Compiladores

.1 Conceitos Básicos de Compiladores

Um compilador é um programa que lê um programa escrito em uma linguagem (linguagem fonte) e o traduz em um outro programa equivalente em outra linguagem (linguagem objeto). Como uma parte importante do processo de compilação, o compilador deve informar ao seu usuário a presença de erros no programa fonte. Um compilador é composto pelas seguintes partes. São elas: [Trem85]

- **Analisador Léxico**: é a primeira fase de um compilador. Sua tarefa principal é ler uma cadeia de caracteres de entrada e produzir como saída uma seqüência de *tokens* que será usada pelo *parser* para efetuar a análise sintática;
- **Analisador Sintático**: recebe os *tokens* do analisador léxico e os encaixa na árvore sintática, verificando se o programa analisado está de acordo com a sintaxe da linguagem;
- **Analisador Semântico**: constitui-se, em linhas gerais, de um verificador de tipos. Entre outras atribuições, é de sua responsabilidade verificar se um operador contém operandos do tipo correto, se estruturas tipo *se* ou e*nquanto* possuem condições *lógicas*, se uma variável foi declarada. Se alguma dessas condições não for satisfeita, o sistema reportará um erro semântico;
- **Geração de Código Intermediário**: a partir da árvore sintática, obtida no processo de análise, caminha-se na árvore em profundidade (*depth first*), gerando código em quádruplas (ou triplas, se for o caso) para cada uma das construções que aparecem na árvore;
- **Otimização de Código**: tem como objetivo melhorar o código quanto à eficiência da execução, tanto no sentido do uso de memória quanto no da velocidade de execução do programa objeto.
- **Geração de Código:** é a ultima etapa do processo de compilação. Ela recebe a representação intermediária do programa fonte e as informações armazenadas na tabela de símbolos e produz como saída um programa objeto equivalente. As informações da tabela de símbolos são usadas para determinar os endereços de execução dos objetos de dados do programa.

.1.1 Análise do Programa Fonte

No processo de compilação, a análise compõe-se de três fases:

- ***Análise linear***, na qual o programa fonte é percorrido caracter a caracter e o analisador extrai os elementos básicos da linguagem chamados *tokens*. Os *tokens* são seqüências de caracteres com sentido coletivo.
- ***Análise hierárquica***, na qual os *tokens* são agrupados hierarquicamente em estruturas aninhadas.
- ***Análise semântica***, na qual certos testes são efetuados para se certificar que os componentes do programa estão agrupados de uma forma correta.

O analisador linear ou léxico é comumente chamado *scanner* e como conseqüência o processo de análise léxica é chamado *scanning*. Na análise léxica, o programa fonte é percorrido para serem extraídos os elementos básicos da linguagem, chamados *tokens*. Os *tokens* ou unidades léxicas são as palavras que formam a linguagem. Por exemplo, na linguagem RS, alguns *tokens* seriam *rsd_prog, rs_prog, emit*,:=, identificadores, constantes, etc. O analisador léxico é chamado por demanda pelo analisador sintático, como veremos mais adiante. Cada vez que o léxico for chamado, ele devolverá um *token* válido da linguagem. Ocasionalmente, quando detectar um símbolo não reconhecido, ele deverá ser capaz de retornar uma indicação de erro apropriado.

O analisador hierárquico ou sintático é comumente chamado de *parser* e como conseqüência o processo de análise sintática é chamado de *parsing*. Na análise sintática, os *tokens* são agrupados em uma estrutura de árvore, chamada árvore sintática, que expressa a ordem em que esses elementos devem ser avaliados. Para as expressões da linguagem, essa árvore deve expressar a precedência dos operadores (ordem em que operações devem ser feitas, por exemplo, '*' antes de '+') e a associatividade dos operadores (ordem em que operações devem ser feitas, quando os operadores possuírem a mesma precedência). Se um *token* não pode ser encaixado na árvore sintática, então ocorre um erro sintático, significando que o programa fonte está incorreto, pois não corresponde à definição da linguagem que o compilador aceita. Portanto, deve-se entender o analisador sintático como um reconhecedor de linguagens. Finalmente, o analisador semântico percorre a árvore sintática procurando por erros semânticos, tais como combinação incorreta de tipos.

.1.2 Modelo de um Compilador

As diversas fases que compõem o processo de compilação estão ilustradas na seguinte figura: [Price98]

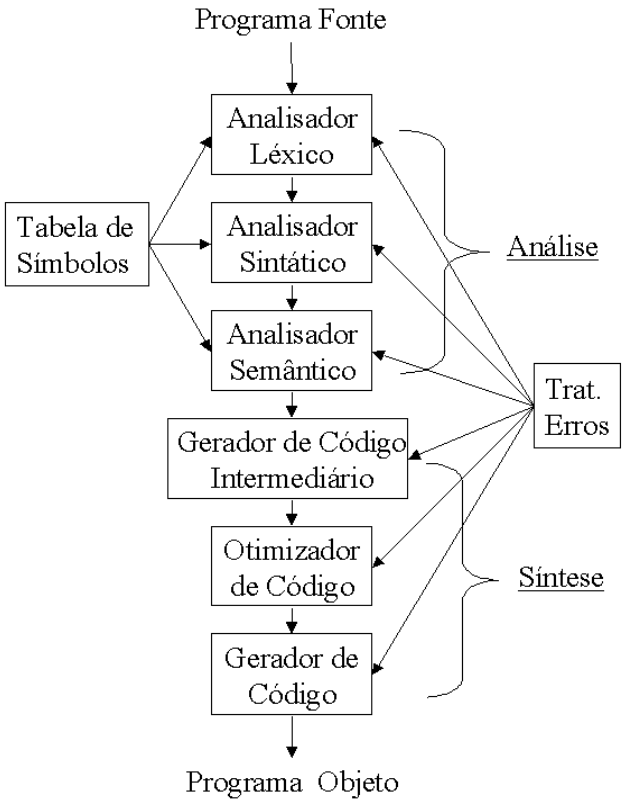

Figura 2.1: Fases de um Compilador.

Observe que em todo o processo de compilação é necessário tratar os erros encontrados, isso é efetuado por um conjunto de rotinas de tratamento de erros.

