

**URICER – Universidade Regional Integrada do Alto Uruguai
e das Missões – Campus de Erechim**

Apostila de COMPILADORES

Erechim, agosto de 2001.

SUMÁRIO

1	CONCEITOS BÁSICOS (revisão)	4
2	LINGUAGENS E SUAS REPRESENTAÇÕES	6
2.1	<i>Conceito</i>	6
2.2	<i>Tipos</i>	7
2.3	<i>Especificação de uma linguagem</i>	8
3	INTRODUÇÃO À COMPILAÇÃO	11
3.1	<i>Tipos de Compilação</i>	11
3.2	<i>Interpretação pura</i>	13
3.3	<i>Sistemas de Interpretação Híbridos</i>	15
3.4	<i>Construção de Compiladores</i>	16
3.5	<i>Processadores de linguagens</i>	16
3.6	<i>Estrutura geral de um compilador</i>	19
3.6.1	<i>Analisador léxico (ou scanner)</i>	19
3.6.2	<i>Analisador sintático (ou parser)</i>	20
3.6.3	<i>Analisador semântico</i>	21
3.6.4	<i>Otimização de código</i>	22
3.6.5	<i>Geração de código</i>	22
3.7	<i>Formas de construção de um compilador</i>	25
3.8	<i>Ferramentas para a construção de compiladores</i>	26
4	ANALISADOR LÉXICO	27
4.1	<i>Função</i>	27
4.2	<i>Reconhecimento de tokens: autômato finito</i>	27
4.3	<i>Implementação</i>	28
5	ANALISADOR SINTÁTICO	29
5.1	<i>Função</i>	29
5.2	<i>Especificação das regras sintáticas: gramática livre de contexto</i>	29
5.2.1	<i>Notações</i>	29
5.2.2	<i>Árvore de derivação ou árvore sintática</i>	30
5.2.3	<i>Análise sintática descendente e ascendente</i>	33
5.3	<i>Autômato de pilha</i>	36
5.4	<i>Tipos de analisadores sintáticos</i>	36
5.4.1	<i>Analisador sintático ascendente (bottom-up)</i>	36
5.4.2	<i>Analisador sintático descendente (top-down)</i>	37
5.5	<i>A Implementação da Análise Shift-Reduce através de pilhas</i>	38
5.5.1	<i>O Analisador de gramáticas de precedência de operadores</i>	39
5.6	<i>Simplificações de Gramáticas Livres de Contexto</i>	41
5.6.1	<i>Símbolos Inúteis ou Improdutivos</i>	41
5.6.2	<i>ϵ - Produções</i>	44
5.6.3	<i>Produções Unitárias</i>	47

5.6.4	Fatoração.....	47
5.6.5	Eliminação de Recursão à Esquerda.....	49
5.7	<i>Tipos Especiais de GLC</i>	50
5.8	<i>Principais Notações de GLC</i>	51
5.9	<i>Conjuntos First e Follow</i>	51
5.9.1	Conjunto First.....	51
5.9.2	Conjunto Follow.....	53
Lista de Exercícios de Simplificação de GLC's		56
BIBLIOGRAFIA.....		61

1 CONCEITOS BÁSICOS (REVISÃO)

Segundo MENEZES (1997), os estudos sobre linguagens formais iniciaram-se na década de 50 com o objetivo de definir matematicamente as estruturas das linguagens naturais. No entanto, verificou-se que a teoria desenvolvida aplicava-se ao estudo das linguagens de programação.

A teoria de linguagens formais engloba, basicamente, o estudo das características, propriedades e aplicações das linguagens formais, bem com a forma de representação da estrutura (sintaxe) e determinação do significado (semântica) das sentenças das linguagens. Segundo FURTADO (1992), a importância desta teoria é dupla: tanto apóia outros aspectos teóricos da teoria da computação, tais como decibilidade, computabilidade, complexidade, etc. como fundamenta diversas aplicações computacionais, como por exemplo processamento de linguagens, reconhecimento de padrões, modelagem de sistemas, etc. Mas, o que é uma linguagem?

- uma linguagem é uma forma de comunicação, usada por sujeitos de uma determinada comunidade;
- uma linguagem é o conjunto de SÍMBOLOS e REGRAS para combinar esses símbolos em sentenças sintaticamente corretas.
- "uma linguagem é formal quando pode ser representada através de um sistema com sustentação matemática" (PRICE; EDELWEISS, 1989).

Assim sendo, são necessários conceitos matemáticos para o estudo das linguagens formais.

- **Símbolo**

Um símbolo é uma entidade abstrata básica sem definição formal. São exemplos de símbolos as letras, os dígitos, etc. Símbolos são ordenáveis lexicograficamente e, portanto, podem ser comparados quanto à igualdade ou precedência. Por exemplo, tomando as letras dos alfabetos, tem-se a ordenação $A < B < C < \dots < Z$. A principal utilidade dos símbolos está na possibilidade de usá-los como elementos atômicos em definições de linguagens.

- **Sentença (ou palavra)**

Uma sentença (ou palavra) é uma seqüência finita de símbolos. Sejam P, R, I, M , e A símbolos, então $PRIMA$ é uma sentença. As sentenças vazias, representadas por ϵ , é uma sentença constituída por nenhum símbolo.

- **Tamanho de uma sentença**

O tamanho (ou comprimento) de uma sentença w , denotado por $|w|$, é dado pelo número de símbolos que compõem w . Assim, o tamanho da sentença $PRIMA$ é 5 e o tamanho da sentença vazia é 0.

- **Alfabeto**

Um alfabeto, denotado por V , é um conjunto finito de símbolos. Assim, considerando os símbolos dígitos, letras, etc., tem-se os seguintes alfabetos $V_{\text{binário}} = \{0, 1\}$; $V_{\text{vogais}} = \{a, e, i, o, u\}$. É importante observar que um conjunto vazio também pode ser considerado um alfabeto.

