Eduardo Belo de Araújo

Analisador ANSI-C

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* Administração de Redes Linux, para a obtenção do título de especialista.

Orientador
Prof. DSc. João Carlos Giacomin

Lavras
Minas Gerais – Brasil
2005

Eduardo Belo de Araújo

Analisador ANSI-C

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências do curso de Pós-Graduação *Lato Sensu* Administração em Redes Linux, para a obtenção do título de especialista.

Aprovada em ___ de _____ de_____

_____
Prof. MSc. Douglas Machado Tavares

_____
Prof. MSc. Fernando Cortez Sica

_____
Prof. MSc. João Carlos Giacomin
(Orientador)

Lavras
Minas Gerais – Brasil

Dedico este trabalho a meus pais e a minha namorada Juliana.

Agradecimentos

Ao professor Giacomin por toda a atenção, boa vontade e orientação que me deu no decorrer deste trabalho.

A minha família e namorada, pela compreensão de todas as vezes que me ausentei para me dedicar a este trabalho.

E a todos os amigos do trabalho e do meio acadêmico, que diretamente ou indiretamente contribuíram de alguma forma.

Resumo

O trabalho em questão visa o desenvolvimento de um analisador de código C, que ofereça ao usuário iniciante da linguagem C um diálogo amigável, reportando as mensagens de erro de programação no idioma português, funcionando como um analisador de código C. O objetivo deste trabalho é que o analisador seja uma ferramenta auxiliar no ensino da linguagem C. É importante destacar, que o analisador irá tratar alguns tipos de erros comumente encontrados nesta linguagem, e não todos os possíveis erros, e não fará a compilação dos programas, a qual deverá ser feita por um compilador que o programador escolher. Dessa forma, espera-se que este trabalho possa contribuir para a comunidade Linux e especialmente para o ensino da linguagem C.

Sumário

1 – Introdução 2
1.1 Cenário ... 3
1.2 Estrutura do Trabalho ... 4

2 - Análise Léxica 6
2.1 Alfabetos, Palavras, Gramáticas e Linguagens ... 8
2.1.1 Definição de Alfabeto ... 8
2.1.2 Definição de Palavra, Cadeia de Caracteres ou Sentença ... 8
2.1.3 Definição de Gramática ... 9
2.1.4 Definição de Linguagem Gerada ... 10
2.1.5 Definição de Gramática Regular ... 11
2.1.6 Definição de Expressão Regular ... 12
2.2 Tokens ... 12
2.3 Tabela de Símbolos ... 14
2.4 Conclusão ... 17

3 - Análise Sintática 18
3.1 Gramáticas Livres do Contexto ... 19
3.1.1 Definição de Gramática Livre do Contexto ... 21
3.1.2 Definição de Árvore de Derivação ... 21
3.1.3 Derivações mais à Esquerda e mais à Direita ... 22
3.1.4 Gramática Ambígua ... 23
3.2 Análise Descendente ... 24
3.2.1 Análise Recursiva com Retrocesso ... 25
3.2.2 Análise Recursiva Preditiva ... 28
3.2.3 Análise Preditiva Tabular ... 29
3.3 Análise Redutiva ... 32
3.3.1 Definição de *Handles* ... 33
3.3.2 Analisadores de Precedência de Operadores ... 35
3.3.3 Analisadores LR ... 37
3.4 Conclusão ... 40

4 - Análise Semântica 41
4.1 Tradução Dirigida pela Sintaxe ... 43
4.2 Gramática de Atributos ... 43
4.3 Geração de Código Intermediário ... 45

5 - Linguagens Formais e Autômatos **48**
5.1 Autômato Finito 48
5.2 Autômatos Finitos do Analisador ANSI C 50
5.2.1 Diretivas do Pré-Processador 51
5.2.1.1 Diretiva #include 52
5.2.1.2 Diretiva #define 52
5.2.2 Tipos de Variáveis 53
5.2.3 Estruturas de Dados 54
5.2.4 Funções e Protótipos de Funções 55
5.2.5 Estruturas de Controle de Fluxo 58
5.2.5.1 If-Else 58
5.2.5.2 Switch 59
5.2.5.3 For 61
5.2.5.4 While 62
5.2.5.5 Do-While 63
5.2.6 Comentários 64
5.3 Conclusão 65

6 - Estrutura Adotada no Projeto do Analisador de Código **66**
6.1 Primeira Etapa do Analisador de Código 67
6.2 Segunda Etapa do Analisador de Código 72
6.2.1 Declaração de Bibliotecas 72
6.2.2 Declaração de Constantes 75
6.2.3 Declaração de Variáveis 77
6.2.4 Declaração de Funções 83
6.2.5 Mensagens de Erro 86
6.3 Terceira Etapa do Analisador de Código 88
6.4 Conclusão 92

7 – Conclusão **93**
7.1 Sugestões para Trabalhos Futuros 95

Referências Bibliográficas 96

Lista de Figuras

2.1 Interação do analisador léxico com o analisador sintático 7
2.2 Exemplo de uma gramática e sua linguagem 10

3.1 Árvores de derivação 22
3.2 Árvores diferentes para uma mesma sentença 23
3.3 Reconhecimento da sentença [a] 26
3.4 Produções dos comandos *if-then*, *while-do* e atribuição de valores 28
3.5 Modelo de um analisador sintático preditivo tabular 30
3.6 Gramática não-ambígua 31
3.7 Eliminando recursividade à esquerda 31
3.8 Gramática que gera expressões lógicas 36
3.9 Estrutura de um analisador sintático LR 39

4.1 Exemplo de erro semântico na verificação de tipos 41
4.2 Exemplo de produções para declaração simples 42
4.3 Árvore de derivação para a expressão 2*3+4= 45
4.4 Posição do gerador de código intermediário 46

5.1 Autômato finito como uma máquina com controle finito 49
5.2 Módulo include 52
5.3 Módulo define 53
5.4 Módulo variáveis 54
5.5 Módulo struct 55
5.6 Módulo function 56
5.7 Módulo protótipo 57
5.8 Módulo if-else 59
5.9 Módulo switch 60
5.10 Módulo for 62
5.11 Módulo while 63
5.12 Módulo do-while 64
5.13 Módulo comentário 65

6.1 Chamada do analisador de código passando o arquivo por parâmetro 67
6.2 Código fonte antes da execução do analisador (Exemplo 1) 69
6.3 Código fonte após a execução do analisador (Exemplo 1) 69
6.4 Código fonte antes da execução do analisador (Exemplo 2) 70
6.5 Código fonte após a execução do analisador (Exemplo 2) 71

6.6 Exemplo com biblioteca que não faz parte do padrão ANSI-C 73
6.7 Mensagem de erro para a biblioteca que não pertence ao padrão ANSI-C. 74
6.8 Exemplo de declaração de constantes 75
6.9 Mensagem de erro para constante declarada com nome inválido 76
6.10 Exemplo de declaração de variáveis (Exemplo 1) 78
6.11 Mensagens de erro nas declarações de variáveis 79
6.12 Exemplo de declaração de variáveis (Exemplo 2) 81
6.13 Mensagens de erro nas declarações de variáveis 82
6.14 Exemplo de declarações de funções 84
6.15 Mensagens de erro nas declarações de funções 85
6.16 Código fonte de exemplo para a terceira etapa 89
6.17 Mensagens de erro geradas para o código da Figura 6.15 91

Lista de Tabelas

2.1 Tabela de símbolos para armazenar informações referentes às funções..... 16
2.2 Tabela de símbolos para armazenar informações referentes às variáveis... 16

3.1 Estruturas da notação BNF 20
3.2 Gramática de exemplo 32
3.3 Redução da sentença *abbcde* 33
3.4 Relações de precedência entre terminais 36
3.5 Tabela de precedência para expressões simplificadas 36

4.1 Esquema de tradução para uma calculadora aritmética 44

6.1 *Tokens* especiais..... 68
6.2 Tabela do pré-compilador para armazenar bibliotecas do padrão ANSI-C 73
6.3 Tabela de símbolos de bibliotecas 74
6.4 Tabela de símbolos de constantes..... 76
6.5 Tabela do analisador de código para armazenar as palavras reservadas da linguagem C..... 80
6.6 Tabela de símbolos de variáveis 82
6.7 Tabela de símbolos de funções 85
6.8 Tabela de mensagens de erros do analisador de código C..... 87
6.9 Tabela de funções do padrão ANSI-C 90

1 - Introdução

Nas décadas de 1980 e 1990, enquanto todo o mundo testemunhava o surgimento do PC e da Internet nas primeiras páginas das revistas e jornais, os métodos de projeto de compiladores desenvolviam-se com menos alarde, avanços vistos principalmente nas publicações técnicas e científicas, e mais importantes ainda, nos compiladores que são usados para processar os softwares atuais.

Esses desenvolvimentos foram encaminhados em parte pelo advento de novos paradigmas de programação, e em parte por uma melhor compreensão das técnicas de geração de códigos, e ainda pelo aparecimento de arquiteturas de hardware mais rápidas e com grande disponibilidade de memória.

Segundo Grune (2001), dentre os projetos de compiladores mais bem sucedidos e mais disseminados em toda à comunidade acadêmica, têm-se como destaque o compilador da linguagem C, ou compilador C, que é o foco principal do tema desse trabalho.

A linguagem C foi uma evolução da B, feita por Dennis Ritchie no *Bell Laboratories* e foi originalmente implementada em um computador DEC PDP-11 em 1972. C inicialmente tornou-se largamente conhecida como a linguagem de desenvolvimento do sistema operacional UNIX. (Giacomin, 2003).

Muitos alunos e programadores iniciantes na linguagem C, têm dificuldade em saber se estão utilizando somente funções do padrão ANSI-C.

A importância de se utilizar apenas funções do padrão ANSI-C é que os programas se tornam portáveis, isto é, se tornam independentes de plataforma, podendo um mesmo programa ser compilado em diferentes computadores e sobre diferentes sistemas operacionais.

Outra questão que merece importância é a dificuldade que iniciantes têm, para entender e interpretar as mensagens de erros exibidas durante a compilação de algum programa, principalmente pelo fato dos diversos compiladores disponíveis hoje exibirem tais mensagens no idioma inglês.

Dessa forma, o trabalho em questão tem como proposta o desenvolvimento de um Pré-Compilador para linguagem C, levando-se em consideração as principais finalidades que são:

- Verificar o nome das bibliotecas e funções indicando suas relações com o padrão ANSI-C;
- Verificar os protótipos, parâmetros e chamadas das funções implementadas no programa-fonte;
- Verificar a ambigüidade de variáveis, tipos e suas localizações dentro do escopo do programa;
- Verificar erros de sintaxe, a falta e/ou excesso de parênteses, chaves e colchetes, variáveis utilizadas sem a prévia declaração e falta de ponto e vírgula no final das linhas de comando;
- Exibir todas as mensagens de notificação de erro no idioma português.

1.1 Cenário

Hoje em dia temos uma grande variedade de tecnologias de linguagens de programação. Porém, por mais que dominemos uma linguagem, sempre haverá a necessidade de aprendermos outras, devido às restrições e recursos que são peculiares a cada uma.

Cada linguagem possui sintaxe, dificuldades e particularidades, que as diferenciam umas das outras, algumas com grau de complexidade maior, outras menor, mas o importante é que quando começamos a estudar uma determinada

linguagem, ora ou outra, nos deparamos com os mais variados tipos de erros, que às vezes nos custam bastante tempo para compreendê-los e corrigi-los.

Muitas das vezes poderíamos recorrer às mensagens de erros reportadas pelo compilador, durante sua execução, para solucionar erros existentes no programa fonte.

O que acontece é que por padrão estas mensagens são exibidas no idioma inglês, e para quem não domina tal idioma, analisar e relacionar tais mensagens aos respectivos erros, pode se tornar uma tarefa difícil.

Diante do cenário exposto acima, o projeto em questão visa propor um ambiente de pré-compilação, que ofereça ao usuário iniciante na linguagem de programação C, um diálogo mais amigável, reportando mensagens de erros no idioma português, estas provenientes de diversas análises realizadas no programa fonte.

Isto é especialmente útil no ensino desta linguagem, quando os alunos têm os primeiros contatos com C, e começam a desenvolver pequenos programas.

Com isso, espera-se oferecer uma ferramenta de grande auxílio para o ensino da linguagem C, e uma contribuição significativa para a comunidade Linux.

1.2 Estrutura do Trabalho

Este trabalho foi estruturado em sete capítulos de maneira a apresentar a introdução, o cenário do trabalho e a sua estrutura no primeiro capítulo.

O segundo capítulo traz os fundamentos e conceitos sobre a Análise Léxica.

O terceiro capítulo apresenta os fundamentos e conceitos da etapa seguinte que é a Análise Sintática.

O quarto capítulo faz uma abordagem à última das etapas das análises que é a Semântica.

O quinto capítulo apresenta as Linguagens Formais e Autômatos, na qual foi planejada e projetada a base do pré-compilador.

O sexto capítulo faz uma abordagem sobre a estrutura adotada no projeto do pré-compilador.

No sétimo capítulo são feitas as considerações finais sobre o trabalho, incluindo as conclusões e sugestões para trabalhos futuros.

2 - Análise Léxica

A análise léxica é a primeira fase do compilador. A função do analisador léxico, também denominado *scanner* é fazer a leitura do programa fonte, caractere a caractere, e traduzi-los para uma seqüência de *símbolos léxicos,* também chamados *tokens.*

Exemplos de símbolos léxicos são as palavras reservadas, os identificadores, as constantes e os operadores da linguagem. (Price, 2001).

Durante o processo de análise léxica, são desprezados caracteres não significativos como comentários e espaços em branco, os últimos sob a forma de espaços, tabulações e caracteres de avanço de linha.

Além de reconhecer os símbolos léxicos, o analisador também realiza outras funções, como armazenar alguns desses símbolos, geralmente identificadores e constantes, em tabelas internas e indicar a ocorrência de erros léxicos.

A seqüência de *tokens* produzida, ou seja, reconhecida pelo analisador léxico é utilizada como entrada pela fase seguinte do compilador, o analisador sintático.

A figura 2.1 representa a interação entre as análises léxica e sintática.

É interessante observar que o mesmo programa fonte é visto pelos analisadores léxico e sintático como sentenças de linguagens diferentes.

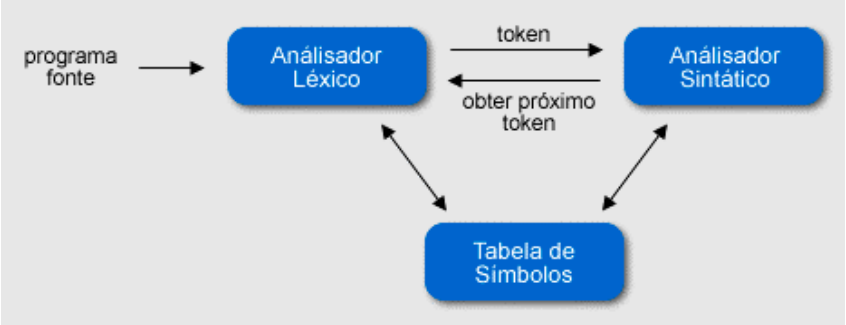

Figura 2.1: Interação do analisador léxico com o analisador sintático (Aho, 1986)

Para o analisador léxico, o programa fonte é uma seqüência de palavras de uma *linguagem regular*. Para o analisador sintático, essa seqüência de *tokens* constitui uma sentença de uma *linguagem livre do contexto* (sentenças formadas por expressões e blocos aninhados, por exemplo). (Price, 2001).

Os analisadores sintáticos serão abordados no Capítulo 3 desse estudo.

Uma *linguagem* é um conjunto de palavras formadas por símbolos de um determinado alfabeto. Os símbolos constituem uma *linguagem regular*.

As linguagens regulares e as livres do contexto são as mais simples, segundo a classificação proposta por Chomsky[1], dentro da Teoria das Linguagens Formais. No que diz respeito ao contexto da tradução de linguagens de programação, as linguagens são usualmente apresentadas através de gramáticas ou de algoritmos (autômatos) que as reconhecem. (Price, 2001).

[1] Segundo a Classificação de Chomsky, em ordem crescente de complexidade e de generalidade, os tipos de linguagens são: linguagens regulares (tipo 3), linguagens livres do contexto (tipo 2), linguagens sensíveis ao contexto (tipo 1) e linguagens recursivamente enumeráveis (tipo 0).

2.1 Alfabetos, Palavras, Gramáticas e Linguagens

Ferreira (2004) define *linguagem* como o uso da palavra articulada ou escrita como meio de expressão e comunicação entre pessoas.

"Entretanto, esta definição não é suficientemente precisa para permitir o desenvolvimento matemático de uma teoria sobre linguagens" (Menezes, 2000).

Dessa forma, se faz necessária a apresentação de algumas definições formais que serão úteis no decorrer desse capítulo.

2.1.1 Definição de Alfabeto

"Um *Alfabeto* é um conjunto finito de *Símbolos*. Um *símbolo* é uma entidade abstrata básica a qual não é definida formalmente. Letras e dígitos são exemplos de símbolos frequentemente usados" (Menezes, 2000).

2.1.2 Definição de Palavra, Cadeia de Caracteres ou Sentença

"Uma *Palavra*, *Cadeia de Caracteres* ou *Sentença* sobre um alfabeto é uma seqüência finita de símbolos justapostos" (Menezes, 2000).

"A *palavra vazia*, representada pelo símbolo ε, é uma palavra sem símbolo. Se $\sum$ representa um alfabeto, então $\sum^*$ denota o conjunto de todas as palavras possíveis sobre $\sum$" (Menezes, 2000).

2.1.3 Definição de Gramática

Uma Gramática é uma quádrupla ordenada G = (N, T, P, S) onde:

- N é o conjunto finito de símbolos *variáveis* ou *não-terminais*;
- T é o conjunto de símbolos *terminais* disjunto de N;
- P é o conjunto finito de pares, denominados *regras de produção* tal que a primeira componente é palavra de (N U T)+ e a segunda componente é a palavra de (N U T)*;
- S é o elemento de N denominado *variável inicial*.

> "Uma regra de produção (α, β) é representada por $\alpha \rightarrow \beta$. As regras de produção definem as condições de geração das palavras da linguagem. Uma seqüência de regras de produção da forma $\alpha \rightarrow \beta 1$, $\alpha \rightarrow \beta 2$, ..., $\alpha \rightarrow \beta n$ (mesma componente no lado esquerdo) pode ser abreviada como uma única produção na forma:
>
> $\alpha \rightarrow \beta 1 \mid \beta 2 \mid ... \mid \beta n$
>
> A aplicação de uma regra de produção é denominado derivação de uma palavra. A aplicação sucessiva de regras de produção permite derivar as palavras da linguagem representada pela gramática" (Menezes, 2000).

A figura 2.2 apresenta o exemplo de uma gramática e a geração de sua respectiva linguagem.