Os *tokens* encontrados no programa podem ser de dois tipos: itens constantes ou itens variáveis. Os itens constantes são formados pelas palavras reservadas, operadores aritméticos, relacionais e pelos delimitadores. Os itens variáveis são formados pelos identificadores, constantes numéricas e *strings* e devem ser inseridos em tabelas (tabela de símbolos, tabela de constantes numéricas e tabela de *strings*).

Alguns compiladores geram, após a análise, uma representação intermediária explícita do programa fonte. Esta representação intermediária deve ser simples de ser otimizada e simples de ser traduzida para código objeto. Ela permite a construção de otimizadores de código independentes da linguagem objeto. Podem ser usadas várias representações para este código intermediário, como por exemplo quádruplas ou triplas. Estas representações usam uma notação pré-fixada para representar as operações executadas pelo programa (operação, operando 1, operando 2, resultado).

.1.3. Ferramentas para Construção de Compiladores

Existem muitas ferramentas para o projeto automatizado de compiladores, tais como o *lex* (gerador de analisador léxico) e o *yacc* (gerador de *parser*) do UNIX. As ferramentas para geração automática de *parsers* utilizam técnicas bem gerais de análise sintática baseadas na abordagem *bottom-up*. Entretanto, em nossa implementação, não iremos utilizar essas ferramentas, mas sim construiremos um *parser* sob medida, usando a abordagem *top-down*. [Aho86]

.2 Compilador RS

.2.1 Análise Léxica

Analisador léxico é a primeira fase de um compilador. Sua tarefa principal é ler uma cadeia de caracteres de entrada e produzir como saída uma seqüência de *tokens* que o *parser* usará para efetuar a análise sintática [Camp97]

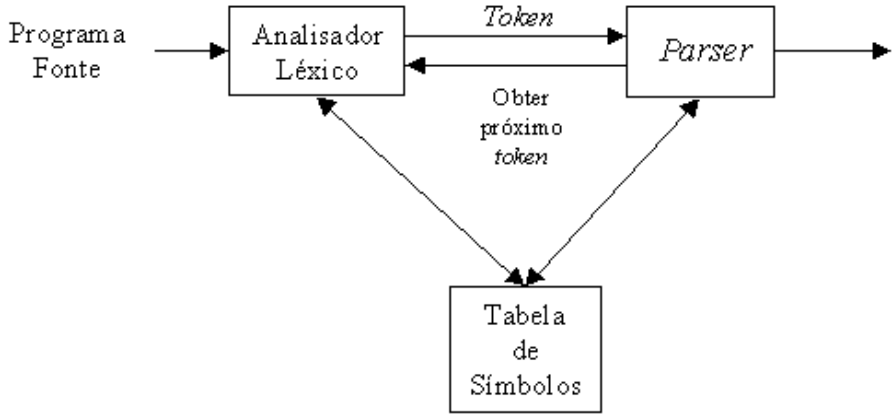

Figura 2.2: Funcionamento do Analisador Léxico.

A decisão de como desenvolver o analisador léxico tem duas opções: ou construir o analisador léxico manualmente, em uma linguagem de programação comum, como o C++, da Borland®, ou desenvolvê-lo a partir da ferramenta Lex, incorporada no LINUX. A opção escolhida foi o desenvolvimento do analisador léxico manualmente. Uma das razões da escolha foi o desconhecimento da ferramenta do LINUX. Apesar de existirem várias referências sobre o assunto, programas extremamente simples, ao serem colocados no Lex, transformam-se em programas em C com milhares de linhas. Se programas extremamente simples, como uma calculadora, necessitariam de muitas linhas de código, no que se transformariam as especificações do FORTALL no Lex? Mesmo que transformassem em

poucos milhares de linhas de código, o entendimento do mesmo seria muito difícil, dificultando alguma mudança no analisador léxico gerado. Outro motivo foi que, com o desenvolvimento a partir do Lex, o FORTALL ficaria restrito ao ambiente em LINUX. A sua especificação inicial é que o FORTALL seja multiplataforma, rodando tanto no DOS como no LINUX.

Analisador Léxico é a parte do compilador que lê o programa fonte, caracter a caracter e constrõe a partir deste entidades primárias chamadas Tokens. O analisador léxico transforma o programa fonte em tiras de tokens.

Tarefas a serem executadas pelo analisador léxico :

- Identificar os símbolos;
- Eliminar os espaços em brancos, caracteres de fim de linha, etc.;
- Eliminar os comentários que acompanham o fonte;
- Criar símbolos intermediários chamados Tokens;
- Avisar de erros que detectar;

Exemplo de uma sentença em FORTALL :

novo := velho + razão * 2

Gera um código simplificado para análise sintática posterior , por exemplo :

< id1 > < := > < id2 > < + > < id3 > < * > < int >

Cada elemento entre <> representa um único token. As abreviaturas id e int significam respectivamente identificadores e inteiros.

2.4.6. Reconhecimento de Tokens

Uma das técnicas de reconhecimento de *tokens* é o uso de diagramas de transição que são representações gráficas de um autômato finito. Um diagrama de transição é um grafo orientado em que os nós representam estados do autômato. No diagrama, os estados são representados por círculos, e o estado final, por um círculo duplo. Os caracteres lidos são representados sobre os arcos que identificam as transições do autômato finito. [Aho86]

A figura 2.4 apresenta um diagrama para reconhecimento dos *tokens* válidos na linguagem FORTALL.