O fechamento reflexivo de um alfabeto V , denotado por V^* , é o conjunto infinito de todas as sentenças que podem ser formadas com os símbolos de V , inclusive a sentença vazia. O fechamento transitivo de um alfabeto V , denotado por V^+ , é dado por $V^* - \{\epsilon\}$. Seja V o alfabeto dos dígitos binários, $V = \{0, 1\}$. O fechamento transitivo de V é $V^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$, enquanto que o fechamento reflexivo de V é $V^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$.

2 LINGUAGENS E SUAS REPRESENTAÇÕES

2.1 CONCEITO

Uma linguagem formal L é um conjunto de sentenças formadas por símbolos tomados de algum alfabeto V . Isto é, uma linguagem sobre o alfabeto V é um subconjunto de V^* ($L \subseteq V^*$). Assim, por exemplo, o conjunto de sentenças válidas da língua portuguesa poderia ser definido extensionalmente como um subconjunto de $\{a, b, c, \dots, z\}^+$.

Uma linguagem pode ser:

- finita: quando é composta por um conjunto finito de sentenças. Seja $V = \{a, b\}$ um alfabeto, L_1, L_2 e L_3 linguagens definidas conforme segue:

$L_1 = \{w \mid w \in V^* \wedge |w| < 3\}$, ou seja, L_1 é uma linguagem constituída por todas as sentenças de tamanho menor que 3 formadas por símbolos de V . Portanto, $L_1 = \{\epsilon, a, b, aa, ab, ba, bb\}$

$L_2 = \emptyset$

$L_3 = \{\epsilon\}$

As linguagens L_2 e L_3 são diferentes.

- infinita: quando é composta por um conjunto infinito de sentenças. Seja $V = \{a, b\}$ um alfabeto, L_1, L_2 e L_3 linguagens definidas conforme segue:

$L_1 = \{w \mid w \in V^* \wedge |w| \text{ MOD }^1 2 = 0\}$, ou seja, L_1 é uma linguagem constituída por todas as sentenças de tamanho par formadas por símbolos de V . Portanto, $L_1 = \{\epsilon, aa, ab, ba, bb, aaaa, aaab, aaba, \dots\}$

$L_2 = \{w \mid w \text{ é uma palíndrome}^2\}$. Portanto $L_2 = \{\epsilon, a, b, aa, bb, aba, baab, \dots\}$

$L_3 =$ linguagem de programação PASCAL, ou seja, o conjunto infinito de programas escritos na linguagem de programação em questão.

Como definir/representar uma linguagem? Uma linguagem finita pode ser definida através da enumeração das sentenças constituintes ou através de uma descrição algébrica. Uma linguagem infinita deve ser definida através de representação finita. Reconhedores ou sistemas geradores são dois tipos de representações finitas. Um reconhedor é um dispositivo formal usado para verificar se uma determinada sentença pertence ou não à linguagem. São exemplos de reconhedores autômatos finitos, autômatos de pilha e máquinas de Turing. Um sistema gerador é um dispositivo formal usado para gerar de forma sistemática as sentenças de uma dada linguagem. São exemplos de sistemas geradores as gramáticas.

Todo reconhedor e todo sistema gerador pode ser representado por um algoritmo.

¹ MOD é uma operação que representa o resto da divisão inteira.

² Uma palíndrome é um palavra que tem a mesma leitura da esquerda para a direita e vice-versa.

2.2 TIPOS

Noam Chomsky definiu uma hierarquia de linguagens como modelos para linguagens naturais. As classes de linguagens são: regular, livre de contexto, sensível ao contexto e irrestrita. As inclusões dos tipos de linguagens são apresentadas na figura abaixo.

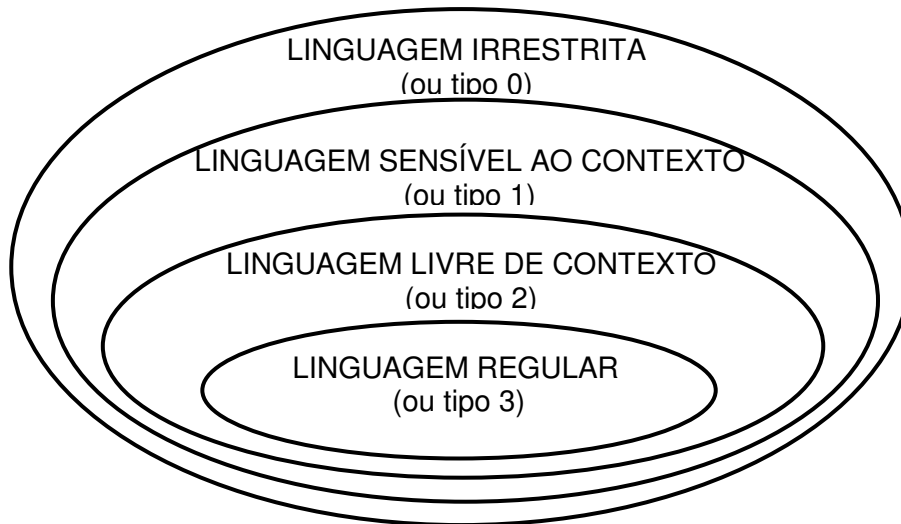


FIGURA 8: hierarquia de Chomsky

Pode-se ver que uma linguagem do tipo 3 é também do tipo 2, uma linguagem do tipo 2 é também do tipo 1, e uma linguagem do tipo 1 é também do tipo 0. Estas inclusões são estritas, isto é, existem linguagens do tipo 0 que não são do tipo 1, existem linguagens do tipo 1 que não são do tipo 2 e existem linguagens do tipo 2 que não são do tipo 3.

As **linguagens regulares** constituem um conjunto de linguagens bastante simples. Essas linguagens podem ser reconhecidas por autômatos finitos, geradas por gramáticas regulares e facilmente descritas por expressões simples, chamadas expressões regulares. Segundo MENEZES (1997), algoritmos para reconhecimento e geração de linguagens regulares são de grande eficiência, fácil implementação e pouca complexidade. HOPCROFT; ULLMAN (1979) afirmam que vários são os problemas cujo desenvolvimento pode ser facilitado pela conversão da especificação feita em termos de expressões regulares na implementação de um autômato finito correspondente. Dentre as várias aplicações, podemos citar: analisadores léxicos e editores de texto. Para descrever precisamente a regra de formação (padrão) de *tokens* mais complexos de linguagens de programação, tais como identificadores, palavras reservadas, constantes e comentários, usa-se a notação das expressões regulares, as quais podem ser automaticamente convertidas em autômatos finitos equivalentes cuja a implementação é trivial. A operação de busca e substituição de palavras (ou trechos de palavras) também pode ser realizada através da implementação de um autômato finito a partir da especificação da expressão regular correspondente.