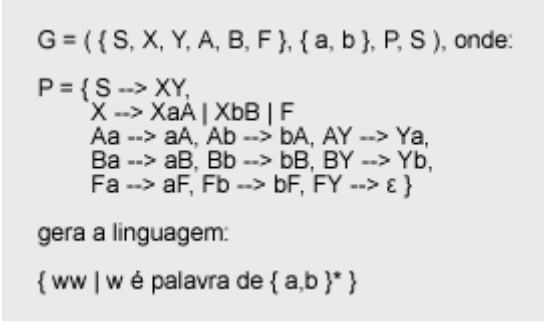

Figura 2.2: Exemplo de uma gramática e sua linguagem (Menezes, 2000)

2.1.4 Definição de Linguagem Gerada

A linguagem gerada por G, denotada por L(G), é o conjunto formado por todas as sentenças de símbolos terminais deriváveis a partir do símbolo inicial S, ou seja

$$L(G) = \{ s \mid s \text{ é um elemento de } T^* \text{ e } S \rightarrow +s \}$$

É importante ressaltar algumas convenções usadas para representar os símbolos gramaticais (terminais e não-terminais) e as formas sentenciais (seqüências de símbolos terminais e não-terminais), que segundo Price (2001) são:

1) Símbolos que representam terminais:
    a. Letras minúsculas do início do alfabeto tais como: a, b, c, ... (as letras finais v, w, x, ... são usadas para representar cadeias de terminais)
    b. Operadores em geral, tais como: +, -, *, ...
    c. Símbolos que representam pontuação, a saber, vírgula, ponto-e-vírgula, ponto e os delimitadores tais como: (,), [,]
    d. Dígitos: 0, 1, 2, 3, 4, ...
    e. Cadeias de caracteres em negrito, como **if**, **while**, **for**, ...

2) Símbolos que representam não-terminais:
    a. Letras maiúsculas do alfabeto: A, B, C, D, ... (em geral, S representa o símbolo inicial)
    b. Qualquer *string* delimitado por "<" e ">"
3) Símbolos que representam formas seqüenciais (seqüências de terminais e não-terminais):
    a. As letras minúsculas do alfabeto grego, como α, β, γ, δ, ...

2.1.5 Definição de Gramática Regular

Uma gramática que tem produções exclusivamente de forma $A \rightarrow wB$ ou $A \rightarrow w$, onde $w \in T^*$ e $A, B \in N$, é classificada como uma *gramática regular.*

Uma gramática tal como definida acima é dita *gramática linear à direita.* As *gramáticas lineares à esquerda*, cujas produções são da forma $A \rightarrow Bw$ ou $A \rightarrow w$, também são regulares. Para qualquer gramática regular, se $|w| <= 1$, então a gramática é dita *unitária*. (Price, 2001).

Tomando como exemplo, a *gramática regular* abaixo gera identificadores que iniciam por letra (l), podendo esta ser seguida por qualquer número de letras e/ou dígitos (d). G é uma gramática linear à direita unitária.

G = (N, T, P, I)
N = { I, R }
T = { l, d }
P = { $I \rightarrow l \mid l\,R$, $R \rightarrow l\,R \mid d\,R \mid l \mid d$ }

As *linguagens regulares* podem ser reconhecidas por máquinas de transição de estados chamados *autômatos finitos.* Fornecendo-se uma sentença a um autômato, este é capaz de responder se a mesma pertence ou não a linguagem que ele representa. (Price, 2001).

Os autômatos serão vistos com ênfase no Capítulo 5 desse trabalho.

2.1.6 Definição de Expressão Regular

Uma *expressão regular* r, sobre um conjunto de símbolos T, representa uma linguagem L(r), a qual pode ser definida indutivamente a partir de expressões básicas, que segundo Price (2001) são:

1) ∅ representa a linguagem vazia (conjunto contendo zero palavras);
2) { ε } representa a linguagem cuja única palavra é a palavra vazia;
3) { x | x ∈ T } representa a linguagem cuja única palavra é x;
4) Se r1 e r2 são expressões regulares definindo as linguagens L(r1) e L(r2), tem-se que:
    a. r1 | r2 é a linguagem cujas palavras constituem o conjunto L(r1) U L(r2);
    b. r1 r2 é a linguagem { vw | v ∈ L(r1) e w ∈ L(r2) }, isto é, a linguagem cujas palavras são formadas pela concatenação de uma palavra de L(r1) com uma palavra de L(r2), nesta ordem;
    c. r1* representa o conjunto L*(r1), isto é, o conjunto de palavras que podem ser formadas concatenando-se zero ou mais palavras de L(r1).

2.2 Tokens

Conforme dito anteriormente, a função do analisador léxico é ler uma seqüência de caracteres que constitui um programa fonte e coletar, dessa seqüência, os *tokens* (palavras de uma linguagem regular) que constituem o programa. Os *tokens* ou símbolos léxicos são as unidades básicas do texto do programa.

Segundo Price (2001) cada *token* é representado internamente por três informações:

1) *Classe do token*, que representa o tipo do *token* reconhecido. Exemplos de classes são: identificadores, constantes numéricas, cadeias de caracteres, palavras reservadas, operadores e separadores.
2) *Valor do token*, o qual depende da classe. Para *tokens* da classe constante inteira, por exemplo, o valor do *token* pode ser o número inteiro representando pela constante. Para *tokens* da classe identificador, o valor pode ser a seqüência de caracteres, lida no programa fonte, que representa o identificador, ou um apontador para a entrada de uma tabela que contém essa seqüência de caracteres. Essa tabela, chamada Tabela de Símbolos, será discutida na Seção 2.3. Algumas classes de *tokens*, como as palavras reservadas, não têm valor associado.
3) *Posição do token*, a qual indica o local do texto fonte (linha e coluna) onde ocorreu o *token*. Essa informação é utilizada, principalmente, para indicar o local de erros.

Em função do campo valor, os *tokens* podem ser divididos em dois grupos:

- *Tokens simples*, que são os *tokens* que não têm um valor associado (como as palavras reservadas, operadores e delimitadores) porque a classe do *token* descreve-o completamente. Esses *tokens* correspondem a elementos fixos da linguagem.
- *Tokens com argumento*, que são os *tokens* que têm um valor associado (como identificadores e constantes). Correspondem aos elementos da linguagem definidos pelo programador como, por exemplo, identificadores, constantes numéricas e cadeias de caracteres.

Tomando como exemplo, tem-se o trecho de programa

while x < 200 x = x + y ;

onde a análise léxica desse segmento produz a seguinte seqüência: [whi,] [id, 5] [<,] [cte, 15] [id, 5] [=,] [id, 5] [+,] [id, 6] [;,].

Para facilitar a compreensão, os *tokens* estão representados por pares. Identificadores e constantes numéricas estão representados pelo par [classe_token, índice_tabela].

As classes para palavras reservadas constituem-se em abreviações dessas, não sendo necessário passar seus valores para o analisador sintático. Para delimitadores e operadores, a classe é o próprio valor do *token*.

Normalmente, os compiladores representam a classe do *token* por um número inteiro para tornar a representação mais compacta. No exemplo apresentado acima, empregou-se uma representação simbólica para facilitar a compreensão.

2.3 Tabela de Símbolos

A tabela de símbolos é uma estrutura de dados gerada pelo compilador com o objetivo de armazenar informações sobre os nomes (identificadores de variáveis, de parâmetros, de funções, de procedimentos, de constantes, etc.) definidos no programa fonte.

Esta tabela associa atributos (tais como tipo, escopo, limites no caso de vetores e número de parâmetros no caso de funções) aos nomes definidos pelo programador. Em geral, ela começa a ser construída durante a análise léxica, quando os identificadores são reconhecidos.

Na primeira vez que um identificador é encontrado, o analisador léxico armazena-o na tabela, mas talvez não tenha ainda condições de associar atributos

a esse identificador. Às vezes, essa tarefa só pode ser executada durante as fases de análise sintática e semântica.

Toda vez que um identificador é reconhecido no programa fonte, a tabela de símbolos é consultada, a fim de verificar se o nome já está registrado; caso não esteja, é feita sua inserção na tabela de símbolos.

Existem vários modos de organizar e acessar tabelas de símbolos. Os mais comuns são através de listas lineares, árvores binárias e tabelas *hash*. Lista linear é o mecanismo mais simples, mas seu desempenho é pobre quando o número de consultas é elevado. Tabelas *hash* têm melhor desempenho, mas exigem mais memória e esforço de programação.

Cada entrada na tabela de símbolos está relacionada com a declaração de um nome. As entradas podem não ser uniformes para classes distintas de identificadores. Por exemplo, entradas para identificadores de funções requerem registro do número de parâmetros, enquanto entradas para identificadores de matrizes requerem registro dos limites inferior e superior de cada dimensão.

Nesses casos, pode-se ter parte da entrada uniforme e usar ponteiros para registros com informações adicionais.

O armazenamento dos nomes pode ser feito diretamente na tabela ou em uma área distinta. No primeiro caso, tem-se um desperdício de memória pela diversidade de tamanho dos identificadores; no segundo, a recuperação de nomes é ligeiramente mais demorada. (Price, 2001).

As Tabelas 2.1 e 2.2 que são mostradas abaixo são exemplos de tabelas de símbolos. A primeira representa uma tabela desenvolvida para armazenar informações referentes às declarações de funções, a segunda, informações referentes às declarações de variáveis.

Tabela 2.1: Tabela de símbolos para armazenar informações referentes às funções

Identificador da função	Tipo de retorno	Quantidade de parâmetros	Linha onde foi declarada
calcularFatorial	int	1	253
gerarNome	char	3	1065
existePalavra	int	2	750
...			

A tabela de símbolos para funções, representada acima, possui quatro colunas que são *Identificador da função*, que armazena o nome da função, *Tipo de retorno*, que armazena o tipo de retorno de dados da função, *Quantidade de parâmetros*, que armazena o número de parâmetros que a função possui, e *Linha onde foi declarada*, que armazena a posição (linha) do código fonte onde a função foi declarada.

Tabela 2.2: Tabela de símbolos para armazenar informações referentes às variáveis

Identificador da variável	Tipo	Escopo	Identificador da função	Linha onde foi declarada
nomeFuncionario	char	local	cadastrarFuncionario	135
enderecoEmpresa	char	local	cadastrarFuncionario	136
arquivoEntrada	FILE	global	-	5
...				

A tabela de símbolos para variáveis, representada acima, possui cinco colunas que são *Identificador da variável*, que armazena o nome da variável, *Tipo*, que armazena o tipo da estrutura de dados da variável, *Escopo*, que armazena a delimitação da variável dentro do contexto do programa, ou seja, local ou global, *Identificador da função*, que armazena; caso a variável seja de escopo local; o nome da função ao qual ela pertence, e por último, *Linha onde foi declarada*, que armazena a posição (linha) do código fonte onde a variável foi declarada.

2.4 Conclusão

Este capítulo apresentou a primeira fase do compilador que é a análise léxica. A análise léxica, também conhecida como *scanner*, tem como tarefa fazer a leitura do programa fonte e traduzir os caracteres existentes para uma seqüência de símbolos léxicos, também conhecidos como *tokens*.

Outro aspecto importante visto neste capítulo foi a tabela de símbolos, que representa uma estrutura de dados gerada pelo compilador, como o objetivo de armazenar informações sobre os identificadores de variáveis, de parâmetros, de funções, de procedimentos e de constantes definidas no programa fonte.

No Capítulo 3, será apresentada a segunda fase do compilador, a análise sintática.

3 - Análise Sintática

A análise sintática constitui a segunda fase de um compilador. Seu objetivo é verificar se as construções usadas no programa estão gramaticalmente corretas. Normalmente, as estruturas sintáticas válidas são especificadas através de uma gramática livre do contexto[2] (GLC). Na Seção 3.1 será feita uma revisão sobre gramáticas livres do contexto.

Dada uma gramática livre do contexto G e uma sentença (programa fonte) **s**, o objetivo do analisador sintático é verificar se a sentença **s** pertence à linguagem gerada por G. o analisador sintático; ou reconhecedor sintático; também chamado *parser*, recebe do analisador léxico a seqüência de *tokens* que constitui a sentença **s** e produz como resultado uma árvore de derivação para **s**, se a sentença é válida, ou emite uma mensagem de erro, caso contrário. (Price, 2001).

Os analisadores sintáticos devem ser projetados para relatarem quaisquer erros de sintaxe de uma forma inteligível. O ideal é que eles possam prosseguir na análise, mesmo que haja erros no programa fonte, a fim de poder continuar processando o resto de sua entrada.

Segundo Aho (1986), existem duas estratégias básicas para a análise sintática que são:

- Estratégia *Top-Down* ou Descendente;
- Estratégia *Bottom-Up* ou Redutiva.

[2] Gramáticas mais simples, como as regulares, não permitem especificar construções do tipo de expressões aritméticas e comandos aninhados.

Os métodos de análise baseados na estratégia *top-down* (descendente) constroem a árvore de derivação a partir do símbolo inicial da gramática (raiz da folha), fazendo a árvore crescer até atingir suas folhas.

A estratégia *bottom-up* (redutiva) realiza a análise no sentido contrário, isto é, a partir dos *tokens* do programa fonte (folhas da árvore de derivação) constrói a árvore até o símbolo inicial da gramática.

Na estratégia *top-down*, em cada passo, um lado esquerdo de produção é substituído por um lado direito (expansão); na estratégia *bottom-up*, em cada passo, um lado direito de produção é substituído por um símbolo não-terminal (redução). (Price, 2001).

Uma ressalva importante a se fazer é que, o analisador léxico vê o programa fonte como uma seqüência de palavras de uma linguagem regular e o reconhece através de um autômato finito, enquanto que o analisador sintático vê o referente programa fonte como uma sentença de uma linguagem livre do contexto.

3.1 Gramáticas Livres do Contexto

As gramáticas livres do contexto, popularizadas pela notação BNF (*Backus Naur Form*), formam a base para a análise sintática das linguagens de programação, pois permitem descrever a maioria das linguagens de programação usadas atualmente. (Price, 2001).

A notação de *Backus-Naur*, conhecida usualmente como o formulário de *Backus-Naur*, é utilizada para expressar gramáticas livres do contexto, isto é, uma maneira formal de descrever *linguagens formais*. BNF é usado extensamente como notação para as gramáticas de linguagens de programação e até mesmo em protocolos de comunicação.

BNF foi desenvolvido por John Backus, revisado e expandido por Peter Naur, que eram dois pioneiros na informática, especialmente na área de projetos de compilador. Eles o desenvolveram inicialmente para descrever a sintaxe da linguagem de programação do ALGOL[3]. (Garshol, 2003).

Abaixo segue a Tabela 3.1 com as estruturas da notação BNF.

Tabela 3.1: Estruturas da notação BNF (Garshol, 2003)

Estrutura	**Descrição**
::=	Identificado como
<...>	Identificadores
{...}	Repetições
(...)	Elementos agrupados
...|...	Escolha entre dois ou mais elementos

É interessante observar a seguinte relação entre linguagens regulares e linguagens livres do contexto: uma linguagem regular pode ser reconhecida por um autômato simples e para a mesma é possível, a partir da expressão regular que a descreve, construir automaticamente um analisador léxico.

Semelhantemente, uma linguagem livre do contexto pode ser reconhecida por um autômato de pilha e para ela é possível, a partir da gramática livre do contexto que a descreve, construir automaticamente um analisador sintático. (Price, 2001).

A seguir serão apresentados conceitos a respeito de gramáticas livres do contexto que serão importantes no decorrer do capítulo.

[3] O ALGOL é uma linguagem de programação desenvolvida originalmente em 1950. Foi projetado para evitar alguns dos problemas existentes no FORTRAN e deu eventualmente a ascensão a muitas outras linguagens de programação, entre elas o Pascal. (Algol, 1996).

3.1.1 Definição de Gramática Livre do Contexto

Uma *Gramática Livre do Contexto* G é uma gramática onde:

G = (V, T, P, S)

com a restrição de que qualquer regra de produção de P é da forma A → α, onde A é uma variável de V e α uma palavra de (V U T)*.

"Portanto, uma Gramática Livre do Contexto é uma gramática onde o lado esquerdo das produções contém exatamente uma variável" (Menezes, 2000).

O exemplo abaixo, retirado de Menezes (2000), mostra expressões aritméticas geradas pela gramática G:

G = ({ E }, { +, -, *, /, (,), x }, P, E) sendo

P = { E → E + E | E – E | E * E | E / E | (E) | x }

3.1.2 Definição de Árvore de Derivação

Árvore de derivação é a representação gráfica de uma derivação de sentença. Essa representação apresenta, de forma explícita, a estrutura hierárquica que originou a sentença.

Segundo Menezes (2000) dada uma GLC, a árvore de derivação para uma sentença é obtida como se segue:

- A raiz da árvore é o símbolo inicial da gramática;
- Os vértices interiores, obrigatoriamente, são não-terminais. Se A → X1X2...Xn é uma produção da gramática, então A será um vértice interior, e X1X2Xn serão os seus filhos ordenados da esquerda para a direita;
- Um *vértice folha* é um símbolo terminal, ou o símbolo vazio. Neste caso, o vazio é o único filho de seu pai (A → ε).

A palavra aabb e a expressão **[x+x]*x** são geradas pelas árvores de derivação mostradas na Figura 3.1.

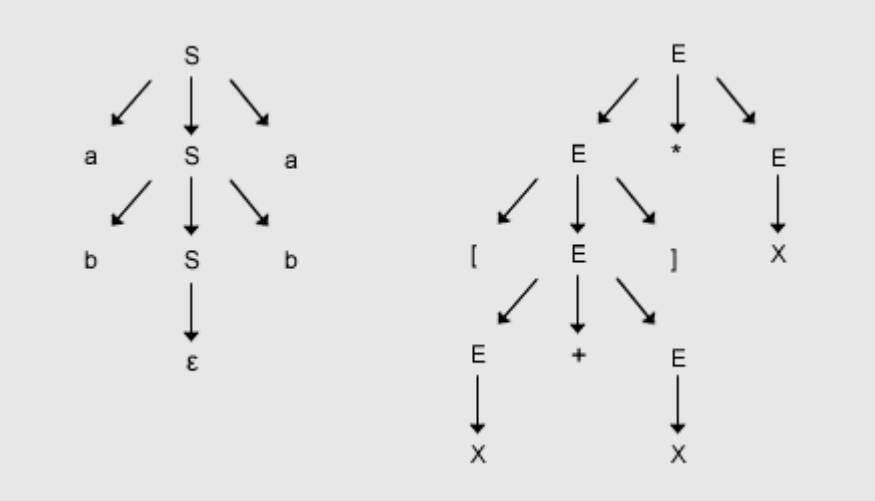

Figura 3.1: Árvores de derivação (Menezes, 2000)

3.1.3 Derivações mais à Esquerda e mais à Direita

Derivação mais à esquerda de uma sentença é a seqüência de formas sentenciais que se obtém derivando sempre o símbolo não-terminal mais à esquerda. Uma *derivação mais à direita* aplica as produções sempre ao não-terminal mais à direita. (Price, 2001).

Abaixo segue o exemplo, onde a sentença **x+x*x** pode ser derivada gerando duas seqüências, a primeira (a) se trata de uma derivação mais à esquerda, e a segunda (b) de uma derivação mais à direita.

a) $E \rightarrow E + E \rightarrow x + E \rightarrow x + E * E \rightarrow x + x * E \rightarrow x + x * x$

b) $E \rightarrow E + E \rightarrow E + E * E \rightarrow E + E * x \rightarrow E + x * x \rightarrow x + x * x$

3.1.4 Gramática Ambígua

"Uma Gramática Livre do Contexto é dita uma *Gramática Ambígua*, se existe uma sentença que possua duas ou mais árvores de derivação" (Menezes, 2000).

A Figura 3.2 mostra que a sentença **x+x*x** pode ser derivada por árvores diferentes.