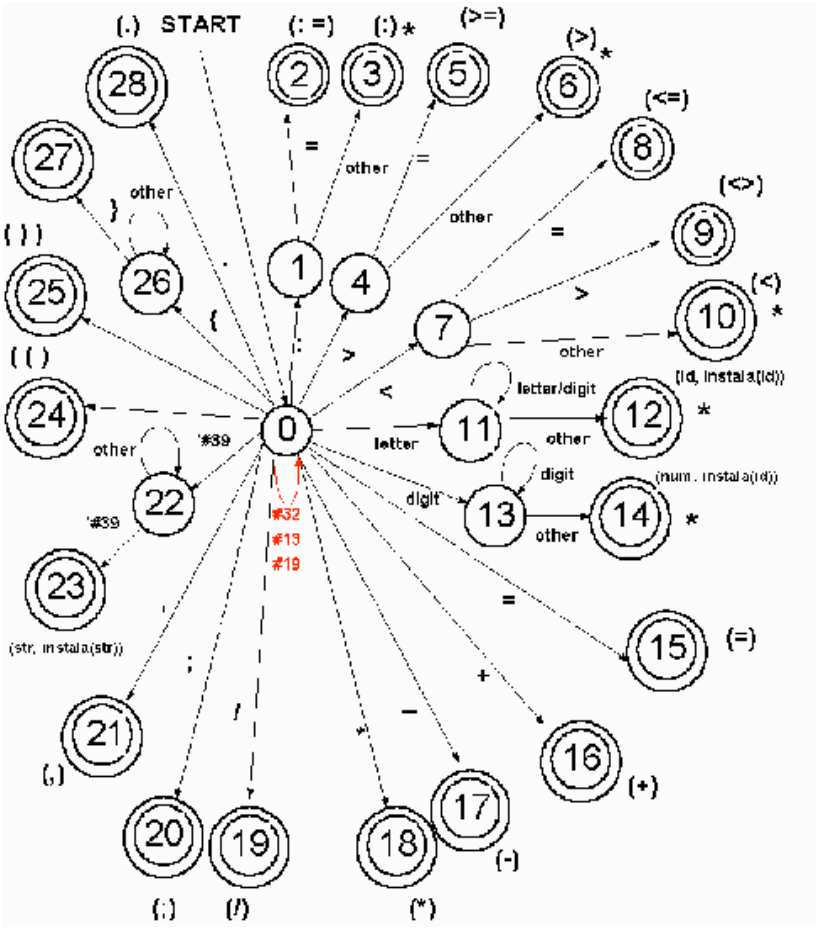

Figura 2.4: Processo do Analisador Léxico

Na figura 2.4, o asterisco em um estado significa que o último caracter lido será devolvido para a entrada, ou seja, o ponteiro do arquivo retornará um caracter atrás. Isso porque o último caracter pode ser o primeiro do próximo *token*. Cada estado final (com círculo duplo) retorna um *token* ao analisador sintático.

À medida que cada *token* for reconhecido, o mesmo poderá ser colocado numa tabela apropriada, como é o caso de identificadores, números e *strings*. O analisador léxico consegue também detectar erros do tipo "*símbolo não-reconhecido pela linguagem"*, ou "*variável duplicada na declaração"* (este último particularmente para a análise sintática).

2.5. Analisador Sintático

O *parser* é responsável por construir uma estrutura hierárquica a partir dos *tokens* recebidos do *scanner*. [Camp97]

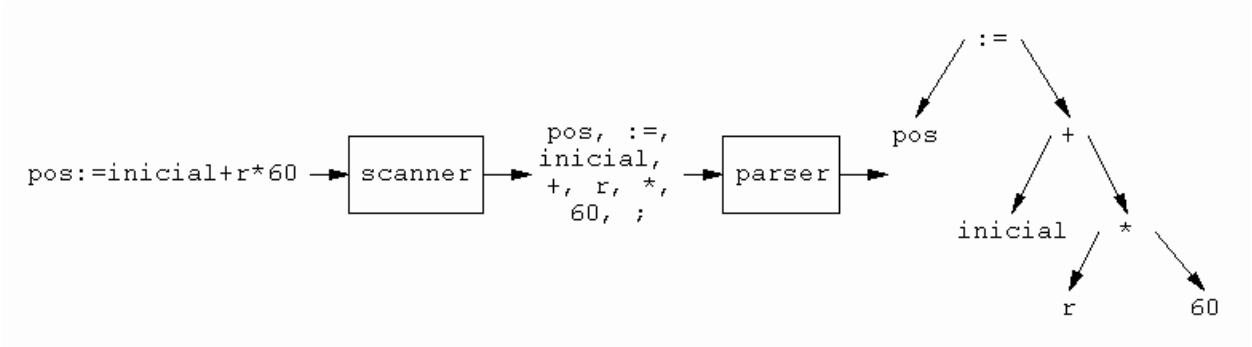

Figura 2.5: Funcionamento do Analisador Sintático

Caso um *token* não possa ser encaixado na árvore então o programa que está sendo analisado não corresponde à linguagem aceita pelo compilador. Neste caso diz-se que ocorreu um erro sintático.

Para a notificação de erros sintáticos, implementamos uma técnica de recuperação, conhecida como modo pânico. Ao descobrir um erro, o *parser* descarta os símbolos da entrada, um por vez, até que um *token* de sincronismo seja encontrado. Os *tokens* de sincronismo são o ponto e vírgula, as declarações *fim*, *início,* e o ponto final. Este método tem como vantagem a simplicidade de implementação. Em contrapartida, alguns dos erros que são detectados podem não ser verdadeiros.

Existem duas abordagens para a construção do *parser*: *top-down* e *bottom-up*. Esses termos referem-se à ordem em que os nós da árvore sintática são construídos, assim como na ordem em que as derivações da gramática da linguagem são usadas. Na abordagem *top-down* a árvore é construída de cima para baixo (da raiz para as folhas) e as derivações são realizadas da esquerda para a direita. Na abordagem *bottom-up* a árvore é construída de baixo para cima e as derivações (reduções) são realizadas da direita para a esquerda.

Para desenvolver o analisador sintático, foi escolhida a análise preditiva tabular [PRI 98], que pertence ao método de *parsers top-down*. A facilidade encontrada devido à vasta bibliografia e ainda o auxílio recebido por parte de colegas desta instituição de ensino e de outras, inclusive, tornou a escolha definitiva.

Analisar sintaticamente uma lista ou cadeia de tokens para encontrar a árvore sintática que tem como raiz o símbolo inicial da gramática e como nodos terminais, uma sucessão ordenada de símbolos que compõe a cadeia analisada. Em caso de não existir esta árvore sintática a cadeia não pertencerá a linguagem e o analisador sintático irá emitir uma mensagem correspondente de erro. Além de reconhecer as seqüências de tokens, e analisar sua estrutura, podem ser realizadas uma série de tarefas adicionais, como por exemplo:

- Recompilar informações dos distintos tokens e armazenar na tabela de símbolos;
- Realizar algum tipo de análise semântica, tal como a verificação de tipos;
- Gerar código intermediário;
- Avisar os erros detectados.