A importância das **linguagens livres de contexto** reside no fato de que especificam adequadamente as estruturas sintáticas das linguagens de programação, tais como parênteses balanceados, construções aninhadas, entre outras. A maioria das linguagens de programação pertence ao conjunto das linguagens livre de contexto e pode ser analisada por algoritmos eficientes. Essas linguagens podem ser reconhecidas por autômatos de pilha e geradas por gramáticas livre de contexto (GLC). Segundo FURTADO (1992), dentre as várias aplicações dos conceitos de linguagens livre de contexto, podemos citar: definição e

especificação de linguagens de programação; implementação eficiente de **analísadores sintáticos**; implementação de tradutores de linguagens e processadores de texto em geral; estruturação formal e análise computacional de linguagens naturais.

Segundo MENEZES (1997), as **linguagens sensíveis ao contexto e irrestritas** "permitem explorar os limites da capacidade de desenvolvimento de reconhecedores ou geradores de linguagens, ou seja, estuda a solucionabilidade do problema da existência de algum reconhecedor ou gerador para determinada linguagem". Essas linguagens podem ser respectivamente reconhecidas por *máquinas de Turing limitadas* e geradas por *gramáticas sensíveis ao contexto*(GSC); reconhecidas por *máquinas de Turing* e geradas por *gramáticas irrestritas*.

MENEZES (1997) observa que nem sempre as linguagens de programação são tratadas adequadamente na hierarquia de Chomsky. Assim, para a especificação de determinadas linguagens de programação pode-se fazer necessário o uso de outros formalismos como por exemplo gramática de grafos.

2.3 ESPECIFICAÇÃO DE UMA LINGUAGEM

Foi dito que uma linguagem L é qualquer subconjunto de sentenças sobre um alfabeto V. Mas, qual subconjunto é esse, como defini-lo? Uma gramática é um dispositivo formal usado para definir qual subconjunto de V^* forma determinada linguagem. A gramática define uma estrutura sobre um alfabeto de forma a permitir que apenas determinadas combinações de símbolos sejam consideradas sentenças.

O que é GRAMÁTICA? É um sistema gerador de linguagens; é um sistema de reescrita; é uma maneira finita de descrever uma linguagem; é um dispositivo formal usado para especificar de maneira finita e precisa uma linguagem infinita.

- **Gramática**

Formalmente uma gramática G é definida como sendo uma quádrupla $G = (N, T, P, S)$, onde: **N** é um conjunto finito de símbolos denominados símbolos não-terminais, usados na descrição da linguagem; **T** é um conjunto finito de símbolos denominados símbolos terminais, os quais são os símbolos propriamente ditos; **P** é conjunto finito de pares (α, β) denominados regras de produção (ou regras gramaticais) que relacionam os símbolos terminais e não-terminais; **S** é o símbolo inicial da gramática pertencente a **N**, a partir do qual as sentenças de uma linguagem podem ser geradas.

As regras gramaticais são representadas por $\alpha ::= \beta$ ou $\alpha \rightarrow \beta$, onde α e β são sentenças sobre V, com α envolvendo pelo menos um símbolo pertencente a V_N . Uma seqüência de regras de produção da forma $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$, ou seja, com a mesma componente do lado esquerdo, pode ser abreviada como uma única produção na forma: $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$. O significado de uma regra de produção $\alpha \rightarrow \beta$ é α produz β ou α é definido por β .

Abaixo se encontra um exemplo de uma gramática:

EXEMPLO 1: a linguagem dos números inteiros sem sinal é gerada pela seguinte gramática G:

$$\begin{aligned} G &= (N, T, P, S) \\ \text{onde} \\ V_N &= \{N, D\} \\ V_T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ P &= \{ \\ &\quad N \rightarrow DN \mid D \\ &\quad D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ &\quad \} \\ S &= N \end{aligned}$$

Observa-se que a gramática apresentada no EXEMPLO 1 não é regular.

- **Gramáticas Regulares (tipo 3)**

Uma gramática G é de tipo 3, ou *regular*, se cada produção é da forma :

$$\begin{aligned} A &::= aB, \\ A &::= a \text{ ou} \\ A &::= \varepsilon, \text{ onde } A \text{ e } B \text{ são não-terminais e } a \text{ é um terminal.} \end{aligned}$$

Exemplo de Gramática Regular:

$$\begin{aligned} S &::= 0A \mid 1A \\ A &::= 0B \mid 1A \\ B &::= 0A \mid 1B \mid \varepsilon \end{aligned}$$

- **Gramáticas Livres de Contexto - GLC (tipo2)**

As linguagens livres de contexto são definidas por **gramáticas livres de contexto** (GLC) que podem ser utilizadas para especificar a parte sintática de uma linguagem de programação. Uma gramática livre de contexto é uma quádrupla $G = (V_N, V_T, P, S)$, onde: $P = \{A \rightarrow \alpha \mid A \in V_N, \alpha \text{ é uma sentença em } (V_N \cup V_T)^*\}$. Em outras palavras, uma gramática livre de contexto admite apenas regras de produção cujo lado esquerdo contém exatamente um não-terminal. Segundo MENEZES (1997), significa que o não-terminal "**A** deriva α **sem depender ('livre')** de qualquer análise dos símbolos que antecedem ou sucedem A ('contexto') na sentença que está sendo derivada".

EXEMPLO: a linguagem $L = \{x \mid x \in \{a, b\}^+ \wedge x \text{ segue o padrão de formação } a^n b^n \wedge n \geq 0\}$ é gerada pela seguinte gramática livre de contexto:

$$S \rightarrow a S b \mid \varepsilon$$

Pode-se estabelecer uma analogia entre esta linguagem e os blocos estruturados do tipo *BEGIN END*, ou as expressões com parênteses balanceados na forma $(^n)^n$.