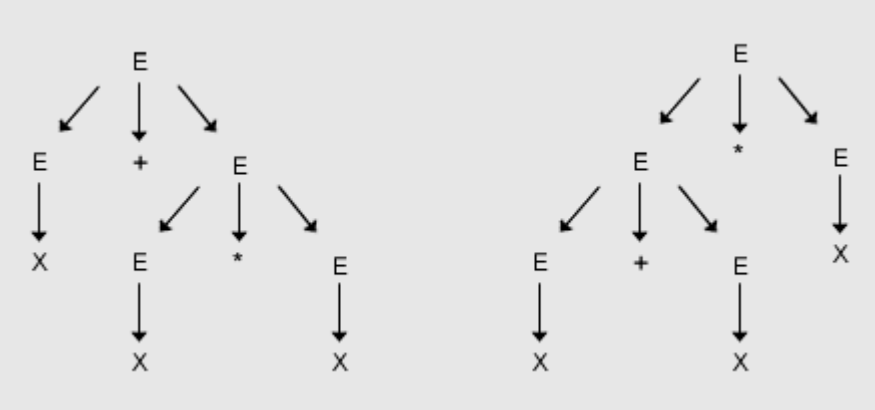

Figura 3.2: Árvores diferentes para uma mesma sentença (Menezes, 2000)

A existência de gramáticas ambíguas torna-se um problema quando os reconhecedores exigem derivações únicas para obter um bom desempenho, ou mesmo para concluir a análise sintática. Se o uso da gramática fosse limitado a determinar se uma seqüência de *tokens* pertence ou não a uma linguagem, a ambigüidade não seria tão problemática.

Em algumas linguagens de programação, parte do significado dos comandos está especificada em sua estrutura sintática, isto é, existe semântica embutida na estrutura do programa fonte. Nesses casos, a ambigüidade precisa ser eliminada.

Na Figura 3.2, as duas árvores de derivação correspondem a diferentes seqüências de avaliação para a expressão. Na árvore da direita, a operação + é realizada em primeiro lugar, e, na da esquerda, essa mesma operação é executada em segundo lugar, o que concorda com a regra padrão da aritmética. Para essa gramática, a ambigüidade pode ser eliminada fazendo com que a multiplicação tenha precedência sobre a soma. (Price, 2001).

De acordo com Lewis (2000), há linguagens livres do contexto com a propriedade de que todas as gramáticas livres do contexto que as geram devem ser ambíguas. Tais linguagens são chamadas *inerentemente ambíguas*. Felizmente, linguagens de programação nunca são inerentemente ambíguas.

Após apresentar os conceitos fundamentais a respeito das gramáticas livres do contexto, será dada continuidade ao trabalho apresentando as duas estratégias básicas para a análise sintática, que são Análise Descendente (*Top-Down*) e Redutiva (*Bottom-Up*) respectivamente.

3.2 Análise Descendente

A análise descendente de uma sentença (ou programa) pode ser vista como uma tentativa de construir uma árvore de derivação em pré-ordem (da esquerda para a direita) para a sentença em questão. A construção dessa árvore é feita da seguinte maneira: cria-se a raiz e, a seguir, as subárvores filhas, da esquerda para a direita. Esse processo produz uma derivação mais à esquerda da sentença em análise. (Price, 2001).

Existem três tipos de analisadores sintáticos descendentes que são:

- Recursivo com retrocesso (*backtracking*);
- Recursivo preditivo;
- Tabular preditivo.

Nos dois primeiros analisadores, cada símbolo não-terminal é implementado por um procedimento que efetua o reconhecimento do(s) lado(s) direito(s) das produções que definem o símbolo. O terceiro tipo, tabular preditivo, é implementado através de um *autômato de pilha*[4] controlado por uma tabela de análise, a qual indica a regra de produção a ser aplicada relativa ao símbolo não-terminal que está no topo da pilha. (Price, 2001).

A seguir serão apresentados os conceitos referentes aos três analisadores sintáticos mencionados anteriormente.

3.2.1 Análise Recursiva com Retrocesso

Esta análise faz a expansão da árvore de derivação a partir da raiz, expandindo sempre o não-terminal mais à esquerda. Quando existe mais de uma regra de produção para o não-terminal a ser expandido, a opção escolhida é a função do símbolo corrente na fita de entrada (*token* sob o cabeçote de leitura).

Se o *token* de entrada não define univocamente a produção a ser usada, então todas as alternativas vão ser tentadas até que se obtenha sucesso (ou até que a análise falhe irremediavelmente). (Price, 2001).

Através da sentença **[a]**, derivada a partir da gramática abaixo, é possível tomar a figura 3.3 como exemplo, para ilustrar o funcionamento de um analisador descendente com retrocesso.

$S \rightarrow a \mid [\ L\ ]$

$L \rightarrow S\ ;\ L \mid S$

[4] Um autômato de pilha é em essência um autômato finito não-determinístico com ε-transições permitidas e uma característica adicional: uma pilha na qual pode armazenar um *string* de "símbolos de pilha". (Hopcroft, 2002).

A análise descendente dessa sentença começa com o símbolo inicial S na raiz da árvore de derivação e com o cabeçote de leitura sobre o primeiro caractere da sentença. Pelo fato do primeiro caractere ser um colchete aberto, a primeira produção a ser aplicada será **S → [L]**. Tal produção é ilustrada na Figura 3.3.a.

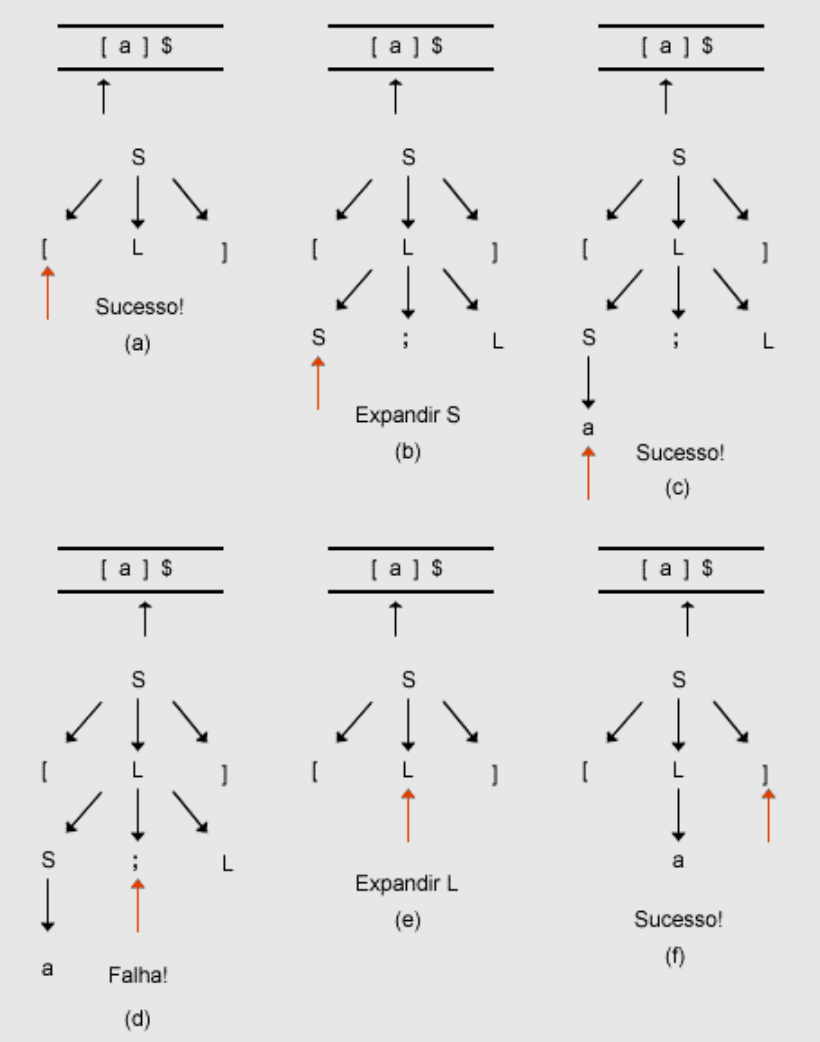

Figura 3.3: Reconhecimento da sentença [a] (Price, 2001)

De acordo com a figura 3.3.a o caractere **[** foi reconhecido com sucesso, prosseguindo a análise com a derivação do caractere **L**, que pode ser realizada utilizando um dos lados direitos (produções) alternativos: **S ; L** ou **S**. A Figura 3.3.b mostra que a alternativa escolhida foi **S ; L**, e posteriormente **S** foi expandido novamente obtendo sucesso como mostra a Figura 3.3.c.

Já na comparação seguinte houve falha ao comparar o caractere **]** com o **;**, resultando falha como mostra a Figura 3.3.d. Neste caso, o analisador tem como tarefa, retroceder na fita de entrada para o ponto em que estava quando foi escolhida a primeira alternativa de derivação de **L**, como mostra a Figura 3.3.e.

Em seguida, é aplicada a segunda alternativa de derivação para **L**, que é **L** $\rightarrow$ **S**. Com essa alternativa houve sucesso na derivação final, resultando na produção **S** $\rightarrow$ **a,** como mostra a Figura 3.3.f.

Segundo Price (2001) o processo de voltar atrás no reconhecimento e tentar produções alternativas dá-se o nome de retrocesso ou *backtracking*. Tal processo é ineficiente, pois leva à repetição da leitura de partes da sentença de entrada e, por isso, em geral, não é usado no reconhecimento de linguagens de programação.

Outra desvantagem que existe nessa classe de analisadores é que, quando um erro é identificado, fica difícil indicar com precisão o local do erro devido aos retrocessos.

Na Seção 3.2.2 será apresentado um analisador descendente que evita o retrocesso.

3.2.2 Análise Recursiva Preditiva

É possível construir um analisador recursivo sem retrocesso. Esses analisadores são chamados recursivos preditivos e, para eles, o principal problema durante a análise preditiva é determinar exatamente qual produção deve ser aplicada na expansão de cada não-terminal.

Segundo Price (2001) esses analisadores exigem que:

- a gramática não tenha recursividade à esquerda;
- a gramática esteja fatorada à esquerda;
- para os não-terminais com mais de uma regra de produção, os primeiros terminais deriváveis sejam capazes de identificar, univocamente, a produção que deve ser aplicada a cada instante da análise.

Para ilustrar o funcionamento de um analisador recursivo preditivo, segue abaixo a Figura 3.4 com o exemplo de produções que definem os comandos *if-then*, *while-do* e atribuição de valores.

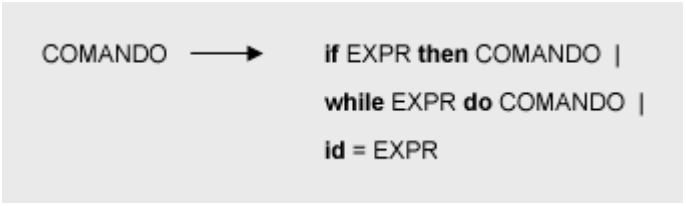

Figura 3.4: Produções dos comandos *if-then*, *while-do* e atribuição de valores

No exemplo mostrado acima, o analisador recursivo preditivo irá analisar as produções, e através dos primeiros terminais dos lados direitos (*if-then*, *while-do* e id), determinará univocamente a produção a ser aplicada para encontrar um determinado comando.

Na Seção 3.2.3 será apresentado um analisador preditivo não recursivo que utiliza explicitamente uma pilha ao invés de fazer chamadas recursivas.

3.2.3 Análise Preditiva Tabular

É possível construir analisadores preditivos *não recursivos* que utilizam uma pilha explícita ao invés de chamadas recursivas implícitas. Esse tipo de analisador implementa um autômato de pilha controlado por uma tabela de análise.

O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que se encontra no topo da pilha. O analisador busca a produção a ser aplicada na tabela de análise, levando em conta o não-terminal no topo da pilha e o *token* sob o cabeçote de leitura. (Price, 2001).

Um analisador preditivo tabular, é composto por uma entrada (fita de entrada), uma pilha e uma tabela sintática, conforme é ilustrado na Figura 3.5

A entrada contém a sentença que será analisada, tendo como último caractere o símbolo **$**, que significa fim de sentença.

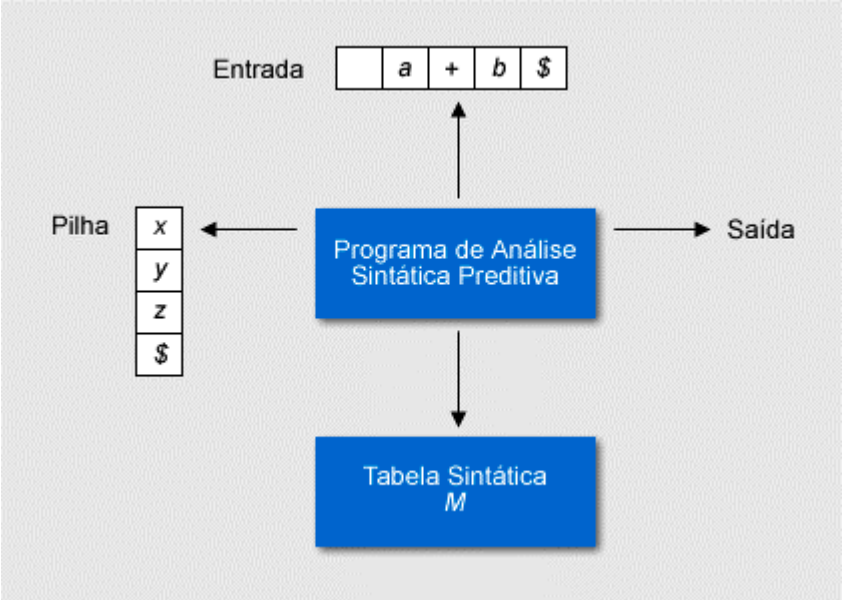

Figura 3.5: Modelo de um analisador sintático preditivo tabular (Aho, 1986)

De acordo com Price (2001), inicialmente a pilha contém o símbolo **$**, que marca a sua base, seguido do símbolo inicial da gramática. A tabela sintática é uma matriz **M** com **n** linhas e **t + 1** colunas, onde **n** é o número de símbolos não-terminais, e **t** é o número de símbolos terminais (a coluna extra corresponde ao símbolo **$**).

Segundo Aho (1996) o comportamento do analisador sintático pode ser descrito em termos de suas *configurações*:

- Se X = a = $, o analisador pára e anuncia o término com sucesso da análise sintática;
- Se X = a ≠ $, o analisador sintático remove X da pilha e avança o apontador da entrada para o próximo símbolo;
- Se X é um não-terminal, o programa consulta a entrada M[X, a] da tabela sintática M. Essa entrada será uma produção X da gramática ou uma entrada de erro.

Para exemplificar o que foi dito anteriormente, segue abaixo a Figura 3.6, que ilustra uma gramática não-ambígua que gera expressões.

Figura 3.6: Gramática não-ambígua (Price, 2001)

Após eliminar a recursividade à esquerda das produções que definem E e T, obtém-se o resultado ilustrado na Figura 3.7.

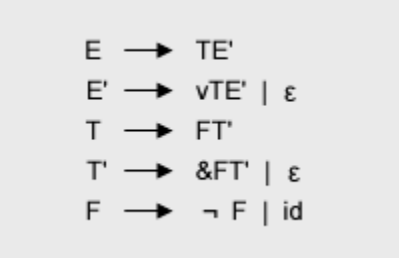

Figura 3.7: Eliminando recursividade à esquerda (Price, 2001)

Depois de apresentar os métodos existentes para a Estratégia Descendente (*Top-Down*), a Seção 3.3 irá discutir os conceitos pertinentes à outra estratégia básica da Análise Sintática que é a Estratégia Redutiva (*Bottom-Up*).

3.3 Análise Redutiva

A análise redutiva de uma sentença (ou programa) pode ser vista como a tentativa de construir uma árvore de derivação a partir de folhas, produzindo uma derivação mais à direita ao reverso.

A denominação redutiva refere-se ao processo que sofre a sentença de entrada, a qual é reduzida até ser atingido o símbolo inicial da gramática (raiz da árvore de derivação). Dá-se o nome de *redução* à operação de substituição do lado direito de uma produção pelo não-terminal correspondente (lado esquerdo). (Price, 2001).

Para melhor compreensão do parágrafo anterior, segue abaixo o exemplo de uma gramática e sua respectiva redução (Tabela 3.1 e 3.2). Este exemplo foi retirado de Aho (1996).

Tabela 3.2: Gramática de exemplo

Gramática
S → aABe
A → Abc | b
B → d

Continuando o exemplo, a sentença *abbcde* pode ser reduzida a *S* pelos passos indicados na tabela 3.3.

A sentença *abbcde* ilustrada no exemplo anterior, procura por uma subcadeia que reconheça o lado direito de alguma produção. As subcadeias *b* e *d* se encaixam nessa condição. Foi escolhido o caractere *b* da sentença mais à esquerda, que logo em seguida foi substituído por *A*, o lado esquerdo da produção $A \rightarrow b$.

Tabela 3.3: Redução da sentença *abbcde*

Sentença	Descrição dos Passos
Abbcde	-
aAbcde	A → b
aAde	A → Abc
aABe	B → d
S	S → aABe

Dessa forma foi obtida a nova cadeia *aAbcde*. Agora as subcadeias *Abc*, *b* e *d* reconhecem o lado direito de alguma produção. Apesar de *b* ser a subcadeia mais à esquerda que reconheça o lado direito de alguma produção, foi escolhida a subcadeia *Abc* para ser substituída por *A*, o lado esquerdo da produção $A \rightarrow Abc$.

Agora a nova cadeia obtida foi *aAde*. Com a substituição da subcadeia *d* por *B*, o lado esquerdo da produção $B \rightarrow d$, obteve-se a cadeia *aABe*. Por último, foi substituída toda a cadeia por *S*, lado esquerdo da produção $S \rightarrow aABe$.

3.3.1 Definição de *Handles*

O analisador redutivo empilha símbolos da sentença de entrada até ter na pilha uma seqüência de símbolos que corresponde à definição de algum não-terminal. Informalmente, é essa seqüência de símbolos (que corresponde ao lado direito de produção) que define o *handle*. (Price, 2001).

Em muitos casos, a subcadeia β mais à esquerda que reconhece o lado direito de uma produção $A \rightarrow \beta$ não é um *handle*, porque uma redução pela produção $A \rightarrow \beta$ produz uma cadeia que não pode ser reduzida ao símbolo de partida. (Aho, 1996).

No processo de análise, os *handles* são as seqüências de símbolos que são lados direitos de produção, tais que suas reduções levam, no final, à redução

para o símbolo inicial da gramática, através do reverso de uma derivação mais à direita. Se uma gramática G é não-ambígua, então toda forma sentencial gerada por G tem exatamente um *handle*. (Price, 2001).

Segundo Aho (1996) existem efetivamente quatro ações possíveis para um reconhecedor empilha-reduz (analisador) realizar:

- Numa ação de *empilhar*, o próximo símbolo de entrada é colocado no topo da pilha;
- Numa ação de *reduzir*, o analisador sabe que o final à direita de um *handle* está no topo da pilha. Precisa, então, localizar o início à esquerda do *handle* dentro da pilha e decidir qual não-terminal irá substituir o *handle*;
- Numa ação de *aceitar*, o analisador anuncia o término com sucesso da operação de decomposição;
- Numa ação de *erro*, o analisador descobre que um erro sintático ocorreu e chama uma rotina de recuperação de erros.

Como dito em Price (2001), existem duas classes de analisadores do tipo empilha-reduz que são:

- *Analisadores de precedência de operadores*, muito eficientes no reconhecimento de expressões aritméticas e lógicas;
- *Analisadores LR*, que reconhecem a maior parte das linguagens livres do contexto.

3.3.2 Analisadores de Precedência de Operadores

Esses analisadores operam sobre a classe das *gramáticas de operadores*. Nessas gramáticas, os não-terminais aparecem sempre separados por símbolos terminais (isto é, nunca aparecem dois não-terminais adjacentes) e, além disso, as produções não derivam a palavra vazia (nenhum lado direito de produção é igual a "ε"). (Price, 2001).