O processo de construção da árvore sintática pode levar a uma situação em que, pela aplicação incorreta de uma produção, seja impossível derivar a *string* desejada. Nesses casos existe a necessidade de voltar atrás na construção da árvore e usar outra produção. A esse mecanismo dá-se o nome de *backtracking*.

2.5.2. Analisador Preditivo Tabular (Top-Down)

O esquema padrão desse tipo de *parser* é mostrado na figura 2.7. [PRI 98]

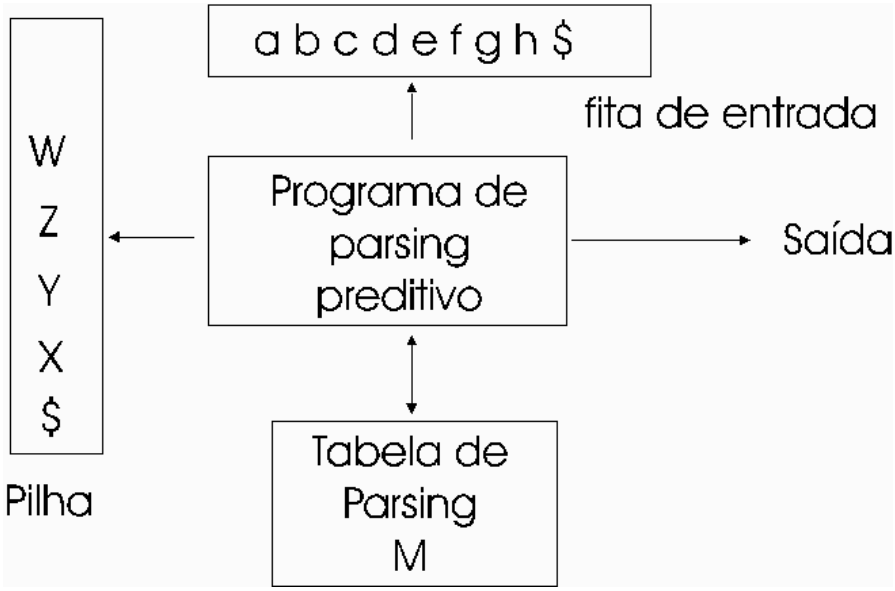

Figura 2.7: Analisador Preditivo Tabular

Para implementarmos esse *parser* tipo *top-down*, devemos construir uma tabela de *parsing* M, que contém as produções da gramática. Se a tabela não apresentar múltiplas entradas, então a gramática é dita LL(1) e o analisador pode ser construído de forma preditiva, ou seja, não necessitará de *backtracking*. O primeiro L de LL(k) significa que a *string* de entrada é lida da esquerda para a direita (*left to right*), o segundo L significa que é feita uma derivação mais à esquerda (*leftmost*) e o k significa o número de símbolos que devem ser lidos para que o *parser* saiba que produção aplicar.

Para a tabela M ser construída, devemos construir inicialmente duas funções especiais: **FIRST** e **FOLLOW**. *FIRST(A)* é o conjunto de terminais que iniciam *strings* derivadas de A. *FOLLOW(B)* é o conjunto de terminais que aparecem imediatamente à direita de B. Os algoritmos para obtenção das duas funções e para a posterior construção da tabela M são mostrados nas seções seguintes.

2.5.2.4. Problemas com o Exemplo da Seção 2.5.1

Para que o analisador sintático possa ser construído, algumas modificações devem ser realizadas na gramática definida anteriormente [Trem85]. Na versão atual, nossa gramática não separa operadores mais precedentes de menos precedentes, nem implementa a associatividade dos mesmos. Tampouco se preocupa com a recursividade à esquerda, como ocorre no caso a seguir:

X->Xa

O que ocorre neste caso é que o não-terminal X ficará sendo expandido para X eternamente, o que não é desejável. Elimina-se a recursividade à esquerda e embute-se a precedência e a associatividade de operadores facilmente. Como nosso método é preditivo, é necessário transformar a gramática, para que não tenha dúvida sobre qual produção usar. Usa-se então um algoritmo de fatoração. Um exemplo de fatoração de gramática seria o seguinte:

E -> abF | acF

Neste caso há um indeterminismo na escolha entre as duas alternativas da produção, já que ambos começam pelo mesmo terminal "a". Eliminando-se este indeterminismo por fatoração ficaria:

E -> aE'

E' -> bF | cF

Observe-se que o processo de fatoração introduz um novo símbolo não-terminal na gramática.

Outro problema presente é a ambigüidade. Uma gramática ambígua pode derivar uma mesma *string* por mais de uma forma, seja mais à esquerda ou mais à direita. Esse problema deve ser eliminado, pois o compilador somente deve trabalhar com linguagens não-ambíguas. Mais adiante, veremos a solução para isso (esse problema acontece em um único caso, na construção *Se...Então...Senão*).

2.5.2.5. Eliminação da Ambigüidade

Muitas vezes uma gramática ambígua pode ser rescrita para eliminar a ambigüidade. Como exemplo eliminaremos a ambigüidade da seguinte gramática: [Camp97]

S → *se* C *então* S

S → *se* C *então* S *senão* S

S → a

C → b

Esse caso é o conhecido "*dangling else*" e nas linguagens de programação é convencionado que o *senão* pertence ao *se* mais próximo. Logo, podemos transformar a gramática da seguinte forma:

S → X | Y

X → *se* C *então* X *senão* X | a

Y → *se* C *então* S | *se* C *então* X *senão* Y

C → b

Na nova gramática, o não terminal X representa estruturas *se-então-senão* casadas e o Y representa as estruturas não casadas. Esta nova gramática gera a mesma linguagem da anterior, porém sem o "dangling else", sendo portanto não ambígua.

2.5.2.6. Funcionamento do Analisador Sintático Preditivo Tabular

O funcionamento do analisador sintático é semelhante ao de um autômato de pilha, que é o reconhecedor de linguagens livres de contexto [AHO 86].

Para tanto, temos que ter uma pilha explícita, para empilharmos os terminais e não-terminais das produções utilizadas. O objeto pilha foi construído a partir do objeto *pilha*, desenvolvido posteriormente na linguagem C. Entre outros, foram implementados os métodos *Push* e *Pop* que, respectivamente, colocam e retiram elementos do topa da pilha.