- **Gramáticas Sensíveis ao Contexto - GSC (tipo 1)**

1. O Símbolo inicial **S** não aparece no lado direito de nenhuma produção, e
2. Para cada produção $\alpha_1 ::= \alpha_2$, é verdade que $|\alpha_1| \leq |\alpha_2|$, ou seja, o lado esquerdo da produção é \leq ao lado direito (com exceção da regra $S ::= \epsilon$), então se diz que **G** é uma gramática do tipo 1, ou *Gramática Sensível ao Contexto*.

EXEMPLO: a linguagem $L = \{x \mid x \in \{a, b, c\}^+ \wedge x \text{ segue o padrão de formação } a^n (b \mid c)^n \wedge n \geq 1\}$ é gerada pela seguinte gramática sensível ao contexto:

$$\begin{aligned} S &\rightarrow a S B C \mid a B C \\ B C &\rightarrow C B \\ C B &\rightarrow B C \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

- **Linguagens Irrestritas (tipo 0)**

As linguagens irrestritas representam todas as linguagens que podem ser reconhecidas mecanicamente e em um tempo finito. São definidas por **gramáticas irrestritas** (GI). Uma gramática irrestrita é uma quádrupla $G = (N, T, P, S)$ que não possui qualquer restrição quanto à forma das regras de produção. Qualquer uma das gramáticas especificadas anteriormente é um exemplo de uma gramática irrestrita.

Observa-se que uma linguagem com dois ou mais símbolos terminais que possuem uma relação quantitativa e posicional não pode ser representada por GR; uma linguagem com três ou mais símbolos terminais que possuem uma relação quantitativa e posicional não pode ser representada por GLC; quanto mais abrangente o tipo da gramática mais complexa é a análise.

3 INTRODUÇÃO À COMPILAÇÃO

3.1 TIPOS DE COMPILAÇÃO

Os programas podem ser traduzidos para linguagem de máquina, a qual pode ser executada diretamente no computador. Isso é chamado de implementação compilada. Esse método tem a vantagem de uma execução de programa muito rápida, assim que o processo de tradução for concluído. A maioria das implementações de linguagens de produção como C, COBOL e Ada dá-se por meio de compiladores.

A linguagem que um compilador traduz é chamada de linguagem-fonte. O processo de compilação desenvolve-se em diversas etapas, a maioria das quais mostrada na figura 3.1.

O analisador léxico reúne os caracteres do programa-fonte em unidades léxicas que são os identificadores, as palavras especiais, os operadores e os símbolos de pontuação. O analisador léxico ignora os comentários no programa-fonte, porque eles não têm nenhuma utilidade para o compilador.

O analisador sintático pega as unidades do analisador léxico e usa-as para construir estruturas hierárquicas chamadas árvores de análise, as quais representam a estrutura sintática do programa (árvores de derivação, lembra?). Em muitos casos, nenhuma estrutura de árvore de análise real é construída; ao contrário, a informação que seria necessária para construí-la é gerada e usada.

O gerador de código intermediário produz um programa em uma linguagem diferente, no nível intermediário entre o programa-fonte e a saída final do compilador, o programa em linguagem de máquina (note que as palavras *linguagem* e *código* muitas vezes são usadas de maneira intercambiável). As linguagens intermediárias, às vezes, parecem-se muito com as linguagens *assembly*. Em outros casos, o código intermediário está em um nível bem mais alto que o *assembly*. O analisador semântico faz parte integralmente do gerador de código intermediário e verifica se há erros difíceis, se não impossíveis, de serem detectados durante a análise sintática, como por exemplo, erros de tipo.

