \Sigma$ e $y \in \Sigma^*$*

Então para $w = 001$ e a máquina A_1 temos:

$$\begin{aligned}
\hat{\delta}(q_0, 001) &= \hat{\delta}(\delta(q_0, 0), 01) \\
&= \hat{\delta}(q_0, 01) \\
&= \hat{\delta}(\delta(q_0, 0), 1) \\
&= \hat{\delta}(q_0, 1) \\
&= \hat{\delta}(\delta(q_0, 1), \lambda) \\
&= \hat{\delta}(q_1, \lambda) \\
&= q_1
\end{aligned}$$

Essa computação também pode ser implementada de forma bastante simples através de uma repetição que consulta a Tabela de Transição T, representada na Tabela 2.1, onde as linhas representam estados e as colunas os símbolos do vocabulário:

```
1 i = 0;
2 estado = 0;
3 while (entrada[i]) {
4     estado = T[estado, entrada[i]];
5     i++;
6 }
```

A partir da função de transição estendida, podemos definir a linguagem L reconhecida por uma autômato finito determinístico A por:

$$L(A) = \{w \in \Sigma^* | \hat{\delta}(i, w) \in F\}$$

2.1.5 Autômatos Finitos Não-Determinísticos

Um outro formalismo é o Autômato Finito Não-Determinístico (AFN) que não aumenta o poder de especificação do AFD, mas pode simplificar a definição de algumas máquinas.

O indeterminismo está representado em mais de um estado inicial, em mais de uma transicão possível para o mesmo estado e símbolo e também pela possibilidade de efetuar a transição de estados sem consumir símbolos da entrada através de transições λ.

Definição 3 *Um AFN é uma quíntupla* $(E, \Sigma, \delta, I, F)$*, onde:*

- E *é o conjunto finito dos estados do autômato;*
- Σ *é um alfabeto;*
- $\delta : E \times \Sigma_\lambda \rightarrow P(E)$ *é a função de transição;*
- $I \subseteq E$ *é o conjunto não vazio de estados iniciais;*
- $F \subseteq E$ *é o conjunto de estados finais.*

A computação no AFN é um pouco diferente, uma vez que precisamos simular a execução em paralelo das múltiplas escolhas de transição que podem ser realizadas no percurso do autômato. A palavra é reconhecida se no conjunto de possibilidades de computação, existe uma que consuma toda entrada e termina num estado final.

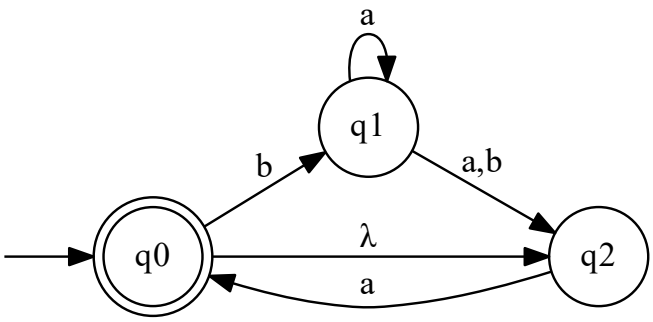

Figura 2.2: Autômato Finito Não-Determinístico

2.1.6 Expressões Regulares

Um outro formalismo que pode ser utilizado para definição de linguagens são as Expressões Regulares (ER). Assim como as expressões aritméticas tem operadores (soma, multiplicação, divisão, ...) que são utilizados para expressar valores numéricos, as **expressões regulares** usam os operadores de concatenação, união (representado pelo símbolo +) e fecho de Kleene (representado pelo símbolo *) para definir linguagens regulares.

Definição 4 *Sejam A e B linguagens. As operações regulares de União, Concatenação e Fecho de Kleene são definidas da seguinte forma:*

- *União:* $A \cup B = \{x | x \in A \text{ ou } x \in B\}$.
- *Concatenação:* $AB = \{xy | x \in A \text{ e } y \in B\}$.
- *Fecho de Kleene:* $A^* = \{x_1 x_2 ... x_k | k \geq 0 \text{ e } x_i \in A\}$.

Definição 5 *Uma ER sobre uma alfabeto Σ é definida recursivamente como:*

1. *$\emptyset$, λ e a para qualquer $a \in \Sigma$ são expressões regulares; tais que ER's denotam, respectivamente, os conjuntos $\emptyset$, $\{\lambda\}$ e $\{a\}$*
2. *Se r e s são expressões regulares, então também são expressões regulares $(r + s)$, (rs) e r^*; tais ER's denotam, respectivamente, $L(r) \cup L(s)$, $L(r)L(s)$ e $L(r)^*$.*

Alguns exemplos de ER considerando o alfabeto $\Sigma = \{a, b\}$:

1. a*ba* = $\{w \in \Sigma^* | w$ contém um único b$\}$
2. (a + λ)b = $\{w \in \Sigma^* | w$ começa com a ou nada seguido de b$\}$
3. a*b = $\{w \in \Sigma^* | w$ contém zero ou mais a´s finalizado por b$\}$
4. a(a+b)*a+a = $\{w \in \Sigma^* | w$ começa e termina com a$\}$

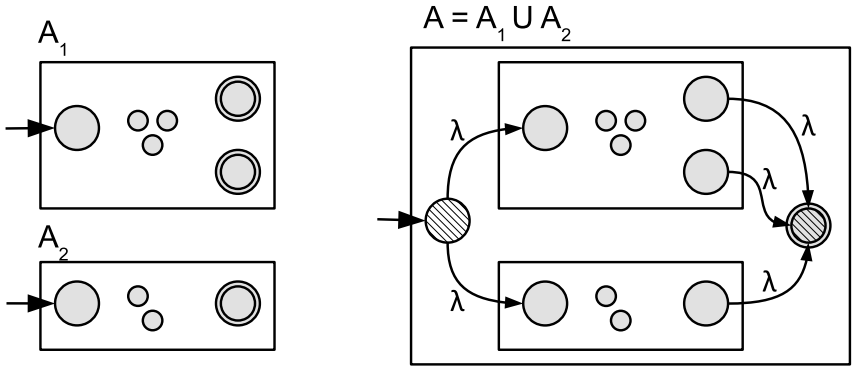

Figura 2.3: Construção de uma AFN A para reconhecer $A_1 \cup A_2$

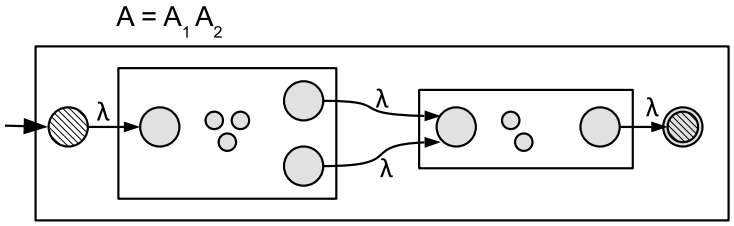

Figura 2.4: Construção de uma AFN A para reconhecer $A_1 A_2$

2.1.7 Conversão de ER para AFN

A classe de Linguagens Regulares é fechada com relação as operações de União, Concatenação e fecho de Kleene [Vieira 2006, Sipser 2012]. A demonstração pode ser feita a partir construção de AFNs para essas operações, conforme ilustrado nas Figuras 2.3, 2.4 e 2.5.

Usamos essa propriedade para transformar as Expressões Regulares em Autômatos Finitos Não-Determinísticos.

2.1.8 Conversão de AFN para AFD

Para todo Autômato Finito Não-Determinístico existe um Autômato Finito Determinístico correspondente [Vieira 2006, Lewis e Papadimitriou 2004, Sipser 2012]. A demonstração pode ser realizada através da simulação da execução em paralelo da com-

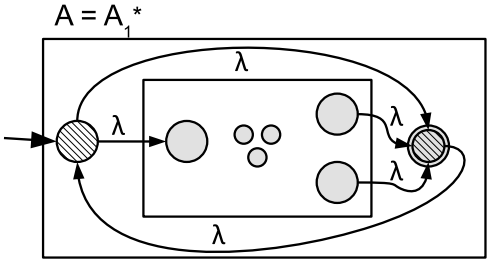

Figura 2.5: Construção de uma AFN A para reconhecer A_1^*

putação do AFN.

Basicamente, ao invés de mudar para um único estado a cada transição, no AFN poderemos nos deslocar para um conjunto de estados possíveis, dada o não-determinismo de algumas transições.

Existem $P(E)$ subconjuntos de estados possíveis para um conjunto E de estados. Assim se $E = \{1, 2, 3\}$, $P(E) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Ou seja, para um AFN com k estados, teremos uma AFD com até 2^k estados correspondentes.

Seja $R \in P(E)$, definimos a função fecho-lambda da seguinte forma: $f\lambda(R) = \{q|q$ pode ser atingido a partir de R através de zero ou mais transições $\lambda\}$.

Através de $P(E)$ e $f\lambda(R)$ podemos construir o AFD $M = \{E', \Sigma, \delta', i', F'\}$ equivalente ao AFN $N = \{E, \Sigma, \delta, I, F\}$ da seguinte forma:

- $E' = P(E)$.
- Para $R \in E'$, $\delta'(R, a) = \bigcup_{r \in R} f\lambda(\delta(r, a))$
- $i = f\lambda(\{I\})$.
- $F' = \{R \in E'|R$ contém um estado de aceitação do $AFN\}$

Para exemplificar todo o processo de conversão de uma expressão regular para um autômato finito não-determinístico e desse para um autômato finito determinístico, faremos uma tradução para $ER = 0^*1 + 0$:

1. A tradução da $ER = 0$ para o AFN está representada na Figura 2.6;

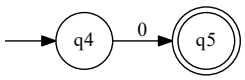

Figura 2.6: AFN correspondente a $ER = 0$

2. A tradução da $ER = 1$ para o AFN está representada na Figura 2.7;

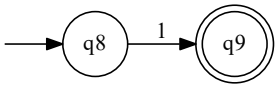

Figura 2.7: AFN correspondente a $ER = 1$

3. A tradução da $ER = 0^*$ para o AFN está representada na Figura 2.8;
4. A tradução da $ER = 0^*1$ para o AFN está representada na Figura 2.9;

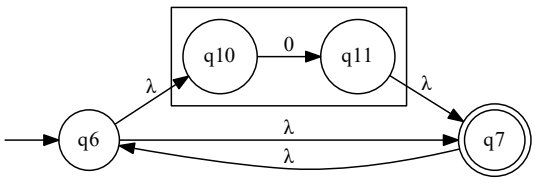

Figura 2.8: AFN correspondente a $ER = 0^*$

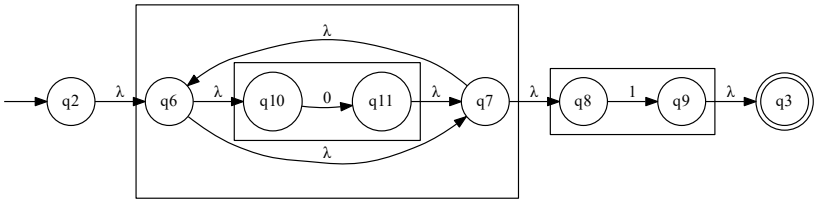

Figura 2.9: AFN correspondente a $ER = 0^*1$

5. A tradução da $ER = 0^*1 + 0$ para o AFN está representada na Figura 2.10;

6. A tradução do AFN = $(\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\}, \{0, 1\}, \delta, q_0, \{q_1\})$ da Figura 2.10 para o AFD = $(\{q'_0, q'_1, q'_2, q'_3\}, \{0, 1\}, \delta', q'_0, \{q'_1, q'_2\})$ correspondente da Figura 2.11 é realizada conforme os passos seguintes. Primeiro o estado inicial para AFD correspondente é calculado como:

$$q'_0 = f\lambda(q_0) = \{q_0, q_2, q_4, q_6, q_{10}, q_7, q_8\}$$

A função de transição δ' é calculada até a idempotência, da seguinte forma:

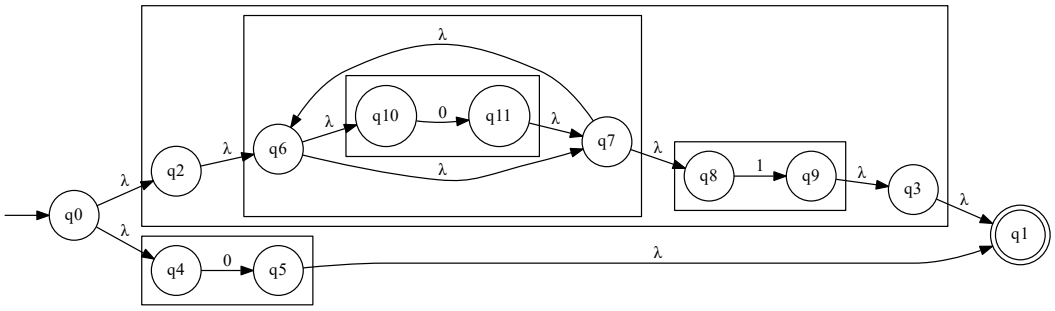

Figura 2.10: AFN correspondente a $ER = 0^*1 + 0$

$$
\begin{aligned}
\delta'(q'_0, 0) = \; & \textstyle\bigcup_{q \in q'_0} f\lambda(\delta(q, 0)) = \\
& f\lambda(\delta(q_4, 0)) \cup f\lambda(\delta(q_{10}, 0)) = \\
& \{q_5, q_1\} \cup \{q_{11}, q_7, q_6, q_{10}, q_8\} = \\
& \{q_5, q_7, q_{10}, q_1, q_8, q_6, q_{11}\} = q'_1 \\
\delta'(q'_0, 1) = \; & \textstyle\bigcup_{q \in q'_0} f\lambda(\delta(q, 1)) = \\
& f\lambda(\delta(q_8, 1)) = \\
& \{q_9, q_3, q_1\} = q'_2 \\
\delta'(q'_1, 0) = \; & \textstyle\bigcup_{q \in q'_1} f\lambda(\delta(q, 0)) = \\
& f\lambda(\delta(q_{10}, 0)) = \\
& \{q_{11}, q_7, q_8, q_6, q_{10}\} = q'_3 \\
\delta'(q'_1, 1) = \; & \textstyle\bigcup_{q \in q'_1} f\lambda(\delta(q, 1)) = \\
& f\lambda(\delta(q_8, 1)) = \\
& \{q_9, q_3, q_1\} = q'_2 \\
\delta'(q'_3, 0) = \; & \textstyle\bigcup_{q \in q'_3} f\lambda(\delta(q, 0)) = \\
& f\lambda(\delta(q_{10}, 0)) = \\
& \{q_{11}, q_7, q_8, q_6, q_{10}\} = q'_3 \\
\delta'(q'_3, 1) = \; & \textstyle\bigcup_{q \in q'_3} f\lambda(\delta(q, 1)) = \\
& f\lambda(\delta(q_8, 1)) = \\
& \{q_9, q_3, q_1\} = q'_2
\end{aligned}
$$

E o conjunto de estados finais $F = \{q'_1, q'_2\}$, pois são os estados do AFD que contém o estado final q_1 do AFN.

O resultado desse processo todo é um autômato finito determinístico correspondente a expressão regular informada.

Todo esse processo pode ser automatizado, executado através de ferramentas que representam essas estruturas (autômatos, expressões regulares) e implementam os algoritmos conforme descritos. São esses algoritmos que estão implementados em ferramentas de geração de analisadores léxicos, como Lex e Flex. A partir de um arquivo de especificação que contém as expressões regulares que pretende-se reconhecer, Lex/Flex traduzem as expressões em autômatos e desse em código em linguagem C, que implementam a programa de reconhecimento dos lexemas.

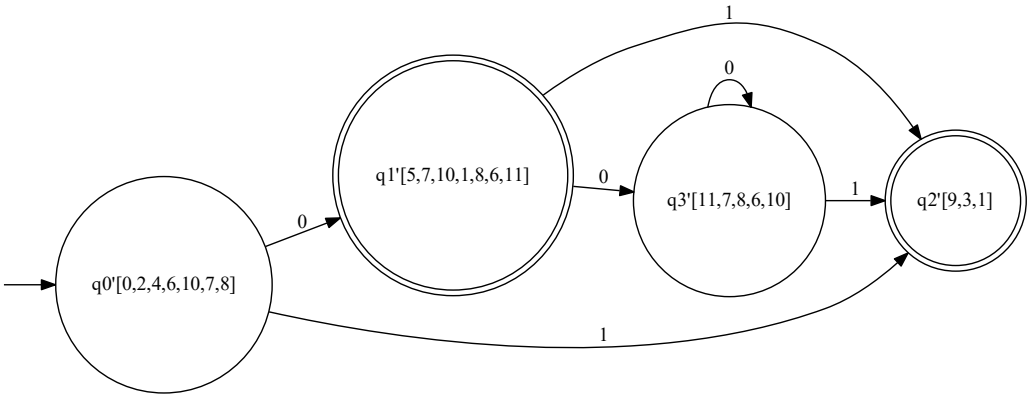

Figura 2.11: AFD equivalente ao AFN da Figura 2.10

2.1.9 Exercícios

1. Construa AFDs para as seguintes linguagens:

   (a) a*ba* = $\{w \in \Sigma^* | w$ contém um único b$\}$

   (b) (a + λ)b = $\{w \in \Sigma^* | w$ começa com a ou nada seguido de b$\}$

   (c) a*b = $\{w \in \Sigma^* | w$ contém zero ou mais a´s finalizado por b$\}$

   (d) a(a+b)*a+a = $\{w \in \Sigma^* | w$ começa e termina com a$\}$

   Sugestão: A partir das Expressões Regulares, construa o AFN e posteriormente, faça a tradução do AFN para o AFD correspondente, conforme apresentado nos exemplos.

2.2 Lex/Flex - Geradores de Analisadores Léxicos [1]

A primeira fase do compilador deve ler o arquivo fonte e converter sequência de caracteres em *tokens*. A construção dessa rotina não é tão complicada de se implementar, no entanto, existem ferramentas que podem ser utilizadas para automatizar essa fase da tradução. Lex é uma ferramenta usada para geração de analisadores léxicos. Especifica-se padrões léxicos para lex usando expressões regulares. E para cada símbolo encontrado associa-se uma ação que pode ser, por exemplo, retornar a codificação numérica (*token*) do símbolo encontrado.

A ferramenta lex funciona basicamente implementando os algoritmos de Conversão de Expressões Regulares em Automatos Finitos Não-Determisticos e esses para Autômatos Finitos Determinístico, conforme apresentado anteriormente. Essa tradução transforma as expressões regulares num código em linguagem C que implementa um autômato de estados finitos para reconhecimento dos símbolos léxicos. O próximo estado é determinado pela indexação numa tabela de estados gerada pelo computador usando o próximo caracter de entrada e o estado corrente.

[1] Tradução do texto de Thomas Niemann (http://epaperpress.com/lexandyacc/index.html).

Metacaracter	**Correspondência**
x	caracter 'x', exceto nova linha
.	Qualquer caracter exceto nova linha
\n	Nova linha
*	Zero ou mais cópias da expressão precedente
+	Uma ou mais cópias da expressão precedente
?	Zero ou uma cópia da expressão precedente
^	Início da linha
$	Fim da linha
a |b	Alternativa "a ou b"
(ab)+	Uma ou mais cópias de ab (agrupamento)
"a+b"	Literal "a+b"
[abc]	Classe de caracteres. Nesse caso, a, b ou c.

Tabela 2.2: Metacaracteres de Lex

Expressão Regular	**Símbolos Reconhecidos**
abc	abc
abc*	ab, abc, abcc, abccc, ...
abc+	abc, abcc, abccc, ...
a(bc)+	abc, abcbc, abcbcbc, ...
a(bc)?	a, abc
[abc]	a ou b ou c
[a-z]	qualquer letra no intervalo a até z.
[a\-z]	a ou - ou z
[^ab]	qualquer símbolo exceto a ou b.
[a^b]	a ou ^ou b
[a|b]	a ou |ou b
a|b	a ou b

Tabela 2.3: Exemplos de Expressões Regulares para Lex

2.2.1 Metacaracteres Lex

Alguns carateres tem um significado especial nas expressões regulares escritas para Lex. Os principais são apresentados na Tabela 2.2.

Alguns exemplos de correspondências entre expressões regulares para Lex e os símbolos que são reconhecidos estão representados na Tabela 2.3.

As expressões regulares em lex são compostas de metacaracteres (Tabela 2.2). Os exemplos de casamento de padrões são mostrados na Tabela 2.3. Com uma classe de caracteres, os caracteres normais perdem seu significado. Os dois caracteres usados em classes de caracteres (entre colchetes) são o hífen ("-") e o circunflexo ("^"). Quando usado entre dois caracteres o hífen significa o intervalo de caracteres. O circunflexo, quando usado como o primeiro caracter, nega a expressão. Se dois padrões reconhecem o mesmo string, o casamento mais longo é utilizado. No caso de dois casamentos de padrões com o mesmo comprimento, então o primeiro padrão listado é usado.