Com o objeto Pilha já pronto, podemos entender o mecanismo de *parsing*. Primeiramente, são empilhados os símbolos $ (símbolo que determina o final do programa) e o símbolo inicial, nessa ordem. Pede-se então um *token* para o léxico. Começa então um processo que se desenvolve até que haja $ no topo da pilha e $ como *token* devolvido.

Se o topo da pilha for um terminal ou $, então

    Se o topo da pilha for igual à entrada então

        O topo é desempilhado;

        O *parser* pede novo *token* ao léxico;

A saída será o reconhecimento do antigo *token*

Senão ocorre erro (topo da pilha e entrada não combinam)

Senão (há um não-terminal no topo da pilha)

Se houver alguma produção na célula [topodapilha, *token*] na Tabela M então

O topo da pilha será retirado;

Pega-se a produção e empilham-se os elementos ao contrário;

A saída será a produção usada

Senão ocorre erro (entrada na tabela M é vazia)

Para que essa fase possa ser completada, devemos ainda construir uma árvore, chamada de árvore sintática que, de forma hierárquica, agrupe os vários *tokens* que forem reconhecidos nessa fase.

2.5.3. Montagem da Árvore Sintática

A montagem da árvore sintática ocorre durante o processo de *parsing*, quando determinadas produções da gramática fazem com que nós (itens constantes) ou folhas (itens variáveis) sejam criadas e encadeadas nos locais certos. [Price98] Para implementarmos esse procedimento, utilizamos uma estrutura de lista, para armazenar os nós e folhas criados. Todos os nós e folhas têm o mesmo formato, com 6 campos, que são:

1. *Token* propriamente dito (; := + * id, etc.);
2. Índice em alguma tabela (se o *token* não possuir tabela, o campo é preenchido com -1);
3. Ponteiro para a subárvore da esquerda;
4. Ponteiro para a subárvore da direita;
5. Linha em que o *token* foi encontrado na análise léxica (para localizarmos erros semânticos diretamente no editor, quando houver);
6. Resultado de um cálculo temporário feito nas folhas (necessário para a geração de código intermediário, principalmente para as expressões aritméticas e lógicas).

Cada estrutura do compilador, como atribuições, condições e outras, tem uma maneira diferente de ser encadeada o que inviabiliza uma explicação mais profunda sobre como as mesmas são acopladas na árvore.

Como exemplo, mostramos a construção da árvore sintática para o seguinte trecho de programa aceito pelo nosso compilador:

```
programa teste;
var x : inteiro;
início
x:=(x+5)*(x+4);
enquanto x>10 faça
      início
      x:=x-1;
      se x<2 então escrever(x)
      senão x:=x;
      fim;
fim.
```

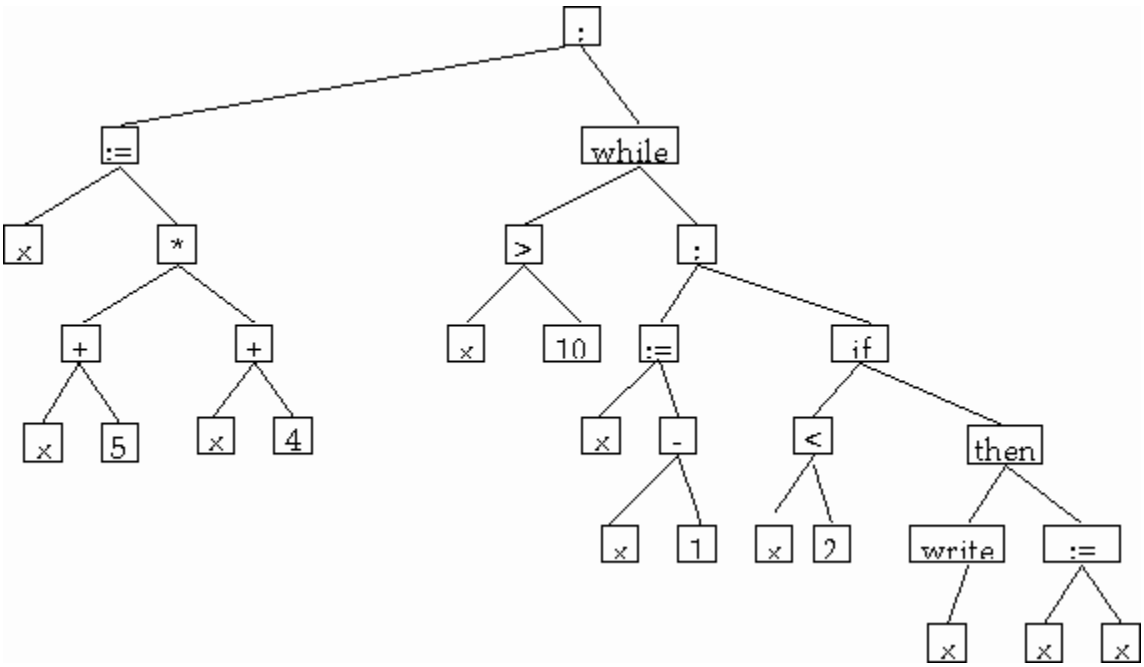

Figura 2.8: Árvore Sintática para Programa Teste.

2.6. Analisador Semântico

Após as análises léxica e sintática terem sido concluídas com sucesso, passa-se então à análise semântica. Este processo marca uma divisa na análise, sendo que deste ponto em diante não é mais utilizado o arquivo que contém o código fonte, mas sim a árvore recentemente gerada pelo processo de *parsing*. [Price98]

A análise semântica constitui-se, em linhas gerais, de um verificador de tipos. É de sua responsabilidade verificar se um operador contém operandos do mesmo tipo, se estruturas *Se* ou *Enquanto* possuem condições *lógicas*, se uma variável foi declarada, entre outras atribuições. Se alguma dessas condições não for satisfeita, o sistema reportará um erro semântico.

Por exemplo, se tivéssemos declarado uma variável **x** como i*nteiro*, uma variável **y** como l*ógico*, e quiséssemos fazer uma atribuição do tipo **x:=x+y**, certamente seria gerado um erro semântico, porque não é possível adicionarmos um valor lógico a um inteiro e atribuí-lo a outro inteiro.