Para ilustrar as propriedades mencionadas no parágrafo anterior, segue como exemplo a gramática:

E → E B E | (E) | id

B → + | - | * | /

A gramática acima não pertence à classe das gramáticas de operadores, pelo fato do lado direito EBE representar três não-terminais adjacentes.

Por outro lado, se for substituído o "B" por uma de suas produções, a gramática passa a ser de operadores.

A análise de precedência de operadores é bastante eficiente e é aplicada, principalmente, no reconhecimento de expressões. (Price, 2001).

Porém, como uma técnica geral de análise sintática, a de precedência de operadores possui uma série de desvantagens. Por exemplo, é difícil tratar os *tokens* como o sinal de menos, que possui duas diferentes precedências (dependendo de ser unário ou binário). (Aho, 1996).

Para identificar o *handle*, os analisadores de precedência de operadores baseiam-se em relações de precedência existentes entre os *tokens* (operandos e operadores). (Price, 2001).

Segundo Aho (1996) na análise sintática de precedência de operadores, são três as relações de precedências entre os terminais: <, > e =.

Essas relações de precedência têm como utilidade guiar a seleção de *handles* e são definidas na Tabela 3.3 exibida logo abaixo.

Tabela 3.4: Relações de precedência entre terminais (Aho, 1996)

Relação	**Significado**
a < b	a "confere precedência" a b
a = b	a "possui a mesma precedência que" b
a > b	a "tem precedência sobre" b

Abaixo seguem uma gramática que gera expressões lógicas (Figura 3.8) e sua respectiva tabela de precedência de operadores (Tabela 3.5).

Figura 3.8: Gramática que gera expressões lógicas (Price, 2001)

Tabela 3.5: Tabela de precedência para expressões simplificadas (Price, 2001)

	id	**v**	**&**	**(**	**)**	**$**
id		>	>		>	>
v	<	>	<	<	>	>
&	<	>	>	<	>	>
(	<	<	<	<	=	
)		>	>		>	>
$	<	<	<	<		

É importante saber que, na Tabela 3.5, os terminais nas linhas representam terminais no topo da pilha, e os terminais nas colunas representam terminais sob o apontador da entrada.

A Seção 3.3.3 irá apresentar a outra classe de analisadores do tipo empilha-reduz que é a de Analisadores LR.

3.3.3 Analisadores LR

Esta seção apresenta uma técnica eficiente de análise sintática *Bottom-Up*, que pode ser utilizada para decompor uma ampla classe de gramáticas livres do contexto.

A técnica é chamada análise sintática LR (k); o "L" significa varredura da entrada da esquerda para a direita (*left-to-right*), o "R", construir uma derivação mais à direita ao contrário (*rightmost derivation*) e o *k*, o número de símbolos de entrada de *lookahead* que são usados ao se tomar decisões na análise sintática. (Aho, 1996).

De acordo com Price (2001), dentre as vantagens identificadas nesses analisadores, destacam-se:

- São capazes de reconhecer, praticamente, todas as estruturas sintáticas definidas por gramáticas livres do contexto;
- O método de reconhecimento LR é mais geral que o de precedência de operadores e que qualquer outro do tipo empilha-reduz e pode ser implementado com o mesmo grau de eficiência;
- Analisadores LR são capazes de descobrir erros sintáticos no momento mais cedo, isto é, já na leitura da sentença em análise.

A principal desvantagem deste método está em ser muito trabalhoso construir um analisador sintático LR manualmente para uma gramática típica de linguagem de programação.

Necessita-se em geral de uma ferramenta especializada – um gerador de analisadores. (Aho, 1996).

Atualmente existem muitos desses geradores disponíveis, por exemplo, o YACC.

Como consta em Price (2001), há basicamente, três tipos de analisadores LR:

- SLR (*Simple* LR), fáceis de implementar, porém aplicáveis a uma classe restrita de gramáticas;
- LR Canônicos, mais poderosos, podendo ser aplicados a um grande número de linguagens livres do contexto;
- LALR (*Look Ahead* LR), de nível intermediário e implementação eficiente, que funciona para a maioria das linguagens de programação. O YACC gera esse tipo de analisador.

YACC é uma parte de *software* de computador que serve como o gerador padrão do *parser* em sistemas operacionais Unix. O nome é um acrônimo para "contudo um outro compilador do compilador".

Gera um *parser* (parte de um compilador que tenta fazer o sentido da entrada) baseado em uma gramática escrita na notação BNF. YACC gera o código para o *parser* na linguagem de programação C. (Johnson, 2005).

A estrutura de um analisador sintático LR é mostrada na Figura 3.9. Ele é composto por uma pilha, uma entrada, uma saída, um programa principal (analisador) e uma tabela sintática que possui duas partes.

A parte da Ação, que contém as ações associadas às transições de estados (empilhar, reduzir, aceitar ou condição de erro); e a parte Transição que contém transições de estados com relação aos símbolos não-terminais.

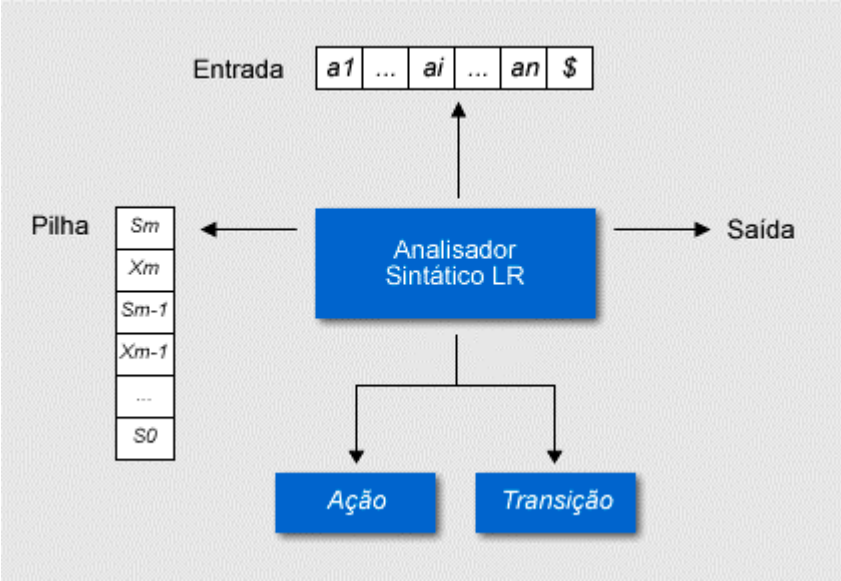

Figura 3.9: Estrutura de um analisador sintático LR (Aho, 1996)

3.4 Conclusão

Esse capítulo se destinou a tratar de conceitos referentes à análise sintática. Como foi visto, a análise sintática constitui a segunda fase de um compilador.

No decorrer do capítulo, além de outros temas, foram estudadas as duas estratégias básicas da análise sintática, que são a estratégia Descendente (*Top-Down*) e a estratégia Redutiva (*Bottom-Up*).

As técnicas de análise sintática, baseadas na estratégia descendente, constroem a árvore de derivação a partir do símbolo inicial da gramática (raiz da árvore), até atingir suas folhas.

Por outro lado, a estratégia Redutiva executa a análise no sentido inverso, isto é, a partir das folhas de derivação constrói a árvore até o símbolo inicial da gramática.

No Capítulo 4, será discutida a terceira etapa de um compilador, que se trata da análise semântica.

4 - Análise Semântica

Embora a análise sintática consiga verificar se uma expressão obedece às regras de formação de uma dada gramática, seria muito difícil expressar através de gramáticas algumas regras usuais em linguagens de programação.

Por exemplo, como "todas as variáveis devem ser declaradas" e situações onde o contexto em que ocorre a expressão ou o tipo da variável deve ser verificado.

O objetivo da análise semântica é trabalhar nesse nível de inter-relacionamento entre as partes distintas do programa.

As tarefas básicas desempenhadas durante a análise semântica incluem a *verificação de tipos*, a *verificação do fluxo de controle* e a *verificação da unicidade da declaração de variáveis e funções*. (Ricarte, 2003).

Abaixo segue a Figura 4.1 que ilustra um exemplo típico de erro semântico.

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

Figura 4.1: Exemplo de erro semântico na verificação de tipos (Ricarte, 2003)

O trecho de código mostrado na Figura 4.1, mostra um erro detectado pelo analisador semântico, devido às regras de verificação de tipos, que indica que o operador % (módulo) não pode ter um operando do tipo real.

Para tornar as ações semânticas mais efetivas, pode-se associar variáveis aos símbolos (terminais e não-terminais) da gramática. Assim, os símbolos

gramaticais passam a conter atributos (ou parâmetros) capazes de armazenar valores durante o processo de reconhecimento.

Toda vez que uma regra de produção é usada no processo de reconhecimento de uma sentença, os símbolos gramaticais dessa regra são "alocados" juntamente com seus atributos.

Isto é, cada referência a um símbolo gramatical, numa regra, faz com que uma cópia desse símbolo seja criada, juntamente com seus atributos. Analogamente a árvore de derivação da sentença sob análise, é como se a cada nó da árvore (símbolo gramatical) correspondesse a uma instanciação de um símbolo e de suas variáveis. (Price, 2001).

A Figura 4.2 mostra o exemplo de produções utilizadas para especificar uma declaração simples e as ações semânticas associadas.

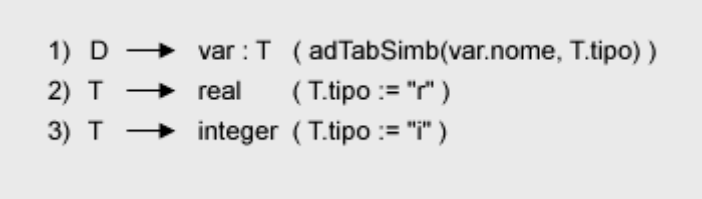

Figura 4.2: Exemplo de produções para declaração simples (Price, 2001)

Segundo Price (2001), durante o processo de reconhecimento, quando a produção (2) ou (3) é reduzida, o atributo *tipo* associado ao não-terminal **T** recebe o valor do tipo reconhecido (caractere "r" ou "i").

A redução da produção (1) adiciona na tabela de símbolos o nome da variável e seu respectivo tipo. O atributo *nome* de var é inicializado quando esse terminal é reconhecido durante o processo de análise léxica.

4.1 Tradução Dirigida pela Sintaxe

Um *Esquema de Tradução Dirigida pela Sintaxe* é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, particionados em dois subconjuntos, chamados de *atributos sintetizados* e *atributos herdados* daquele símbolo gramatical. (Aho, 1996).

Um atributo de um símbolo pode conter um valor numérico, uma cadeia de caracteres, um tipo de dado, um endereço de memória, etc.

Segundo Price (2001), num esquema de tradução, para cada produção $A \rightarrow \alpha$, podem existir várias regras semânticas da forma b := f(c1, c2, ..., ck), onde f é uma função, e b, c1, c2, ..., ck são atributos. Diz-se que:

- b é um *atributo sintetizado* se ele é um atributo de A (símbolo do lado esquerdo da produção) e c1, c2, ..., ck são atributos associados aos símbolos do lado direito da produção (filhos de A);
- b é um *atributo herdado* se ele é atributo de um dos símbolos do lado direito da produção. Nesse caso, c1, c2, ..., ck podem pertencer a quaisquer dos demais símbolos da produção (pai ou irmãos).

4.2 Gramática de Atributos

Uma *gramática de atributos* é uma definição dirigida pela sintaxe, na qual as funções nas regras semânticas não têm efeitos colaterais[5], ou seja, as ações semânticas são atribuições ou funções envolvendo apenas os atributos do esquema.

[5] Uma função, procedimento ou operação é dita produzir efeitos colaterais (*side effects*) quando altera um ou mais de seus parâmetros ou modifica uma variável global. (Pratt, 1984).

A seguir segue a Tabela 4.1 como exemplo, para especificar um interpretador de uma calculadora, o qual reconhece uma expressão aritmética e imprime seu valor. A expressão é formada por dígitos decimais, parênteses, operadores aritméticos "+" e "*", e também o *token* "=", por exemplo 5*(2+3)=.

Tabela 4.1: Esquema de tradução para uma calculadora aritmética (Price, 2001)

Produções	**Regras Semânticas**
L → E =	{ print (E.val) }
E → E1 + T	{ E.val := E1.val + T.val }
E → T	{ E.val := T.val }
T → T1 * F	{ T.val := T1.val * F.val }
T → F	{ T.val := F.val }
F → (E)	{ F.val := E.val }
F → digit	{ F.val := digit.lexval }

As regras semânticas mostradas na Tabela 4.1, associam um atributo sintetizado chamado *val* a cada símbolo não-terminal E, T, e F. O símbolo terminal *digit* tem um atributo sintetizado *lexval* cujo valor é atribuído pelo analisador léxico. A regra semântica associada à produção "L → E =" é uma chamada a uma função que imprime o valor da expressão aritmética reconhecida.

De acordo com a definição, este exemplo deixa de ser uma gramática de atributos, pois na primeira regra de produção, a chamada feita à função *print(E.val)* produz efeitos colaterais.

A Figura 4.3 mostra a árvore de derivação correspondente à seqüência de *tokens* **2*3+4=**. O resultado que é impresso, no momento da redução para o símbolo inicial da gramática, é o valor *E.val* do filho à esquerda da raiz.

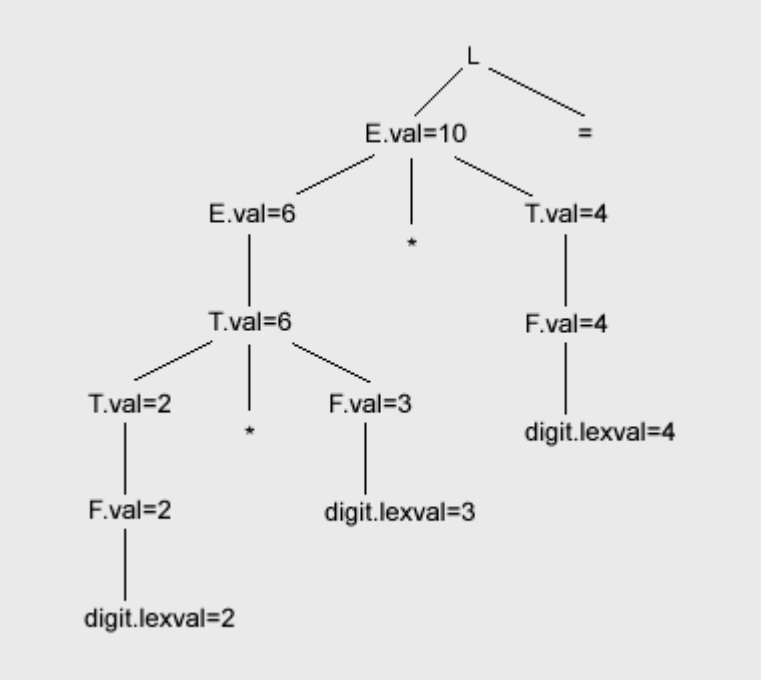

Figura 4.3: Árvore de derivação para a expressão 2*3+4= (Price, 2001)

Conforme Price (2001), num esquema de tradução, os símbolos terminais possuem apenas atributos sintetizados, os quais são calculados pelo analisador léxico. No outro extremo da árvore (na raiz da árvore), ocorre algo semelhante, pois o símbolo inicial da gramática não pode ter atributos herdados (ele não tem de quem herdar).

4.3 Geração de Código Intermediário

A tradução do código de alto nível para o código do processador está associada a traduzir para a linguagem-alvo a representação da árvore gramatical obtida para as diversas expressões do programa.

Embora tal atividade possa ser realizada para a árvore completa, após a conclusão da análise sintática, em geral ela é efetivada através das ações semânticas associadas à aplicação das regras de reconhecimento do analisador sintático. (Ricarte, 2003).

Essa geração de código pode, eventualmente, ser o código objeto final, mas, na maioria das vezes, constitui-se num código intermediário, pois a tradução de código fonte para código objeto final em mais de um passo apresenta algumas vantagens (Price, 2001):

- Possibilita a otimização do código intermediário, de modo a obter-se o código objeto final mais eficiente;
- Simplifica a implementação do compilador, resolvendo, gradativamente, as dificuldades da passagem de código fonte para objeto (alto-nível para baixo-nível), já que o código fonte pode ser visto como um contexto condensado que "explode" em inúmeras instruções elementares de baixo nível;
- Possibilita a tradução do código intermediário para diversas máquinas.

A Figura 4.4 mostra a posição do gerador de código intermediário.

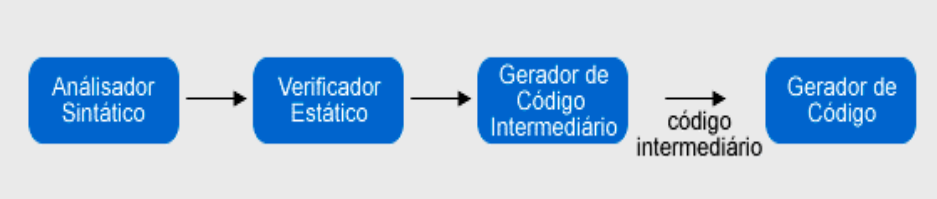

Figura 4.4: Posição do gerador de código intermediário (Aho, 1996)

A desvantagem de gerar código intermediário é que o compilador requer um passo a mais. A tradução direta do código fonte para objeto leva a uma compilação mais rápida.

A grande diferença entre o código intermediário e o código objeto final é que o intermediário não especifica detalhes da máquina alvo, tais como quais registradores serão usados, quais endereços de memória serão referenciados, etc. (Price, 2001).

Dessa forma, quando se utiliza um compilador que gera código intermediário, é possível desenvolver programas para diversas arquiteturas de máquinas, bastando gerar o código intermediário, e através deste, terminar a tradução para o código objeto final na máquina de destino.

No Capítulo 5, serão apresentados os conceitos que envolvem a Teoria de Linguagens Formais e Autômatos.

5 - Linguagens Formais e Autômatos

"A Teoria de Linguagens Formais e Autômatos foi originalmente desenvolvida na década de 1950, com o objetivo de desenvolver teorias relacionadas com as linguagens naturais" (Menezes, 2000).

"Entretanto, logo foi verificado que esta teoria era importante para o estudo de linguagens artificiais e, em especial, para as linguagens originárias na Ciência da Computação" (Menezes, 2000).

A partir de então, o estudo das Linguagens Formais evoluiu significamente e com diversos enfoques, com destaque para aplicações em Análise Léxica, Sintática e Semântica de linguagens de programação.

A seguir serão abordados conceitos sobre os autômatos finitos, os quais foram a base fundamental para o desenvolvimento desse trabalho.

5.1 Autômato Finito

Segundo Menezes (2000), um Autômato Finito Determinístico ou simplesmente Autômato Finito pode ser visto como uma máquina composta, basicamente, de três partes:

- *Fita*. Dispositivo de entrada que contém a informação a ser processada;
- *Unidade de Controle*. Reflete o estado corrente da máquina. Possui uma unidade de leitura (cabeça da fita) a qual acessa uma célula da fita de cada vez e movimenta-se exclusivamente para a direita;
- *Programa* ou *Função de Transição*. Função responsável por comandar as leituras e definir o estado da máquina.

> "A fita é finita (à esquerda e à direita), sendo dividida em células, onde cada uma armazena um símbolo. Os símbolos pertencem a um alfabeto de entrada. Não é possível gravar sobre a fita (e não existe memória auxiliar).
> A unidade de controle possui um número finito e predefinido de estados. A unidade de leitura lê o símbolo de uma célula de cada vez. Após a leitura, a *cabeça da fita* move-se uma célula para a direita. Inicialmente, a cabeça está posicionada na célula mais à esquerda da fita, como ilustrado na Figura 5.1" (Menezes, 2000).