O arquivo de entrada para Lex tem o seguinte formato:

```
1 ... definicoes ...
2 %%
```

```
3 ... regras ...
4 %%
5 ... subrotinas ...
```

O arquivo de entrada Lex é dividido em três seções, com %% dividindo as seções. Vamos ilustrar isto com um exemplo. O primeiro exemplo é o menor arquivo de entrada Lex possível:

```
1 %%
```

A entrada é copiada para a saída, um caracter por vez. O primeiro %% é obrigatório, iniciando a seção de regras. Se nenhuma regra é especificada então a ação padrão é reconhecer tudo e copiar para a saída. Os arquivos de entrada e saída padrão são stdin e stdout, respectivamente. Aqui o mesmo exemplo, com os valores default explicitamente codificados:

```
1  %%
2      /* match everything except newline */
3  .   ECHO;
4      /* match newline */
5  \n  ECHO;
6
7  %%
8
9  int yywrap(void) {
10     return 1;
11 }
12
13 int main(void) {
14     yylex();
15     return 0;
16 }
```

Dois padrões foram especificados na seção de regras. Cada padrão tem que começar na primeira coluna do arquivo texto. Estes são seguidos por caracter em branco (espaço, tabulação ou nova linha) e uma ação opcional associada com o padrão. A ação pode ser um único comando C ou múltiplos comandos delimitados por chaves. Qualquer comando que não começa na primeira coluna é copiado como está para o arquivo C gerado. Podemos usar comentários no arquivo lex. Neste exemplo há dois padrões "." e "\n", com a ação ECHO associada com cada padrão. Várias macros e variáveis são pré-definidas por lex. ECHO é uma macro que escreve o código reconhecido pelo padrão léxico. Esta é a ação default para qualquer string não reconhecido. Normalmente, ECHO é definido como:

```
1 #define ECHO fwrite(yytext, yyleng, 1, yyout)
```

A variável yytext é um ponteiro para o string reconhecido (terminado por NULL), e yyleng é o tamanho do string reconhecido. A variável yyout é o arquivo de saída, e o default é stdout. A função yywrap é chamada por lex quando a entrada é esgotada. Retorna um se terminou, ou 0 se mais processamento é necessário. Todo programa C completo precisa de uma função main. Neste exemplos, nós simplesmente chamamos yylex, o ponto de entrada principal para lex. Algumas implementações de lex incluem cópias de main e yywrap na biblioteca, eliminando a necessidade de codificá-las explicitamente.

Algumas das funções, macros e variáveis pré-definidas no arquivo gerado por Lex estão listados na Tabela 2.4.

Nome	Função
int yylex(void)	Função principal do analisador que retorna o próximo token.
char *yytext	Ponteiro para o padrão léxico (string) reconhecido.
yyleng	Tamanho do string reconhecido.
yylval	Valor associado com o token.
int yywrap(void)	wrapup, retorna 1 se acabou, 0 se não acabou.
FILE *yyout	Arquivo de saída.
FILE *yyin	Arquivo de entrada.
INITIAL	Condição inicial.
BEGIN condition	Troca a condição inicial.
ECHO	Escreve o string reconhecido.

Tabela 2.4: Variáveis, Macros e Funções geradas por Lex

2.2.2 Alguns Exemplos de Lex

O primeiro exemplo apresenta um arquivo de especificação lex que gera um programa que não faz nada. Todas as entradas são reconhecidas mas não existe nenhuma ação associada com qualquer padrão, tal que nada será apresentado na saída.

```
1 %%
2 .
3 \n
```

O seguinte exemplo numera cada linha do arquivo. Algumas implementações de lex predefinem e calculam a variável yylineno. O arquivo de entrada para lex é yyin, e o arquivo de entrada default é stdin.

```
1  %{
2      int yylineno;
3  %}
4  %%
5  ^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
6  %%
7  int main(int argc, char *argv[]) {
8      yyin = fopen(argv[1], "r");
9      yylex();
10     fclose(yyin);
11 }
```

A seção de definições é composta de substituições, códigos e estados iniciais. O código na seção de definições é copiado como está para o topo do arquivo C gerado e precisa ser delimitado com os marcadores "%{"and "%}". As substituições simplificam as regras para casamento de padrões. Por exemplo, podemos definir dígitos e letras como segue:

```
1  digit      [0-9]
2  letter     [A-Za-z]
3  %{
4      int count;
5  %}
6  %%
7      /* match identifier */
8  {letter}({letter}|{digit})*     count++;
9  %%
10 int main(void) {
```

```
11      yylex();
12      printf("number of identifiers = %d\n", count);
13      return 0;
14  }
```

Espaços em branco precisam separar o termo definido e a expressão associada. As referências para as substituições nas regras devem estar delimitadas por chaves ({letter}) para diferenciá-las de literais. Quando o padrão léxico é reconhecido nas regras, o código C associado é executado. Aqui um exemplo de analisador que conta o número de caracteres, palavras e linhas no arquivo (similar ao comando Unix wc):

```
1  %{
2      int nchar, nword, nline;
3  %}
4  %%
5  \n          { nline++; nchar++; }
6  [^ \t\n]+   { nword++, nchar += yyleng; }
7  .           { nchar++; }
8  %%
9  int main(void) {
10     yylex();
11     printf("%d\t%d\t%d\n", nchar, nword, nline);
12     return 0;
13 }
```

2.3 Analisador Léxico para Linguagem Simples

A linguagem Simples®, na sua versão inicial, é composta dos seguintes símbolos léxicos:

- **Palavras Reservadas**: programa, inicio, fimprograma, leia, escreva, se, entao, senao, fimse, enquanto, faca, fimenquanto.
- **Operadores**: +, -, *, div, >, <, +, e, ou, nao, <−, (,)
- **tipos**: inteiro, logico
- **constantes**: V, F
- **identificadores**: (letra)(letra|digito)*
- **números**: (digito)+

Utilizando a ferramenta flex pode-se construir um classificador do código da linguagem Simples. Esse classificador é uma versão básica do analisador léxico que recebe como entrada um código de Simples e apresenta na saída uma tabela onde todos os símbolos léxicos encontrados no código são classificados. Dado o seguinte código Simples:

```
1  programa teste
2    inteiro A B
3  inicio
4    A <- 5
5    leia B
6    B <- A * B
7    escreva B
8  fimprograma
```

Obtem-se como saída a seguinte tabela:

```
1     programa: palavra reservada
2        teste: identificador
3      inteiro: palavra reservada
4            A: identificador
5            B: identificador
6       inicio: palavra reservada
7            A: identificador
8           <-: operador de atribuicao
9            5: numero
10        leia: palavra reservada
11           B: identificador
12           B: identificador
13          <-: operador de atribuicao
14           A: identificador
15           *: operador aritmetico multiplicacao
16           B: identificador
17     escreva: palavra reservada
18           B: identificador
19 fimprograma: palavra reservada
```

O arquivo de especificação Lex que implementa esse classificador de símbolos da linguagem Simples é:

```
1  identificador   [a-zA-Z]([a-zA-Z0-9])*
2  numero          [0-9]+
3  espaco          [ \t]+
4  novalinha       [\n]
5
6  %%
7
8  programa          printf("%11s: palavra reservada\n", yytext);
9  inicio            printf("%11s: palavra reservada\n", yytext);
10 fimprograma       printf("%11s: palavra reservada\n", yytext);
11
12 leia              printf("%11s: palavra reservada\n", yytext);
13 escreva           printf("%11s: palavra reservada\n", yytext);
14
15 se                printf("%11s: palavra reservada\n", yytext);
16 entao             printf("%11s: palavra reservada\n", yytext);
17 senao             printf("%11s: palavra reservada\n", yytext);
18 fimse             printf("%11s: palavra reservada\n", yytext);
19
20 enquanto          printf("%11s: palavra reservada\n", yytext);
21 faca              printf("%11s: palavra reservada\n", yytext);
22 fimenquanto       printf("%11s: palavra reservada\n", yytext);
23
24 "+"               printf("%11s: operador aritmetico soma\n", yytext);
25 "-"               printf("%11s: operador aritmetico subtracao\n", yytext);
26 "*"               printf("%11s: operador aritmetico multiplicacao\n", yytext)
       ;
27 div               printf("%11s: operador aritmetico divisao\n", yytext);
28
29 ">"               printf("%11s: operador relacional maior\n", yytext);
30 "<"               printf("%11s: operador relacional menor\n", yytext);
31 "="               printf("%11s: operador relacional igual\n", yytext);
32
33 e                 printf("%11s: operador logico conjuncao\n", yytext);
```

```
34 ou                printf("%11s: operador logico disjuncao\n", yytext);
35 nao               printf("%11s: operador logico negacao\n", yytext);
36
37 "<-"              printf("%11s: operador de atribuicao\n", yytext);
38 "("               printf("%11s: simbolo abre parenteses\n", yytext);
39 ")"               printf("%11s: simbolo fecha parenteses\n", yytext);
40
41 inteiro          printf("%11s: palavra reservada\n", yytext);
42 logico           printf("%11s: palavra reservada\n", yytext);
43 V                printf("%11s: constante logica de verdade\n", yytext);
44 F                printf("%11s: constante logica de falsidade\n", yytext);
45
46 {identificador} printf("%11s: identificador\n", yytext);
47 {numero}         printf("%11s: numero\n", yytext);
48 {espaco}         /* nao faz nada */
49 {novalinha}      /* nao faz nada */
50 .                printf("%11s: ERRO - SIMBOLO NAO RECONHECIDO!\n", yytext);
51
52 %%
53
54 int yywrap(void) { return 1; }
55
56 int main (void) {
57     yylex();
58     return 0;
59 }
```

A exclusão dos comentários tem um tratamento especial na sintaxe Lex/Flex. A marca de início de comentário, normalmente representada por uma sequência de símbolos como /* e (*, determina uma configuração especial para o autômato. A partir dessa sequência, o analisador léxico deve descartar todos os símbolos, exceto a marca de fim de linha que é utilizada na contagem de linhas do arquivo fonte. Essa contagem é necessária para que as rotinas de tratamento de erro possam indicar a linha correta do erro. E quando o autômato identifica a sequência que fecha o comentário (/* ou *)), o autômato deve retornar ao seu estado normal. A Figura 2.12 representa o autômato para o tratamento de comentário de linhas. O autômato vai para o estado X quando encontra a marca de início de comentário. Nesse estado, o autômato consome os símbolos da entrada, quando encontra a sequência que marca o fim do comentário, o autômato volta ao estado inicial de aceitação.

O novo arquivo de especificação Flex que inclui o tratamento para comentário de linha (//), comentário de múltiplas linhas (/* ... */) e a contagem do número das linhas no arquivo fonte é:

```
1  identificador [a-zA-Z]([a-zA-Z0-9])*
2  numero [0-9]+
3  espaco [ \t]+
4  novalinha [\n]
5
6  %x coment
7
8  %{
9          int nlin = 1;
10 %}
11
12 %%
13 programa       printf("%3d|%11s: reservada\n", nlin, yytext);
```

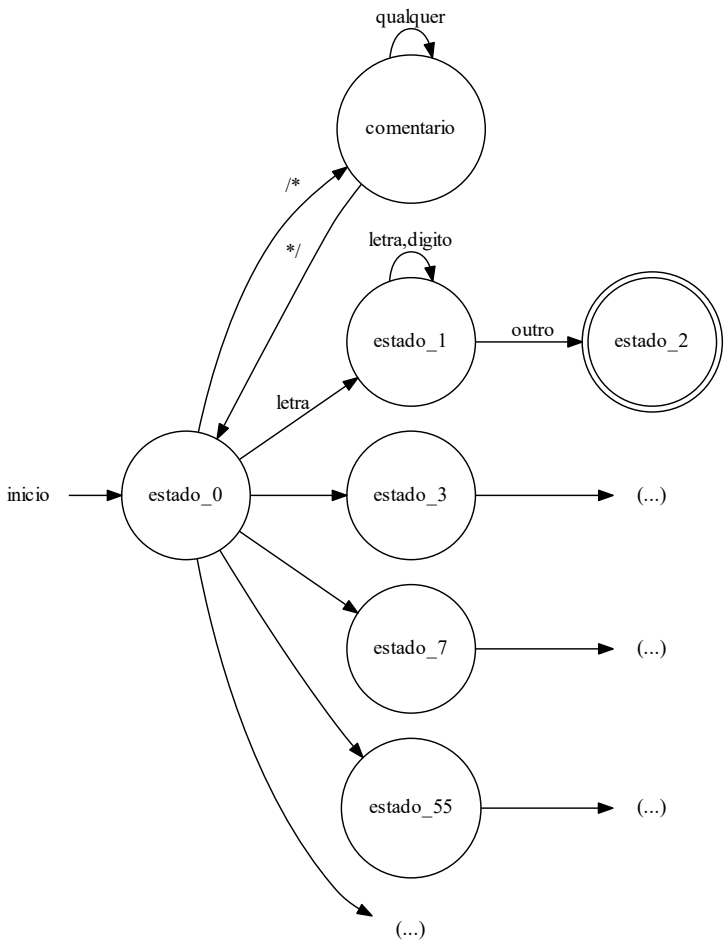

Figura 2.12: Comentário para autômatos de estados finitos

```
14 inicio          printf("%3d|%11s: reservada\n", nlin, yytext);
15 fimprograma     printf("%3d|%11s: reservada\n", nlin, yytext);
16 leia            printf("%3d|%11s: reservada\n", nlin, yytext);
17 escreva         printf("%3d|%11s: reservada\n", nlin, yytext);
18 se              printf("%3d|%11s: reservada\n", nlin, yytext);
19 entao           printf("%3d|%11s: reservada\n", nlin, yytext);
20 senao           printf("%3d|%11s: reservada\n", nlin, yytext);
21 fimse           printf("%3d|%11s: reservada\n", nlin, yytext);
22 enquanto        printf("%3d|%11s: reservada\n", nlin, yytext);
23 fimenquanto     printf("%3d|%11s: reservada\n", nlin, yytext);
24 inteiro         printf("%3d|%11s: reservada (tipo)\n", nlin, yytext);
25 logico          printf("%3d|%11s: reservada (tipo)\n", nlin, yytext);
26
27 "+"            printf("%3d|%11s: soma\n", nlin, yytext);
28 "-"            printf("%3d|%11s: subtracao\n", nlin, yytext);
29 "*"            printf("%3d|%11s: multiplicacao\n", nlin, yytext);
30 "<-"           printf("%3d|%11s: atribuicao\n", nlin, yytext);
31 div            printf("%3d|%11s: divisao\n", nlin, yytext);
32 ">"            printf("%3d|%11s: maior\n", nlin, yytext);
33 "<"            printf("%3d|%11s: menor\n", nlin, yytext);
34 "="            printf("%3d|%11s: igual\n", nlin, yytext);
35 e              printf("%3d|%11s: e logico\n", nlin, yytext);
36 ou             printf("%3d|%11s: ou logico\n", nlin, yytext);
37 nao            printf("%3d|%11s: negacao\n", nlin, yytext);
38 "("            printf("%3d|%11s: abre\n", nlin, yytext);
39 ")"            printf("%3d|%11s: fecha\n", nlin, yytext);
40
41 "V"            printf("%3d|%11s: constante\n", nlin, yytext);
42 "F"            printf("%3d|%11s: constante\n", nlin, yytext);
43
44 "//".*            ;
45 "/*"              BEGIN(coment);
46 <coment>"*/"      BEGIN(INITIAL);
47 <coment>.         ;
48 <coment>\n        nlin++;
49
50 {identificador}    printf("%3d|%11s: identificador\n", nlin, yytext);
51 {numero}           printf("%3d|%11s: numero\n", nlin, yytext);
52 {espaco}           ;
53 {novalinha}        nlin++;
54
55 %%
56 int yywrap() { return 1; }
57 int main () {
58         yylex();
59         return 0;
60 }
```

Para o arquivo Simples de entrada:

```
1 /*--------------------
2 Programa exemplo em linguagem simples
3 --------------------*/
4 programa um
5   inteiro a b  // declaracao das variaveis
6 inicio
7   leia a  // leitura de a
8   leia b  // leitura de b
9   enquanto a < b faca
```

```
10      escreva a
11      a <- a + 1
12    fimenquanto
13  fimprograma
```

A saída produzida como resultado do classificador de símbolos (analisador léxico simplificado) é:

```
1     4|    programa: reservada
2     4|          um: identificador
3     5|     inteiro: reservada (tipo)
4     5|           a: identificador
5     5|           b: identificador
6     6|      inicio: reservada
7     7|        leia: reservada
8     7|           a: identificador
9     8|        leia: reservada
10    8|           b: identificador
11    9|    enquanto: reservada
12    9|           a: identificador
13    9|           <: menor
14    9|           b: identificador
15    9|       faca: identificador
16   10|     escreva: reservada
17   10|           a: identificador
18   11|           a: identificador
19   11|          <-: atribuicao
20   11|           a: identificador
21   11|           +: soma
22   11|           1: numero
23   12|fimenquanto: reservada
24   13|fimprograma: reservada
```

Capítulo 3

Análise Sintática

A análise sintática é executada pelo *parser* e sua função principal é agrupar os *tokens*, retornados do analisar léxico, em estruturas sintáticas (comando, bloco, expressão, identicador, número, etc). Uma estrutura que pode ser empregada é a árvore sintática. A árvore representa a aplicação das regras sintáticas da linguagem e definem, de certa forma, um significado para a estrutura do programa compilado. O programa está sintaticamente correto se for possível construir uma única árvore sintática, no qual a raiz é o símbolo inicial da gramática da linguagem (símbolo de partida) e as folhas são os tokens retornados do analisado léxico.

Apresentamos na próxima seção uma breve revisão de conceitos que serão fundamentais para essa fase da compilação, tais como: gramática, gramática livre de contexto, derivação, redução, árvore de derivação e árvore sintática.

3.1 Breve revisão

3.1.1 Gramática

A gramática pode ser definida de maneira formal como uma quádrupla $G = (V, \Sigma, R, P)$, onde V é vocabulário de símbolos não-terminais de G, também chamados de variáveis da gramática. Σ é o vocabulário de símbolos terminais de G, disjunto de V. R são as regras de produção ou regras de sintaxe. P é um símbolo de V (raiz, símbolo inicial ou símbolo de partida da gramática G).

3.1.2 Gramática Regular

A gramática regular é uma gramática, em que cada regra tem uma das formas: $X \to a$ ou $X \to aY$ ou $X \to \lambda$, onde X e Y são símbolos não-terminais e a é um símbolo terminal. Gramáticas Regulares, Expressões Regulares e Autômatos finitos são três formalismos alternativos usados para especificar linguagens regulares. Usamos essas especificações para definir os padrões léxicos que são reconhecidos pelo analisador léxico.

3.1.3 Gramática Livre de Contexto

A gramática livre de contexto (GLC) é uma gramática cujas as regras de produção são da forma: $X \to \alpha$, onde X é qualquer símbolo não terminal e α é uma elemento de $\{V \cup \Sigma\}^*$. As GLC´s são usadas para especificar, através de definições indutivas, todas as construções sintáticas válidas para as linguagens de programação. Usamos GLC para especificar a sintaxe das linguagens que são reconhecidas pelo analisador sintático de compilador. Exemplo:

$$G = (\{C, E, I, O\}, \{a, b, +, *, (,), \underline{:=}, \underline{;}\}, R, C\}$$

onde:

$$R = \left\{ \begin{array}{rcl} C & \rightarrow & I := E|C;C \\ E & \rightarrow & I|EOE|(E) \\ O & \rightarrow & +|* \\ I & \rightarrow & a|b \end{array} \right\}$$

3.1.4 Derivação

A derivação, representada através da relação $\Rightarrow$ significa a aplicação de uma regra de produção numa sequência que contém símbolos não-terminais a fim de se obter uma sequência só de símbolos terminais. Exemplo: Considere a gramática de expressões (para abreviar a notação da gramática estamos indicando apenas o conjunto de regras de produção):

$$G_1 : E \rightarrow a|b|E + E|E * E|(E).$$

A sentença $\underline{(a + b) * a}$ pode ser obtida por várias derivações distintas:

- $E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E \Rightarrow (a + E) * E \Rightarrow (a + b) * E \Rightarrow (a + b) * a$
- $E \Rightarrow E * E \Rightarrow E * a \Rightarrow (E) * a \Rightarrow (E + E) * a \Rightarrow (E + b) * a \Rightarrow (a + b) * a$

Usando as definições de derivação e gramática podemos definir novamente uma linguagem da seguinte forma:

$$L(G) = \{\alpha \in \Sigma^* | P \stackrel{*}{\Rightarrow} \alpha\}$$

3.1.5 Árvores de Derivação

Se observarmos o exemplo anterior, verificamos que o que muda numa sequência de derivação para outra é a ordem na qual as produções são aplicadas. Num certo sentido todas as derivações são equivalentes. Um meio usado para representar todas as derivações que indicam a mesma estrutura são as árvores de derivação (árvores sintáticas). Na árvore de derivação a aplicação da regra $E \rightarrow E_1E_2...E_n$ é representada da seguinte forma, o símbolo não terminal E, do lado esquerdo da regra é a raiz da subárvore e os símbolos $E_1, E_2, ..., E_n$, os símbolos não terminais e/ou terminais do lado direito da regra são os filhos desta raiz. Exemplo: Para gramática G_1 e a sequência $\underline{(a + b) * a}$, temos a árvore de derivação, conforme ilustrado na Figura 3.1.

3.1.6 Gramática Ambígua

Consideremos a cadeia $\underline{a+b*a}$ para gramática G_1. Existem duas árvores de derivação distintas, conforme ilustrado na Figura 3.2.