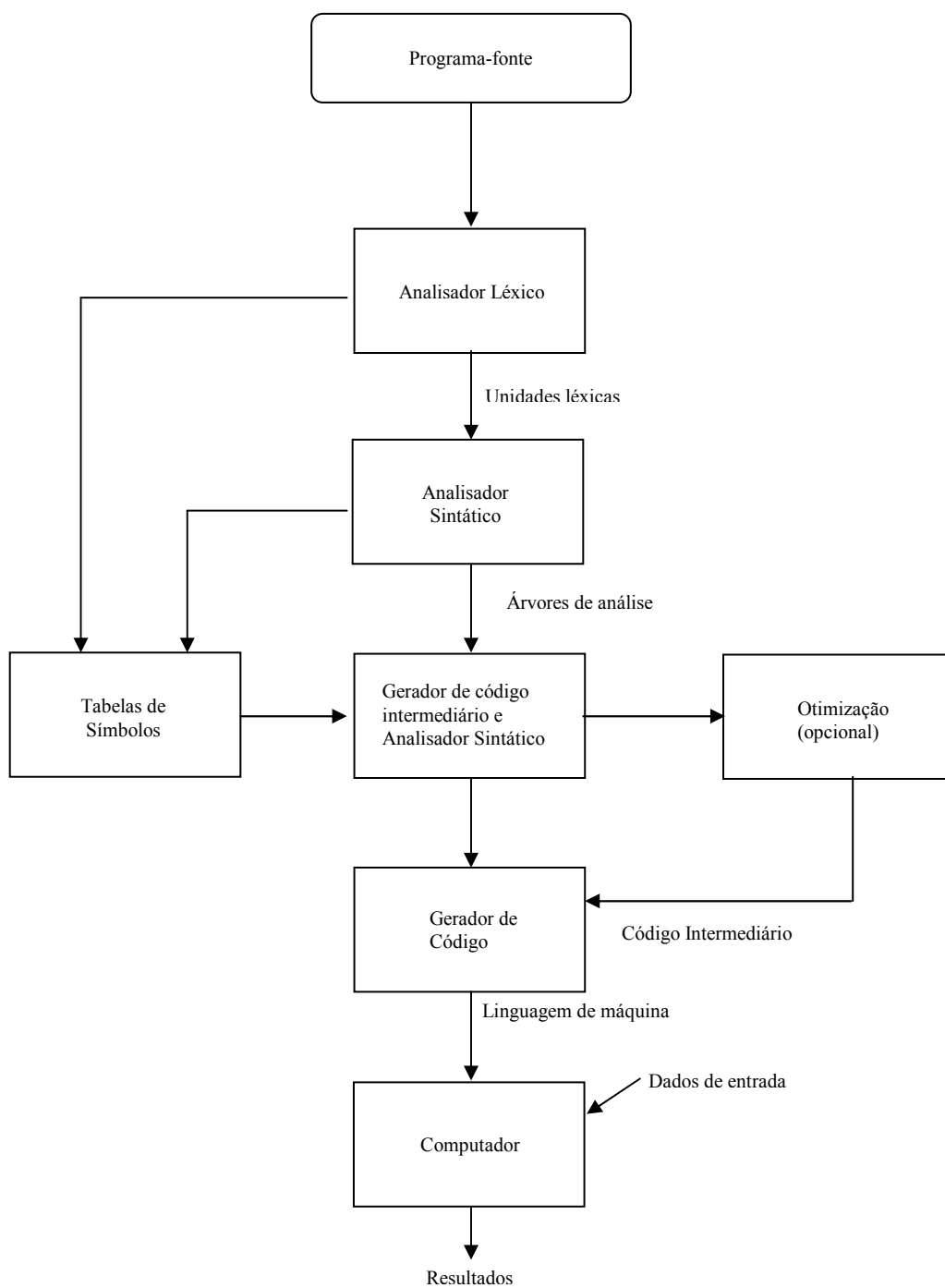


Figura 3.1 - O processo de compilação.

A otimização, que melhora os programas tornando-os menores ou mais rápidos, ou ambos, muitas vezes, é uma parte opcional da compilação. De fato, alguns compiladores são incapazes de fazer qualquer otimização significativa. Esse tipo de compilador seria usado em situações em que a velocidade de execução do programa traduzido é bem menos importante do que a velocidade de compilação. Um exemplo dessa situação é um laboratório de computação para programadores principiantes. Na maioria das situações comerciais e industriais, a velocidade de execução é mais importante do que a velocidade de compilação, de modo que a otimização é rotineiramente desejável. Uma vez que muitos tipos de otimização não podem ser feitos em linguagem de máquina, a maioria das otimizações é feita no código intermediário.

O gerador de código converte a versão do código intermediário otimizado do programa para um programa em linguagem de máquina equivalente.

A tabela de símbolos serve como um banco de dados para o processo de compilação. Seu principal conteúdo são informações sobre tipos e atributos de cada nome definido pelo usuário no programa. Essas informações são colocadas na tabela de símbolos pelos analísadores léxico e sintático e usadas pelo analisador semântico e pelo gerador de código.

Conforme afirmou-se acima, não obstante a linguagem de máquina gerada por um compilador possa ser executada diretamente no hardware, quase sempre ela deve ser executada juntamente com algum outro código. A maioria dos programas de usuário também exige programas do sistema operacional. Entre os mais comuns, estão aqueles para entrada (input) e saída (output) de dados. O compilador cria chamadas a programas do sistema necessários quando o programa de usuário necessita deles. Antes que os programas em linguagem de máquina produzidos pelo compilador possam ser executados, os programas necessários do sistema operacional devem ser encontrados e vinculados ao usuário. A operação de vinculação conecta o programa de usuário aos de sistema, colocando os endereços dos pontos de entrada dos programas de sistema nas chamadas a eles no de usuário. O código de usuário e o de sistemas juntos, às vezes, são chamados de módulo de carga ou imagem de executável. O processo de coletar programas de sistema e vinculá-los aos programas de usuário é chamado de vinculação e carregamento ou, às vezes, apenas de vinculação. Ele é realizado por um programa de sistema chamado linkeditor.

Além dos programas de sistemas, os programas de usuário muitas vezes devem ser vinculados a programas de usuário compilados anteriormente, que residem em **bibliotecas.** Assim, o *linkeditor* não somente vincula algum dado programa aos programas de sistema, mas também o vincula a outros programas de usuário.

3.2 INTERPRETAÇÃO PURA

Na extremidade oposta dos métodos de implementação, os programas podem ser interpretados por outro programa chamado interpretador, sem nenhuma conversão. O programa interpretador age como uma simulação de software de uma máquina cujo ciclo buscar-executar lida com instruções de programa em linguagem de alto nível em vez de instruções de máquina. Essa simulação de software, evidentemente, fornece uma máquina virtual para a linguagem.

Essa técnica, chamada de **interpretação pura** ou, simplesmente de **interpretação**, tem a vantagem de permitir uma fácil implementação de muitas operações de depuração do código-fonte, porque todas as mensagens de erro em tempo de execução podem referir-se a unidades do código. Por exemplo, se for considerado que um índice de *array* está fora da faixa, a mensagem de erro poderá facilmente indicar a linha da fonte e o nome do *array*. Por outro lado, esse método tem a séria desvantagem de que a execução é de 10 a 100 vezes mais lenta que em sistemas compilados. A principal causa dessa lentidão é a decodificação das instruções de alto nível, bem mais complexas do que as instruções em linguagem de máquina (não obstante possa haver um número menor de comandos do que de instruções de código de máquina

equivalente). Portanto, a decodificação de comandos, em vez da conexão entre o processador e a memória, é o gargalo de um interpretador puro.

Outra desvantagem da interpretação pura é que ela freqüentemente exige mais espaço. Além do programa-fonte, a tabela de símbolos deve estar presente na interpretação. Além disso, o programa-fonte deve ser armazenado em uma forma projetada para permitir fácil acesso e modificação, em vez de um tamanho mínimo.

A interpretação é um processo difícil em programas escritos em uma linguagem complicada, porque o significado de cada expressão e instrução deve ser determinado diretamente do programa-fonte em tempo de execução. Linguagens com estruturas mais simples prestam-se à interpretação pura. Por exemplo, a APL e a LIST, às vezes são implementadas como sistemas interpretativos puros. A maioria dos comandos do sistema operacional, como por exemplo, o conteúdo dos *scripts* do *shell* do UNIX e dos arquivos **.bat** do DOS, são implementados com interpretadores puros. Linguagens mais complexas, como o FORTRAN e o C, raramente são implementadas com interpretadores puros. O processo de interpretação pura é mostrado na figura 3.2.