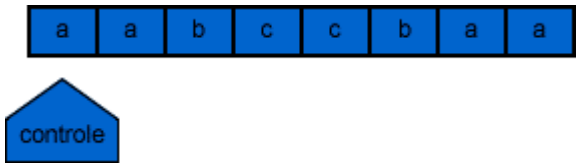

Figura 5.1: Autômato finito como uma máquina com controle finito (Menezes, 2000)

> "O programa é uma função parcial que, dependendo do estado corrente e do símbolo lido, determina o novo estado do autômato. Deve-se reparar que o Autômato Finito não possui memória de trabalho. Portanto, para armazenar as informações passadas necessárias ao processamento, deve-se usar o conceito de estado" (Menezes, 2000).

De acordo com Price (2001) um autômato finito M sobre um alfabeto $\sum$ é uma quíntupla (K, $\sum$, δ, e0, F), onde:

- K é um conjunto finito de estados;
- $\sum$ é o alfabeto dos símbolos da linguagem;
- $\delta : K \times \sum \rightarrow K$ é a função de transição dos estados;
- e0 é o estado inicial;
- F é o conjunto de estados finais.

"O processamento de um autômato finito M, para uma palavra de entrada w, consiste na sucessiva aplicação da função programa para cada símbolo de w (da esquerda para a direita) até ocorrer uma condição de parada" (Menezes, 2000).

Diz-se que uma sentença é aceita por um autômato (pertence à linguagem que ele representa) se, após seu processamento, o autômato pára em um estado final. Se, durante o processamento da sentença, ocorre uma situação de indefinição, o autômato pára e a sentença não é aceita. (Price, 2001).

A seguir serão apresentados os autômatos finitos desenvolvidos para representar a estrutura gramatical da linguagem de programação C.

5.2 Autômatos Finitos do Pré-Compilador ANSI C

O objetivo deste trabalho é o desenvolvimento de uma ferramenta de auxilio ao ensino da linguagem C. Não se pretende desenvolver um novo compilador para esta linguagem, entretanto, deve-se destacar as limitações do trabalho proposto:

- O pré-compilador será utilizado para analisar programas contidos em um único arquivo;
- O pré-compilador identificará apenas alguns erros, os mais comuns, que ocorrem frequentemente com programadores iniciantes;
- Embora as instruções em linguagem C possam ser complexas, o pré-compilador não as analisará completamente, verificando apenas sua estrutura básica, como por exemplo, verificar a presença de pares de parênteses no comando *if*, mas não a condição de decisão.

Assim, o pré-compilador ANSI-C em questão, terá como base, um conjunto de autômatos finitos desenvolvidos para representar gramaticalmente partes da linguagem C.

Como dito anteriormente, o motivo de representar partes da linguagem C, é o fato deste trabalho ter como foco a apresentação de um pré-compilador, que irá tratar alguns tipos de erros comumente encontrados nesta linguagem, e não todos os possíveis erros.

As partes da linguagem C que serão tratadas pelo pré-compilador são as seguintes:

- Diretivas do Pré-Processador: *include* e *define*;
- Tipos de Variáveis: declaração e inicialização de variáveis;
- Estruturas de Dados: declaração de estruturas;
- Funções: declaração de funções e protótipos;
- Estruturas de Controle de Fluxo: *if-else*, *switch*, *for*, *while* e *do-while*;
- Comentários: definição de comentários.

5.2.1 Diretivas do Pré-Processador

Pode-se incluir diversas instruções do compilador no código-fonte de um programa em C. Elas são chamadas de diretivas do pré-processador e, embora não sejam realmente parte da linguagem de programação C, expandem o escopo do ambiente de programação em C. (Schildt, 1996).

5.2.1.1 Diretiva #include

A diretiva #include instrui o compilador a ler outro arquivo-fonte adicionado àquele que contém a diretiva #include. O nome do arquivo adicional deve estar entre aspas ou símbolos de maior e menor.

Formalmente, a sintaxe é:

#include "biblioteca"
#include <biblioteca>

Ambas instruem o compilador a ler e compilar o arquivo de cabeçalho para as rotinas de arquivos em disco da biblioteca. (Schildt, 1996).

Abaixo é mostrado o autômato que representa a diretiva #include.

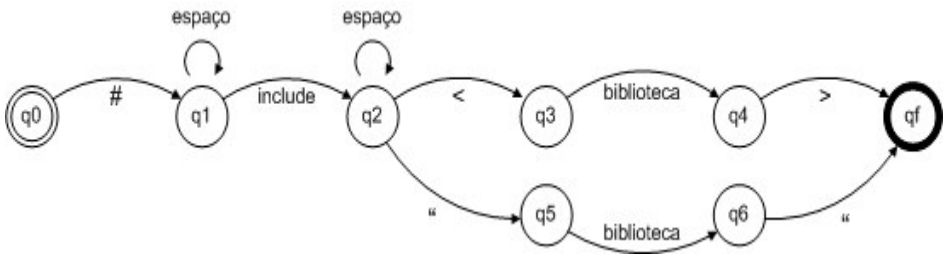

Figura 5.2: Módulo include

5.2.1.2 Diretiva #define

A diretiva #define define um identificador e uma *string* que o substituirá toda vez que for encontrado no arquivo-fonte. O padrão ANSI C refere-se ao identificador como um nome de macro e ao processo de substituição como substituição de macro. (Schildt, 1996).

Formalmente, a sintaxe é:

#define nome_macro string

É importante destacar que não há nenhum ponto-e-vírgula nesse comando. Pode haver qualquer número de espaços entre o identificador e a *string*, mas, assim que a *string* começar, será terminada apenas por uma nova linha. (Schildt, 1996).

Abaixo é mostrado o autômato que representa a diretiva #define.

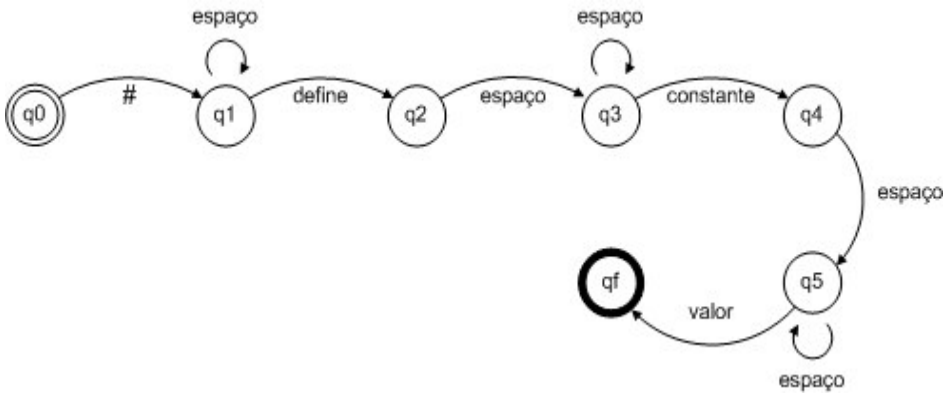

Figura 5.3: Módulo define

5.2.2 Tipos de Variáveis

Uma variável é uma posição nomeada de memória, que é usada para guardar um valor que pode ser modificado pelo programa. Todas as variáveis em C devem ser declaradas antes de serem usadas. (Schildt, 1996).

Formalmente, a sintaxe é:

tipo lista_de_variáveis;

No exemplo acima, *tipo* deve ser um tipo de dado válido em C mais quaisquer modificadores; e *lista_de_variáveis* pode consistir em um ou mais nomes de identificadores separados por vírgula.

Um identificador não pode ser igual a uma palavra-chave de C e não deve ter o mesmo nome que as funções escritas no programa ou as que estão nas bibliotecas ANSI C. (Schildt, 1996).

Abaixo é mostrado o autômato que representa as variáveis da linguagem C.

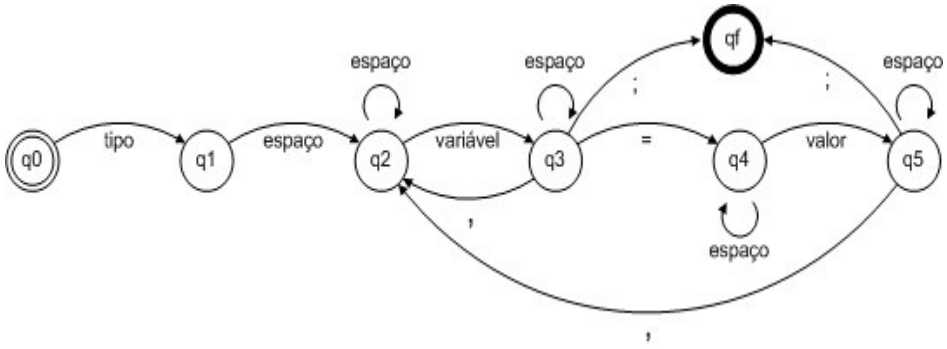

Figura 5.4: Módulo variáveis

5.2.3 Estruturas de Dados

Em C, uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas.

Uma definição de estrutura forma um modelo que pode ser usado para criar variáveis de estruturas. As variáveis que compreendem a estrutura são chamadas membros da estrutura. (Schildt, 1996).

Formalmente, a sintaxe é:

```
struct identificador {
   tipo nome_da_variável;
   tipo nome_da_variável;
   tipo nome_da_variável;
} variáveis_estrutura;
```

O *identificador* ou *variáveis_estrutura* podem ser omitidos, mas não ambos. (Schildt, 1996).

Abaixo é mostrado o autômato que representa a estrutura de dados struct da linguagem C.

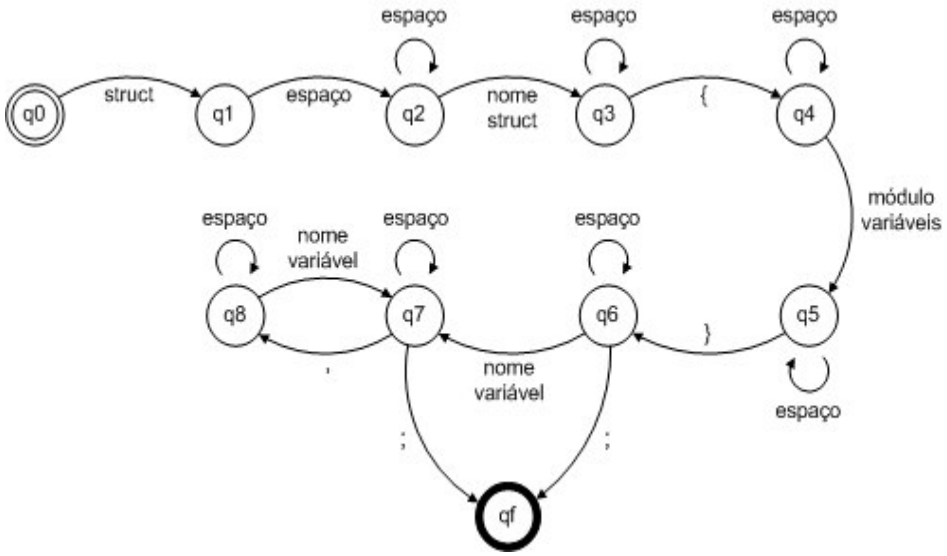

Figura 5.5: Módulo struct

5.2.4 Funções e Protótipos de Funções

Funções são os blocos de construção de C e o local onde toda a atividade do programa ocorre. Elas representam uma das características mais importantes de C. (Schildt, 1996).

Formalmente, a sintaxe é:

```
especificador_de_tipo nome_da_função (lista_de_parâmetros) {
  corpo_da_função
}
```

O *especificador_de_tipo* especifica o tipo de valor que o comando *return* da função devolve, podendo ser qualquer tipo válido.

Se nenhum tipo é especificado, o compilador assume que a função devolve um resultado inteiro. A *lista_de_parâmetros* é uma lista de nomes de variáveis separadas por vírgulas e seus tipos associados que recebem os valores dos argumentos quando a função é chamada.

Uma função pode não ter parâmetros, neste caso a *lista_de_parâmetros* é vazia. No entanto, mesmo que não existam parâmetros, os parênteses ainda são necessários. (Schildt, 1996).

A Figura 5.6 mostra o autômato que representa a declaração de funções da linguagem C.

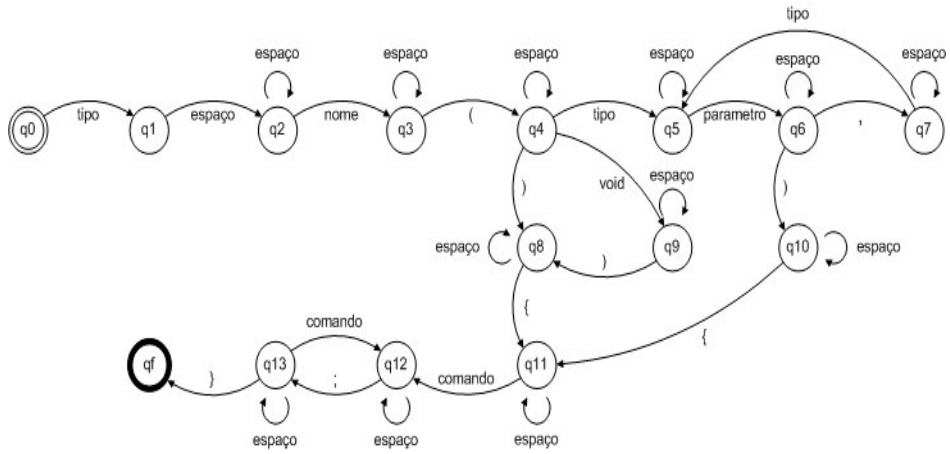

Figura 5.6: Módulo function

As declarações de protótipos de funções são em parte, exatamente iguais às de funções, a única diferença é que nos protótipos a declaração termina com o ponto e vírgula, e não é seguida por chaves aberta.

Formalmente, a sintaxe é:

especificador_de_tipo nome_da_função (lista_de_parâmetros);

Abaixo é mostrado o autômato que representa a declaração de protótipos de funções da linguagem C.

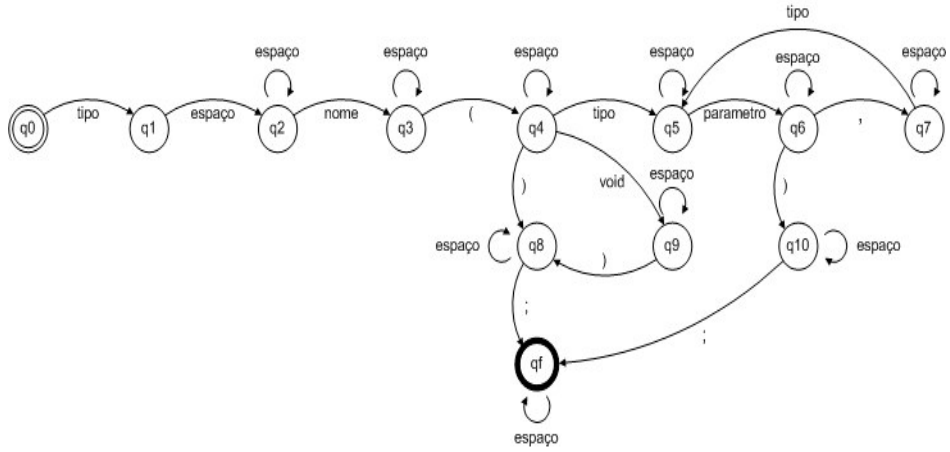

Figura 5.7: Módulo protótipo

5.2.5 Estruturas de Controle de Fluxo

Os comandos de fluxo de controle de uma linguagem especificam a ordem em que a computação é feita, ou seja, gerenciam o fluxo de execução de um programa.

Entre eles podem ser citados: if-else, switch, for, while e do-while.

5.2.5.1 If-Else

O comando if-else é utilizado para expressar decisões. Formalmente, a sintaxe é:

```
if (expressão)
  comando1;
else
  comando2;
```

No exemplo acima a parte do *else* é opcional. A *expressão* é avaliada; se for verdadeira (isto é, se *expressão* tiver um valor diferente de zero), *comando1* é executado. Se for falsa (*expressão* é zero) e se houver uma parte *else*, *comando2* é executado. (Kernighan, 1988).

Abaixo é mostrado o autômato que representa a estrutura de controle if-else da linguagem C.

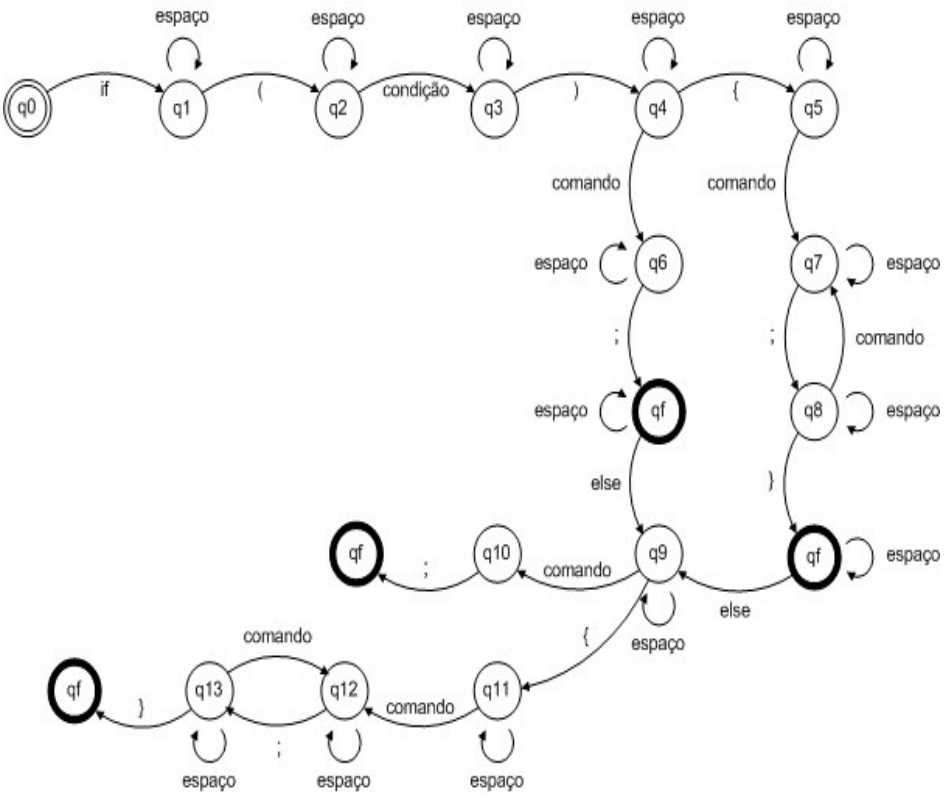

Figura 5.8: Módulo if-else

5.2.5.2 Switch

O comando switch é uma estrutura de decisão múltipla que testa se uma expressão casa um de vários valores inteiros *constantes*, e desvia de acordo com o resultado. (Kernighan, 1988).

Formalmente, a sintaxe é:

```
switch (expressão) {
  case expr-const : comando1;
  case expr-const : comando2;
  default : comando3;
}
```

Cada caso é rotulado por uma ou mais constantes de valor inteiro ou expressões constantes. Se um caso combina com o valor da expressão, a execução inicia nesse caso.

Todas as expressões de caso devem ser diferentes umas das outras. O caso intitulado default é executado se nenhum outro for satisfeito. Um default é opcional; se não estiver presente e nenhum dos casos combinarem com a expressão, nenhuma ação é tomada. (Kernighan, 1988).