Uma gramática G é dita ambígua se a linguagem $L(G)$ contém uma sentença para qual existe mais de uma árvore de derivação, usando a gramática G. Consideremos agora a seguinte sentença $\underline{a+a+a}$:

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow a + E + E \Rightarrow a + a + E \Rightarrow a + a + a$$

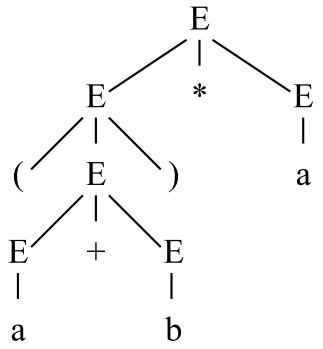

Figura 3.1: Árvore de Derivação

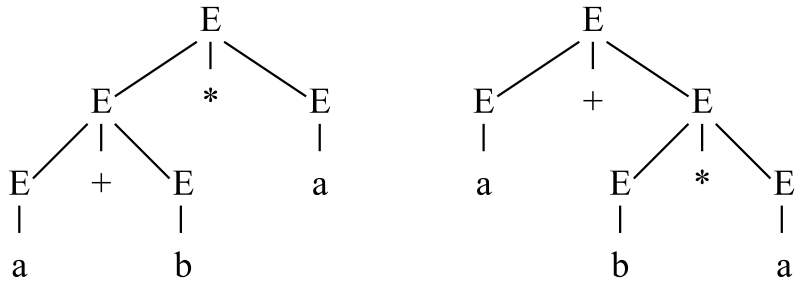

Figura 3.2: Exemplo de ambiguidade

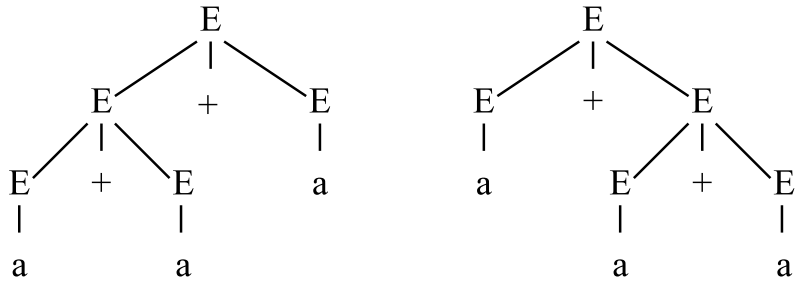

Figura 3.3: Exemplo de ambiguidade

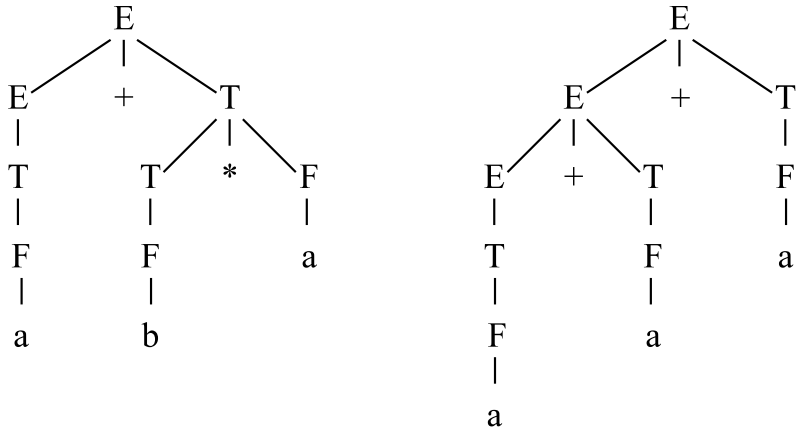

Figura 3.4: Exemplo de árvore não ambígua

Devido à maneira como foi definido o conceito de derivação não está claro no passo E+E+E, qual das duas ocorrências de E foi substituída. Esta derivação corresponde a duas árvores distintas, conforme ilustrado na Figura 3.3.

A causa da ambiguidade está no fato dela não indicar as precedências entre os operadores * e +. Consideremos agora a seguinte gramática G_2:

$$\begin{array}{lcl} E & \rightarrow & E+T \mid T \\ T & \rightarrow & T*F \mid F \\ F & \rightarrow & a \mid b \mid (E) \end{array}$$

Pode-se demonstrar que a gramática G_1 e a G_2 definem a mesma linguagem e que G_2 não é ambígua. Há várias derivações para as sentenças a+b+a e a+a+a:

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+b+F \Rightarrow a+b+a$

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*a \Rightarrow E+F*a \Rightarrow E+b*a \Rightarrow F+b*a \Rightarrow a+b*a$

Há porém, uma única árvore de derivação para cada sentença, conforme ilustrado na Figura 3.4.

Deve-se notar que nem sempre é possível eliminar ambiguidade. Um outro exemplo famoso de ambiguidade é o chamado "else pendente". Consideremos a seguinte gramática G_3:

$$C \rightarrow a \mid if \ b \ then \ C \ else \ C \mid if \ b \ then$$

Esta gramática é ambígua, pois temos duas árvores de derivação para a sentença if b then if b then a else a, conforme ilustrado na Figura 3.5.

Isto se deve ao fato da parte else a poder ser associada tanto com o primeiro como com o segundo if. Ainda neste caso podemos eliminar a ambiguidade:

$$\begin{array}{lcl} C & \rightarrow & a \mid if \ b \ then \ D \ else \ C \mid if \ b \ then \ C \\ D & \rightarrow & a \mid if \ b \ then \ D \ else \ D \end{array}$$

Note-se que esta gramática sempre associa o else com o if mais próximo possível.

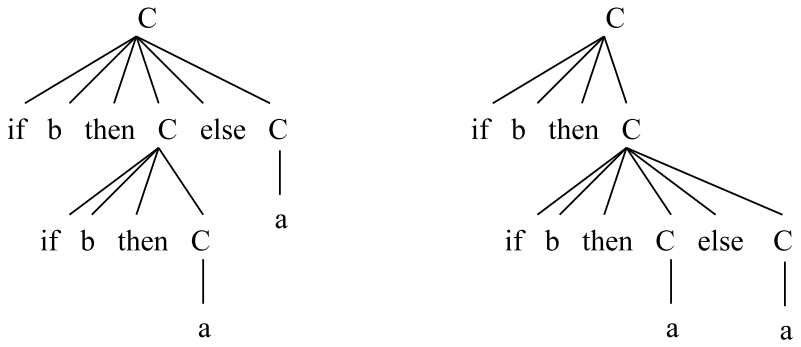

Figura 3.5: Ambiguidade do else pendente

3.1.7 Exercícios

1. Escreva as gramáticas para gerar as linguagens:

   a) $L_1 = \{a^m b^n | m \geq n \geq 1\}$

   b) $L_2 = \{a^n b | n \geq 1\}$

2. Enumere as derivações possíveis para a cadeia $\underline{b * a + a}$, usando as gramáticas G1 e G2 abaixo:

$$G_1 : E \rightarrow a|b|E + E|E * E|(E).$$

$$\begin{array}{rcl} G_2 : E & \rightarrow & E + T \quad | \quad T \\ T & \rightarrow & T * F \quad | \quad F \\ F & \rightarrow & a \quad | \quad b \quad | \quad (E) \end{array}$$

3. Determine duas sentenças válidas para a gramática: $S \rightarrow aS \quad | \quad bS \quad | \quad c$.

4. Escreva uma gramática qualquer para os símbolos terminais a, b e c e determine duas sentenças válidas para esta gramática.

3.2 Implementação de Analisadores Sintáticos

O objetivo principal da análise sintática é decidir se uma cadeia (sentença, palavra) pertence ou não à linguagem definida por uma gramática. Existem duas importantes estratégias para o problema de análise sintática:

- Análise Sintática Ascendente
- Análise Sintática Descendente

Na primeira, a árvore sintática é construída partindo-se da cadeia a ser analisada, "subindo-se" até atingir o símbolo inicial da gramática. Na segunda, parte-se do símbolo inicial e vai-se "descendo" até atingir todos os símbolos da cadeia que está sendo analisada.

Apresentamos nessa seção uma estratégia (existem outras) para análise descendente e uma estratégia para análise ascendente.

3.3 Análise Sintática LL

Os analisadores descendentes (*top-down*) podem ser construídos com uma classe de gramática chamada LL(1). O primeiro "L"se refere a forma como é lida a cadeia de entrada na análise, nesse caso da esquerda para direita (*Left-to-right*). O segundo "L"se refere a forma como é obtida a sequência de derivação para obtenção da sentença avaliada, nesse caso derivação mais à esquerda (*leftmost*). O número "1"se refere ao número de símbolos a frente deve-se olhar para decidir que regra da gramática deve ser utilizada.

Para construção de Analisadores LL(k), são necessárias duas funções sobre símbolos da gramática: função *FIRST* e *FOLLOW*. A Figura 3.6 é usada para ilustrar o cálculo dessas funções. Intuitivamente, todos os símbolos que iniciam derivações de A compõe o conjunto *FIRST* de A (por exemplo, o terminal c da Figura). Todo símbolo terminal que segue o símbolo A em qualquer derivação faz parte do conjunto *FOLLOW* de A (por exemplo, o terminal a da Figura 3.6)

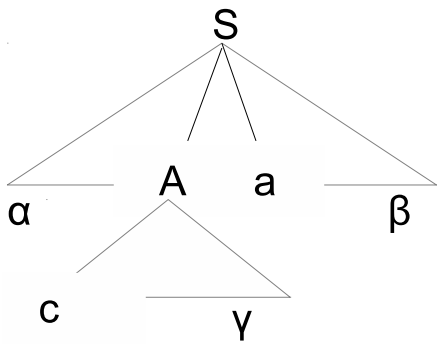

Figura 3.6: Ilustração para os conjuntos First/Follow

3.3.1 Função $FIRST$

Para calcular $FIRST$ de todos os simbolos X de uma gramática, execute as seguintes regras até que nenhum novo símbolo possa ser acrescentado a qualquer conjunto $FIRST$.

1. Se X é um símbolo terminal, então $FIRST(X) = \{X\}$
2. Se X é um não-terminal e $X \rightarrow Y_1Y_2...Y_k$ é uma regra de produção para algum $k \geq 1$, então acrescente a a $FIRST(X)$ se, para algum i, a estiver em $FIRST(Y_i)$, e λ estiver em todos os $FIRST(Y_1), ..., FIRST(Y_{i-1})$. Se λ está em $FIRST(Y_j)$ para todo $j = 1, 2, ..., k$, então adicione λ a $FIRST(X)$.
3. Se $X \rightarrow \lambda$ é um regra de produção, então acrescente λ a $FIRST(X)$

3.3.2 Função $FOLLOW$

Para calcular $FOLLOW$ de todos os simbolos NÃO-TERMINAIS S de uma gramática, execute as seguintes regras até que nenhum novo símbolo possa ser acrescentado a qualquer conjunto $FOLLOW$.

1. Coloque # em $FOLLOW(S)$, onde S é o símbolo inicial da gramática e # é o marcador de fim de sentença, que é incluido antes da avaliação da sentença.

2. Se houver uma produção $A \to \alpha B \beta$, então tudo que está em $FIRST(\beta)$ exceto λ, deve estar em $FOLLOW(B)$.

3. Se houver uma produção $A \to \alpha B$ ou $A \to \alpha B \beta$, onde $FIRST(\beta)$ contém λ, então inclua $FOLLOW(A)$ em $FOLLOW(B)$.

Exemplo:
Considere a seguinte gramática:

$$E \to TE'$$
$$E' \to +TE' | \lambda$$
$$T \to FT'$$
$$T' \to *FT' | \lambda$$
$$F \to a | (E)$$

Conjuntos FIRST e FOLLOW:

	FIRST	FOLLOW
E	{(, a}	{), #}
E'	{+, λ}	{), #}
T	{(, a}	{+,), #}
T'	{$*$, λ}	{+,), #}
F	{(, a}	{$*$, +,), #}

Algoritmo para construir a Tabela de Análise Preditiva LL(1):

- Para cada produção $A \to \alpha$ da gramática, faça:

  1. Para cada terminal a de $FIRST(\alpha)$, adicione a produção $A \to \alpha$ a $T[A, a]$
  2. Se $FIRST(\alpha)$ inclui a palavra vazia, então adicione $A \to \alpha$ a $T[A, b]$ para cada b em $FOLLOW(A)$.

A tabela LL(1) é:

	a	+	$*$	(	)	#
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \lambda$	$E' \to \lambda$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \lambda$	$T' \to *FT'$		$T' \to \lambda$	$T' \to \lambda$
F	$F \to a$			$F \to (E)$		

A seguinte definição permite identificar as gramáticas LL(1):

Definição 6 *Uma gramática G não recursiva à esquerda é LL(1)se e somente se, sempre que $A \to \alpha$ e $A \to \beta$ são regras de produção de G, ocorre que:*

1. *a interseção dos conjuntos $FIRST(\alpha)$ e $FIRST(\beta)$ é vazia;*

2. *no máximo um dos dois, α ou β, deriva a palavra vazia; e*

3. *se $\beta \stackrel{*}{\Rightarrow} \lambda$, então a interseção de $FIRST(\alpha)$ e $FOLLOW(A)$ é vazia;*

3.4 Análise LR

Os analisadores ascendentes (*bottom-up*) podem ser construídos com uma classe de gramática chamada LR(1). A letra "L" se refere a forma como é lida a cadeia de entrada na análise, nesse caso da direita para esquerda (*Left-to-right*). A letra "R" do nome se refere a forma como é obtida a sequência de derivação para obtenção da sentença avaliada, nesse caso derivação mais à direita (*rigthmost*). O número "1" se refere ao número de símbolos a frente deve-se olhar para decidir que regra da gramática deve ser utilizada.

O funcionamento de um algoritmo de análise sintática ascendente pode ser descrito, informalmente da seguinte maneira:

1. α = cadeia dada
2. Decompor $\alpha = \beta X_1 X_2 ... X_n \gamma$ tal que exista uma regra de produção $X \rightarrow X_1 X_2 ... X_n$. Adotar a cadeia $\alpha = \beta X \gamma$, associando-se uma árvore onde X é a raiz e $X_1 X_2 ... X_n$ são as subárvores da raiz X. (Este processo inverso da derivação é denominado REDUÇÃO).
3. O passo 2 é repetido até que o valor de α seja reduzido para o símbolo inicial da gramática.

Na descrição genérica do algoritmo de análise ascendente usa-se a intuição para decidir que regra de produção utilizar para redução. Em algumas situações temos mais de uma redução possível. Para automatizar o processo de análise sintática existem alguns algoritmos que utilizam uma tabela (matriz) representando a gramática para decidir de forma automática as reduções. A idéia básica deste método é que para cada símbolo da cadeia de entrada é feita uma consulta na tabela. O valor obtido da tabela (um estado) é empilhado. Em cada passo, o último estado empilhado é utilizado para decidir sobre uma eventual redução. O processo continua até que seja encontrada uma situação de erro, ou então, até que a cadeia de entrada seja reconhecida.

A tabela de análise é uma matriz retangular cujas linhas são indexadas pelos estados, e as colunas pelos símbolos do vocabulário da gramática (terminais e não-terminais). Os elementos da matriz indicam as ações que podem ser tomadas pelo algoritmo que podem ser:

- Empilhar o estado e_i
- Reduzir usando a j-ésima regra de produção.
- Aceitar
- Rejeitar

Consideremos a gramática abaixo cujas produções foram numeradas para fins de referência:

$$\begin{array}{lllll} (1) & E & \rightarrow & +EE \\ (2) & E & \rightarrow & *EE \\ (3) & E & \rightarrow & a \\ (4) & E & \rightarrow & b \end{array}$$

Tabela	E	+	*	a	b	#
e_o	e_1	e_2	e_3	e_4	e_5	
e_1						a
e_2	e_6	e_2	e_3	e_4	e_5	
e_3	e_7	e_2	e_3	e_4	e_5	
e_4		r_3	r_3	r_3	r_3	r_3
e_5		r_4	r_4	r_4	r_4	r_4
e_6	e_8	e_2	e_3	e_4	e_5	
e_7	e_9	e_2	e_3	e_4	e_5	
e_8		r_1	r_1	r_1	r_1	r_1
e_9		r_2	r_2	r_2	r_2	r_2

Tabela 3.1: Tabela de Análise LR

A tabela de análise LR(k) para esta gramática é:

O símbolo # é utilizado para marcar o fim da sentença. As ações indicadas em cada célula da tabela são:

- e_i = significa empilhar o estado e_i.
- r_j = significa reduzir usando a j-ésima regra de produção.
- a = aceitar.
- *branco* = rejeitar.

Considerando essa tabela, o algoritmo que reconhece automaticamente as construções válidas para a gramática pode ser resumido da seguinte forma:

```
1  Inicio
2    P[0] ← e_o
3    i ← 0
4    termino ← falso
5    reduzido ← falso
6    Simbolo ← PROXIMO ( )
7    Repita
8      Se reduzido
9          Entao s ← SimboloReduzido
10         Senao s ← Simbolo
11     Fim-se
12     Caso Tabela[P[i],s] de
13         Empilha (e_j):
14               i ← i + 1
15               P[i] ← e_j
16               Se reduzido
17                    Entao reduzido ← falso
18                    Senao Simbolo ← PROXIMO ( )
19         Reduzir (A → α):
20                i ← i - | α |
21                Reduzido ← verdadeiro
22                SimboloReduzido ← A
23         Aceitar: termino ← verdadeiro
24         Rejeitar: ERRO ( )
25     Fim-caso
26   Ate termino
27 Fim
```

As estruturas e variáveis desse algoritmo são:

- P: é a pilha de estados
- i: é o topo da pilha de estados
- termino: condição de parada do algoritmo. Determina a aceitação da sentença que está sendo avaliada.
- reduzido: variável booleana que indica se houve redução no passo anterior do algoritmo
- Simbolo: é o último símbolo lido na sentença de entrada.
- PROXIMO ( ): função que retorna o próximo símbolo da sentença de entrada (corresponde a uma versão primitiva do analisador léxico, i. e., função yylex() gerada pela ferramenta FLEX).
- Tabela: é a tabela de análise LR, em questão.
- ERRO ( ): função de tratamento de erro.

A Tabela 3.2 apresenta o acompanhamento dos passos do algoritmo de análise LR(k) usando a tabela para a sentença +*a+baa#

Algumas observações:

- Conforme pode ser observado no algoritmo, o próximo símbolo a ser consultado pode ser o próximo símbolo da cadeia de entrada ou o não-terminal da última redução (se houve redução no passo anterior).
- A fim de tornar o processo de análise mais claro, os estados empilhados são indicados por s_i, onde s representa o símbolo que determinou a inclusão do estado na pilha e i representa o estado propriamente dito.
- Na redução são removidos k estados da pilha, onde k representa o número de símbolos no lado direito da regra de produção considerada.

Exercício: Fazer o acompanhamento do algoritmo de análise LR(k) usando a gramática e a tabela anterior, para a cadeia de entrada ++*abaa#.