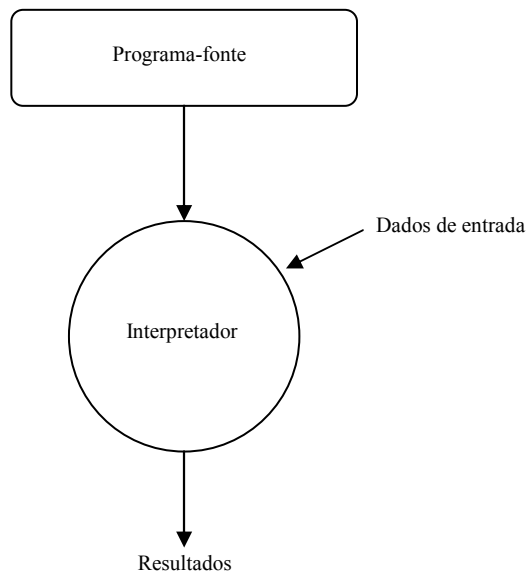


Figura 3.2 - Interpretação pura.

3.3 SISTEMAS DE INTERPRETAÇÃO HÍBRIDOS

Alguns sistemas de implementação de linguagem são um meio-termo entre os compiladores e os interpretadores puros; eles traduzem programas em linguagem de alto nível para uma linguagem intermediária projetada para permitir fácil interpretação. Esse método é mais rápido do que a interpretação pura porque as instruções da linguagem fonte são decodificadas somente uma vez. Essas implementações são chamadas de **sistemas de implementação híbridos**.

O processo usado em um sistema de implementação híbrido é mostrado na figura 3 em vez de traduzir código em linguagem intermediária para código de máquina, ele simplesmente interpreta o código intermediário.

A Perl é implementada com um sistema híbrido. Ela se desenvolveu a partir da linguagem imperativa **sh** e **awk**, mas é parcialmente compilada para detectar erros antes da interpretação e para simplificar o interpretador.

As implementações iniciais de Java eram todas híbridas. Sua forma intermediária, chamada código de bytes, oferece portabilidade a qualquer máquina que tenha um interpretador de código de bytes e um sistema *run-time* associado. Juntos, eles são chamados de Java Virtual Machine. Agora há sistemas que traduzem código de bytes Java para código de máquina para permitir uma execução mais rápida. Porém os *applets* Java são sempre descarregados do servidor Web na forma de código de bytes.

Às vezes, um interpretador pode oferecer tanto implementações compiladas como interpretadas para uma linguagem. Nesses casos, o interpretador é usado para desenvolver e para depurar programas. Então, depois de que um estado (relativamente) livre de problemas (*bugs*) é alcançado, os programas são compilados para aumentar sua velocidade de execução.

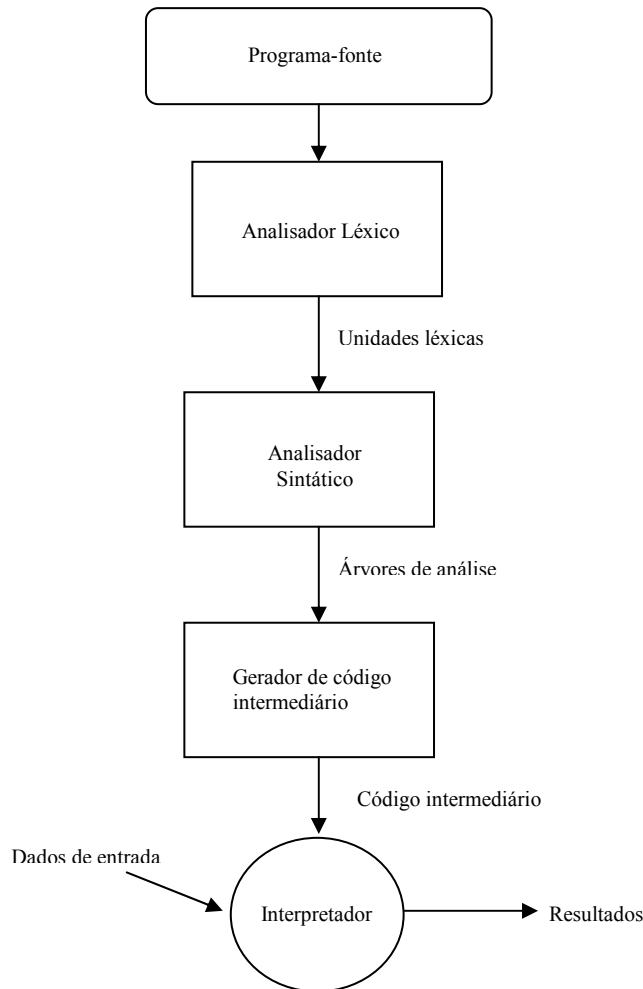


Figura 3.3 - Sistema de implementação híbrido.

3.4 CONSTRUÇÃO DE COMPILADORES

A construção de compiladores engloba várias áreas desde teoria de linguagens de programação até engenharia de *software*, passando por arquitetura de máquina, sistemas operacionais e algoritmos. Algumas técnicas básicas de construção de compiladores podem ser usadas na construção de ferramentas variadas para o processamento de linguagens, como por exemplo:

- **compiladores para linguagens de programação:** um compilador traduz um programa escrito numa *linguagem fonte* em um programa escrito em uma *linguagem objeto*;
- **formatadores de texto:** um formatador de texto manipula um conjunto de caracteres composto pelo documento a ser formatado e por comandos de formatação (parágrafos, figuras, fórmulas matemáticas, **negrito**, *itálico*, etc). São exemplos de textos formatados os documentos escritos em editores convencionais, os documentos escritos em HTML (*HyperText Markup Language*), os documentos escritos em Latex, etc.;
- **interpretadores de *queries* (consultas a banco de dados):** um interpretador de *queries* traduz uma *query* composta por operadores lógicos ou relacionais em comandos para percorrer um banco de dados.