Abaixo é mostrado o autômato que representa a estrutura de controle switch da linguagem C.

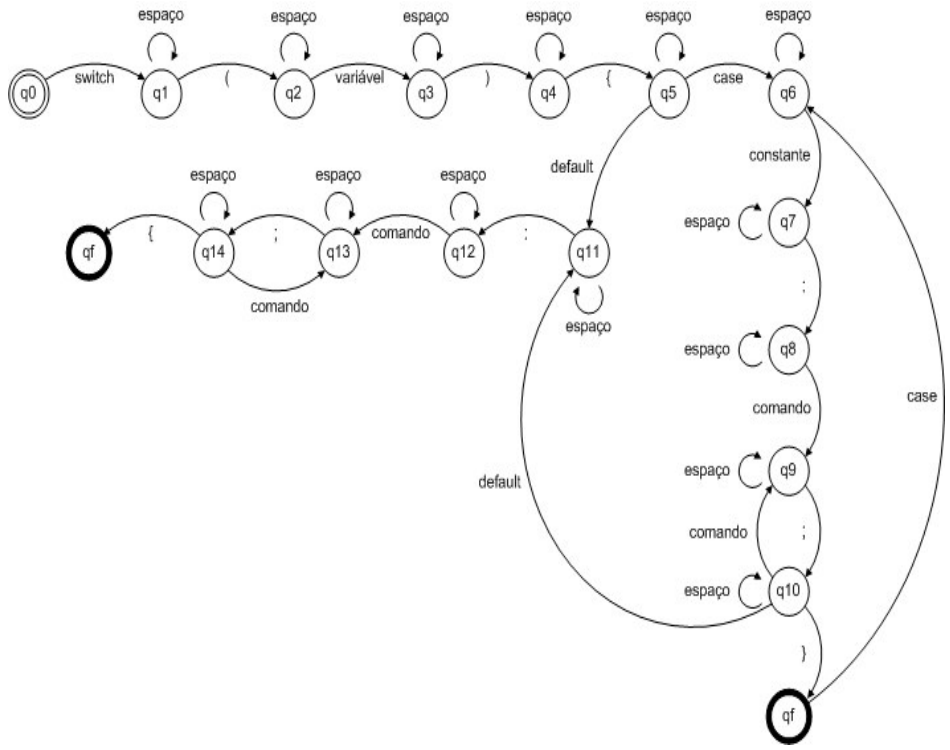

Figura 5.9: Módulo switch

5.2.5.3 For

O comando for é normalmente utilizado para executar repetidamente um conjunto de comandos por um número pré-determinado de vezes.

Formalmente, a sintaxe é:

for (expr1; expr2; expr3)
comando;

Gramaticalmente, os três componentes de um laço for são expressões. Normalmente, *expr1 e expr3* são atribuições ou chamadas de função e *expr2* é uma expressão relacional.

Qualquer uma das três pode ser omitida, embora os ponto-e-vírgulas devam permanecer. Se *expr1* ou *expr3* forem omitidas, ela é simplesmente desconsiderada. Se o teste *expr2*, não está presente, é considerada permanentemente verdadeira de forma que é um laço "infinito". (Kernighan, 1988).

Abaixo é mostrado o autômato que representa a estrutura de controle for da linguagem C.

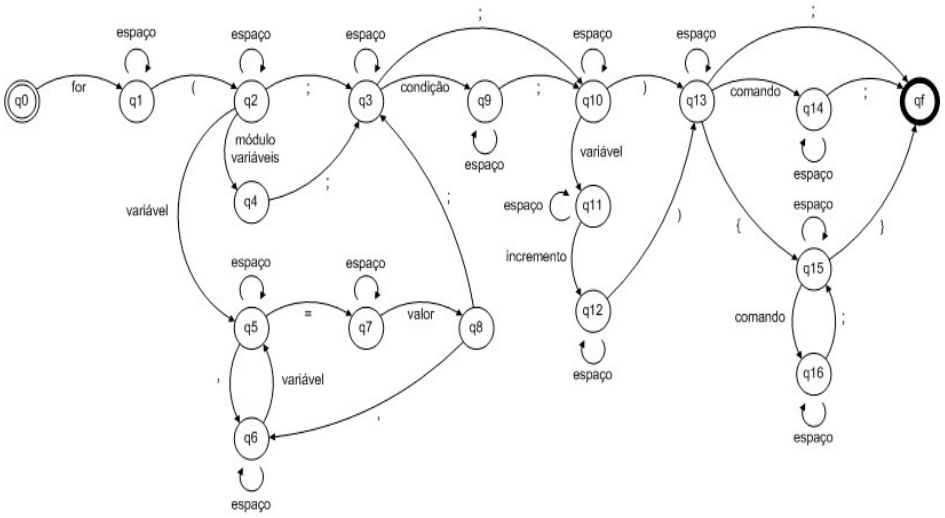

Figura 5.10: Módulo for

5.2.5.4 While

O comando while também pode ser utilizado para executar repetidamente um conjunto de comandos.

Formalmente, a sintaxe é

while (expressão)
  comando;

No exemplo acima, a *expressão* é avaliada. Se for diferente de zero, *comando* é executado e *expressão* é reavaliada. Este ciclo continua até *expressão* se tornar zero, continuando então a execução após *comando*. (Kernighan, 1988).

Abaixo é mostrado o autômato que representa a estrutura de controle while da linguagem C.

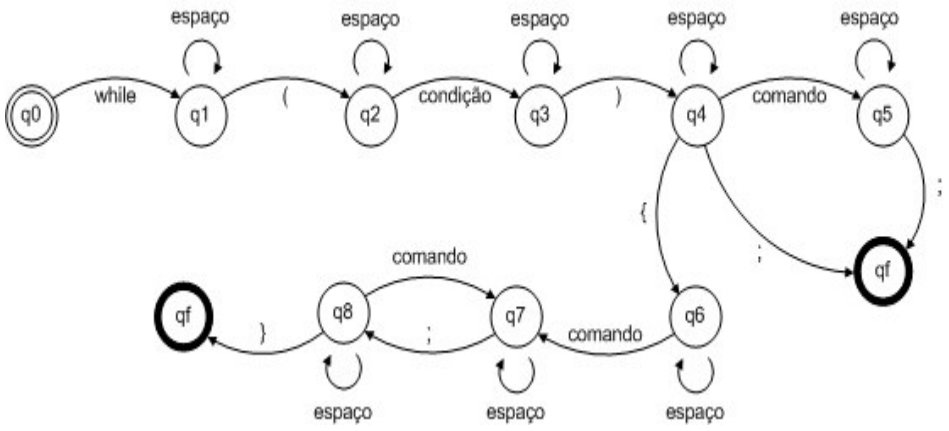

Figura 5.11: Módulo while

5.2.5.5 Do-While

Como foi dito anteriormente, os laços de repetição while e for testam a condição de término no início. Ao contrário, o *loop* do-while, testa ao final, depois de fazer cada passagem pelo corpo do laço; o corpo é sempre executado pelo menos uma vez. (Kernighan, 1988).

Formalmente, a sintaxe é:

do
  comando;
while (expressão);

O *comando* é executado, então *expressão* é avaliada. Se for verdadeira, *comando* é executado novamente, e assim por diante. Se a *expressão* se tornar falsa, o laço termina. (Kernighan, 1988).

Abaixo é mostrado o autômato que representa a estrutura de controle while da linguagem C.

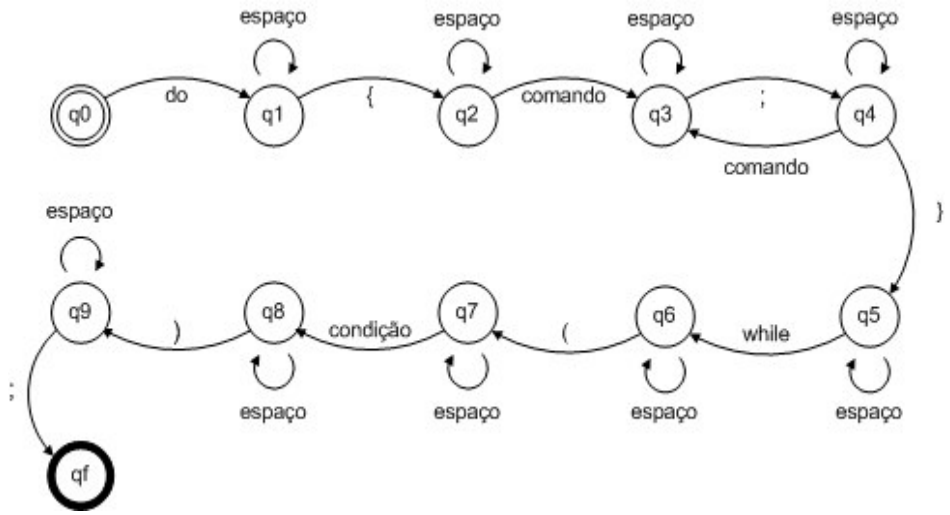

Figura 5.12: Módulo do-while

5.2.6 Comentários

Em C, todo comentário começa com o par de caracteres /* e termina com */. Não deve haver nenhum espaço entre o asterisco e a barra. O compilador ignora qualquer texto entre os símbolos de comentário. (Schildt, 1996).

Formalmente, a sintaxe é:

*/*comentário comentário comentário*/*

Os comentários podem ser colocados em qualquer lugar em um programa desde que não apareçam no meio de uma palavra-chave ou identificador. (Schildt, 1996).

Abaixo é mostrado o autômato que representa os comentários na linguagem C.

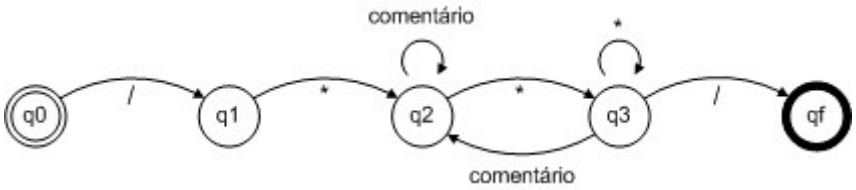

Figura 5.13: Módulo comentário

5.3 Conclusão

Esse capítulo teve como objetivo, apresentar os conceitos que envolvem a Teoria de Linguagens Formais e Autômatos, mais especificamente os Autômatos Finitos.

No decorrer do capítulo, foram apresentados também, os autômatos finitos desenvolvidos para delimitar e representar o escopo da linguagem de programação C, que será tratada pelo pré-compilador em questão.

O pré-compilador não irá tratar todos os possíveis erros, que possam existir em códigos fontes da linguagem C, más sim os que estiverem dentro do escopo, apresentado nesse capítulo por intermédio dos autômatos finitos.

No Capítulo 6, será apresentada a forma com a qual a estrutura do pré-compilador foi planejada e desenvolvida.

6 - Estrutura Adotada no Projeto do Analizador de Código

Esse capítulo tem como finalidade, abordar toda a estrutura utilizada para o desenvolvimento do trabalho em questão. Trata-se da implementação de um analisador de código para a linguagem de programação C, o qual foi desenvolvido para analisar códigos fontes de programas escritos na linguagem C, baseando-se sempre no padrão ANSI desta linguagem.

O analisador também foi desenvolvido na linguagem C, sendo adotado o padrão ANSI-C durante toda a fase de construção. O objetivo de se adotar este padrão, é o fato de tornar o analisador portável para diversas plataformas de sistemas operacionais, bastando para tal, apenas compilá-lo na plataforma desejada.

Este analisador foi compilado e executado sobre os sistemas operacionais Linux e Windows, utilizando-se um computador PC AMD *Athlon* XP 2000.

Para utilizar o analisador de código, basta realizar a chamada do mesmo, via *prompt* de comando[6], passando por parâmetro o caminho completo do arquivo com o código fonte que será analisado. Este arquivo deverá possuir a extensão "c".

O analisador fará uma leitura de todo o código contido no arquivo cujo nome foi passado como parâmetro e verificará a existência de um conjunto de erros comuns de programação. Os erros encontrados serão reportados ao usuário na tela do computador.

---

[6] Em sistemas operacionais baseados em comando, o ***prompt*** é constituído por um ou mais símbolos que indicam o local a partir do qual o usuário deve digitar uma instrução.

Abaixo segue a Figura 6.1, que ilustra a chamada do analisador de código, passando por parâmetro um determinado arquivo “c” que será analisado.

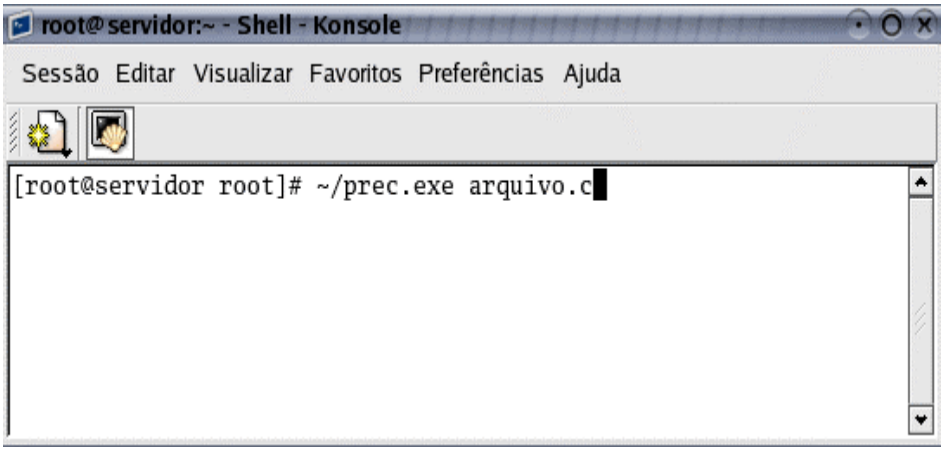

Figura 6.1: Chamada do analisador de código passando o arquivo por parâmetro

O analisador de código C foi organizado em três etapas, sendo cada uma responsável por uma série de verificações e tratamentos, a serem descritas nas próximas seções.

6.1 Primeira Etapa do Analisador de Código

A primeira etapa do analisador, recebe o arquivo passado por parâmetro, abre o mesmo, e em seguida começa a ler e analisar suas linhas.

O objetivo dessa etapa, é preparar o código fonte, ou seja, formatá-lo, visando facilitar o trabalho das etapas posteriores.

A primeira tarefa desta etapa é, identificar uma série de *tokens* especiais e aplicar espaçamento entre os mesmos, gerando assim uma melhor organização do código fonte. A Tabela 6.1 mostra estes *tokens* especiais.

Tabela 6.1: *Tokens* especiais

Token	**Descrição**
(,)	Parênteses
[,]	Colchetes
,	Vírgula
&&, ||, !	Operadores lógicos
>, >=, <, <=, ==, !=	Operadores relacionais
+, -, *, /, %	Operadores aritméticos
++, --	Incremento e decremento
+=, *=, -=, %=, /=, =	Comandos de atribuição
.	Ponto
?	Operador ternário
;	Ponto e vírgula
#	*Sharp*
&	Referência a endereços de memória
“, ‘	Aspas simples e duplas

A segunda tarefa dessa etapa é, identificar e retirar caracteres não significativos como espaços em branco, tabulações, linhas vazias e comentários. Ao final dessa tarefa, todas as linhas do código fonte estarão alinhas à esquerda.

Para ilustrar as duas tarefas dessa etapa mencionas acima, a Figura 6.2 mostra o exemplo da edição de um código fonte, e logo em seguida, a Figura 6.3 mostra o mesmo código fonte depois de ter sido submetido à execução do analisador.

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

/* o programa começa aqui */

int main()
{
   int numero1=0;
   int numero2=3;

   if(numero1>=numero2)
      numero2++;
   else
      numero1++;

   return 1;
}
```

Figura 6.2: Código fonte antes da execução do analisador (Exemplo 1)

```
# include <stdio.h>
# include <string.h>
int main (  )
{
int numero1 = 0 ;
int numero2 = 3 ;
if ( numero1 >= numero2 )
numero2 ++  ;
else
numero1 ++  ;
return 1 ;
}
```

Figura 6.3: Código fonte após a execução do analisador (Exemplo 1)

Como pode ser visto na Figura 6.3, o analisador identificou alguns *tokens* especiais, e colocou espaçamento entre eles, como é o caso do *token* "#" na linha um. Nessa mesma linha, existe o caractere "." (ponto) que é um *token* especial, porém nesse caso ele se refere ao nome de uma biblioteca, portanto nesse contexto não haverá espaçamento.

Na linha sete, comando "*if*", o *token* ">=" que é um operador relacional foi identificado, e consequentemente houve espaçamento antes e depois do mesmo. Já na linha oito, foi detectado o *token* "++", que diz respeito ao incremento da variável *numero2*. Nesse caso também houve espaçamento.

Nota-se ainda que, o comentário que havia no código fonte foi retirado, assim como as linhas vazias e espaços em branco no início de algumas linhas. Por fim, todas as linhas do código fonte ficaram alinhas à esquerda.

A terceira e última tarefa dessa etapa é, identificar a existência de mais de um comando por linha e separá-los em linhas distintas.

A Figura 6.4 ilustra o exemplo de um código fonte com mais de um comando por linha. Posteriormente a Figura 6.5 mostra o mesmo código fonte depois de ter sido submetido à execução do analisador.

```
#include<stdio.h>

int main()
{
   int i;

   for(i=0;i<5;i++)
   {
      printf(i); printf(i+1);
   }

   return 1;
}
```

Figura 6.4: Código fonte antes da execução do analisador (Exemplo 2)

```
# include <stdio.h>
int main (  )
{
int i ;
for ( i = 0 ; i < 5 ; i ++  )
{
printf ( i )  ;
printf ( i + 1 )  ;
}
return 1 ;
}
```

Figura 6.5: Código fonte após a execução do analisador (Exemplo 2)

Como pode ser visto na Figura 6.5, ao encontrar mais de um comando por linha, o analisador os separou, colocando-os em linhas distintas (linhas sete e oito). Dessa forma, pode-se garantir que, ao término da execução da primeira etapa, só haverá um comando por linha no código fonte.

Todas as alterações realizadas no código fonte, conseqüente das regras aplicadas pelas tarefas da primeira etapa, são efetivamente salvas em um outro arquivo que o analisador cria no início dessa etapa.

O motivo pelo qual se cria esse arquivo auxiliar é manter a integridade do arquivo fonte original, haja visto que após a execução da primeira etapa, a organização do código fonte é totalmente diferente da inicial.

Dessa forma, as etapas posteriores do analisador utilizarão o mesmo arquivo gerado pela primeira etapa.

A seguir, na Seção 6.2, serão discutidas as regras que compõe a segunda etapa do analisador de código C.

6.2 Segunda Etapa do Analisador de Código

A segunda etapa do analisador de código C recebe por parâmetro, o arquivo fonte gerado pela primeira etapa. Através desse arquivo serão feitas todas as análises e tratamentos responsáveis pela segunda etapa.

6.2.1 Declaração de Bibliotecas

A primeira tarefa da segunda etapa é identificar as declarações de bibliotecas feitas no código fonte. Ao encontrar uma declaração de biblioteca, as seguintes verificações são realizadas:

- Verificar se a biblioteca faz parte das bibliotecas pertencentes ao padrão ANSI-C;
- Verificar se a biblioteca foi declarada mais de uma vez no programa.

Para verificar se a biblioteca identificada pertence ao conjunto de bibliotecas definidas pelo padrão ANSI-C, o analisador realiza uma consulta em uma de suas tabelas, a tabela de bibliotecas do padrão ANSI-C.

A Tabela 6.2 é a tabela que foi construída no analisador, para armazenar todas as bibliotecas que fazem parte deste padrão.

Tabela 6.2: Tabela do analisador para armazenar bibliotecas do padrão ANSI-C

Código	Descrição da Biblioteca
0	assert.h
1	ctype.h
2	errno.h
3	float.h
4	limits.h
5	locale.h
6	math.h
7	setjmp.h
8	signal.h
9	stdarg.h
10	stddef.h
11	stdio.h
12	stdlib.h
13	string.h
14	time.h

O objetivo de se construir essa tabela, foi possibilitar à segunda etapa, averiguar se as bibliotecas identificas nos programas pertencem ou não às definidas pelo padrão ANSI-C. A referência utilizada para obter os nomes das bibliotecas incorporadas ao padrão ANSI-C foi Richardson (2003).

A Figura 6.6 mostra o exemplo de um código fonte, que contém a declaração de uma biblioteca que não faz parte do padrão ANSI-C.

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

int main(void)
{
   printf("Exemplo de bibliotecas.");
}
```

Figura 6.6: Exemplo com biblioteca que não faz parte do padrão ANSI-C

O exemplo acima mostra na linha um, a declaração da biblioteca *calculos.h*, que por sua vez não pertence ao padrão ANSI-C. Dessa forma ao executar esse código fonte no analisador, será exibida uma mensagem notificando a ocorrência do erro conforme é exibido na Figura 6.7.