3.4.1 Implementação do Algoritmo LR

Na listagem de código seguinte é apresentada uma implementação para o algoritmo LR. Na implementação em questão a cadeia de entrada é restrita a cada símbolo ocupando uma única posição de caracter no string *sentenca*. O código apresenta várias tabelas para diversas gramáticas distintas. Ao final da execução, se a sentença verificada for aceita, o programa apresenta a sequência de derivações mais à direita que produz a sentença.

```
1  /*+-------------------------------------------------------------+
2    | Implementacao do algoritmo de analise sintatica LR(K).      |
3    | Por Luiz Eduardo da Silva                                   |
4    +-------------------------------------------------------------+*/
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
```

Passo	**Pilha**	**Símbolo Reduzido**	**Cadeia de Entrada**	**Ação**
0	e_o		$\underline{+}$*a+baa#	e2
1	e_o+_2		$\underline{*}$a+baa#	e3
2	$e_o +_2 *_3$		$\underline{a}$+baa#	e4
3	$e_o +_2 *_3 a_4$		$\underline{+}$baa#	r3
4	$e_o +_2 *_3$	$\underline{E}$	+baa#	e7
5	$e_o +_2 *_3 E_7$		$\underline{+}$baa#	e2
6	$e_o +_2 *_3 E_7 +_2$		$\underline{b}$aa#	e5
7	$e_o +_2 *_3 E_7 +_2 b_5$		$\underline{a}$a#	r4
8	$e_o +_2 *_3 E_7 +_2$	$\underline{E}$	aa#	e6
9	$e_o +_2 *_3 E_7 +_2 E_6$		$\underline{a}$a#	e4
10	$e_o +_2 *_3 E_7 +_2 E_6 a_4$		$\underline{a}$#	r3
11	$e_o +_2 *_3 E_7 +_2 E_6$	$\underline{E}$	a#	e8
12	$e_o +_2 *_3 E_7 +_2 E_6 E_8$		$\underline{a}$#	r1
13	$e_o +_2 *_3 E_7$	$\underline{E}$	a#	e9
14	$e_o +_2 *_3 E_7 E_9$		$\underline{a}$#	r2
15	$e_o +_2$	$\underline{E}$	a#	e6
16	$e_o +_2 E_6$		$\underline{a}$#	e4
17	$e_o +_2 E_6 a_4$		$\underline{\#}$	r3
18	$e_o +_2 E_6$	$\underline{E}$	#	e8
19	$e_o +_2 E_6 E_8$		$\underline{\#}$	r1
20	e_o	$\underline{E}$	#	e1
21	$e_o E_1$		$\underline{\#}$	ACEITAR

Tabela 3.2: Acompanhamento do algoritmo LR

```
8
9  /*+-------------------------------------------------------------+
10   | Vocabulario, Regras de Producao de uma gramatica            |
11   | e a Tabela de Analise LR(k) para a esta gramatic            |
12   +-------------------------------------------------------------+*/
13
14 #define NSIMBOLOS 7
15 #define NREGRAS   4
16 #define NESTADOS  9
17 char alfabeto[NSIMBOLOS+1] = "SLa[];#";
18 char *regras[NREGRAS] =
19       { "S::=a",
20         "S::=[L]",
21         "L::=S",
22         "L::=L;S" };
23
24 struct {
25   char acao;
26   int indice;
27 } TabSint[NESTADOS][NSIMBOLOS] =
28    { 'e',1, '␣',0, 'e',2, 'e',3, '␣',0, '␣',0, '␣',0,
29      '␣',0, '␣',0, '␣',0, '␣',0, '␣',0, '␣',0, 'a',0,
30      '␣',0, '␣',0, 'r',1, 'r',1, 'r',1, 'r',1, 'r',1,
31      'e',4, 'e',5, 'e',2, 'e',3, '␣',0, '␣',0, '␣',0,
32      '␣',0, '␣',0, 'r',3, 'r',3, 'r',3, 'r',3, 'r',3,
33      '␣',0, '␣',0, '␣',0, '␣',0, 'e',6, 'e',7, '␣',0,
34      '␣',0, '␣',0, 'r',2, 'r',2, 'r',2, 'r',2, 'r',2,
35      'e',8, '␣',0, 'e',2, 'e',3, '␣',0, '␣',0, '␣',0,
36      '␣',0, '␣',0, 'r',4, 'r',4, 'r',4, 'r',4, 'r',4,
37    };
38
39 /*
40 #define NSIMBOLOS 6
41 #define NREGRAS   4
42 #define NESTADOS  10
43 char alfabeto[NSIMBOLOS+1] = "E+*ab#";
44 char *regras[NREGRAS] =
45       { "E::=+EE",
46         "E::=*EE",
47         "E::=a",
48         "E::=b" };
49
50 struct {
51   char acao;
52   int indice;
53 } TabSint[NESTADOS][NSIMBOLOS] =
54    { 'e',1, 'e',2, 'e',3, 'e',4, 'e',5, ' ',0,
55      ' ',0, ' ',0, ' ',0, ' ',0, ' ',0, 'a',0,
56      'e',6, 'e',2, 'e',3, 'e',4, 'e',5, ' ',0,
57      'e',7, 'e',2, 'e',3, 'e',4, 'e',5, ' ',0,
58      ' ',0, 'r',3, 'r',3, 'r',3, 'r',3, 'r',3,
59      ' ',0, 'r',4, 'r',4, 'r',4, 'r',4, 'r',4,
60      'e',8, 'e',2, 'e',3, 'e',4, 'e',5, ' ',0,
61      'e',9, 'e',2, 'e',3, 'e',4, 'e',5, ' ',0,
62      ' ',0, 'r',1, 'r',1, 'r',1, 'r',1, 'r',1,
63      ' ',0, 'r',2, 'r',2, 'r',2, 'r',2, 'r',2
64    };
65
```

```
66  #define NSIMBOLOS 5
67  #define NREGRAS   2
68  #define NESTADOS  6
69  char alfabeto[NSIMBOLOS+1] = "Sacb#";
70  char *regras[NREGRAS] =
71          { "S::=aSc",
72            "S::=b"};
73
74  struct {
75    char acao;
76    int indice;
77  } TabSint[NESTADOS][NSIMBOLOS] =
78     { 'e',1, 'e',2, ' ',0, 'e',3, ' ',0,
79       ' ',0, ' ',0, ' ',0, ' ',0, 'a',0,
80       'e',4, 'e',2, ' ',0, 'e',3, ' ',0,
81       ' ',0, 'r',2, 'r',2, 'r',2, 'r',2,
82       ' ',0, ' ',0, 'e',5, ' ',0, ' ',0,
83       ' ',0, 'r',1, 'r',1, 'r',1, 'r',1,
84     };
85
86  #define NSIMBOLOS 7
87  #define NREGRAS   4
88  #define NESTADOS  9
89  char alfabeto[NSIMBOLOS+1] = "SL(),a#";
90  char *regras[NREGRAS] =
91          { "S::=(L)",
92            "S::=a",
93            "L::=L,S",
94            "L::=S"};
95
96  struct {
97    char acao;
98    int indice;
99  } TabSint[NESTADOS][NSIMBOLOS] =
100    { 'e',1, ' ',0, 'e',2, ' ',0, ' ',0, 'e',3, ' ',0,
101      ' ',0, ' ',0, ' ',0, ' ',0, ' ',0, ' ',0, 'a',0,
102      'e',4, 'e',5, 'e',2, ' ',0, ' ',0, 'e',3, ' ',0,
103      ' ',0, ' ',0, 'r',2, 'r',2, 'r',2, 'r',2, 'r',2,
104      ' ',0, ' ',0, 'r',4, 'r',4, 'r',4, 'r',4, 'r',4,
105      ' ',0, ' ',0, ' ',0, 'e',6, 'e',7, ' ',0, ' ',0,
106      ' ',0, ' ',0, 'r',1, 'r',1, 'r',1, 'r',1, 'r',1,
107      'e',8, ' ',0, 'e',2, ' ',0, ' ',0, 'e',3, ' ',0,
108      ' ',0, ' ',0, 'r',3, 'r',3, 'r',3, 'r',3, 'r',3,
109    };
110
111
112 #define NSIMBOLOS 7
113 #define NREGRAS   4
114 #define NESTADOS  9
115 char alfabeto[NSIMBOLOS+1] = "SLa[];#";
116 char *regras[NREGRAS] =
117         { "S::=a",
118           "S::=[L]",
119           "L::=S",
120           "L::=L;S"};
121
122 struct {
123   char acao;
```

```
124     int indice;
125  } TabSint[NESTADOS][NSIMBOLOS] =
126      { 'e',1, ' ',0, 'e',2, 'e',3, ' ',0, ' ',0, ' ',0,
127        ' ',0, ' ',0, ' ',0, ' ',0, ' ',0, ' ',0, 'a',0,
128        ' ',0, ' ',0, 'r',1, 'r',1, 'r',1, 'r',1, 'r',1,
129        'e',4, 'e',5, 'e',2, 'e',3, ' ',0, ' ',0, ' ',0,
130        ' ',0, ' ',0, 'r',3, 'r',3, 'r',3, 'r',3, 'r',3,
131        ' ',0, ' ',0, ' ',0, ' ',0, 'e',6, 'e',7, ' ',0,
132        ' ',0, ' ',0, 'r',2, 'r',2, 'r',2, 'r',2, 'r',2,
133        'e',8, ' ',0, 'e',2, 'e',3, ' ',0, ' ',0, ' ',0,
134        ' ',0, ' ',0, 'r',4, 'r',4, 'r',4, 'r',4, 'r',4,
135      };
136  */
137
138  /*+-------------------------------------------------------------------+
139    | Pilha sintatica utilizada pelo algoritmo de analise LR(K          |
140    +-------------------------------------------------------------------+*/
141  struct {
142    char elem;
143    int ind;
144  } P[20];
145
146  void traco (int);
147  void strins (char *, int, char *, int);
148
149  /*+-------------------------------------------------------------------+
150    | Programa principal que implementa o algoritmo de analise LR(K)    |
151    | Este programa LE uma sentenca e verifica se esta sentenca eh      |
152    | valida ou para a gramatica representada na tabela de analise.     |
153    | Durante o processo de analise o programa apresenta, de forma      |
154    | tabular, os valores das estruturas a cada passo.                  |
155    +-------------------------------------------------------------------+*/
156  int main()
157  {
158    int i, j, k, termino, reduzido, indice, ind, tam, passo,
159        nreducao, indreduz = -1, reducoes[50];
160    char sentenca[40], pilha[60], cadeia[40], str[40];
161    char s, simboloreduzido = ' ', acao;
162
163    P[0].elem = 'e';
164    P[0].ind = 0;
165    i = termino = reduzido = 0;
166
167    printf ("\nDigite a sentenca: ");
168    gets (sentenca);
169
170    strcat (sentenca, "#");
171    indice = 0;
172    passo = 0;
173    printf ("PASSO %-30s S.R. %-15s %s\n", "PILHA", "SENTENCA", "ACAO");
174    traco (79);
175    while (!termino) {
176      if (reduzido)
177         s = simboloreduzido;
178      else
179         s = sentenca[indice];
180      for (j=0; alfabeto[j] != s && j <= strlen(alfabeto); j++);
181      if (alfabeto[j] != s) {
```

```
182         printf ("\nERRO: o simbolo <%c> nao e reconhecido nesta linguagem",
                s);
183         printf ("\n\nDigite '.' para terminar!");
184         while (getchar()!='.');
185         exit(10);
186      }
187      acao = TabSint[P[i].ind][j].acao;
188      ind = TabSint[P[i].ind][j].indice;
189      for (j = 0, k = indice; sentenca[k]; j++, k++)
190         cadeia [j] = sentenca[k];
191      cadeia [j] = '\0';
192      strcpy (pilha, "");
193      for (k = 0; k <= i; k++) {
194         sprintf (str,"%c%d", P[k].elem, P[k].ind);
195         strcat (pilha, str);
196      }
197      printf ("%3d   %-30s %c    %-15s  %c%d\n",
198               passo++, pilha, simboloreduzido, cadeia, acao, ind);
199      switch (acao) {
200         case 'e':  i++;
201                    P[i].elem = s;
202                    P[i].ind = ind;
203                    if (reduzido) {
204                        reduzido = 0;
205                        simboloreduzido = ' ';
206                    }
207                    else
208                       indice++;
209                    break;
210         case 'r':  tam = strlen (regras[ind-1]);
211                    i = i - tam + 4;
212                    reduzido = 1;
213                    reducoes[++indreduz] = ind-1;
214                    simboloreduzido = regras[ind-1][0];
215                    break;
216         case 'a':  termino = 1;
217                    printf ("\nA sentenca <%s> esta correta", sentenca);
218                    break;
219         case ' ':  printf ("\nA sentenca <%s> NAO e reconhecida", sentenca);
220                    printf ("\n\nDigite '.' para terminar!");
221                    while (getchar()!='.');
222                    exit (1);
223      }
224      /*getchar();*/
225   }
226   /*- Mostra a serie de derivacoes mais a direita para produzir a sentenca
          -*/
227   printf ("\n\nGramatica:");
228   printf ("\n==========\n");
229   for (i = 0; i < NREGRAS; i++)
230      printf ("(%d).  %s\n", i+1, regras[i]);
231   printf ("\n\nSequencia de Derivacoes mais a direita:");
232   printf ("\n=================================\n\n");
233   sentenca[0] = regras[0][0];
234   sentenca[1] = '\0';
235   printf ("%s ", sentenca);
236   while (indreduz >= 0)
237   {
```

```
238     i = strlen (sentenca) - 1;
239     while (sentenca[i] < 'A' || sentenca[i] > 'Z') i--;
240     nreducao = reducoes[indreduz--];
241     strins (regras[nreducao], 4, sentenca, i);
242     printf ("=%d=> %s ", nreducao+1, sentenca);
243   }
244   printf ("\n\nDigite '.' para terminar!");
245   while (getchar()!='.');
246 }
247
248 /*+-------------------------------------------------------------+
249   | Desenha uma linha de hifens na tela.                        |
250   +-------------------------------------------------------------+*/
251 void traco (int i) {
252   int k;
253   for (k = 0; k < i; k++)
254      printf ("-");
255   printf ("\n");
256 }
257
258 /*+-------------------------------------------------------------+
259   |Insere substring da pos1 ate o final de s1 na pos2 do string s2 |
260   +-------------------------------------------------------------+*/
261 void strins (char *s1, int pos1, char *s2, int pos2) {
262   int i, tam_s1, tam_s2;
263   for (tam_s1 = 0; s1[tam_s1]; tam_s1++);
264   for (tam_s2 = 0; s2[tam_s2]; tam_s2++);
265   for (i = tam_s2; i >= pos2; i--)
266      s2[i+tam_s1-pos1-1] = s2[i];
267   for (i = pos1; i < tam_s1; i++)
268      s2[i+pos2-pos1] = s1[i];
269   s2[tam_s1+tam_s2-pos1]='\0';
270 }
```

O resultado da execução desse programa, considerando a entrada da sentença [a;[a;a]] é:

PASSO	PILHA	S.R.	SENTENCA	ACAO
0	e0		[a;[a;a]]#	e3
1	e0[3		a;[a;a]]#	e2
2	e0[3a2		;[a;a]]#	r1
3	e0[3	S	;[a;a]]#	e4
4	e0[3S4		;[a;a]]#	r3
5	e0[3	L	;[a;a]]#	e5
6	e0[3L5		;[a;a]]#	e7
7	e0[3L5;7		[a;a]]#	e3
8	e0[3L5;7[3		a;a]]#	e2
9	e0[3L5;7[3a2		;a]]#	r1
10	e0[3L5;7[3	S	;a]]#	e4
11	e0[3L5;7[3S4		;a]]#	r3
12	e0[3L5;7[3	L	;a]]#	e5
13	e0[3L5;7[3L5		;a]]#	e7
14	e0[3L5;7[3L5;7		a]]#	e2
15	e0[3L5;7[3L5;7a2		]]#	r1
16	e0[3L5;7[3L5;7	S	]]#	e8
17	e0[3L5;7[3L5;7S8		]]#	r4
18	e0[3L5;7[3	L	]]#	e5

```
 19    e0[3L5;7[3L5                          ]]#          e6
 20    e0[3L5;7[3L5]6                        ]#           r2
 21    e0[3L5;7                        S     ]#           e8
 22    e0[3L5;7S8                            ]#           r4
 23    e0[3                            L     ]#           e5
 24    e0[3L5                                ]#           e6
 25    e0[3L5]6                              #            r2
 26    e0                              S     #            e1
 27    e0S1                                  #            a0

A sentenca <[a;[a;a]]#> esta correta

Gramatica:
==========
(1).  S::=a
(2).  S::=[L]
(3).  L::=S
(4).  L::=L;S


Sequencia de Derivacoes mais a direita:
=======================================

S =2=> [L] =4=> [L;S] =2=> [L;[L]] =4=> [L;[L;S]] =1=> [L;[L;a]] =3=> [L;[S
    ;a]] =1=> [L;[a;a]] =3=> [S;[a;a]] =1=> [a;[a;a]]
```

3.4.2 Construção da Tabela de Análise Sintática LR

Já temos um método automático para verificar a sintaxe em linguagens de programação: o algoritmo de análise LR(k). Nesse algoritmo, todo trabalho do analisador sintático é orientado por uma tabela de análise que é construída a partir da gramática LR para linguagem. A questão é: como definir as ações da tabela de análise LR(k). Antes de apresentar o algoritmo para o cálculo da coleção de estados (linhas da tabela) e para definição dos valores da tabela, precisamos fazer algumas definições.

Definições:

1. **Item**: É uma regra de produção na qual foi marcada uma posição na cadeia do lado direito; esta posição será indicada por meio do símbolo • (ponto). Exemplo: Seja a gramática:

$$\begin{aligned} E &\rightarrow +EE \\ E &\rightarrow *EE \\ E &\rightarrow a \\ E &\rightarrow b \end{aligned}$$

   O conjunto de itens derivados desta gramática é: $\{E \rightarrow \bullet + EE | + \bullet EE | + E \bullet E | + EE \bullet | \bullet *EE | * \bullet EE | * E \bullet E | * EE \bullet | \bullet a | a \bullet | \bullet b | b \bullet\}$. O conjunto de itens para uma gramática é sempre finito e será utilizado para construir os estados da tabela.

2. **Estado**: É um conjunto de itens. A presença no topo da pilha de um estado contendo um item da forma $A \rightarrow \alpha \bullet \beta$ indica que já foi processada e deslocada para pilha a parte inicial *alpha* do redutendo $\alpha\beta$. O estado contendo o item da forma $A \rightarrow \alpha \bullet$ indica um redutendo completo (item completo), o que indica que a próxima ação será uma REDUÇÃO.

3. **Fecho**: Diremos que um conjunto K de itens é fechado se para todo item K da forma $A \to \alpha \bullet B\beta$, todos os itens da forma $B \to \bullet\gamma$ estão em K. Denotaremos por FECHO(K) o menor conjunto fechado que contém K. Exemplo: Consideremos os seguintes conjuntos de itens:

$$\begin{array}{lcl} K_1 & = & \{E \to + \bullet EE\} \\ K_2 & = & \{E \to +E \bullet E | * \bullet EE | \bullet a\} \\ K_3 & = & \{E \to \bullet b\} \end{array}$$

Para a gramática:

$$\begin{array}{lcl} E & \to & +EE \\ E & \to & *EE \\ E & \to & a \\ E & \to & b \end{array}$$

Então seus fechos são:

$$\begin{array}{lcl} FECHO(K_1) & = & \{E \to + \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} \\ FECHO(K_2) & = & \{E \to +E \bullet E | * \bullet EE | \bullet a | \bullet +EE | \bullet *EE | \bullet b\} \\ FECHO(K_3) & = & \{E \to \bullet b\} \end{array}$$

4. **Transfere**: Vamos agora analisar como determinar as entradas da forma e_j na tabela de análise. Suponhamos que um estado e_i contenha um item incompleto da forma $A \to \alpha \bullet X\beta$. A presença deste estado no topo da pilha de análise indica que se o próximo símbolo a ser consultado for X, então terá sido processada a parte αX do redutendo $\alpha X\beta$, devendo ser empilhado portando um estado que contenha o item $A \to \alpha X \bullet \beta$ (com a marca depois do símbolo X). Definiremos então a função TRANSFERE (K,X), como sendo o conjunto fechado de todos os itens da forma $A \to \alpha X \bullet \beta$ tais que o item $A \to \alpha \bullet X\beta$ está em K.

   Consideremos a gramática anterior e os conjuntos:

$$\begin{array}{lcl} K_1 & = & \{E \to * \bullet EE\} \\ K_2 & = & FECHO(K_1) = \{E \to + \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} \\ K_3 & = & \{E \to \bullet b\} \end{array}$$

   Tem-se então:

$$\begin{array}{lcl} TRANSFERE(K_1, *) & = & \{\} \\ TRANSFERE(K_1, E) & = & \{E \to * \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} \\ TRANSFERE(K_2, +) & = & \{E \to + \bullet EE | \bullet +EE | \bullet *EE | \bullet a | \bullet b\} = K_2 \\ TRANSFERE(K_2, a) & = & \{E \to a\bullet\} \\ TRANSFERE(K_3, E) & = & \{\} \\ TRANSFERE(K_3, B) & = & \{E \to b\bullet\} \end{array}$$

As funções FECHO e TRANSFERE permitem a construção dos estados que serão identificados como as linhas da tabela de análise. Assim:

Algoritmo para determinar a coleção C de estados de uma gramática:

1. Adota-se o estado $e_0 = FECHO(\{S' \rightarrow \bullet S\#\})$ como sendo o valor inicial da coleção C. Observe que deve ser acrescentada à gramática, uma regra para caracterizar o instante que a sentença toda será reduzida para o símbolo inicial. O símbolo terminal # é artificialmente acrescentado a gramática para marcar o fim da sentença que será analisada.

2. Se existe um estado e de C e um símbolo X de Σ (vocabulário) tais que $e' = TRANSFERE(e, X) \neq \emptyset$ e $e' \notin C$, então e' é acrescentado à coleção C.

3. O passo 2 é repetido até que não se possam acrescentar mais estados à coleção C.

C é o conjunto de estado tipo LR(0) da gramática.

Exemplo: Consideremos a gramática:

$$\begin{array}{lcl} E' & \rightarrow & E\# \\ E & \rightarrow & +EE \\ E & \rightarrow & *EE \\ E & \rightarrow & a \\ E & \rightarrow & b \end{array}$$

Aplicando-se o algoritmo anterior obtém-se os seguintes estados para esta gramática:

$$
\begin{aligned}
e_0 = \quad & FECHO(\{E' \to \bullet E\#\}) = \{E' \to \bullet E\# \\
& \qquad\qquad E \to \bullet + EE| \bullet * EE| \bullet a| \bullet b\} \\
e_1 = \quad & TRANSFERE(e_0, E) = \{E' \to E \bullet \#\} \\
e_2 = \quad & TRANSFERE(e_0, +) = \{E \to + \bullet EE| \bullet + EE| \bullet * EE| \bullet a| \bullet b\} \\
e_3 = \quad & TRANSFERE(e_0, *) = \{E \to * \bullet EE| \bullet + EE| \bullet * EE| \bullet a| \bullet b\} \\
e_4 = \quad & TRANSFERE(e_0, a) = \{E \to a\bullet\} \\
e_5 = \quad & TRANSFERE(e_0, a) = \{E \to b\bullet\} \\
e_6 = \quad & TRANSFERE(e_2, E) = \{E \to +E \bullet E| \bullet + EE| \bullet * EE| \bullet a| \bullet b\} \\
& TRANSFERE(e_2, +) = e_2 \\
& TRANSFERE(e_2, *) = e_3 \\
& TRANSFERE(e_2, a) = e_4 \\
& TRANSFERE(e_2, b) = e_5 \\
e_7 = \quad & TRANSFERE(e_3, E) = \{E \to *E \bullet E| \bullet + EE| \bullet * EE| \bullet a| \bullet b\} \\
& TRANSFERE(e_3, +) = e_2 \\
& TRANSFERE(e_3, *) = e_3 \\
& TRANSFERE(e_3, a) = e_4 \\
& TRANSFERE(e_3, b) = e_5 \\
e_8 = \quad & TRANSFERE(e_6, E) = \{E \to +EE\bullet\} \\
& TRANSFERE(e_6, +) = e_2 \\
& TRANSFERE(e_6, *) = e_3 \\
& TRANSFERE(e_6, a) = e_4 \\
& TRANSFERE(e_6, b) = e_5 \\
e_9 = \quad & TRANSFERE(e_7, E) = \{E \to *EE\bullet\} \\
& TRANSFERE(e_7, +) = e_2 \\
& TRANSFERE(e_7, *) = e_3 \\
& TRANSFERE(e_7, a) = e_4 \\
& TRANSFERE(e_7, b) = e_5
\end{aligned}
$$

A tabela de análise LR construída com essa coleção de estados é:
Observações:

- Não foi calculado o estado $TRANSFERE(e_1, \#)$ pois esta situação corresponde à aceitação da sentença de entrada.
- Os estados calculados contêm apenas um item completo (e_4, e_5, e_8, e_9), ou apenas itens incompletos ($e_0, e_1, e_2, e_3, e_6, e_7$). Os estados constituídos de itens completos indicam que os últimos estados empilhados correspondem a um redutendo, e que portanto deve haver redução, independente do próximo símbolo.