Para podermos determinar esses erros, temos que "caminhar" na árvore, a partir do nó corrente, para a subárvore da esquerda e para a subárvore da direita, recursivamente.

2.7. Gerador de Código Intermediário

O gerador de código intermediário será acionado quando o programa for analisado léxica, sintática e semanticamente, e estiver correto do ponto de vista das três análises citadas. Neste ponto do processo, finda-se a parte de análise e entra-se no processo de síntese. [Camp97]

Para essa geração, deve-se percorrer a árvore em profundidade (*depth first*), para que o código seja gerado das folhas para os nós. A geração de código se dá através do uso de quádruplas, ou seja, quatro campos que já deixam o código numa representação adequada para a posterior tradução para código de máquina. Os quatro campos são: Operador, Operando 1, Operando 2 e Resultado. Além disso, temos um quinto campo, que guarda a linha da quádrupla, a qual será útil para endereços que necessitem de resolução mais adiante no processo (endereços não resolvidos).

2.8. Otimização de Código

Após o gerador de código intermediário encontra-se o otimizador que tem como objetivo otimizar o código intermediário quanto à eficiência da execução, tanto no sentido do uso de memória quanto no da velocidade de execução do programa objeto. [Trem85]

Sabe-se que a geração de código totalmente otimizado é um problema de difícil solução. De fato, em geral, os algoritmos de otimização de código não obtêm a melhor otimização possível.

2.9. Gerador de Código Objeto

O gerador de código objeto é a última parte de um compilador, conforme indicado na figura 2.9. No entanto, muitas vezes ainda existe mais uma etapa, que é a otimização de código objeto. Para não adicionarmos complexidade ao nosso trabalho, não implementamos otimizadores de códigos. [Aho86]

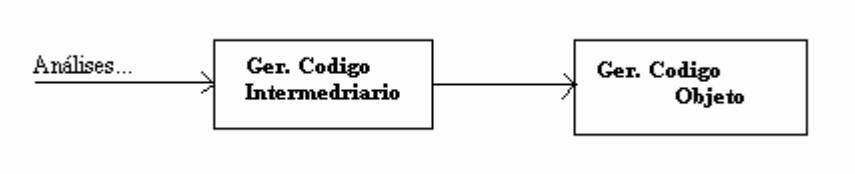

Figura 2.9: Final do Processo de um Compilador

A tarefa central desse gerador de código é transformar uma especificação de código intermediário vinda da etapa anterior para uma especificação de código *assembly*, para poder ser executado num PC. Como vimos, um gerador de código intermediário disponibiliza um conjunto de quádruplas, com as informações de *label*, operador, operandos e resultado.

3.1. O Compilador FORTALL

Este trabalho tem por finalidade construir um compilador de linguagem de programação, um mini-Pascal, o qual chamaremos de **FORTALL**. O compilador será feito em ambiente *DOS*, usando para tanto a linguagem de Programação ***C***++ da *Borland International*®.

O compilador tem por função receber um programa fonte escrito na linguagem FORTALL e analisá-lo sintaticamente, verificando a estrutura do programa e detectando os possíveis erros de programação do arquivo fonte. Se o programa estiver correto sintaticamente, o mesmo é traduzido para uma linguagem de programação, o Pascal, e compilado pela segunda vez, agora no compilador Pascal da *Borland International*® que fará a geração de código. Esta segunda compilação deve ocorrer sem erros, pois os erros sintáticos devem ser identificados pelo compilador FORTALL.

3.2. Especificações do Trabalho

Inicialmente, devemos ter em mente quais estruturas e dados o compilador deve aceitar. Por ser este um trabalho acadêmico, o nosso Pascal ficou extremamente reduzido, com as seguintes especificações:

Tipos:

Inteiro (-32768 .. 32767)

Lógico (1 *byte* [*Verdadeiro, Falso*])

Entrada/Saída:

Ler

Escrever

Laço:

Enquanto .. Faça

Condicional:

Se .. Então

Se .. Então .. Senão

Operadores Aritméticos:

+, -, *, /

Operadores Relacionais:

=,<> <, <=, >, >=

Comentários:

{ ... }

Atribuições:

:=

Strings:

'...'

A linguagem, inicialmente, não aceitará construções *repita .. até, goto, caso .. seja*, procedimentos, funções, vetores, seqüência de caracteres*, registros*, nem outros tipos de dados.

Para implementarmos uma gramática que reconheça essas especificações, foi utilizado duas ferramentas: EBNF (*Extended Backus-Naur Form*) e Notação Formal, esta última usando GLC (Gramática Livre de Contexto).

3.2.1. Elementos Básicos e Especificações de EBNF

Conjunto de Símbolos Terminais: são os símbolos que formam a linguagem. Ex.: *Início, Enquanto*, etc.;

Conjunto de Símbolos Não-Terminais: são os símbolos que não pertencem à linguagem, são usados para representar definições intermediárias usadas nas produções da gramática. São cercados por '<' e '>';

Conjunto de Produções: é a definição de um símbolo não-terminal.

Forma Geral:

X::=Y

onde

X - símbolo não-terminal e Y - seqüência de símbolos

Símbolo Alvo (ou Inicial): não-terminal especial.

Obs.:

Todo símbolo não-terminal deve aparecer na esquerda de alguma produção.

O símbolo alvo não pode aparecer na direita de produção.

Pode-se representar ainda:

Alternativa: | *Opcional*: [ ] *Repetição*: { }

Obedecendo essas regras, a gramática implementada em EBNF tem 14 não-terminais e pode ser vista na tabela abaixo.