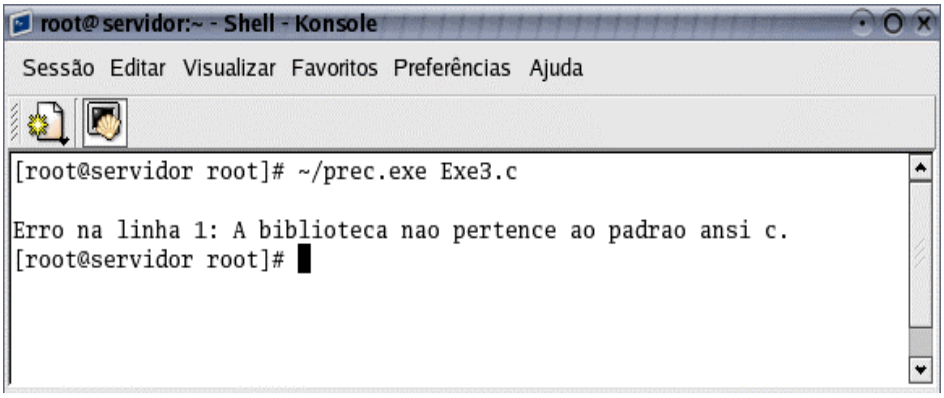

Figura 6.7: Mensagem de erro para a biblioteca que não pertence ao padrão ANSI-C

Com relação à verificação de duplicidade nas declarações de bibliotecas, o analisador faz tal verificação através da tabela de símbolos de bibliotecas, que toda vez que identifica uma declaração de biblioteca no código fonte, insere um novo registro.

Após a execução do código fonte, mostrado na Figura 6.6, a tabela de símbolos de bibliotecas estará preenchida com as seguintes informações conforme é ilustrado na Tabela 6.3.

Tabela 6.3: Tabela de símbolos de bibliotecas

Posição	Nome da Biblioteca
1	calculos.h
2	stdio.h

6.2.2 Declaração de Constantes

A segunda tarefa da segunda etapa é identificar as declarações de constantes feitas no código fonte. Ao encontrar uma declaração de constante, a seguinte verificação é realizada:

- Verificar o caractere inicial do nome dado à constante. O nome da constante deve iniciar com algum caractere compreendido entre os caracteres existentes no alfabeto da língua portuguesa, ou seja, da letra "a" até a letra "z", independente se ser maiúsculo ou minúsculo, ou então pelo caractere "_" *underscore*.

Todas as constantes identificadas no código fonte do programa, são inseridas na tabela de símbolos de constantes existente no analisador.

A Figura 6.8 ilustra o exemplo de um código fonte, que contém a declaração de algumas constantes, sendo uma delas declarada com um nome cujo caractere inicial não é válido.

```
#include<stdio.h>

#define tamanhoNome 50
#define ?valor 30

int main()
{
   /* Exemplo de constantes */
   return 1;
}
```

Figura 6.8: Exemplo de declaração de constantes

Ao executar o código fonte acima no analisador, será identificado um erro na linha onde é feita a declaração da constante *?valor*, pois a mesma inicia

seu nome com o caractere “?”, que não é um caractere válido para iniciar o nome de uma constante.

A Figura 6.9 mostra a saída de erro gerada pelo analisador conforme foi mencionado no parágrafo anterior.

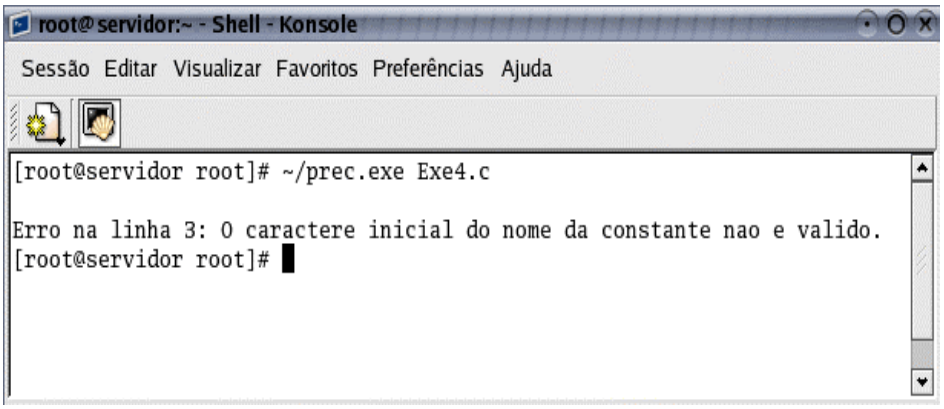

Figura 6.9: Mensagem de erro para constante declarada com nome inválido

Com relação à tabela de símbolos de constantes, após a execução do código fonte, mostrado na Figura 6.8, a mesma estará preenchida com as seguintes informações conforme é ilustrado na Tabela 6.4.

Tabela 6.4: Tabela de símbolos de constantes

Posição	Nome da Constante
1	tamanhoNome
2	?valor

6.2.3 Declaração de Variáveis

A terceira tarefa da segunda etapa é identificar as declarações de variáveis feitas no código fonte. Ao encontrar uma declaração de variável, as seguintes verificações são realizadas:

- Verificar se existe algum caractere no nome da variável que não seja válido. Os caracteres válidos para nome de variáveis são os mesmos mencionados anteriormente para nome de constantes;
- Se a variável corresponder a um vetor, verificar a paridade de colchetes;
- Se a variável corresponder a um vetor e estiver utilizando como definição de tamanho alguma constante, verificar se a referida constante foi previamente declarada;
- Se houver algum comando de atribuição para a variável, verificar se há algum valor para ser atribuído;
- Verificar se o nome da variável é igual ao de alguma palavra reservada da linguagem C;
- Verificar se a variável foi declarada mais de uma vez no programa dentro do mesmo escopo.

Para ilustrar algumas das verificações feitas nas declarações de variáveis, a Figura 6.10 mostra o exemplo de um código fonte que contém alguns erros.

```
#include<stdio.h>

#define tamanho 50

/* Exemplo de erros nas declarações de
   de variáveis */

int main()
{
   int vetor1[tamanho];
   int vetor2[valor];
   int vetor3[[tamanho];
   char palavra = ;

   printf("Teste de variáveis!");

   return 1;
}
```

Figura 6.10: Exemplo de declaração de variáveis (Exemplo 1)

O exemplo da Figura 6.10 contém alguns erros nas declarações das variáveis. Ao executar este código fonte no analisador, o mesmo identificará a primeira declaração de variável, no caso a variável *vetor1*. Esta variável está declarada de forma correta, pois o tamanho dela, que é definido pela constante *tamanho*, foi declarada no início do programa.

Ao prosseguir a execução, o analisador identificará a segunda declaração de variável, no caso a variável *vetor2*. Esta variável contém um erro, pois a constante que define seu tamanho, chamada de *valor*, não foi declarada no programa.

A próxima declaração de variável a ser identificada será a variável *vetor3*. Esta variável por sua vez, possui um erro na paridade de colchetes, ou seja, existem mais colchetes abertos do que fechados.

Por último, o analisador encontrará a declaração da variável *palavra*. Esta variável possui um erro na atribuição de valor, pois após seu nome existe

um operador de atribuição (operador =), porém após este operador não há a ocorrência de valor.

Abaixo segue a Figura 6.11, que ilustra o resultado gerado pelo analisador, na execução do código fonte exibido na Figura 6.10.

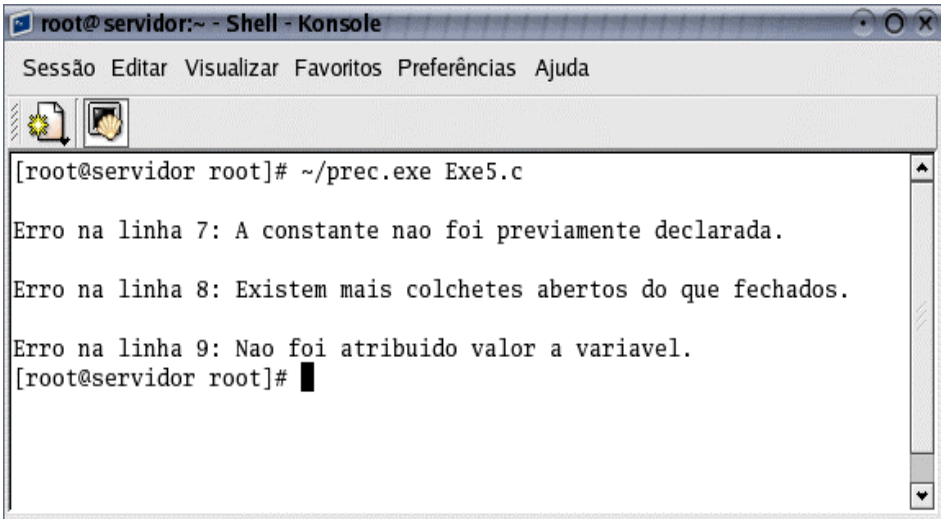

Figura 6.11: Mensagens de erro nas declarações de variáveis

Para verificar se o nome de uma variável é igual ao de alguma palavra reservada da linguagem C, o analisador realiza uma consulta em uma de suas tabelas, a tabela de palavras reservadas. Essa consulta consiste em verificar se o nome da variável é igual ao de alguma palavra reservada da linguagem C.

A Tabela 6.5 é a tabela que foi construída no analisador, para armazenar todas as palavras reservadas da linguagem C.

Tabela 6.5: Tabela do analisador de código para armazenar as palavras reservadas da linguagem C

Posição	Nome da Palavra
1	Auto
2	Break
3	Case
4	Char
5	Const
6	Continue
7	Default
8	Do
9	Double
10	Else
...	...
32	While

A referência utilizada para obter os nomes das palavras reservadas da linguagem C foi O'Hare (2004).

A última verificação que o analisador de código C realiza na declaração de variáveis, é a duplicidade de variáveis dentro de um mesmo escopo, como por exemplo, duas variáveis globais com mesmo nome e tipo.

Toda vez que o analisador identifica no código fonte uma declaração de variável, ele insere algumas informações a respeito dessa variável em uma tabela de símbolos de variáveis.

Portanto, através dessa tabela de símbolos de variáveis, é possível ao encontrar uma declaração de variável, consultar nessa tabela, se já existe a ocorrência dessa variável, levando-se em consideração o nome, o tipo e o escopo, ou seja, local (dentro de alguma função) ou global.

A Figura 6.12 mostra como exemplo, um código fonte com declarações de variáveis, sendo algumas com erros de duplicidade e outras com erro de nomes iguais ao de palavras reservadas da linguagem C.

```
#include<stdio.h>

char nome[30];
char palavra;
int y;
int x;
char nome[30];

void imprimeNumero()
{
   int x;
   int while;

   for(x=0; x<10; x++)
      printf("%d", x);
}
```

Figura 6.12: Exemplo de declaração de variáveis (Exemplo 2)

Como foi apresentando no exemplo da Figura 6.12, existem erros em algumas declarações de variáveis desse código fonte. Ao identificar as declarações de variáveis, o analisador irá detectar dois erros.

O primeiro erro diz respeito à duplicidade na declaração da variável *nome*, pois a mesma aparece duas vezes declarada no escopo de variável global. O segundo erro diz respeito à declaração da variável *while*, que foi declarada com o nome igual ao de uma palavra reservada da linguagem C.

A Figura 6.13 ilustra o resultado gerado pelo analisador, ao executar o código fonte exibido na Figura 6.12.

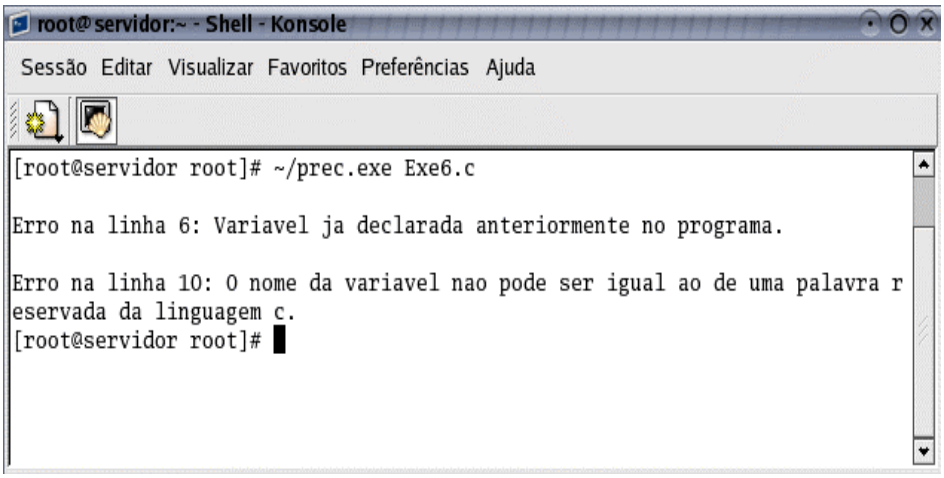

Figura 6.13: Mensagens de erro nas declarações de variáveis

Com relação à tabela de símbolos de variáveis, após a execução do código fonte, mostrado na Figura 6.12, a mesma estará preenchida com as seguintes informações conforme é ilustrado na Tabela 6.6.

Tabela 6.6: Tabela de símbolos de variáveis

Posição	Linha	Escopo	Tipo	Nome da Variável	Nome da Função
1	2	global	char	nome	-
2	3	global	char	palavra	-
3	4	global	int	y	-
4	5	global	int	x	-
5	6	global	char	nome	-
6	9	local	int	x	imprimeNumero
7	10	local	int	while	imprimeNumero

6.2.4 Declaração de Funções

A quarta tarefa da segunda etapa é identificar as declarações de funções feitas no código fonte. Ao encontrar uma declaração de função, as seguintes verificações são realizadas:

- Verificar se existe algum caractere no nome da função que não seja válido. Os caracteres válidos para nome de funções são os mesmos mencionados anteriormente para nome de constantes;
- Verificar se o nome da função é igual ao de alguma palavra reservada da linguagem C;
- Verificar se a função foi declarada mais de uma vez no programa;
- Se o tipo de retorno da função for diferente de "*void*", verificar se há algum retorno de valor dentro da mesma.

Para verificar caracteres inválidos e o uso de nomes de palavras reservadas da linguagem C nos nomes das funções, o analisador utiliza as mesmas técnicas mencionadas na Seção 6.2.3.

Toda vez que o analisador identifica no código fonte uma declaração de função, ele insere algumas informações a respeito dessa função em uma tabela de símbolos de funções.

Através dessa tabela de símbolos de funções, o analisador consegue realizar consultas para verificar se uma determinada função já foi declarada anteriormente no programa.

Para ilustrar alguns erros em declarações de funções, como por exemplo, duplicidade e falta de retorno de dados, segue abaixo a Figura 6.13.

```
#include<stdio.h>

int intFuncao1()
{
   printf("Função 1!");
   return 1;
}

char strFuncao2()
{
}

int intFuncao1()
{
   printf("Função 1 novamente!");
   return 0;
}
```

Figura 6.14: Exemplo de declarações de funções

O código fonte mostrado no exemplo da Figura 6.14, possui dois erros no que diz respeito à declaração de funções. O primeiro erro é com relação à função *intFuncao1*. Ela foi declarada duas vezes no programa com o mesmo nome, tipo de retorno e número de parâmetros.

Já o segundo erro é com relação à função *strFuncao2*. Ela foi declarada com o tipo de retorno *char*, ou seja, a função exige o retorno de dado do tipo *char*. Como não houve o esperado retorno de dado, fica caracterizado o erro.

Ao executar o código fonte da Figura 6.14, o analisador gerou como resultado duas mensagens de erro, conforme foi previsto nos parágrafos anteriores. Abaixo segue a Figura 6.15 com as mensagens geradas pelo analisador.

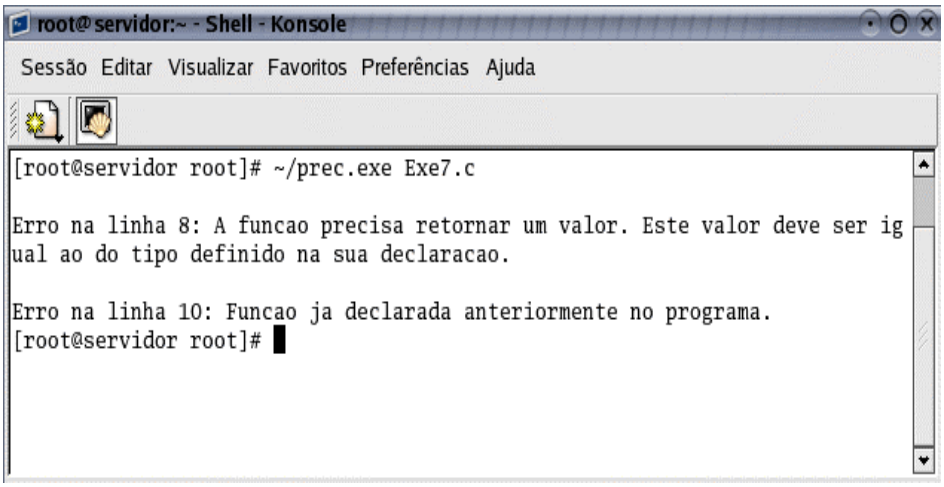

Figura 6.15: Mensagens de erro nas declarações de funções

Com relação à tabela de símbolos de funções, após a execução do código fonte, mostrado na Figura 6.14, a mesma estará preenchida com as seguintes informações conforme é ilustrado na Tabela 6.7.

Tabela 6.7: Tabela de símbolos de funções

Posição	Linha Inicial	Linha Final	Número de Parâmetros	Tipo de Retorno	Nome da Função
1	2	6	0	int	intFuncao1
2	7	9	0	char	strFuncao2
3	10	14	0	int	intFuncao1

6.2.5 Mensagens de Erro

Como foi visto no decorrer desse capítulo, toda vez que o analisador de código C identifica a ocorrência de algum erro em um código fonte, ele exibe uma mensagem notificando a linha onde o erro foi encontrado e a descrição detalhada do mesmo.

Para que essas notificações de erros pudessem ser exibidas, foi necessário construir uma tabela de mensagens de erros no analisador. Dessa forma, o analisador ao identificar algum tipo de erro, realiza uma consulta na tabela de erros, a fim de verificar qual mensagem corresponde ao erro identificado.