Exercícios

1. Calcule a tabela de análise LR(0) para a seguinte gramática:

Tabela	E	+	*	a	b	#
e_o	e_1	e_2	e_3	e_4	e_5	
e_1						a
e_2	e_6	e_2	e_3	e_4	e_5	
e_3	e_7	e_2	e_3	e_4	e_5	
e_4		r_3	r_3	r_3	r_3	r_3
e_5		r_4	r_4	r_4	r_4	r_4
e_6	e_8	e_2	e_3	e_4	e_5	
e_7	e_9	e_2	e_3	e_4	e_5	
e_8		r_1	r_1	r_1	r_1	r_1
e_9		r_2	r_2	r_2	r_2	r_2

Tabela 3.3: Tabela de Análise LR

$$\begin{array}{lcl} E & \rightarrow & T* \\ T & \rightarrow & \$E \\ T & \rightarrow & a \end{array}$$

2. Usando a tabela de análise da questão anterior e o algoritmo de análise LR(k), faça um acompanhamento para verificar a validade da sentença $a*#

3.5 Yacc/Bison - Geradores e Analisadores Sintáticos [1]

As gramáticas para Yacc estão definidas usando uma notação similar a notação de Backus Naur Form (BNF). Esta técnica foi criada por John Backus e Peter Naur, e usada para descrever a sintaxe de ALGOL60. Uma gramática BNF pode ser usada para especificar gramáticas livres de contexto. Muitas construções sintáticas de linguagens de programação modernas podem ser expressas usando BNF. Por exemplo, a gramática de expressões que multiplica e adiciona números é:

$$\begin{array}{lcl} (1) \quad E & \rightarrow & E + E \\ (2) \quad E & \rightarrow & E * E \\ (3) \quad E & \rightarrow & id \end{array}$$

Três regras de produção foram especificadas. Os termos que aparecem no lado esquerdo das regras de produção, como E, são não-terminais. Termos como id (identificadores) são terminais (tokens retornados pelo analisador léxico) e somente podem aparecer no lado direito de uma regra de produção. Esta gramática especifica que expressões podem ser a soma de duas expressões, o produto de duas expressões ou um identificador. Nós podemos usar esta gramática para gerar expressões:

$$\begin{array}{lcll} E & \Rightarrow & E * E & (r2) \\ & \Rightarrow & E * z & (r3) \\ & \Rightarrow & E + E * z & (r1) \\ & \Rightarrow & E + y * z & (r3) \\ & \Rightarrow & x + y * z & (r3) \end{array}$$

[1] Tradução do texto de Thomas Niemann (http://epaperpress.com/lexandyacc/index.html).

A cada passo expande-se um termo, trocando um símbolo não-terminal no lado esquerdo de um regra pela sequência de símbolos terminais e/ou não terminais no lado direito da mesma regra. O número no lado direito indica que regra foi aplicada. Para analisar a expressão, nós usamos uma operação inversa. Ao invés de começar com o símbolo não-terminal inicial e gerar a expressão da gramática, nós devemos reduzir a expressão para um único símbolo não-terminal (o símbolo inicial). Este método, mais simples de implementar é chamado de análise ascendente ou análise de redução e deslocamento (bottom-up or shift-reduce parsing), e usa uma pilha para armazenar os termos. Aqui está o mesma da derivação, mas em ordem inversa:

1	$\bullet x + y * z$	$shift$		
2	$x \bullet + y * z$	$reduce(r3)$		
3	$E \bullet + y * z$	$shift$		
4	$E + \bullet y * z$	$shift$		
5	$E + y \bullet * z$	$reduce(r3)$		
6	$E + E \bullet * z$	$shift$		
7	$E + E * \bullet z$	$shift$		
8	$E + E * z \bullet$	$reduce(r3)$		
9	$E + E * E \bullet$	$reduce(r2)$	$emit$	$multiply$
10	$E + E \bullet$	$reduce(r1)$	$emit$	add
11	$E \bullet$	$accept$		

Os termos do lado esquerdo do ponto estão na pilha, enquanto o restante da entrada está no lado direito do ponto. Quando o topo da pilha corresponde ao lado direito de uma regra nós trocamos os tokens correspondentes com o símbolo não terminal do lado esquerdo da regra de produção. Conceitualmente, os tokens do lado direito são desempilhados e o símbolo não terminal do lado esquerdo da regra é empilhado. Os seqüência de tokens encontrados são chamados de redutendo (handle), e nós estamos reduzindo o redutendo para o símbolo do lado esquerdo da regra de produção. Este processo continua até que tenhamos deslocado todos os símbolos para a pilha e somente o símbolo não terminal inicial permanecer na pilha. No passo 1 nós deslocamos o x para a pilha. No passo 2 nós aplicamos a regra r3 para a pilha, trocando o símbolo x por um E. Nós continuamos deslocando e reduzindo, até que um único não terminal, o símbolo inicial, permaneça na pilha. No passo 9, quando nós reduzimos usando a regra r2, escrevemos a instrução de multiplicação. Da mesma forma, escrevemos a instrução de soma no passo 10. Portanto a multiplicação tem precedência maior que a soma.

Considere, entretanto, o deslocamento no passo 6. Ao invés de deslocar, nós poderíamos ter reduzido, aplicando a regra r1. Isto resultaria na adição tendo maior precedência que a multiplicação. Isto é conhecido como conflito de redução-deslocamento (shift-reduce conflict). Nossa gramática é ambígua, pois existe mais de uma derivação que produzirá a expressão. Neste caso, a precedência dos operadores é afetada. Outro exemplo, a associatividade na regra:

$$E \rightarrow E + E$$

é ambígua, pois podemos recursivamente substituir o não terminal mais à esquerda ou mais à direita na regra. Para resolver esta situação, nós deveríamos reescrever a gramática, ou indicar para Yacc a precedência que existe entre os operadores. O último método é mais simples e será demonstrado numa prática posterior.

A seguinte gramática tem um conflito de redução e redução. Com um id na pilha nós podemos reduzir para T ou reduzir para E.

$$\begin{aligned} E &\rightarrow T \\ E &\rightarrow id \\ T &\rightarrow id \end{aligned}$$

Yacc executa uma ação default quando há um conflito. Para conflito de redução-deslocamento, yacc deslocará o símbolo para pilha. Para conflito de redução-redução ele usará a primeira regra da listagem. Ele também apresenta uma mensagem indicando a existência do conflito. O aviso pode ser excluído tornando a gramática não ambígua. Vários métodos para remover ambigüidade serão apresentados na seção seguinte.

Implementação de uma calculadora de expressões usando Yacc

```
1 ... definicoes ...
2 %%
3 ... regras ...
4 %%
5 ... subrotinas ...
```

O arquivo de entrada para yacc é dividido em três seções. A seção de definições consiste de declaração de tokens, e código C delimintado por "%"e "%". A gramática BNF é colocada na seção de regras e as rotinas do usuário são colocados na seção de subrotinas.

Podemos ilustrar isto melhor através de um exemplo que contrói uma pequena calculadora que pode adicionar e subtrair números. Começaremos examinando a ligação entre lex e yacc. Abaixo está a seção de definição para o arquivo de entrada yacc:

```
1 %token INTEGER
```

Esta definição declara um token INTEGER. Quando nós rodamos yacc, ele gera o analisador sintático (parser) no arquivo y.tab.c, e também cria um arquivo (include file), y.tab.h:

```
1 #ifndef YYSTYPE
2 #define YYSTYPE int
3 #endif
4 #define INTEGER 258
5 extern YYSTYPE yylval;
```

Lex inclui este arquivo e utiliza a definição para o valor do token. Para obter tokens, yacc chama yylex. A função yylex é do tipo int, e retorna o código numérico do token. Valores associados com o token são retornados por lex na variável yylval. Por exemplo,

```
1 [0-9]+        {
2                   yylval = atoi(yytext);
3                   return INTEGER;
4               }
```

armazenaria o valor do inteiro em yylval, e retorna o token INTEGER para yacc. O tipo de yylval é determinado por YYSTYPE. Como o tipo default é inteiro, o valor retornado está correto para este caso. Os valores de tokens entre 0 e 255 são reservados para os caracteres. Por exemplo, se você tem um regra como

```
[-+]        return *yytext;     /* return operator */
```

será retornado o valor caracter para os operadores de subtração e soma. Note que nós colocamos o caracter menos antes do caracter mais porque senão ele poderia ser confundido com o designador de intervalo numa classe de caracteres. Os valores do tokens gerados começam normalmente do valor 258 (Lex ainda reseva alguns valores para fim de arquivo e códigos de erro). Aqui está uma especificação lex completa para nossa calculadora:

```
%{
    #include "y.tab.h"
    #include <stdlib.h>
    void yyerror(char *);
%}

%%

[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

[-+\n]      return *yytext;

[ \t]       ; /* skip whitespace */

.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Internamente, Yacc mantém duas pilhas na memória, a pilha de análise e a pilha de valores. A pilha de análise contém terminais e não-terminais e representa o estado de análise corrente. A pilha de valores é um vetor de elementos do tipo YYSTYPE, e associa um valor com cada elemento da pilha de análise. Por exemplo, quando lex retorna um token INTEGER, yacc desloca este token para a pilha de análise. Ao mesmo tempo, o yylval correspondente é deslocado para a pilha de valores. A pilha de análise e de valores estão sempre sincronizadas. Abaixo está a especificação do arquivo de entrada yacc para a nossa calculadora:

```
%{
    int yylex(void);
    void yyerror(char *);
%}


%token INTEGER

%%

program:
        program expr '\n'       { printf("%d\n", $2); }
        |
        ;
```

```
15
16 expr:
17         INTEGER                      { $$ = $1; }
18         | expr '+' expr              { $$ = $1 + $3; }
19         | expr '-' expr              { $$ = $1 - $3; }
20         ;
21
22 %%
23
24 void yyerror(char *s) {
25     printf("%s\n", s);
26 }
27
28 int main(void) {
29     yyparse();
30     return 0;
31 }
```

A seção de regras descreve a gramática BNF discutida anteriormente. O lado esquerdo da regra de produção, ou não-terminal, deve estar todo justificado à esquerda, seguido por dois pontos. Este é seguido pelo lado direito da regra de produção. As ações associadas as regras devem estar entre chaves. Pela utilização da recursão à esquerda, nós especificamos que um programa consiste de zero ou mais expressões. Cada expressão terminada por nova linha (ENTER). Quando uma nova linha é detectada, nós escrevemos o valor associado com a expressão. Quando nós aplicamos a regra:

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

Nós trocamos o lado direito da regra de produção na pilha de análise com o símbolo não terminal no lado esquerdo da mesma regra. Neste caso nós desempilhamos "expr '+' expr"e empilhamos "expr". Nós reduzimos a pilha desempilhando três termos e empilhando de volta um termo. Nós podemos referenciar as posições na pilha de valores no código em C especificando "$1"para o primeiro termo no lado direito da regra de produção, "$2"para o segundo termo, e assim por diante. "$$"determina o valor do topo da pilha depois que a redução é executada. As ações acima adicionam os valores associados com as duas expressões, desempilha três termos da pilha, e empilha de volta a soma. Então a pilha de análise e de valores permanecem sincronizadas.

Os valores numéricos estão inicialmente todos na pilha quanto reduzimos de INTEGER para expr. Depois que INTEGER é deslocado para a pilha, aplicamos a regra:

```
1 expr: INTEGER                        { $$ = $1; }
```

O token INTEGER é desempilhado da pilha de análise, seguido pelo empilhamento de expr. Para a pilha de valores, nós desempilhamos o valor inteiro e então empilhamos ele de volta. Em outras palavras, nós não fazemos nada. De fato , esta é uma operação default e não precisa ser especificada. Finalmente, quando uma nova linha é encontrada, o valor associado com expr será apresentado na saída.

No caso de erro de sintaxe, yacc chama a função definida pelo usuário yyerror. Se você necessita modificar a interface para yyerror, você pode alterar a implementação que Yacc inclui automaticamente para satisfazer as suas necessidades. A última função em nossa especificação Yacc é main. Este exemplo tem uma gramática ambígua. Yacc emitirá avisos de conflito shift-reduce, mas processará a gramática usando o deslocamento como operação default.

Nova versão da calculadora

Nesta seção nós estenderemos a calculadora da seção anterior para incorporar algumas novas funcionalidades. As novas características incluem operadores aritméticos de multiplicação e divisão. Parênteses pode ser usado para alterar a precedência das operações e variáveis de uma única letra podem ser especificadas para comandos de atribuição. O exemplo seguinte ilustra algumas entradas e saídas da calculadora:

```
user:  3 * (4 + 5)
calc:  27
user:  x = 3 * (5 + 4)
user:  y = 5
user:  x
calc:  27
user:  y
calc:  5
user:  x + 2*y
calc:  37
```

O analisador léxico retorna os tokens VARIABLE e INTEGER. Para variável, yylval especifica o índice para sym, nossa tabela de símbolos.Para este programa, sym simplesmente pega o valor da variável associada. Quando o token INTEGER é retornado, yylval contém o número encontrado. Abaixo a especificação para lex:

```
%{
    #include <stdlib.h>
    #include "y.tab.h"
    void yyerror(char *);
%}

%%

    /* variables */
[a-z]       {
                yylval = *yytext - 'a';
                return VARIABLE;
            }

    /* integers */
[0-9]+      {
                yylval = atoi(yytext);
                return INTEGER;
            }

    /* operators */
[-+()=/*\n] { return *yytext; }

    /* skip whitespace */
[ \t]        ;

    /* anything else is an error */
.           yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

A especificação para Yacc é apresentada em seguida. Os tokens para INTEGER e VARIABLE são usados por yacc para criar definições do tipo #defines no arquivo y.tab.h para usar em lex. Esta é seguida pela definição da aritmética de operadores. Nós podemos especificar %left, para associatividade à esquerda ou %right, para associatividade à direita.A última definição listada é a quem tem maior precedência. Então multiplicação e divisão tem precedência maior que adição e subtração. Todos os quatro operadores são associados à esquerda. Usando esta técnica simples, nós tiramos a ambiguidade de nossa gramática:

```
1  %token INTEGER VARIABLE
2  %left '+' '-'
3  %left '*' '/'
4
5  %{
6      void yyerror(char *);
7      int yylex(void);
8      int sym[26];
9  %}
10
11 %%
12
13 program:
14         program statement '\n'
15         |
16         ;
17
18 statement:
19         expr                        { printf("%d\n", $1); }
20         | VARIABLE '=' expr         { sym[$1] = $3; }
21         ;
22
23 expr:
24         INTEGER
25         | VARIABLE                  { $$ = sym[$1]; }
26         | expr '+' expr             { $$ = $1 + $3; }
27         | expr '-' expr             { $$ = $1 - $3; }
28         | expr '*' expr             { $$ = $1 * $3; }
29         | expr '/' expr             { $$ = $1 / $3; }
30         | '(' expr ')'              { $$ = $2; }
31         ;
32
33 %%
34
35 void yyerror(char *s) {
36     printf("%s\n", s);
37 }
38
39 int main(void) {
40     yyparse();
41     return 0;
42 }
```

3.6 Analisador Sintático para a Linguagem Simples

A gramática da linguagem Simples é:

$$
\begin{aligned}
\mathit{programa} &\rightarrow \mathit{cabecalho}\ \ \mathit{variaveis}\ \ \mathit{T\_INICIO}\ \ \mathit{lista\_comandos}\ \ \mathit{T\_FIM}\\
\mathit{cabecalho} &\rightarrow \mathit{T\_PROGRAMA}\ \ \mathit{T\_IDENTIF}\\
\mathit{variaveis} &\rightarrow \lambda\\
&\mid\ \mathit{declaracao\_variaveis}\\
\mathit{declaracao\_variaveis} &\rightarrow \mathit{tipo}\ \ \mathit{lista\_variaveis}\ \ \mathit{declaracao\_variaveis}\\
&\mid\ \mathit{tipo}\ \ \mathit{lista\_variaveis}\\
\mathit{tipo} &\rightarrow \mathit{T\_LOGICO}\\
&\mid\ \mathit{T\_INTEIRO}\\
\mathit{lista\_variaveis} &\rightarrow \mathit{T\_IDENTIF}\ \ \mathit{lista\_variaveis}\\
&\mid\ \mathit{T\_IDENTIF}\\
\mathit{lista\_comandos} &\rightarrow\ \ \mathit{comando}\ \ \mathit{lista\_comandos}\\
&\mid\ \lambda\\
\mathit{comando} &\rightarrow \mathit{entrada\_saida}\\
&\mid\ \mathit{repeticao}\\
&\mid\ \mathit{selecao}\\
&\mid\ \mathit{atribuicao}\\
\mathit{entrada\_saida} &\rightarrow \mathit{leitura}\ \mid\ \mathit{escrita}\\
\mathit{leitura} &\rightarrow \mathit{T\_LEIA}\ \ \mathit{T\_IDENTIF}\\
\mathit{escrita} &\rightarrow \mathit{T\_ESCREVA}\ \ \mathit{expressao}\\
\mathit{repeticao} &\rightarrow \mathit{T\_ENQTO}\ \ \mathit{expressao}\ \ \mathit{T\_FACA}\\
&\quad\ \mathit{lista\_comandos}\ \ \mathit{T\_FIMENQTO}\\
\mathit{selecao} &\rightarrow \mathit{T\_SE}\ \ \mathit{expressao}\ \ \mathit{T\_ENTAO}\ \ \mathit{lista\_comandos}\\
&\quad\ \mathit{T\_SENAO}\ \ \mathit{lista\_comandos}\ \ \mathit{T\_FIMSE}\\
\mathit{atribuicao} &\rightarrow \mathit{T\_IDENTIF}\ \ \mathit{T\_ATRIB}\ \ \mathit{expressao}\\
\mathit{expressao} &\rightarrow \mathit{expressao}\ \ \mathit{T\_VEZES}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_DIV}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_MAIS}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_MENOS}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_MAIOR}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_MENOR}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_IGUAL}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_E}\ \ \mathit{expressao}\\
&\mid\ \mathit{expressao}\ \ \mathit{T\_OU}\ \ \mathit{expressao}\\
&\mid\ \mathit{termo}\\
\mathit{termo} &\rightarrow \mathit{T\_IDENTIF}\\
&\mid\ \mathit{T\_NUMERO}\\
&\mid\ \mathit{T\_V}\\
&\mid\ \mathit{T\_F}\\
&\mid\ \mathit{T\_NAO}\ \ \mathit{termo}\\
&\mid\ \mathit{T\_ABRE}\ \ \mathit{expressao}\ \ \mathit{T\_FECHA}
\end{aligned}
$$

Os símbolos não terminais (variáveis da gramática) estão grafados com letras minúsculas e são: *programa*, *cabecalho*, *variaveis*, *declaracao_variaveis*, *tipo*, *lista_variaveis*, *lista_comandos*, *comando*, *entrada_saida*, *leitura*, *escrita*, *repeticao*, *condicao*, *atribuicao*, *expressao* e *termo*. O símbolo de partida é o não-terminal *programa*.

Os símbolos terminais (alfabeto da gramática) estão representados pelas constantes: T_PROGRAMA, T_INICIO, T_FIM, T_IDENTIF, T_INTEIRO, T_LOGICO, T_LEIA, T_ESCREVA, T_ENQTO, T_FACA, T_FIMENQTO, T_SE, T_ENTAO, T_SENAO, T_FIMSE, T_ATRIB, T_VEZES, T_DIV, T_MAIS, T_MENOS, T_MAIOR, T_MENOR, T_IGUAL, T_E, T_OU, T_V, T_F, T_NAO, T_ABRE e T_FECHA. Essas constantes representam codificações numéricas dos símbolos que são reconhecidos e retornados pelo analisador léxico.

Para bison, as constantes que definem as codificações numéricas dos tokens são definidas através da seguinte declaração:

```
1  %token T_PROGRAMA
2  %token T_INICIO
3  %token T_FIM
4  %token T_IDENTIF
5  %token T_LEIA
6  %token T_ESCREVA
7  %token T_ENQTO
8  %token T_FACA
9  %token T_FIMENQTO
10 %token T_SE
11 %token T_ENTAO
12 %token T_SENAO
13 %token T_FIMSE
14 %token T_ATRIB
15 %token T_VEZES
16 %token T_DIV
17 %token T_MAIS
18 %token T_MENOS
19 %token T_MAIOR
20 %token T_MENOR
21 %token T_IGUAL
22 %token T_E
23 %token T_OU
24 %token T_V
25 %token T_F
26 %token T_NUMERO
27 %token T_NAO
28 %token T_ABRE
29 %token T_FECHA
30 %token T_LOGICO
31 %token T_INTEIRO
```

Os códigos utilizados aqui são os mesmos retornados pelo analisador léxico. Basta incluir o arquivo de cabeçalho que é gerado com a declaração dessas contantes pela ferramenta bison.