<prog>::=programa id; [<declarações>] início <lista comandos> fim.
<declarações>::=var id {,id} : <tipo>; {id {,id} : <tipo>;}
<tipo>::=inteiro | lógico
<lista comandos>::=<comando> ; {<comando>;}
<comando>::=<atribuição> | <leitura> | <escrita> | <composto> | <condicional> | <repetição>
<atribuição>::=id := <expr>
<leitura>::=ler (id {,id}) | ler [(id {,id})]

<escrita>::=Escrever (<stringvar> {,<stringvar>}) | Escrever [(<stringvar> {,<stringvar>})]
<composto>::=Início <lista comandos> Fim
<condicional>::=Se <exprLogico> Então <comando> [Senão <comando>]
<repetição>::=Enquanto <exprLogico> Faça <comando>
<expr>::=<expr> + <expr> | <expr> - <expr> | <expr> * <expr> | <expr> / <expr> | - <expr> | (<expr>) | id | num
<exprLogico>::=<expr> < <expr> | <expr> <= <expr> | <expr> > <expr> | <expr> >= <expr> | <expr> = <expr> | <expr> <> <expr> | id
<stringvar>::=str | <expr>

Tabela 3.1: Gramática Implementada em EBNF

3.2.2. Elementos Básicos e Especificações de Notação Formal

Uma gramática G é definida pela quádrupla:

G = (N,T,P,S)

onde:

N = conjunto finito de não-terminais;

T = conjunto finito de terminais;

P = conjunto finito de regras de produção;

S = símbolo inicial.

3.2.3. Especificação de GLC

Para G sendo a quádrupla (N,T,P,S) e V+ sendo N unido com T exceto o vazio temos:

P = {a -> b | a pertence a N e b pertence a V+}

Obedecendo essas especificações, foi construído uma gramática em notação formal, que pode ser vista na tabela abaixo. Foram utilizados 17 não-terminais.

S → **Programa id**; *S'*

S’	→	*DM.* | *M.*
D	→	**var** *V*
V	→	*I*: *T*; *V’*
V’	→	*V* | ε
T	→	**Inteiro** | **lógico**
I	→	**id** *I’*
I’	→	,*I* | ε
L	→	*N*;*L’*
L’	→	*L* | ε
C	→	**id** := *A* | **ler** *R* | **escrever** *W* | *M* | **enquanto** *B* **faca** *N*
A	→	*E* | **VERDADEIRO** | **FALSO**
R	→	**(***I***)** | ε
W	→	**(***F***)** | ε
F	→	*GF’*
F’	→	,*F* | ε
G	→	**str** | *E*
M	→	**Inicio** *M’*
M’	→	*L* **fim**
N	→	**se** *B* **então** *N N’* | *C*
N’	→	**senão** *N* | ε
E	→	*XE’*
E’	→	+ *XE’* | - *XE’* | ε
X	→	YX’
X’	→	* *YX’* | / *YX’* | ε
Y	→	- *K* | *K*
K	→	**(***E***)** | **id** | **num**
B	→	*EB’*

B’ → = *E* | < E | <> E | <= E | > *E* | >= *E* | ε

Tabela 3.2: GLC do Compilador FORTALL

Com relação a EBNF anteriormente mostrada, pode-se fazer a seguinte ligação EBNF>GLC:

S=<Programa>
W=<escrita>
D=<declarações>
F=lista de <stringvar>
V=lista de variáveis na declaração
G=<stringvar>
T=<tipo>
M=<composto>
I=lista de ids
N=<condicional>
L=<lista comandos>
P=<repetição>
C=<comando>
E=<expr>
A=<atribuição>
B=<exprLogico>
R=<leitura>

Tabela 3.3: Ligação EBNF -> GLC

3.9. Tabela de símbolos

É onde são armazenadas todas as informações referentes às variáveis e objetos do programa que está compilando. Em certos momentos do processo de compilação devemos fazer uso de certas informações referentes aos identificadores e aos números que aparecem

na sentença, como seu tipo, sua posição de armazenamento na memória, etc. Estas informações é que são armazenadas na tabela de símbolos.

3.10. Rotinas de Erros

Estão incluídas em cada um dos processos de compilação (análise léxica, sintática e semântica) e se encarregam de informar os erros que encontram no programa fonte. Por exemplo, o analisador semântico poderia emitir um erro ou uma mensagem quando detectasse uma diferença nos tipos de uma expressão.

3.11. Conceitos de Consistências do Ambiente FORTALL

Os tokens são as unidades significativas pequenas de texto num programa em FORTALL e são categorizadas como símbolos especiais, identificadores, dígitos e string constante.

Um programa em FORTALL é composto de símbolos e separadores, onde um separador é qualquer espaço em branco ou comentário. Dois tokens adjacentes devem ser separados por um ou mais separadores, se cada token for uma palavra reservada, um identificador ou um dígito.

3.11.1. Tokens Especiais e Palavras Reservadas

FORTALL utiliza os seguintes caracteres da tabela ASCII :

Letras - O alfabeto inglês, A.....Z e a.....z

Dígitos - Os numerais arábicos 0.....9

Lacunas - O caracter de espaço (ASCII 32) e todos caracteres de controle de ASCII (ASCII 0.....31), incluindo o final-de-linha ou (ASCII 13, caractere de retorno).

A seguir, na figura 3.3, são apresentados diagramas de sintaxe por letra, algarismo e algarismos hexadecimais. Caminhos alternativos são freqüentemente encontrados.

Os nomes em caixas retangulares ficam para construções reais. As palavras reservadas, operadores e pontuação que são apresentados nas caixas circulares são os termos reais a serem usados no programa.

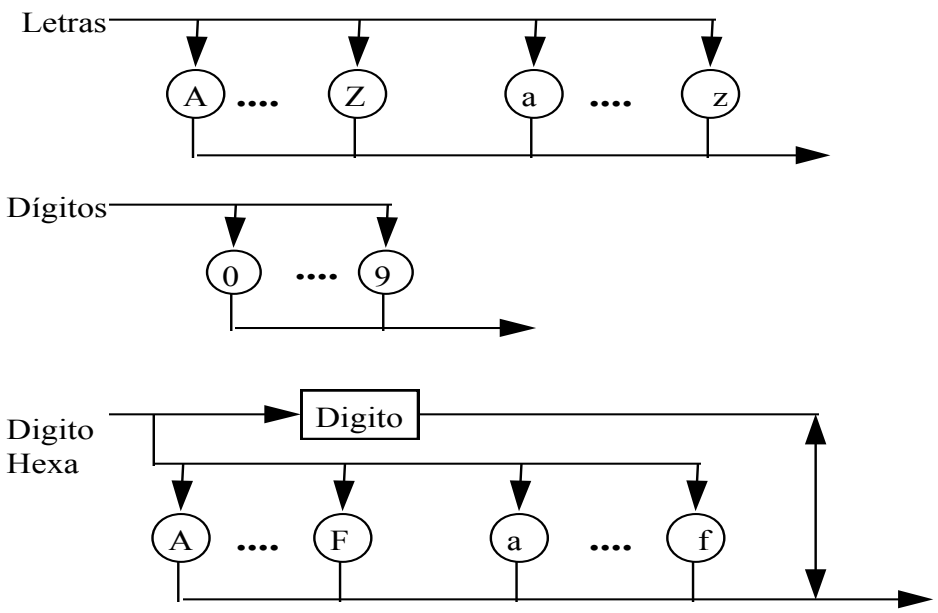

Figura 3.3: Diagramas de Sintaxe

Tokens especiais e palavras reservadas possuem significados fixos. Estes são os Tokens especiais :