Outra característica importante é que todas as mensagens de erro foram desenvolvidas no idioma português. A Tabela 6.8 mostra algumas mensagens de erros.

A referência completa da tabela de mensagens de erros do analisador pode ser vista no Anexo A.

Na Seção 6.3, serão discutidas as regras que compõe a terceira etapa do analisador de código C.

Tabela 6.8: Tabela de mensagens de erros do analisador de código C

Código do Erro	Mensagem de Erro
1	A finalização da diretiva include deve ser feita com o caractere ' > ', portanto este deve ser o último caractere da linha.
2	Entre o caractere ' < ' e o nome da biblioteca não pode haver espaço.
3	Depois da diretiva include deve vir o nome da biblioteca.
...	...
8	Depois do caractere ' # ' deve vir a sintaxe 'include'.
9	A diretiva include deve iniciar com o caractere ' # '.
10	A biblioteca não pertence ao padrão ANSI-C.
...	...
16	Nenhum valor foi atribuído a variável.
...	...
22	O nome da função não pode ser igual ao de uma palavra reservada da linguagem C.
23	Depois do nome da função devem vir os parênteses ' () ', com ou sem parâmetros.
...	...
31	O caractere inicial do nome da constante não é válido.
32	A constante não foi previamente declarada.
...	...
36	Na declaração de vetores e matrizes, a definição de tamanho deve terminar com o colchete fechado '] '.
40	A linha possui mais parênteses fechados do que abertos.
41	A linha possui mais aspas duplas que o necessário.
42	Na declaração de função não pode haver o sinal de ponto e vírgula ' ; '.
43	A função não foi previamente declarada ou pertence a alguma biblioteca que não foi utilizada no programa.
...	...

6.3 Terceira Etapa do Analisador de Código

A terceira etapa do analisador de código C tem como objetivo, receber por parâmetro o arquivo fonte gerado pela primeira etapa, e a partir desse arquivo, analisá-lo a fim de verificar algumas regras necessárias.

As regras que devem ser analisadas pela terceira etapa são:

- Identificar a obrigatoriedade de ponto e vírgula no final das linhas de comando;
- Verificar a paridade de parênteses, chaves, colchetes e aspas duplas;
- Identificar a utilização de variáveis e posteriormente verificar se as mesmas foram previamente declaradas;
- Identificar a chamada de funções e posteriormente verificar se as mesmas foram previamente declaradas. Caso não façam parte das funções declaradas, verificar se elas pertencem a alguma biblioteca utilizada no programa.

Para ilustrar melhor as regras mencionadas acima, segue a Figura 6.16 com um código fonte de exemplo.

O código fonte apresentado na Figura 6.16, possui alguns erros que serão analisados. O primeiro erro se encontra na linha do retorno da função *imprimeValor*. Nessa linha, onde se encontra a sintaxe *return*, é obrigatório o ponto e vírgula no final, que, no entanto, não foi colocado.

O segundo erro se encontra dentro da função *main*, na linha da declaração da variável *raiz*. Nessa linha também é obrigatório o ponto e vírgula no final.

O terceiro erro se encontra na linha da condição *if*, o que ocorre é que foram colocados mais parênteses fechados do que abertos, ocasionando assim mais um erro.

```
#include<stdio.h>

int imprimirValor(char valor)
{
   printf(valor);
   return 1
}

int main()
{
   int numero1[10][5];
   double raiz

   if (numero1[1][1] > 0))
      raiz = sqrt(numero1[1][1]);

   imprimirValor(numero2);

   numero1[[10][5] = 0;

   printf("Fim do programa!);

   return 1;
}
```

Figura 6.16: Código fonte de exemplo para a terceira etapa

O quarto erro se encontra dentro da condição *if*, na linha onde é feita a chamada da função *sqrt*. Acontece que essa função é uma função pertencente a uma biblioteca do padrão ANSI-C, chamada *math.h*. Porém, como essa biblioteca não foi utilizada no referido programa, ocasionará um erro de função não declarada.

Para que o analisador consiga identificar se uma determinada função pertence ou não a uma biblioteca do padrão ANSI-C, foi necessário construir uma tabela de funções do padrão ANSI-C. Ver Tabela 6.9.

Tabela 6.9: Tabela de funções do padrão ANSI-C

Posição	Tipo da Função	Nome da Função	Biblioteca Pertencente
1	int	isalnum	ctype.h
2	int	isalpha	ctype.h
...	...	...	...
12	int	tolower	ctype.h
13	int	toupper	ctype.h
...	...	...	...
16	double	exp	math.h
17	double	log	math.h
...	...	...	...
45	FILE	fopen	stdio.h
46	FILE	freopen	stdio.h
...	...	...	...
56	int	printf	stdio.h
57	int	sprintf	stdio.h
...	...	...	...
139	size_t	strftime	time.h

Já o quinto erro, se encontra na linha da chamada da função *imprimeValor*. A variável chamada *numero2*, a qual é passada por parâmetro nessa função, não foi previamente declarada, acarretando em mais uma ocorrência de erro.

O analisador consegue verificar se uma determinada variável encontrada no programa, já foi declarada anteriormente, através da tabela de símbolos de variáveis abordada na Seção 6.2.

Em seguida na linha posterior, aparece o sexto erro. A variável *numero1*, à qual é atribuído o valor “0” (zero) possui erro na paridade de colchetes, pois existem mais colchetes abertos do que fechados.

Por último, existe um erro na chamada da função *printf*, localizada no final do programa. Nessa função foi passado um texto como parâmetro, só que houve erro de paridade de aspas duplas, pois o texto iniciou-se com as aspas abertas e não terminou com as aspas fechadas.

Diante desse contexto, a Figura 6.17 mostra o resultado gerado pelo analisador, ao executar o código fonte exibido na Figura 6.16.

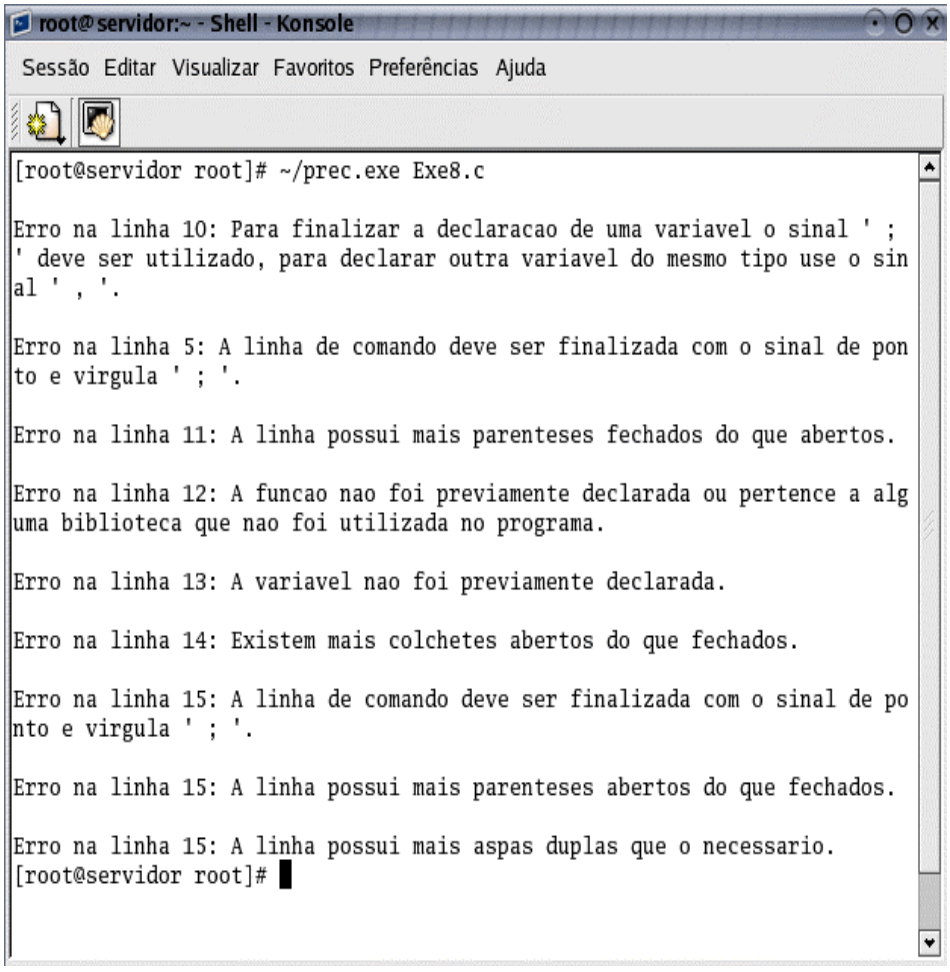

Figura 6.17: Mensagens de erro geradas para o código da Figura 6.16

Através da tabela de funções do padrão ANSI-C, o analisador conseguiu, por exemplo, identificar que as chamadas da função *printf* eram referentes à função *printf* pertencente à biblioteca *stdio.h*, que foi declarada no início do programa.

A referência utilizada para obter as informações referentes às funções do padrão ANSI-C foi Richardson (2003).

6.4 Conclusão

Este capítulo apresentou a forma com a qual a estrutura do analisador de código C em questão foi desenvolvida.

O analisador foi dividido em três etapas, cada uma responsável por validar um conjunto de regras da linguagem C.

Um dos critérios adotados no desenvolvimento do analisador foi que o mesmo fosse portável para diversos sistemas operacionais.

No decorrer deste capítulo, para cada uma das três etapas do analisador, foram apresentados exemplos de códigos fontes com diversos erros de programação.

Para cada exemplo de código fonte com erro apresentado, foi demonstrada logo em seguida, a execução dos mesmos pelo analisador, a fim de mostrar as mensagens dos erros identificados.

Outro aspecto importante, que também foi descrito neste capítulo, diz respeito à forma com a qual o analisador utiliza e manipula as tabelas de símbolos de variáveis, de funções, de constantes e de bibliotecas.

Dessa forma, foi possível apresentar como o analisador foi estruturado, descrevendo em detalhes as particularidades de cada uma de suas etapas.

7 - Conclusão

O tema *Compiladores* diz respeito a uma das áreas da computação. A área de compiladores é bem complexa e extensa, pois agrega conceitos e técnicas que já vem sendo discutidas a algum tempo.

A proposta deste trabalho não foi desenvolver um novo compilador para a linguagem C, mas sim uma ferramenta prática que auxiliasse no ensino desta linguagem.

Essa ferramenta trata-se de um analisador de código, que teve seus objetivos focados nas análises dos erros mais comuns, geralmente cometidos pelos iniciantes na linguagem C.

Entre os erros mais comuns encontrados nos códigos fontes escritos em C, pode-se destacar a falta e/ou excesso de parênteses, chaves, colchetes, vírgulas, a falta de ponto e vírgula no final das linhas de comando, utilização de variáveis, constantes e funções sem terem sido inicialmente declaradas, declarar mais de uma vez a mesma variável etc.

Diante desse cenário, viu-se a necessidade de construir um analisador de código, que fosse capaz de ler a analisar códigos fontes escritos na linguagem C, com o intuito de identificar e notificar os possíveis erros de programação.

Ao identificar um erro, o analisador de código não interrompe o processo de análise, ele continua a execução até o final do código fonte.

Isso é importante, pois na ocorrência de vários erros em um código fonte, é possível corrigi-los com uma única execução do analisador, do contrário, para cada erro encontrado, seria necessário corrigi-lo e posteriormente submeter o código novamente à execução do analisador.

Uma característica importante, é que as mensagens de erro notificas ao usuário, pelo analisador de código C, são exibidas no idioma português. O fato

de exibir tais mensagens neste idioma contribui para que o usuário tenha melhor compreensão do erro identificado.

Para facilitar o processo de construção e obter uma melhor modularização, o analisador foi dividido em três etapas. Cada etapa possui tarefas distintas, como já foram vistas no Capítulo 6.

Outra característica do analisador, é que o mesmo foi desenvolvido na linguagem C, adotando-se o padrão ANSI desta linguagem. Este critério foi adotado para que o analisador seja portável para diversos sistemas operacionais.

Após a construção do analisador de código C, o mesmo foi submetido a uma seqüência de testes, realizados em sistemas operacionais Linux e Windows, cujo objetivo foi analisar diversos códigos fontes escritos na linguagem C, cada um com diversos tipos de erros.

Em todos os testes, o analisador identificou os erros de programação existentes nos códigos, lembrando-se que estes erros foram elaborados, com base nos erros que o analisador é capaz de tratar.

Diante disso, pode-se dizer que este trabalho superou os objetivos propostos, e que o analisador de código está pronto para auxiliar no ensino da linguagem C.

7.1 Sugestões para Trabalhos Futuros

Abaixo seguem algumas sugestões para trabalhos futuros:

- Tratar a sintaxe dos comandos *if*, *while*, *do-while*, *for* e *switch*;
- Tratar as condições dos comandos *if*, *while*, *do-while*, *for* e *switch*;
- Tratar os tipos atribuídos a variáveis, verificando se há compatibilidade de tipos;
- Tratar a estrutura de dados *struct*.

Referências Bibliográficas

ALGOL. *The ALGOL Programming Language.* [on-line]. Disponível na internet via www. url: http://www.engin.umd.umich.edu/cis/course.des/cis400/algol/algol.html. Arquivo capturado em 21 de junho de 2005.

AHO, Alfred V., Sethi, Ravi and Ullman, Jeffrey D. *Compiler Principles Techniques and Tools*. Addison-Wesley, 1986. 344p.

FERREIRA, Aurélio Buarque de Holanda. *Novo Dicionário Aurélio da Língua Portuguesa*. Curitiba: Positivo, 2004. 2120p.

GARSHOL, Lars Marius. *BNF and EBNF: What are they and how do they work?* [on-line]. Disponível na internet via www. url: http://www.garshol.priv.no/download/text/bnf.html. Arquivo capturado em 20 de junho de 2005.

GIACOMIN, João Carlos. *Introdução à Linguagem C.* Lavras: UFLA/FAEPE, 2003. 112p.

GRUNE, Dick. *Projeto Moderno de Compiladores*. São Paulo: Campus, 2001. 671p.

HOPCROTF, John E., Motwani, Rajeev & Ullman, Jeffrey D. *Introdução à teoria dos autômatos, linguagens e computação*. Rio de Janeiro: Campus, 2002. 560p.

KERNIGHAN, Brian W. and Ritchie, Dennis M. *The C Programming Language*. New Jersey: Prentice Hall, EUA, 1988. 272 p.

JOHNSON, Stephen C. *YACC*. [on-line]. Disponível na internet via www. url: http://en.wikipedia.org/wiki/Yacc. Arquivo capturado em 17 de julho de 2005.

LEWIS, Harry R. & Papadimitriou, Christos H. *Elementos de teoria da computação*. Porto Alegre: Bookman, 2000. 339p.

MENEZES, Paulo Fernando Blauth. *Linguagens formais e autômatos*. Porto Alegre: Sagra Luzzatto, Instituto de Informática da UFRGS, 2000. 165p.

O'HARE, Anthony B. *Reserved Words and Program Filenames*. [on-line]. Disponível na internet via www. url: http://www-teaching.physics.ox.ac.uk/computing/ProgrammingResources/Oxford/handbook_C_html/node64.html. Arquivo capturado em 07 de agosto de 2005.

PRATT, T. W. *Programming Languages: Design and Implementation*. New Jersey: Prentice Hall, EUA, 1984.

PRICE, Ana Maria de Alencar & Toscani, Simão Sirineo. *Implementação de linguagens de programação: Compiladores*. Porto Alegre: Sagra Luzzatto, Instituto de Informática da UFRGS, 2001. 216p.

RICARTE, Ivan Luiz Marques. *Análise Semântica*. [on-line]. Disponível na internet via www. url: http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node71.html. Arquivo capturado em 28 de julho de 2005.

RICHARDSON, Ross Leon. *C Standard Library*. [on-line]. Disponível na internet via www. url: http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html. Arquivo capturado em 06 de agosto de 2005.

SCHILDT, Herbert. *C, completo e total*. São Paulo: Pearson Education do Brasil, 1996. 827p.

Anexo A

Tabela de mensagens de erros do analisador de código C

Código do Erro	Mensagem de Erro
1	A finalização da diretiva include deve ser feita com o caractere ' > ', portanto este deve ser o último caractere da linha.
2	Entre o caractere ' < ' e o nome da biblioteca não pode haver espaço.
3	Depois da diretiva include deve vir o nome da biblioteca.
4	Depois do nome da biblioteca deve vir o caractere ' \" ', não pode haver espaço entre eles.
5	A finalização da diretiva include deve ser feita com o caractere ' \" ', portanto este deve ser o último caractere da linha.
6	Entre o caractere ' \" ' e o nome da biblioteca não pode haver espaço.
7	Depois da diretiva include deve vir o caractere ' < ' ou o caractere ' \" '.
8	Depois do caractere ' # ' deve vir a sintaxe 'include'.
9	A diretiva include deve iniciar com o caractere ' # '.
10	A biblioteca não pertence ao padrão ANSI-C.
11	Depois do nome da biblioteca de vir o caractere ' > '.
12	Tipo não definido pela linguagem C.
13	O caractere inicial do nome da variável não e valido.
14	Variável já declarada anteriormente no programa.
15	Para finalizar a declaração de uma variável o sinal ' ; ' deve ser utilizado, para declarar outra variável do mesmo tipo use o sinal ' , '.
16	Nenhum valor foi atribuído a variável.
17	Para atribuir valor a uma variável o sinal ' = ' deve vir precedido, para declarar outra variável do mesmo tipo use o sinal ' , ', para finalizar a declaração utilize o sinal ' ; '.
18	Apos o tipo deve ser especificado o nome da variável.
19	O nome da variável não pode ser igual ao de uma palavra reservada da linguagem C.
20	Apos o tipo deve ser especificado o nome da função.
21	O caractere inicial do nome da função não é válido.

Tabela de mensagens de erros do analisador de código C (cont.)

Código do Erro	Mensagem de Erro
22	O nome da função não pode ser igual ao de uma palavra reservada da linguagem C.
23	Depois do nome da função devem vir os parênteses ' () ', com ou sem parâmetros.
24	O parêntese de parâmetros da função não foi fechado ') '.
25	Para iniciar o código da função deve ser utilizada a chaves aberta ' { '.
26	A função não foi finaliza, é necessário utilizar a chaves fechada ' } '.
27	A função precisa retornar um valor. Este valor deve ser igual ao do tipo definido na sua declaração.
28	A diretiva define deve iniciar com o caractere ' # '.
29	Depois do caractere ' # ' deve vir a sintaxe 'define'.
30	Depois da diretiva define deve vir o nome da constante.
31	O caractere inicial do nome da constante não é válido.
32	A constante não foi previamente declarada.
33	Existem mais colchetes abertos do que fechados.
34	Existem mais colchetes fechados do que abertos.
35	Existem caracteres inválidos no nome da variável.
36	Na declaração de vetores e matrizes, a definição de tamanho deve terminar com o colchete fechado '] '.
37	Existem caracteres inválidos no nome da função.
38	A linha de comando deve ser finalizada com o sinal de ponto e vírgula ' ; '.
39	A linha possui mais parênteses abertos do que fechados.
40	A linha possui mais parênteses fechados do que abertos.
41	A linha possui mais aspas duplas que o necessário.
42	Na declaração de função não pode haver o sinal de ponto e vírgula ' ; '.
43	A função não foi previamente declarada ou pertence a alguma biblioteca que não foi utilizada no programa.
44	A variável não foi previamente declarada.
45	Função já declarada anteriormente no programa.
46	Biblioteca já declarada anteriormente no programa.