Como pode ser observado, a gramatica de expressões usada na linguagem Simples é ambígua. Essa ambiguidade é resolvida definindo-se prioridade entre as operações para a ferramenta Yacc. Fazemos isso com a seguinte declaração:

```
1 %left T_E T_OU
2 %left T_IGUAL
3 %left T_MAIOR T_MENOR
```

```
4 %left T_MAIS T_MENOS
5 %left T_VEZES T_DIV
```

Com essa declaração, definimos que a multiplicação e a divisão tem associatividade mais à esquerda e precedência sobre a soma e a subtração. Soma e subtração tem precedência sobre Operadores relacionais Maior, Menor e Igual. Os operadores relacionais tem precedência sobre os operadores lógicos E e OU.

As regras de produção da gramática SIMPLES usando a notação de bison fica:

```
1  programa
2        : cabecalho variaveis T_INICIO lista_comandos T_FIM
3        ;
4
5  cabecalho
6        : T_PROGRAMA T_IDENTIF
7        ;
8
9  variaveis
10       :  /* vazio */
11       | declaracao_variaveis
12       ;
13
14 declaracao_variaveis
15       : tipo lista_variaveis declaracao_variaveis
16       | tipo lista_variaveis
17       ;
18
19 tipo
20       : T_LOGICO
21       | T_INTEIRO
22       ;
23
24 lista_variaveis
25       : T_IDENTIF lista_variaveis
26       | T_IDENTIF
27       ;
28
29 lista_comandos
30       : /* vazio */
31       | comando lista_comandos
32       ;
33
34 comando
35       : entrada_saida
36       | repeticao
37       | selecao
38       | atribuicao
39       ;
40
41 entrada_saida
42       : leitura
43       | escrita
44       ;
45
46 leitura
47       : T_LEIA T_IDENTIF
48       ;
49
```

```
50 escrita
51         : T_ESCREVA expressao
52         ;
53
54 repeticao
55         : T_ENQTO expressao T_FACA lista_comandos T_FIMENQTO
56         ;
57
58 selecao
59         : T_SE expressao T_ENTAO lista_comandos T_SENAO lista_comandos
             T_FIMSE
60         ;
61
62 atribuicao
63         : T_IDENTIF T_ATRIB expressao
64         ;
65
66 expressao
67         : expressao T_VEZES expressao
68         | expressao T_DIV expressao
69         | expressao T_MAIS expressao
70         | expressao T_MENOS expressao
71         | expressao T_MAIOR expressao
72         | expressao T_MENOR expressao
73         | expressao T_IGUAL expressao
74         | expressao T_E expressao
75         | expressao T_OU expressao
76         | termo
77         ;
78
79 termo
80         : T_IDENTIF
81         | T_NUMERO
82         | T_V
83         | T_F
84         | T_NAO termo
85         | T_ABRE expressao T_FECHA
86         ;
```

Capítulo 4

MVS - Máquina Virtual Simples

O objetivo principal do compilador para uma linguagem de programação é construir uma ferramenta de tradução que transforme os códigos numa linguagem de alto nível para instruções numa linguagem de máquina para, assim, possibilitar a sua execução.

Entretanto, traduzir para a linguagem de máquina de um computador real é, em geral, uma tarefa muito trabalhosa. Envolve o conhecimento profundo da máquina alvo. Por exemplo, umma máquina INTEL contém centenas de instruções, que variam em cada versão do processador e que contém uma dezena de detalhes que devem ser observados para a sua apropriada utilização.

Ao invés de traduzir, então, os programas-fonte para a linguagem de máquina de um computador real, definiremos uma máquina hipotética, mais conveniente para o desenvolvimento de um compilador didático. Desta forma, poderemos dar mais atenção para os detalhes do compilador sem termos que nos preocupar com detalhes de uma máquina real. Como efeito colateral, estaremos desenvolvendo um compilador de código 100% portável (como acontece com o código-objeto JAVA). A partir do momento que o programa estiver traduzido para código desta máquina hipotética (máquina virtual), basta ter o programa de interpreta estes códigos para executar o programa em qualquer plataforma (arquitetura+sistema operacional).

4.1 Características gerais da MVS

A Máquina Virtual Simples (MVS) é uma máquina baseada na Máquina de Execução Pascal, proposta por Tomás Kowaltowski [Kowaltowski 1983]. Assim como MEPA e a máquina virtual Java (JVM), MVS é uma máquina baseada em pilha. Essa característica é muito apropriada para tradução de estruturas encontradas em linguagens de programação C-Like e Pascal-Like, como recursividade, estruturas aninhadas, etc.

A memória da MVS é composta de duas regiões:

1. A região de programa P que conterá as instruções do programa em MVS que a máquina está executando.

2. A região de pilha de dados M que conterá os valores manipulados pelas instruções MVS.

As regiões de memória P e M funcionam como vetores com índices numerados de zero até uma tamanho máximo. Além disso, MVS tem três registradores que são usados para indexar posições no programa P, nos dados M e endereçar as variáveis locais. São eles:

1. O registrador de programa i contém o endereço da próxima instrução a ser executada, que será, portanto, $P[i]$. Esse registrador é incrementado automaticamente

após a execução de cada instrução. Exceto para as instruções de desvio, que alteram de forma absoluta o valor de i.

2. O registrador s indicará o elemento no topo da pilha cujo valor será dado, portanto, por $M[s]$. Grande parte das instruções da MVS são relativas as posições do topo da pilha M, conforme será apresentado na próxima seção.

3. O registrador de base d que contém o endereço de base no qual a variável está inserida. Esse único registrador é suficiente para generalizar a forma de endereçamento das variáveis locais e globais no programa. O endereçamento de variável global pode ser mais simples, no entanto, por questão de simplificação da máquina, usaremos endereçamento indireto para variáveis locais e globais.

Uma vez que o programa MVS está carregado na região P, e os registradores têm seus valores iniciais, o funcionamento da máquina é muito simples. As instruções indicadas pelo registrador i são executadas até que seja encontrada uma instrução de parada ou um erro. A execução de cada instrução incrementa i de uma unidade para posicionar na próxima instrução do programa, $i \leftarrow i + 1$, exceto as instruções que envolvem desvio.

4.1.1 Descrição das instruções MVS

Apresentamos o efeito de cada instrução MVS numa pseudo-linguagem de programação, indicando as modificações no estado dos registradores e da memória na MVS. Omitimos nesta descrição a operação $i \leftarrow i + 1$ que está implícita em todas as instruções, exceto quando há desvio. Adotaremos, também, a convenção de representar os valores booleanos (lógicos) por inteiros: true por 1 e false por 0.

As instruções de MVS estão sintetizadas na Tabela 4.1.

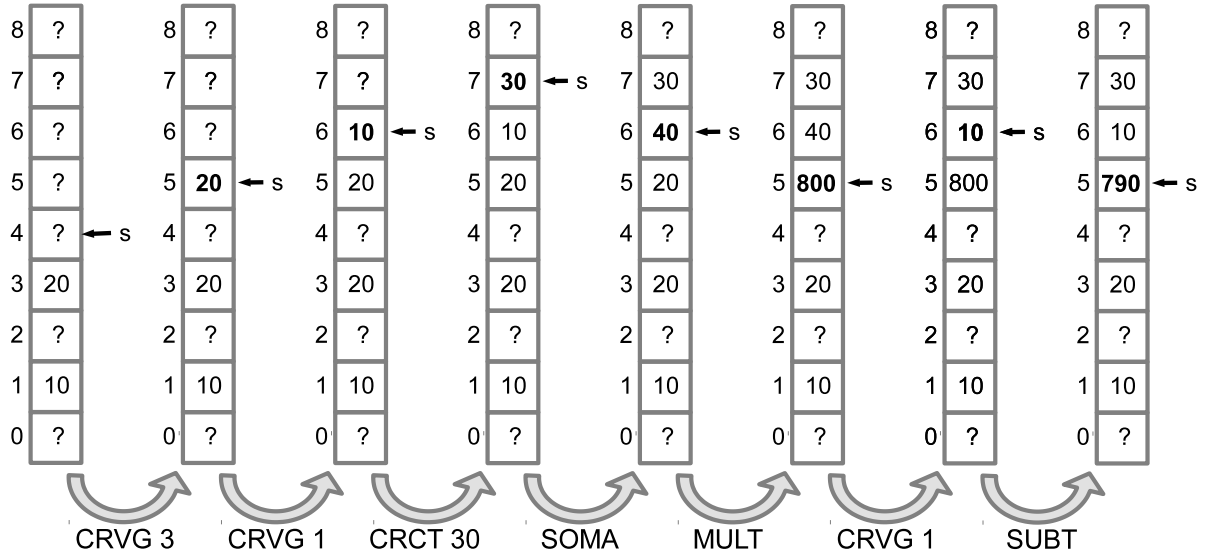

Figura 4.1: Configurações da pilha de dados M na avaliação de uma expressão

Tradução de Expressões

Como a máquina MVS é baseada em pilha, as expressões em notação infixa (com o operação entre os operandos) da linguagem fonte devem ser traduzidas para uma sequência

Tabela 4.1: Resumo das instruções da Máquina Virtual Simples

Instrução	**Micro-código**
CRCT k	$s \leftarrow s + 1$ $M[s] \leftarrow k$
CRVG n	$s \leftarrow s + 1$ $M[s] \leftarrow M[n]$
ARZG n	$M[n] \leftarrow M[s]$ $s \leftarrow s - 1$
CMMA	SE $M[s-1] > M[s]$ ENTAO $M[s-1] \leftarrow 1$ SENAO $M[s-1] \leftarrow 0$; $s \leftarrow s - 1$
CMME	SE $M[s-1] < M[s]$ ENTAO $M[s-1] \leftarrow 1$ SENAO $M[s-1] \leftarrow 0$ $s \leftarrow s - 1$
CMIG	SE $M[s-1] = M[s]$ ENTAO $M[s-1] \leftarrow 1$ SENAO $M[s-1] \leftarrow 0$ $s \leftarrow s - 1$
DISJ	SE $M[s-1]$ ou $M[s]$ ENTAO $M[s-1] \leftarrow 1$ SENAO $M[s-1] \leftarrow 0$ $s \leftarrow s - 1$
CONJ	SE $M[s-1]$ e $M[s]$ ENTAO $M[s-1] \leftarrow 1$ SENAO $M[s-1] \leftarrow 0$ $s \leftarrow s - 1$
NEGA	$M[s] \leftarrow 1 - M[s]$
SOMA	$M[s-1] \leftarrow M[s-1] + M[s]$ $s \leftarrow s - 1$
SUBT	$M[s-1] \leftarrow M[s-1] - M[s]$ $s \leftarrow s - 1$
MULT	$M[s-1] \leftarrow M[s-1] * M[s]$ $s \leftarrow s - 1$
DIVI	$M[s-1] \leftarrow M[s-1]/M[s]$ $s \leftarrow s - 1$
DSVS p	$i \leftarrow p$
DSVF p	SE $M[s] = 0$ ENTAO $i \leftarrow p$ SENAO $i \leftarrow i + 1$ $s \leftarrow s - 1$
LEIA	$s \leftarrow s + 1$ "$M[s] \leftarrow$ Entrada"
ESCR	"Escreve $M[s]$ " $s \leftarrow s - 1$
NADA	"Não faz nada"
INPP	$s \leftarrow -1$ $i \leftarrow 1$ $D \leftarrow 0$;
FIMP	"Finaliza a execução"
AMEM n	$s \leftarrow s + n$

de instruções em NPR (Notação Polonesa Reversa, na qual a operação é colocada após os seus operandos). A avaliação de expressões em NPR pela máquina baseada em pilha é trivial.

Exemplo 1 *Considere a expressão B * (A + 30) - A e suponha que os endereços atribuídos pelo compilador as variáveis A e B são 1 e 3, respectivamente. Considere ainda que o valor da variável A é 10 e que o valor da variável B é 20.*

A tradução para MVS da expressão desse exemplo é:

```
1    CRVG   3
2    CRVG   1
3    CRCT   10
4    SOMA
5    MULT
6    CRVG   1
7    SUBT
```

Na execução dessa sequência de instruções pela MVS, obtemos as configurações da pilha de dados M, conforme ilustrado na Figura 4.1. Para essa simulação consideramos o valor inicial do registrador s igual a 4.

Tradução do comando de atribuição

Para tradução da atribuição na forma V <− E, onde V representa uma variável qualquer e E representa uma expressão, usa-se a seguinte instrução de armazenamento:

Instrução	**Micro-código**
ARZG n	$M[n] \leftarrow M[s]$ $s \leftarrow s - 1$

O compilador deve traduzir a expressão, conforme já foi discutido anteriormente, e após essa tradução, o resultado da expressão deve ser armazenado na variável no lado esquerdo da atribuição, usando a instrução **ARZG**.

Exemplo 2 *Considere o comando de atribuição A <− A + B, onde os endereços das variáveis A e B são 10 e 12 respectivamente.*

A tradução MVS para essa atribuição é:

```
1    CRVG   10
2    CRVG   12
3    SOMA
4    ARZG   12
```

Tradução do comando de repetição e seleção

Para traduzir os comandos de repetição e seleção são usados as instruções de desvio condicional DSVF (Desvia se Falso), o desvio incondicional DSVS (Desvia Sempre) e a instrução **NADA** para receber o rótulo para onde será realizado o desvio da execução do programa MVS. O micro-código dessas instruções está sintetizado na tabela seguinte:

Instrução	**Micro-código**
DSVS p	$i \leftarrow p$
DSVF p	SE $M[s] = 0$ ENTAO $i \leftarrow p$ SENAO $i \leftarrow i + 1$ $s \leftarrow s - 1$
NADA	"Não faz nada"

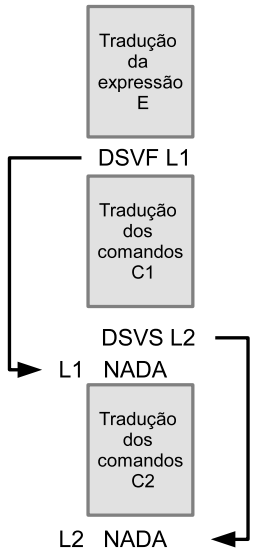

Figura 4.2: Esquema de tradução do comando de seleção para MVS

O operando p nas instruções DSVF e DSVS determinam um rótulo dentro do próprio código MVS para onde a execução deverá ou não ser desviada.

A instrução NADA é usada somente para receber o rótulo. Numa fase de otimização a instrução NADA pode ser trocada pela próxima instrução no programa MVS.

O comando de seleção da forma se E entao C1 senao C2, onde E representa uma expressão e C1 e C2 representam listas de comandos, pode ser traduzido usando o esquema de Figura 4.2.

Observe que, na execução do código MVS, se o resultado que restar da avaliação da expressão for falso, o desvio condicional DSVF L1 é realizado e somente os comandos C2 são executados. De outra forma, após a execução da lista de comandos C1, o desvio incondicional DSVS L2 é realizado para evitar a execução da lista de comandos C2.

Exemplo 3 *Considere o seguinte trecho de programa com um comando de seleção em linguagem Simples:*

```
1    A <- V
2    se A
3        entao A <- F
4        senao A <- V
5    fimse
```

A tradução MVS para essa seleção é (o endereço da variável A é zero):

```
1          CRCT    1
2          ARZG    0
3          CRVG    0
4          DSVF    L1
5          CRCT    0
6          ARZG    0
7          DSVS    L2
8  L1      NADA
9          CRCT    1
10         ARZG    0
11 L2      NADA
```

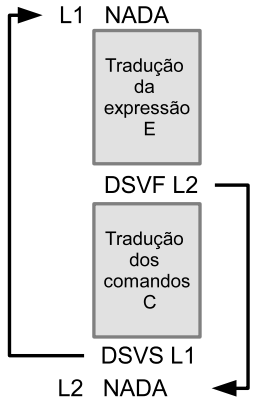

Figura 4.3: Esquema de tradução do comando de repetição para MVS

O comando de repetição da forma enquanto E faca C fimenquanto, onde E representa uma expressão e C representa uma lista de comandos, pode ser traduzido usando o esquema de Figura 4.3.

Observe que a palavra reservada enquanto marca um local programa (rótulo) para onde a execução deve ser desviada de forma incondicional toda vez que a lista de comandos for executada. Por outro lado, se o resultado da avaliação da expressão que condiciona a repetição for falso, o programa deve desviar a execução para fora da repetição.

Exemplo 4 *Considere o seguinte trecho de programa com um comando de repetição em linguagem Simples:*

```
1    x <- 1
2    enquanto x < 10 faca
3        x <- 2 * x
4    fimenquanto
```

A tradução MVS para essa repetição é (o endereço da variável x é zero):

```
1          CRCT    1
2          ARZG    0
3   L1     NADA
4          CRVG    0
5          CRCT    10
6          CMME
7          DSVF    L2
8          CRCT    2
9          CRVG    0
10         MULT
11         ARZG    0
12         DSVS    L1
13  L2     NADA
```

Tradução dos comandos de entrada e saída

Para tradução dos comandos leia V e escreva E, onde V é uma variável e E uma expressão, usaremos as instruções MVS:

Instrução	Micro-código
LEIA	$s \leftarrow s + 1$ "$M[s] \leftarrow$ Entrada"
ESCR	"Escreve $M[s]$ " $s \leftarrow s - 1$

Para tradução do leia X, primeiro gera-se uma instrução LEIA e o resultado lido deve ser armazenado na variável X.

Para tradução de escreva A + B, primeiro traduz-se a expressão A + B e, ao final, gera-se uma instrução ESCR, para que o resultado a expressão seja apresentado na saída padrão.

Exemplo 5 *Considere o seguinte trecho de programa com comandos de leitura e escrita em linguagem Simples:*

```
1        leia A
2        leia B
3        escreva A + B
```

A tradução MVS é:

```
1        LEIA
2        ARZG    0
3        LEIA
4        ARZG    1
5        CRVG    0
6        CRVG    1
7        SOMA
8        ESCR
```

Considere, nessa tradução, que o endereço atribuído pelo compilador à variável A é zero e à variável B é um.

Tradução dos comandos de entrada e saída

Para tradução de um programa em linguagem Simples completo usaremos as seguintes instruções MVS:

Instrução	Micro-código
INPP	$s \leftarrow -1$ $i \leftarrow 1$ $D \leftarrow 0$;
FIMP	"Finaliza a execução"
AMEM n	$s \leftarrow s + n$

Onde:

- INPP - instrução MVS que serve para colocar a máquina de execução numa configuração inicial;
- FIMP - instrução que finaliza a execução de um programa.
- AMEM - instrução que aloca memória para as variáveis globais do programa.

Exemplo 6 *Considere o seguinte programa Simples:*

```
1  programa repete
2          inteiro i j
3  inicio
4       i <- 1
5       enquanto i < 10 faca
6               j <- 1
7               enquanto j < 10 faca
8                       escreva i + j
9                       j <- j + 1
10              fimenquanto
11              i <- i + 1
12      fimenquanto
13 fimprograma
```

A tradução MVS é:

```
1            INPP
2            AMEM    2
3            CRCT    1
4            ARZG    0
5  L1        NADA
6            CRVG    0
7            CRCT    10
8            CMME
9            DSVF    L2
10           CRCT    1
11           ARZG    1
12 L3        NADA
13           CRVG    1
14           CRCT    10
15           CMME
16           DSVF    L4
17           CRVG    0
18           CRVG    1
19           SOMA
20           ESCR
21           CRVG    1
22           CRCT    1
23           SOMA
24           ARZG    1
25           DSVS    L3
26 L4        NADA
27           CRVG    0
28           CRCT    1
29           SOMA
30           ARZG    0
31           DSVS    L1
32 L2        NADA
33           FIMP
```

Capítulo 5

Análise Semântica e Geração de Código

Nesse capítulo são apresentadas as modificações do analisador léxico e analisador sintático para possibilitar algumas verificações semânticas e a geração do código Simples para Máquina MVS.