+ - * / = < > [ ] . , ( ) : ; { }

Também são considerados símbolos especiais os seguintes pares de caracteres :

< = > = : = (* *)

Alguns símbolos especiais são também operadores. Um colchete esquerdo ([) é equivalente ao par esquerdo de parêntese. Do mesmo modo, colchete direito (]) equivale ao caracter par de parêntese direito (]).

A tabela 3.4 apresenta as palavras reservadas do FORTALL.

Início	Fim	Programa	Var
Enquanto	Faça	Se	Então
Senão	VERDADEIRO	FALSO	Inteiro
Lógico	Ler	Escrever	

Tabela 3.4 - Palavras Reservadas do FORTALL

3.11.2. Identificadores

Identificador significa constantes, variáveis, procedimentos, funções, unidades, programas e campos em registros.

Um identificador pode ser de qualquer comprimento, mas só os 63 primeiros caracteres são significativos. Um identificador deve começar com uma letra ou outro símbolo especial. Não deverá começar com dígitos ou espaço. A figura 3.4 apresenta o diagrama de identificadores:

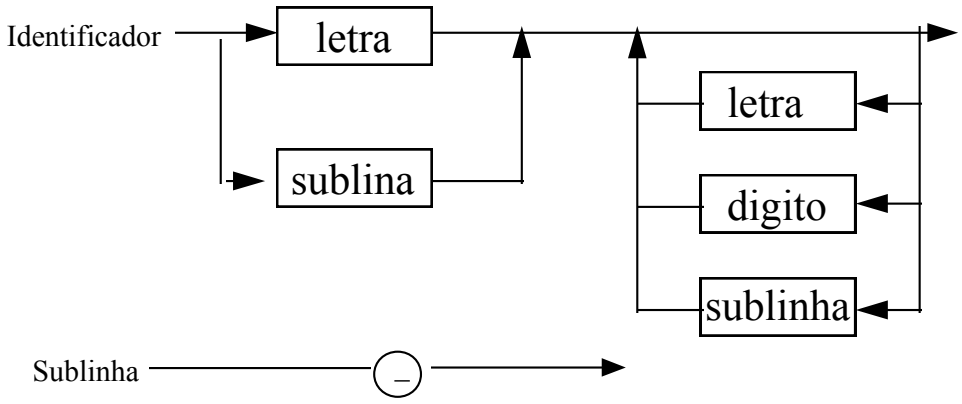

Figura 3.4: Diagrama de Identificadores

3.11.3. String De Caracteres

Uma string é uma seqüência de zero ou mais caracteres escritos em uma linha de programa que fica reservado entre apóstrofos. Toda string que se encontra no meio de apóstrofos é considerada String nula, ou seja, é desconsiderada pelo compilador, o diagrama de sintaxe é apresentado na figura 3.6:

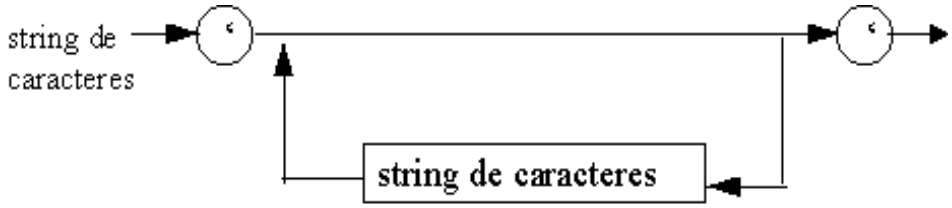

Figura 3.6 - Diagrama de string de caracteres

Abaixo seguem alguns exemplos de strings:

‘Turbo’

‘;’

‘ ‘

3.11.4. Declaração De Constantes

Uma constante quando declarada é marcada dentro do bloco pelo identificador, contendo a sua declaração, conforme é apresentado na figura 3.7:

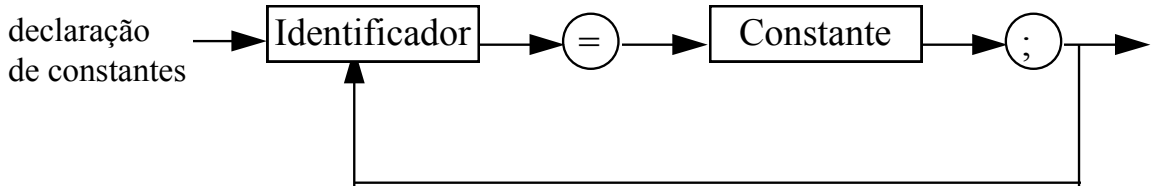

Figura 3.7 - Diagrama de declaração de constantes

3.11.5. Comentários

Os comentários são ignorados pelo compilador. Os símbolos usados para identificar uma comentários são :

{ }

Exemplos :

{ Qualquer texto que possua um símbolo chave aberto esquerdo e outro esquerdo fechado}

3.12. Exemplo de Programa do Compilador FORTALL

Aqui está um exemplo de um programa aceito pela linguagem FORTALL, com alguns recursos que o compilador fornece ao seu usuário.

```
programa testeMedia;
var
  cont, x, Media : inteiro;
início
   cont:=1;
   Media:=0;
   enquanto (cont<>5) faca
     início
        escrever('Valor ',cont,': ');
        ler(x);
        Media := Media + x;
        cont:=cont+1;
    fim;
```

```
  escrever('Soma dos valores: ')
  escrever(Media);
  escrever('Media dos valores: ')
  escrever(Media/5);
fim.
```