5.1 Funções Utilitárias

```
1  /*--------------------------------------------------------------
2   *  Estruturas e Rotinas Utilitarias do Compilador
3   *
4   *  Por Luiz Eduardo da Silva
5   *-------------------------------------------------------------*/
6
7  /*--------------------------------------------------------------
8   *  Limites das estruturas
9   *-------------------------------------------------------------*/
10 #define TAM_TSIMB 100  /* Tamanho da tabela de simbolos */
11 #define TAM_PSEMA 100  /* Tamanho da pilha semantica    */
12
13 /*--------------------------------------------------------------
14  *  Variaveis globais
15  *-------------------------------------------------------------*/
16 int TOPO_TSIMB     = 0;  /* TOPO da tabela de simbolos */
17 int TOPO_PSEMA     = 0;  /* TOPO da pilha semantica */
18 int ROTULO         = 0;  /* Proximo numero de rotulo */
19 int CONTA_VARS     = 0;  /* Numero de variaveis */
20 int POS_SIMB;            /* Pos. na tabela de simbolos */
21 int aux;                 /* variavel auxiliar */
22 int numLinha = 1; /* numero da linha no programa */
23 char atomo[30];   /* nome de um identif. ou numero */
24
25 /*--------------------------------------------------------------
26  *  Rotina geral de tratamento de erro
27  *-------------------------------------------------------------*/
28 void ERRO (char *msg, ...) {
29   char formato [255];
30   va_list arg;
31   va_start (arg, msg);
32   sprintf (formato, "\n%d: ", numLinha);
33   strcat (formato, msg);
34   strcat (formato, "\n\n");
35   printf ("\nERRO no programa");
36   vprintf (formato, arg);
```

```
37    va_end (arg);
38    exit (1);
39  }
40
41  /*--------------------------------------------------------------------
42   *  Tabela de Simbolos
43   *------------------------------------------------------------------*/
44  struct elem_tab_simbolos {
45    char id[30];
46    int desloca;
47  } TSIMB [TAM_TSIMB], elem_tab;
48
49  /*--------------------------------------------------------------------
50   *  Pilha Semantica
51   *------------------------------------------------------------------*/
52  int PSEMA[TAM_PSEMA];
53
54  /*--------------------------------------------------------------------
55   * Funcao que BUSCA um simbolo na tabela de simbolos.
56   *       Retorna -1 se o simbolo nao esta' na tabela
57   *       Retorna i, onde i e' o indice do simbolo na tabela
58   *                  se o simbolo esta' na tabela
59   *------------------------------------------------------------------*/
60  int busca_simbolo (char *ident)
61  {
62    int i = TOPO_TSIMB-1;
63    for (;strcmp (TSIMB[i].id, ident) && i >= 0; i--);
64    return i;
65  }
66
67  /*--------------------------------------------------------------------
68   * Funcao que INSERE um simbolo na tabela de simbolos.
69   *     Se ja' existe um simbolo com mesmo nome e mesmo nivel
70   *     uma mensagem de erro e' apresentada e o  programa  e'
71   *     interrompido.
72   *------------------------------------------------------------------*/
73  void insere_simbolo (struct elem_tab_simbolos *elem)
74  {
75    if (TOPO_TSIMB == TAM_TSIMB) {
76       ERRO ("OVERFLOW - tabela de simbolos");
77    }
78    else {
79       POS_SIMB = busca_simbolo (elem->id);
80       if (POS_SIMB != -1) {
81         ERRO ("Identificador [%s] duplicado", elem->id);
82       }
83       TSIMB [TOPO_TSIMB] = *elem;
84       TOPO_TSIMB++;
85    }
86  }
87
88
89  /*--------------------------------------------------------------------
90   * Funcao de insercao de uma variavel na tabela de simbolos
91   *------------------------------------------------------------------*/
92  void insere_variavel (char *ident) {
93     strcpy (elem_tab.id, ident);
94     elem_tab.desloca = CONTA_VARS;
```

```
95      insere_simbolo (&elem_tab);
96  }
97
98  /*-------------------------------------------------------------
99   * Rotinas para manutencao da PILHA SEMANTICA
100  *-------------------------------------------------------------*/
101 void empilha (int n) {
102   if (TOPO_PSEMA == TAM_PSEMA) {
103     ERRO ("OVERFLOW - Pilha Semantica");
104   }
105   PSEMA[TOPO_PSEMA++] = n;
106 }
107
108 int desempilha () {
109   if (TOPO_PSEMA == 0) {
110     ERRO ("UNDERFLOW - Pilha Semantica");
111   }
112   return PSEMA[--TOPO_PSEMA];
113 }
```

5.2 Modificação do analisador léxico

```
1   %{
2
3   #include "sintatico.h"
4
5   %}
6
7   identificador   [a-zA-Z]([a-zA-Z0-9])*
8   numero          [0-9]+
9   espaco          [ \t]+
10  novalinha       [\n]
11
12  %x comentario
13
14  %%
15
16  programa        return T_PROGRAMA;
17  inicio          return T_INICIO;
18  fimprograma     return T_FIM;
19
20  leia            return T_LEIA;
21  escreva         return T_ESCREVA;
22
23  se              return T_SE;
24  entao           return T_ENTAO;
25  senao           return T_SENAO;
26  fimse           return T_FIMSE;
27
28  enquanto        return T_ENQTO;
29  faca            return T_FACA;
30  fimenquanto     return T_FIMENQTO;
31
32  "+"             return T_MAIS;
33  "-"             return T_MENOS;
34  "*"             return T_VEZES;
```

```
35 div                 return T_DIV;
36
37 ">"                 return T_MAIOR;
38 "<"                 return T_MENOR;
39 "="                 return T_IGUAL;
40
41 e                   return T_E;
42 ou                  return T_OU;
43 nao                 return T_NAO;
44
45 "<-"                return T_ATRIB;
46 "("                 return T_ABRE;
47 ")"                 return T_FECHA;
48
49 inteiro             return T_INTEIRO;
50 logico              return T_LOGICO;
51 V                   return T_V;
52 F                   return T_F;
53
54 {identificador} { strcpy (atomo, yytext);
55                    return T_IDENTIF; }
56 {numero}          { strcpy (atomo, yytext);
57                    return T_NUMERO; }
58 {espaco}          /* nao faz nada */
59 {novalinha}       numLinha++;
60
61 "//".*            /* comentario de linha */
62
63 "/*"               BEGIN(comentario);
64 <comentario>"*/"  BEGIN(INITIAL);
65 <comentario>.      /* nao faz nada */
66 <comentario>\n     numLinha++;
67
68 .                  ERRO ("ERRO_LEXICO");
69
70 %%
```

5.3 Modificação do analisador sintático

```
1  /*-------------------------------------------------------------
2   *       A N A L I S A D O R   S I N T A T I C O
3   *
4   *      Por Luiz Eduardo da Silva
5   *-------------------------------------------------------------*/
6
7  %{
8  #include <stdio.h>
9  #include <string.h>
10 #include <stdlib.h>
11 #include <stdarg.h>
12
13 #include "utils.c"
14 #include "lexico.c"
15
16 int yylex();
17 void yyerror(char *);
```

```
18
19 %}
20
21 // Simbolo de partida
22 %start programa
23
24 // Simbolos terminais
25 %token T_PROGRAMA
26 %token T_INICIO
27 %token T_FIM
28 %token T_IDENTIF
29 %token T_LEIA
30 %token T_ESCREVA
31 %token T_ENQTO
32 %token T_FACA
33 %token T_FIMENQTO
34 %token T_SE
35 %token T_ENTAO
36 %token T_SENAO
37 %token T_FIMSE
38 %token T_ATRIB
39 %token T_VEZES
40 %token T_DIV
41 %token T_MAIS
42 %token T_MENOS
43 %token T_MAIOR
44 %token T_MENOR
45 %token T_IGUAL
46 %token T_E
47 %token T_OU
48 %token T_V
49 %token T_F
50 %token T_NUMERO
51 %token T_NAO
52 %token T_ABRE
53 %token T_FECHA
54 %token T_LOGICO
55 %token T_INTEIRO
56
57 // Precedencia e associatividade
58 %left T_E T_OU
59 %left T_IGUAL
60 %left T_MAIOR T_MENOR
61 %left T_MAIS T_MENOS
62 %left T_VEZES T_DIV
63
64 %%
65
66 // Regras de producao
67 programa
68       : cabecalho
69           { printf ("\tINPP\n"); }
70         variaveis
71         T_INICIO lista_comandos
72         T_FIM
73           { printf ("\tFIMP\n"); }
74       ;
75
```

```
76  cabecalho
77        : T_PROGRAMA T_IDENTIF
78        ;
79
80  variaveis
81        :  /* vazio */
82        |  declaracao_variaveis
83        ;
84
85  declaracao_variaveis
86        : declaracao_variaveis
87          tipo
88            { CONTA_VARS = 0; }
89          lista_variaveis
90            { printf ("\tAMEM\t%d\n", CONTA_VARS); }
91        | tipo
92            { CONTA_VARS = 0; }
93          lista_variaveis
94            { printf ("\tAMEM\t%d\n", CONTA_VARS); }
95        ;
96
97  tipo
98        : T_LOGICO
99        | T_INTEIRO
100       ;
101
102 lista_variaveis
103       : lista_variaveis
104         T_IDENTIF
105           { insere_variavel (atomo); CONTA_VARS++; }
106
107       | T_IDENTIF
108           { insere_variavel (atomo); CONTA_VARS++; }
109       ;
110
111 lista_comandos
112       : /* vazio */
113       | comando lista_comandos
114       ;
115
116 comando
117       : entrada_saida
118       | repeticao
119       | selecao
120       | atribuicao
121       ;
122
123 entrada_saida
124       : leitura
125       | escrita
126       ;
127
128 leitura
129       : T_LEIA
130         T_IDENTIF
131           {
132             POS_SIMB = busca_simbolo (atomo);
133             if (POS_SIMB == -1)
```

```
134                     ERRO ("Variavel [%s] nao declarada!",
135                                 atomo);
136                 else {
137                     printf ("\tLEIA\n");
138                     printf ("\tARZG\t%d\n",
139                             TSIMB[POS_SIMB].desloca);
140                 }
141               }
142         ;
143
144 escrita
145         : T_ESCREVA expressao
146           { printf ("\tESCR\n"); }
147         ;
148
149 repeticao
150         : T_ENQTO
151             {
152               printf ("L%d\tNADA\n", ++ROTULO);
153               empilha (ROTULO);
154             }
155           expressao
156           T_FACA
157             {
158               printf ("\tDSVF\tL%d\n",++ROTULO);
159               empilha (ROTULO);
160             }
161           lista_comandos
162           T_FIMENQTO
163             {
164               aux = desempilha();
165               printf ("\tDSVS\tL%d\n", desempilha());
166               printf ("L%d\tNADA\n", aux);
167             }
168         ;
169
170 selecao
171         : T_SE
172           expressao
173             {
174               printf ("\tDSVF\tL%d\n", ++ROTULO);
175               empilha (ROTULO);
176             }
177           T_ENTAO
178           lista_comandos
179           T_SENAO
180             {
181               printf ("\tDSVS\tL%d\n", ++ROTULO);
182               printf ("L%d\tNADA\n", desempilha());
183               empilha (ROTULO);
184             }
185           lista_comandos
186           T_FIMSE
187             {
188               printf ("L%d\tNADA\n", desempilha());
189             }
190         ;
191
```

```
192 atribuicao
193       : T_IDENTIF
194           {
195             POS_SIMB = busca_simbolo (atomo);
196             if (POS_SIMB == -1)
197                 ERRO ("Variavel [%s] nao declarada!",
198                       atomo);
199             else
200                 empilha (TSIMB[POS_SIMB].desloca);
201           }
202         T_ATRIB
203         expressao
204           { printf ("\tARZG\t%d\n", desempilha()); }
205       ;
206
207 expressao
208       : expressao T_VEZES expressao
209           { printf ("\tMULT\n"); }
210       | expressao T_DIV expressao
211           { printf ("\tDIVI\n"); }
212       | expressao T_MAIS expressao
213           { printf ("\tSOMA\n"); }
214       | expressao T_MENOS expressao
215           { printf ("\tSUBT\n"); }
216       | expressao T_MAIOR expressao
217           { printf ("\tCMMA\n"); }
218       | expressao T_MENOR expressao
219           { printf ("\tCMME\n"); }
220       | expressao T_IGUAL expressao
221           { printf ("\tCMIG\n"); }
222       | expressao T_E expressao
223           { printf ("\tCONJ\n"); }
224       | expressao T_OU expressao
225           { printf ("\tDISJ\n"); }
226       | termo
227       ;
228
229 termo
230       : T_IDENTIF
231           {
232             POS_SIMB = busca_simbolo (atomo);
233             if (POS_SIMB == -1)
234                ERRO ("Variavel [%s] nao declarada!",
235                          atomo);
236             else {
237                printf ("\tCRVG\t%d\n",
238                         TSIMB[POS_SIMB].desloca);
239             }
240           }
241       | T_NUMERO
242           { printf ("\tCRCT\t%s\n", atomo); }
243       | T_V
244           { printf ("\tCRCT\t1\n"); }
245       | T_F
246           { printf ("\tCRCT\t0\n"); }
247       | T_NAO termo
248           { printf ("\tNEGA\n"); }
249       | T_ABRE expressao T_FECHA
```

```
250         ;
251
252 %%
253 /*+-------------------------------------------------------------+
254   |            Corpo principal do programa COMPILADOR           |
255   +-------------------------------------------------------------+*/
256
257 int yywrap () {
258    return 1;
259 }
260
261 void yyerror (char *s)
262 {
263   ERRO ("ERRO SINTATICO");
264 }
265
266 int main ()
267 {
268   return yyparse ();
269 }
```

5.4 Simulador da Máquina MVS

```
1  /*-------------------------------------------------------------
2   *       Simulador da Maquina Virtual Simples (MVS)
3   *       Por Luiz Eduardo da Silva
4   *-------------------------------------------------------------*/
5  #include <stdio.h>
6  #include <string.h>
7  #include <stdlib.h>
8
9  /*-------------------------------------------------------------
10  * Conjunto de instrucoes mnemonicas da MVS
11  *-------------------------------------------------------------*/
12 #define TOTALINST 21
13 char *inst[TOTALINST] = {
14   "CRVG", /* Carrega valor */
15   "CRCT", /* Carrega constante */
16   "SOMA", /* Soma */
17   "SUBT", /* Subtrai */
18   "MULT", /* Multiplica */
19   "DIVI", /* Divide (divisao inteira) */
20   "CMIG", /* Compara se igual */
21   "CMMA", /* Compara se maior */
22   "CMME", /* Compara se menor */
23   "CONJ", /* Conjuncao - e logico */
24   "DISJ", /* Disjuncao - ou logico */
25   "NEGA", /* Negacao - nao logico */
26   "ARZG", /* Armazena na variavel */
27   "DSVS", /* Desvia sempre */
28   "DSVF", /* Desvia se falso */
29   "NADA", /* Nada */
30   "ESCR", /* Escreve */
31   "LEIA", /* Le */
32   "INPP", /* Inicia Programa Principal */
33   "AMEM", /* Aloca memoria */
```

```
34  "FIMP"   /* Fim do programa */
35 };
36
37 enum codinst {CRVG, CRCT, SOMA, SUBT, MULT, DIVI, CMIG,
38 CMMA, CMME, CONJ, DISJ, NEGA, ARZG, DSVS, DSVF, NADA, ESCR,
39 LEIA, INPP, AMEM, FIMP};
40
41 /*-------------------------------------------------------
42  * Regioes do ambiente de execucao da MVS
43  *-------------------------------------------------------*/
44 struct prog { int r, i, o; }
45   P[500]; /* Regiao do Programa */
46 int
47   L[50];  /* Posicao dos rotulos */
48 int
49   M[500]; /* Pilha M - Dados do programa */
50
51 /*-------------------------------------------------------
52  * Rotinas da MVS
53  *-------------------------------------------------------*/
54 int busca_instrucao (char *s);
55 int carrega_programa (char *s, int *nro);
56 void mostra_programa (int nro);
57 void executa_programa (void);
58
59 /*-------------------------------------------------------
60  * Funcao principal.
61  * Carrega o codigo passado em linha de comando e executa
62  *-------------------------------------------------------*/
63 int main (int argc, char *argv[]) {
64   int i, nroinst;
65   argc--;
66   argv++;
67   if (argc < 1)  {
68     puts ("\nMVS - Maquina Virtual Simples");
69     puts ("USO: mvs <nomedoarquivo.mvs>\n\n");
70     exit (0);
71   }
72   if (carrega_programa (argv[0], &nroinst))
73   {
74     mostra_programa(nroinst);
75     executa_programa();
76     printf ("Fim do programa.\n");
77   }
78   return 1;
79 }
80
81 /*-------------------------------------------------------
82  * Funcao que codifica as instrucoes.
83  * Recebe o nome da instrucao e retorna um numero.
84  *-------------------------------------------------------*/
85 int busca_instrucao (char *s)
86 {
87   int i;
88   for (i = TOTALINST-1; i >= 0 && strcmp(inst[i], s); i--);
89   return i;
90 }
91
```

```
92  /*-------------------------------------------------------
93   * Carrega o programa MVS.
94   * Recebe o nome do arquivo com o programa para MVS e
95   * retorna o numero de linhas do programa.
96   *-------------------------------------------------------*/
97  int carrega_programa (char *s, int *nro)
98  {
99    FILE *arq;
100   char linha[100], *p, str[6];
101   int j;
102
103   if ((arq = fopen (s, "r")) == NULL)
104   {
105     puts ("\n\nErro na abertura do arquivo\n\n");
106     return 0;
107   }
108   *nro = 0;
109   while (!feof (arq))
110   {
111     fgets (linha, 100, arq);
112     linha[strlen(linha)-1]='\0';
113     /*-------------------------------------
114       Tres tipos de instrucoes:
115        1. Com rotulo:
116              L1:\tNADA\n
117        2. Sem rotulo nem argumento:
118              \tINPP\n
119        3. Com argumento:
120              \tCRCT\t4\n    ou
121              \tDSVF\tL1\n
122     -------------------------------------*/
123     if (linha[0] == 'L') /*--- tipo 1 ---*/
124     {
125       p = (char *) strtok (linha, "\t");
126       p++;
127       P[*nro].r = atoi (p);
128       P[*nro].i = busca_instrucao ("NADA");
129       P[*nro].o = -1;
130       L[P[*nro].r] = *nro;
131     }
132     else /*--- tipo 2 ou 3 ---*/
133     {
134       p = (char *) strtok (linha, "\t");
135       P[*nro].r = -1;
136       P[*nro].i = busca_instrucao (p);
137       p = (char *) strtok (NULL, "\t");
138       if (!p) /*--- tipo 2 ---*/
139             P[*nro].o = -1;
140       else     /*--- tipo 3 ---*/
141       {
142             if (*p == 'L') p++;
143             P[*nro].o = atoi (p);
144       }
145       if (P[*nro].i == TOTALINST-1)
146       {
147             (*nro)++;
148             break;  /*-- INSTRUCAO FIM --*/
149       }
```

```
150       }
151       (*nro)++; // proxima instrucao
152     }
153     fclose (arq);
154     return 1;
155   }
156
157   void mostra_programa (int nro) {
158      int i;
159      for (i = 0; i < nro; i++) {
160           printf ("%d\t%s\t%d\n", P[i].r, inst[P[i].i], P[i].o);
161      }
162   }
163
164   /*-------------------------------------------------------------
165    * Funcao que executa o programa MVS
166    * A partir do vetor P (programa) preenchido, executa uma
167    * a uma as instrucoes do programa.
168    *-------------------------------------------------------------*/
169   void executa_programa (void)
170   {
171     int i = 0, s;
172     char numstr[6];
173     while (1) {
174       switch (P[i].i) {
175            case CRCT : s++; M[s] = P[i].o; i++;
176                        break;
177            case CRVG : s++; M[s] = M[P[i].o]; i++;
178                        break;
179            case ARZG : M[P[i].o] = M[s]; s--; i++;
180                        break;
181            case SOMA : M[s-1] = M[s-1] + M[s]; s--; i++;
182                        break;
183            case SUBT : M[s-1] = M[s-1] - M[s]; s--; i++;
184                        break;
185            case MULT : M[s-1] = M[s-1] * M[s]; s--; i++;
186                        break;
187            case DIVI : M[s-1] = M[s-1] / M[s]; s--; i++;
188                        break;
189            case CONJ : if (M[s-1] && M[s]) M[s-1]=1;
190                        else M[s-1] = 0;
191                        s--; i++;
192                        break;
193            case DISJ : if (M[s-1] || M[s]) M[s-1] = 1;
194                        else M[s-1] = 0;
195                        s--; i++;
196                        break;
197            case NEGA : M[s] = 1-M[s]; i++;
198                        break;
199            case CMME : if (M[s-1] < M[s]) M[s-1] = 1;
200                        else M[s-1] = 0;
201                        s--; i++;
202                        break;
203            case CMMA : if (M[s-1] > M[s]) M[s-1] = 1;
204                        else M[s-1] = 0;
205                        s--; i++;
206                        break;
207            case CMIG : if (M[s-1] == M[s]) M[s-1] = 1;
```

```
208                         else M[s-1] = 0;
209                         s--; i++;
210                         break;
211             case DSVS : i = L[P[i].o];
212                         break;
213             case DSVF : if (M[s] == 0) i = L[P[i].o];
214                         else i++; s--;
215                         break;
216             case NADA : i++;
217                         break;
218             case FIMP : printf("\n\n"); return;
219             case LEIA : s++; printf ("? ");
220                         fgets (numstr, 5, stdin);
221                         M[s] = atoi (numstr); i++;
222                         break;
223             case ESCR : printf ("\nSaida = %d", M[s]); s--; i++;
224                         break;
225             case AMEM : s = s + P[i].o; i++;
226                         break;
227             case INPP : s = -1; i++;
228                         break;
229             default : puts ("Instrucao MVS desconhecida!");
230                       exit (0);
231       }
232     }
233 }
```

Capítulo 6

Rotinas e Passagem de Parâmetro

6.1 Instruções MVS para tradução de rotinas

Instrução	**Descrição**	**Micro-código**
CRVL n	Carrega valor local	$s \leftarrow s + 1$ $M[s] \leftarrow M[d + n]$
ARZL n	Armazena Local	$M[d + n] \leftarrow M[s]$ $s \leftarrow s - 1$
CREL n	Carrega endereço local	$s \leftarrow s + 1$ $M[s] \leftarrow d + n$
CREG n	Carrega endereço global	$s \leftarrow s + 1$ $M[s] \leftarrow n$
CRVI n	Carrega valor indireto	$s \leftarrow s + 1$ $M[s] \leftarrow M[M[d + n]]$
ARMI n	Armazena Indireto	$M[M[d + n]] \leftarrow M[s]$ $s \leftarrow s - 1$
SVCP	Salva Contador do Programa	$s \leftarrow s + 1$ $M[s] \leftarrow i + 2$
ENSP	Entrada sub-programa	$s \leftarrow s + 1$ $M[s] \leftarrow d$ $d \leftarrow s + 1$
RTSP n	Retorno sub-programa	$d \leftarrow M[s]$ $i \leftarrow M[s - 1]$ $s \leftarrow s - (n + 2)$
DMEM n	Desaloca memória	$s \leftarrow s - n$

6.2 Modificação da gramática para incluir rotinas

#	**Regra**
1	programa: cabecalho variaveis **rotinas** T_INICIO lista_comandos T_FIM
2	cabecalho: T_PROGRAMA identificador
3	identificador: T_IDENTIF
4	variaveis: λ
5	| declaracao_variaveis
6	declaracao_variaveis: tipo lista_variaveis declaracao_variaveis
7	| tipo lista_variaveis
8	tipo: T_LOGICO

9	| T_INTEIRO
10	lista_variaveis: identificador lista_variaveis
11	| identificador
12	**rotinas:** λ
13	| **lista_rotinas**
14	**lista_rotinas: lista_rotinas rotina**
15	| **rotina**
16	**rotina: funcao**
17	| **procedimento**
18	**funcao: T_FUNC tipo identificador T_ABRE lista_parametros T_FECHA variaveis T_INICIO lista_comandos T_FIMFUNC**
19	**procedimento: T_PROC identificador T_ABRE lista_parametros T_FECHA variaveis T_INICIO lista_comandos T_FIMPROC**
20	**lista_parametros:** λ
21	| **parametro lista_parametros**
22	**parametro: mecanismo tipo identificador**
23	**mecanismo:** λ
24	| **T_REF**
25	lista_comandos: λ
26	| comando lista_comandos
27	comando: entrada_saida
28	| repeticao
29	| selecao
30	| atribuicao
31	| **chamada_procedimento**
32	entrada_saida: leitura
33	| escrita
34	leitura: T_LEIA identificador
35	escrita: T_ESCREVA expressao
36	repeticao: T_ENQTO expressao T_FACA lista_comandos T_FIMENQTO
37	selecao: T_SE expressao T_ENTAO lista_comandos T_SENAO lista_comandos T_FIMSE
38	atribuicao: identificador T_ATRIB expressao
39	**chamada_procedimento: identificador T_ABRE lista_argumentos T_FECHA**
40	**lista_argumentos: lista_argumentos argumento**
41	| λ
42	**argumento: expressao**
43	expressao: expressao T_VEZES expressao
44	| expressao T_DIV expressao
45	| expressao T_MAIS expressao
46	| expressao T_MENOS expressao
47	| expressao T_MAIOR expressao
48	| expressao T_MENOR expressao

49	| expressao T_IGUAL expressao
50	| expressao T_E expressao
51	| expressao T_OU expressao
52	| termo
53	**chamada:** λ
54	| **T_ABRE lista_argumentos T_FECHA**
55	termo: identificador **chamada**
56	| T_NUMERO
57	| T_V
58	| T_F
59	| T_NAO termo
60	| T_ABRE expressao T_FECHA

6.3 Modificação da Tabela de Símbolos

A Figura 6.1 ilustra as informações que devem estar disponíveis na Tabela de Símbolos, no início do bloco da rotina e no início do corpo principal para um programa exemplo.

As colunas da Tabela de Símbolos são:

- # - identifica a posição do símbolo na tabela
- **id** - é o nome do identificador, o nome escolhido pelo programador para a entidade do programa
- **Esc** - escopo da variável. No caso da linguagem simples os símbolos só podem ter escopo global (G) ou local (L).
- **Dsl** - deslocamento (ou endereço) é o operando que acompanhará as variáveis na instrução CRVG, CRVL. As funções também tem valor para esse campo.
- **Rot** - rótulo atribuído pelo compilador para a função ou procedimento. Essa informação é necessária para tradução da instrução de desvio (DSVS) na chamada do procedimento ou função.
- **Cat** - categoria do símbolo que pode ser: VAR = variável, PRO = procedimento, FUN = função, PAR = parâmetro.
- **Tip** - tipo do símbolo (INT = inteiro ou LOG = lógico).
- **Mec** - mecanismo de passagem para parâmetros que pode ser REF = referência ou VAL = valor.
- **Par** - lista encadeada dos parâmetros da rotina, com seus Tipos e Mecanismos de passagem de parâmetro. Essa informação é necessária para traduzir de forma correta os parâmetros na chamada da rotina. Observe que no programa principal as variáveis locais e parâmetros das rotinas são excluídos da tabela de símbolos

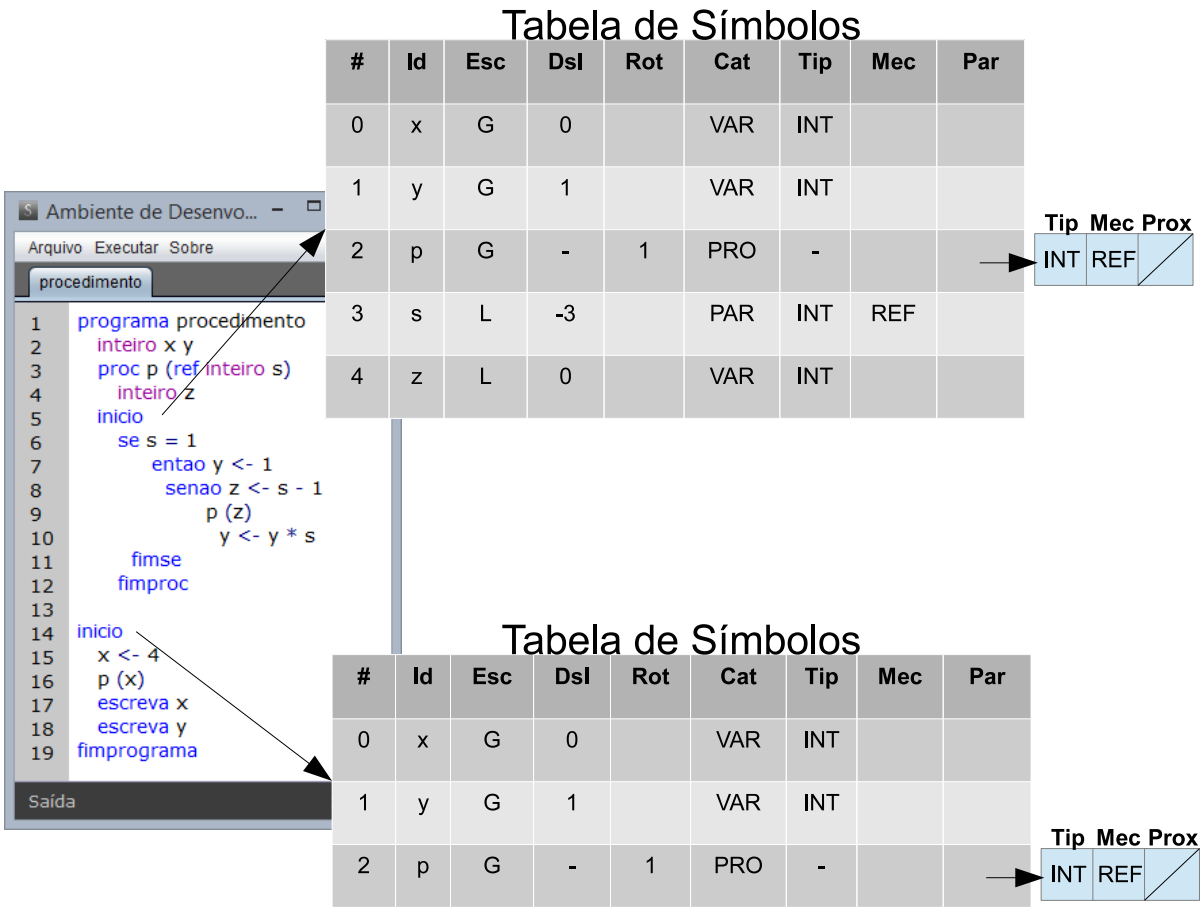

Figura 6.1: Modificação da Tabela de Símbolos para permitir a tradução de procedimento e função na linguagem Simples

6.4 Exemplos de Tradução de Procedimentos e Funções

6.4.1 Teste 1 - Simples

```
1  programa teste1
2     inteiro x
3     proc muda (ref inteiro a)
4     inicio
5           a <- 7
6     fimproc
7  inicio
8     muda (x)
9     escreva x
10 fimprograma
```

Código MVS correspondente:

```
1          INPP
2          AMEM    1
3          DSVS    L0
4  L1      ENSP
5          CRCT    7
6          ARMI    -3
7          RTSP    1
8  L0      NADA
9          CREG    0
10         SVCP
11         DSVS    L1
12         CRVG    0
13         ESCR
14         DMEM    1
15         FIMP
```

6.4.2 Teste 2 - Simples

```
1  programa teste2
2     proc somatudo (inteiro a inteiro b inteiro c inteiro d)
3     inicio
4         escreva a + b + c + d
5     fimproc
6  inicio
7     somatudo (10 3 4 8)
8  fimprograma
```

Código MVS correspondente:

```
1          INPP
2          DSVS    L0
3  L1      ENSP
4          CRVL    -6
5          CRVL    -5
6          SOMA
7          CRVL    -4
```

```
8           SOMA
9           CRVL    -3
10          SOMA
11          ESCR
12          RTSP    4
13  L0      NADA
14          CRCT    10
15          CRCT    3
16          CRCT    4
17          CRCT    8
18          SVCP
19          DSVS    L1
20          FIMP
```

6.4.3 Teste 3 - Simples

```
1  programa teste3
2     inteiro x
3     proc recursivo (inteiro n)
4     inicio
5        se n > 0
6           entao escreva n
7                 recursivo (n - 1)
8           senao
9        fimse
10    fimproc
11 inicio
12    leia x
13    recursivo (x)
14 fimprograma
```

Código MVS correspondente:

```
1           INPP
2           AMEM    1
3           DSVS    L0
4   L1      ENSP
5           CRVL    -3
6           CRCT    0
7           CMMA
8           DSVF    L2
9           CRVL    -3
10          ESCR
11          CRVL    -3
12          CRCT    1
13          SUBT
14          SVCP
15          DSVS    L1
16          DSVS    L3
17  L2      NADA
18  L3      NADA
19          RTSP    1
20  L0      NADA
21          LEIA
22          ARZG    0
```

```
23          CRVG    0
24          SVCP
25          DSVS    L1
26          DMEM    1
27          FIMP
```

6.4.4 Teste 4 - Simples

```
1  programa teste4
2     inteiro n
3     func inteiro fatorial (inteiro n)
4     inicio
5        se n = 0
6           entao fatorial <- 1
7           senao fatorial <- n * fatorial (n - 1)
8        fimse
9     fimfunc
10 inicio
11    leia n
12    escreva fatorial (n)
13 fimprograma
```

Código MVS correspondente:

```
1           INPP
2           AMEM    1
3           DSVS    L0
4  L1       ENSP
5           CRVL    -3
6           CRCT    0
7           CMIG
8           DSVF    L2
9           CRCT    1
10          ARZL    -4
11          DSVS    L3
12 L2       NADA
13          CRVL    -3
14          AMEM    1
15          CRVL    -3
16          CRCT    1
17          SUBT
18          SVCP
19          DSVS    L1
20          MULT
21          ARZL    -4
22 L3       NADA
23          RTSP    1
24 L0       NADA
25          LEIA
26          ARZG    0
27          AMEM    1
28          CRVG    0
29          SVCP
30          DSVS    L1
31          ESCR
```

```
32          DMEM    1
33          FIMP
```

6.4.5 Teste 5 - Simples

```
1  programa teste5
2          inteiro x y z
3      proc soma (inteiro a inteiro b ref inteiro c)
4      inicio
5         c <- a + b
6      fimproc
7  inicio
8          leia x
9      leia y
10     soma (x y z)
11     escreva z
12 fimprograma
```

Código MVS correspondente:

```
1           INPP
2           AMEM    3
3           DSVS    L0
4  L1       ENSP
5           CRVL    -5
6           CRVL    -4
7           SOMA
8           ARMI    -3
9           RTSP    3
10 L0       NADA
11          LEIA
12          ARZG    0
13          LEIA
14          ARZG    1
15          CRVG    0
16          CRVG    1
17          CREG    2
18          SVCP
19          DSVS    L1
20          CRVG    2
21          ESCR
22          DMEM    3
23          FIMP
```

6.4.6 Teste 6 - Simples

```
1  programa teste6
2  inteiro z x
3
4  proc g (inteiro t)
5    inteiro y
6  inicio
7     y <- t * t
```

```
8     z <- z + x + y
9     escreva z
10 fimproc
11
12 proc f1 (inteiro x inteiro y)
13     inteiro t
14 inicio
15     t <- z + x + y
16     g(t)
17     z <- t
18 fimproc
19
20 proc h (inteiro y)
21   inteiro x
22 inicio
23   x <- y + 1
24   f1(x y)
25   g(z + x)
26 fimproc
27
28 inicio
29   z <- 1
30   x <- 3
31   h(x)
32   g(x)
33   escreva x
34   escreva z
35 fimprograma
```

Código MVS correspondente:

```
1          INPP
2          AMEM    2
3          DSVS    L0
4  L1      ENSP
5          AMEM    1
6          CRVL    -3
7          CRVL    -3
8          MULT
9          ARZL    0
10         CRVG    0
11         CRVG    1
12         SOMA
13         CRVL    0
14         SOMA
15         ARZG    0
16         CRVG    0
17         ESCR
18         DMEM    1
19         RTSP    1
20 L2      ENSP
21         AMEM    1
22         CRVG    0
23         CRVL    -4
24         SOMA
25         CRVL    -3
26         SOMA
```

```
27          ARZL    0
28          CRVL    0
29          SVCP
30          DSVS    L1
31          CRVL    0
32          ARZG    0
33          DMEM    1
34          RTSP    2
35  L3      ENSP
36          AMEM    1
37          CRVL    -3
38          CRCT    1
39          SOMA
40          ARZL    0
41          CRVL    0
42          CRVL    -3
43          SVCP
44          DSVS    L2
45          CRVG    0
46          CRVL    0
47          SOMA
48          SVCP
49          DSVS    L1
50          DMEM    1
51          RTSP    1
52  L0      NADA
53          CRCT    1
54          ARZG    0
55          CRCT    3
56          ARZG    1
57          CRVG    1
58          SVCP
59          DSVS    L3
60          CRVG    1
61          SVCP
62          DSVS    L1
63          CRVG    1
64          ESCR
65          CRVG    0
66          ESCR
67          DMEM    2
68          FIMP
```

6.4.7 Teste 7 - Simples

```
1  programa nome_programa
2     inteiro x y
3     proc le (ref inteiro a ref inteiro b)
4     inicio
5         leia a
6         leia b
7     fimproc
8  inicio
9       le (x y)
10      escreva x + y
11 fimprograma
```

Código MVS correspondente:

```
1          INPP
2          AMEM    2
3          DSVS    L0
4  L1      ENSP
5          LEIA
6          ARMI    -4
7          LEIA
8          ARMI    -3
9          RTSP    2
10 L0      NADA
11         CREG    0
12         CREG    1
13         SVCP
14         DSVS    L1
15         CRVG    0
16         CRVG    1
17         SOMA
18         ESCR
19         DMEM    2
20         FIMP
```

6.4.8 Teste 8 - Simples

```
1  programa nome_programa
2     inteiro x y
3     proc soma (inteiro a inteiro b)
4       inteiro s
5     inicio
6        s <- a + b
7        escreva s
8     fimproc
9  inicio
10    leia x
11    leia y
12    soma (x y)
13 fimprograma
```

Código MVS correspondente:

```
1          INPP
2          AMEM    2
3          DSVS    L0
4  L1      ENSP
5          AMEM    1
6          CRVL    -4
7          CRVL    -3
8          SOMA
9          ARZL    0
10         CRVL    0
11         ESCR
12         DMEM    1
13         RTSP    2
14 L0      NADA
```

```
15          LEIA
16          ARZG    0
17          LEIA
18          ARZG    1
19          CRVG    0
20          CRVG    1
21          SVCP
22          DSVS    L1
23          DMEM    2
24          FIMP
```

6.4.9 Teste 9 - Simples

```
1  programa t_subprogramas
2    inteiro a b c
3    logico f1 k4 y5
4
5    proc test(logico lol inteiro a inteiro b)
6      inteiro a1 a2 a3 a4 a5
7    inicio
8      se lol entao
9        escreva a
10     senao
11           escreva b
12     fimse
13   fimproc
14
15   func inteiro soma(inteiro a inteiro b)
16     inteiro a1 a2 a3 a4 a5
17     inteiro b1 b2 b3 b4 b5
18   inicio
19     soma <- a + b
20   fimfunc
21
22
23   proc proc_soma(ref inteiro c inteiro a inteiro b)
24   inicio
25     c <- a + b
26   fimproc
27
28   func logico maior(inteiro a inteiro b)
29   inicio
30     maior <- a > b
31   fimfunc
32
33
34 inicio
35   leia a
36   b <- 7
37
38   proc_soma(c a b)
39   escreva c
40
41   escreva soma (a b)
42
43   test(maior(a b) a b )
```

```
44
45    se maior(a b) entao
46            escreva 1
47    senao
48            escreva 0
49    fimse
50
51  fimprograma
```

Código MVS correspondente:

```
1           INPP
2           AMEM    6
3           DSVS    L0
4   L1      ENSP
5           AMEM    5
6           CRVL    -5
7           DSVF    L2
8           CRVL    -4
9           ESCR
10          DSVS    L3
11  L2      NADA
12          CRVL    -3
13          ESCR
14  L3      NADA
15          DMEM    5
16          RTSP    3
17  L4      ENSP
18          AMEM    10
19          CRVL    -4
20          CRVL    -3
21          SOMA
22          ARZL    -5
23          DMEM    10
24          RTSP    2
25  L5      ENSP
26          CRVL    -4
27          CRVL    -3
28          SOMA
29          ARMI    -5
30          RTSP    3
31  L6      ENSP
32          CRVL    -4
33          CRVL    -3
34          CMMA
35          ARZL    -5
36          RTSP    2
37  L0      NADA
38          LEIA
39          ARZG    0
40          CRCT    7
41          ARZG    1
42          CREG    2
43          CRVG    0
44          CRVG    1
45          SVCP
46          DSVS    L5
```

```
47          CRVG    2
48          ESCR
49          AMEM    1
50          CRVG    0
51          CRVG    1
52          SVCP
53          DSVS    L4
54          ESCR
55          AMEM    1
56          CRVG    0
57          CRVG    1
58          SVCP
59          DSVS    L6
60          CRVG    0
61          CRVG    1
62          SVCP
63          DSVS    L1
64          AMEM    1
65          CRVG    0
66          CRVG    1
67          SVCP
68          DSVS    L6
69          DSVF    L7
70          CRCT    1
71          ESCR
72          DSVS    L8
73  L7      NADA
74          CRCT    0
75          ESCR
76  L8      NADA
77          DMEM    6
78          FIMP
```

6.4.10 Teste 10 - Simples

```
1  programa nome_programa
2     func inteiro soma (inteiro a inteiro b)
3     inicio
4        soma <- a + b
5     fimfunc
6  inicio
7     escreva soma (10 20)
8     escreva soma (5 100)
9  fimprograma
```

Código MVS correspondente:

```
1           INPP
2           DSVS    L0
3   L1      ENSP
4           CRVL    -4
5           CRVL    -3
6           SOMA
7           ARZL    -5
8           RTSP    2
```

```
 9  L0      NADA
10          AMEM    1
11          CRCT    10
12          CRCT    20
13          SVCP
14          DSVS    L1
15          ESCR
16          AMEM    1
17          CRCT    5
18          CRCT    100
19          SVCP
20          DSVS    L1
21          ESCR
22          FIMP
```

Referências Bibliográficas

[Aho et al. 2008]AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas*. São Paulo: Pearson Addison-Wesley, 2008.

[J.L.Gersting 2004]J.L.GERSTING. *Fudamentos matemáticos para a Ciência da Computação: um tratamento moderno de matemática discreta*. Rio de Janeiro: LTC, 2004.

[Kowaltowski 1983]KOWALTOWSKI, T. *Implementação de linguagens de programação*. Rio de Janeiro: Guanabara, 1983.

[Lewis e Papadimitriou 2004]LEWIS, H. R.; PAPADIMITRIOU, C. H. *Elementos da Teoria da Computação*. Porto Alegre: Bookman, 2004.

[Louden 2004]LOUDEN, K. *Compiladores Princípios e Práticas*. São Paulo: Thomson, 2004.

[Price 2008]PRICE, A. M. A. *Implementação de Linguagens de Programação:Compiladores*. Rio Grande do Sul: Sagra-Luzzatto, 2008.

[Sipser 2012]SIPSER, M. *Introdução a Teoria da Computação*. São Paulo: Cengage Learning, 2012.

[Vieira 2006]VIEIRA, N. J. *Introdução aos fundamentos da computação*. São Paulo: Pioneira Thompson Learning, 2006.

[Ziviani 1999]ZIVIANI, N. *Projeto de Algoritmos: com implementação em Pascal e C*. São Paulo: Pioneira, 1999.