3.5 PROCESSADORES DE LINGUAGENS

Um **processador** é um programa que permite ao computador “entender” os comandos de alto nível escritos pelos usuários.

Existem dois tipos principais de processadores de linguagem: os **interpretadores** e os **tradutores**.

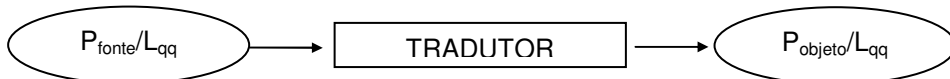
Um **interpretador** (FIGURA 3.4) é um programa que aceita como entrada um programa escrito em uma linguagem chamada *linguagem fonte* e executa diretamente as instruções dadas nesta linguagem.



FIGURA 3.4: interpretador

Um **tradutor** (FIGURA 3.5) é um programa que aceita como entrada um programa escrito em uma *linguagem fonte* e produz como saída um programa escrito em uma *linguagem objeto*. Muitas vezes a *linguagem objeto* é a própria linguagem de máquina do computador. Nesse caso, o *programa objeto* pode ser diretamente executado pela máquina.

FIGURA 3.5: tradutor



Tradutores são divididos em dois tipos: **montadores** e **compiladores**, os quais traduzem linguagens de baixo nível e linguagens de alto nível, respectivamente. Existem também o **pré-processador** que traduz uma linguagem de alto nível em outra linguagem de alto nível e o **cross-compiler** que gera código para outra máquina diferente da utilizada para compilação. A figura abaixo (FIGURA 3.6) esquematiza os três tipos de tradutores.

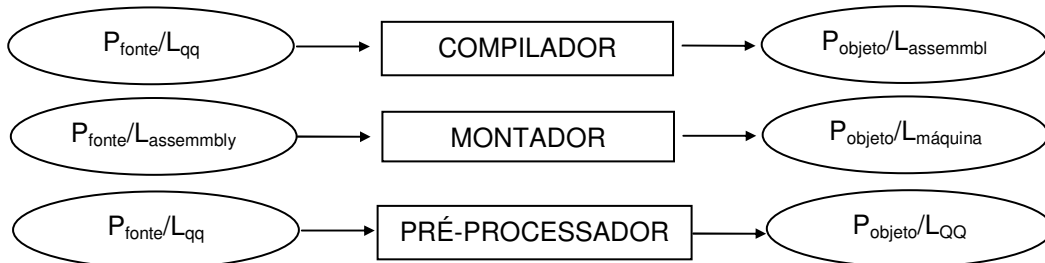


FIGURA 3.6: tipos de tradutores

No processamento de linguagens pode ser necessário o uso de vários processadores para traduzir um *programa fonte* composto por módulos em um *programa objeto*. A figura abaixo (FIGURA 3.7) apresenta o exemplo de um sistema de processamento de linguagem:

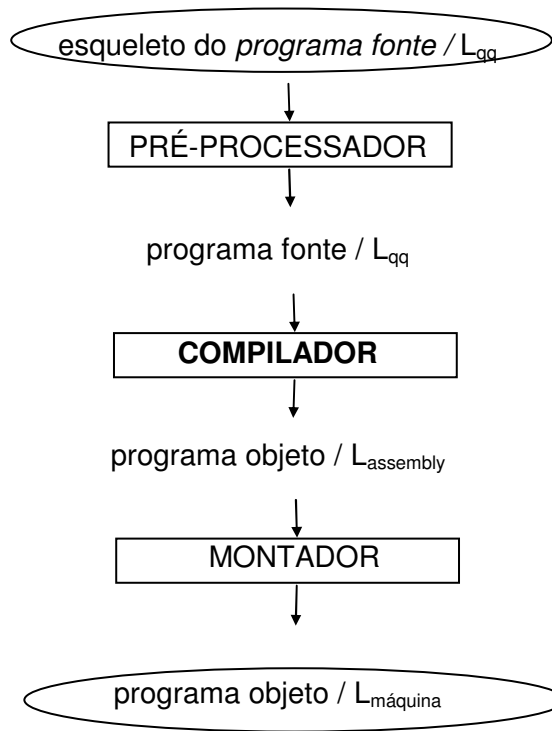


FIGURA 3.7: um sistema de processamento de linguagem

3.6 ESTRUTURA GERAL DE UM COMPILADOR

O objetivo de um compilador é traduzir as seqüências de caracteres que representam o programa fonte em código executável. Essa tarefa é complexa o suficiente de forma que um compilador pode ser dividido em processos menores interconectados. A FIGURA 3.8 mostra os processos constituintes de um compilador:

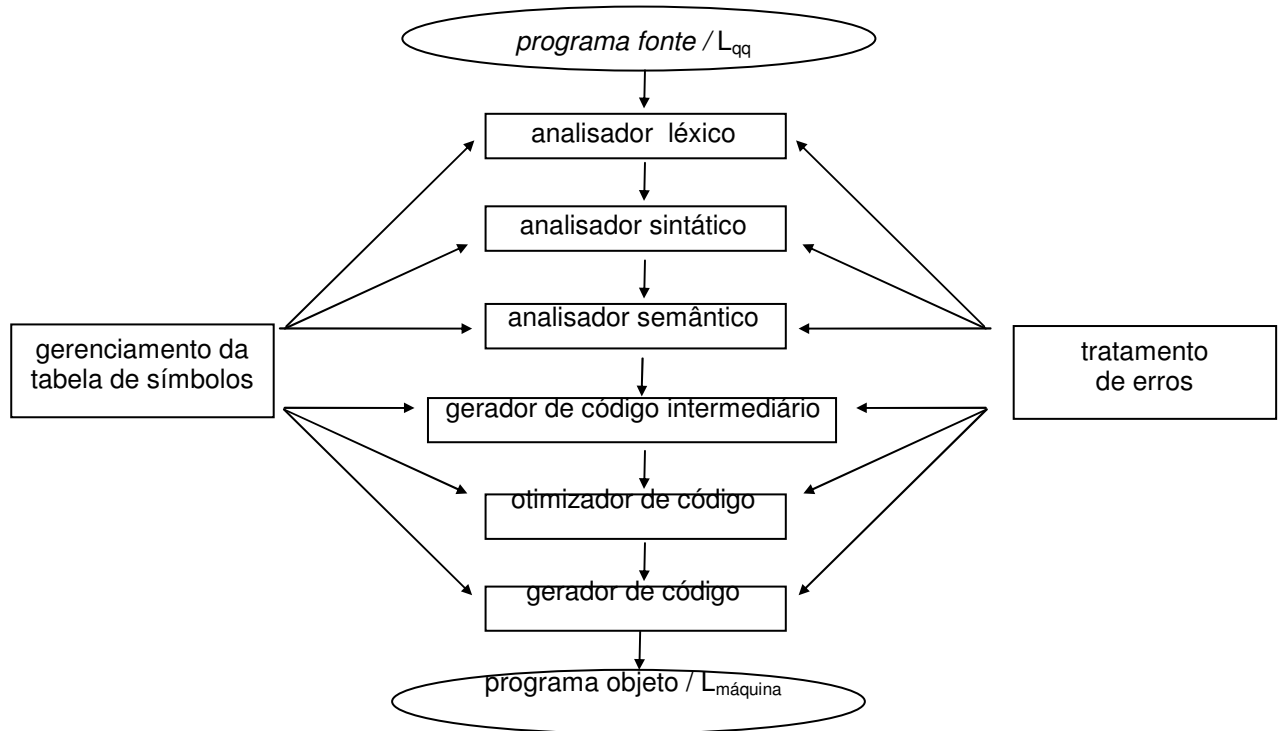


FIGURA 3.8: estrutura geral de um compilador

Cada um dos processos tem acesso a tabelas de informações globais sobre o programa fonte. Na compilação existe também um módulo responsável pelo tratamento ou recuperação de erros cuja função é diagnosticar através de mensagens adequadas os erros léxicos, sintáticos e semânticos encontrados.

3.6.1 Analisador léxico (ou *scanner*)

O **analisador léxico** separa a seqüência de caracteres que representa o programa fonte em entidades ou *tokens*, símbolos básicos da linguagem. Durante a análise léxica, os *tokens* são classificados como palavras reservadas, identificadores, símbolos especiais, constantes de tipos básicos (inteiro real, literal, etc.), entre outras categorias. Considere, por exemplo, a seqüência de caracteres:

SOMA := SOMA + 35

os quais podem ser agrupados, pelo analisador léxico, em 5 entidades:

(VALOR)	(CLASSE)
SOMA	identificador
:=	comando de atribuição
SOMA	identificador
+	operador aritmético de adição
35	constante numérica inteira

Um *token* consiste de um par ordenado (valor, classe). A *classe* indica a natureza da informação contida em *valor*.

Outras funções atribuídas ao analisador léxico são: ignorar espaços em branco e comentários, e detectar erros léxicos.

3.6.2 Analisador sintático (ou *parser*)

O **analisador sintático** agrupa os *tokens* fornecidos pelo analisador léxico em estruturas sintáticas, construindo a árvore sintática correspondente. Para isso, utiliza uma série de regras de sintaxe, que constituem a gramática da linguagem fonte. É a gramática da linguagem que define a estrutura sintática do programa fonte. Por exemplo, para a lista de *tokens* exemplificados na seção anterior, o analisador sintático construiria a árvore de derivação apresentada abaixo:

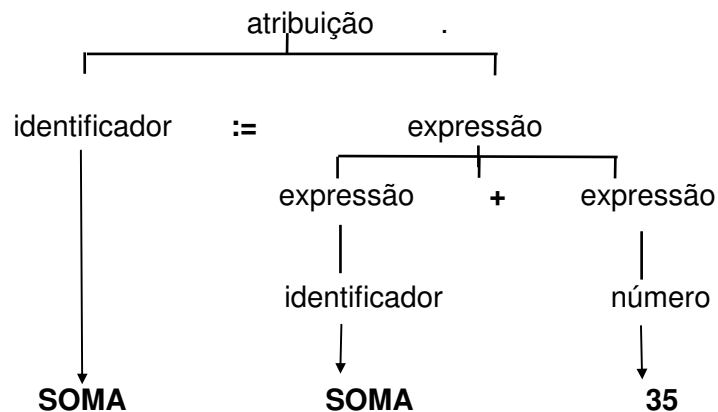


FIGURA 3.9: árvore de derivação

O analisador sintático tem também por tarefa o reconhecimento de erros sintáticos, que são construções do programa fonte que não estão de acordo com as regras de formação de estruturas sintáticas como especificado pela gramática.

3.6.3 Analisador semântico

O compilador executa ainda a análise semântica³. O analisador semântico utiliza a árvore sintática determinada pelo analisador sintático para: identificar operadores e operandos das expressões, reconhecer erros semânticos, fazer verificações de compatibilidade de tipo, analisar o escopo das variáveis, fazer verificações de correspondência entre parâmetros atuais e formais. Por exemplo, para o comando de atribuição `SOMA:= SOMA + 35`, é necessário fazer a seguinte análise:

- o identificador SOMA foi declarado? em caso negativo, erro semântico.
- o identificador SOMA é uma variável? em caso negativo, erro semântico.
- qual o escopo da declaração da variável SOMA: local ou global?
- qual o tipo da variável SOMA? o valor atribuído no lado direito do comando de atribuição é compatível?

Fundamentalmente, a análise semântica trata os aspectos sensíveis ao contexto da sintaxe das linguagens de programação. Por exemplo, não é possível representar em uma gramática livre de contexto uma regra como "Todo identificador deve ser declarado antes de ser usado.", e a verificação de que essa regra foi aplicada cabe à análise semântica. A regra pode ser representada em uma gramática sensível ao contexto, mas não existem algoritmos rápidos para essa classe de gramáticas. A idéia fundamental é a de usar tabelas externas ao processo de análise sintática, em que as informações são coletadas para posterior consulta. Por exemplo, uma *tabela de símbolos* (ou tabela de identificadores) pode guardar informações sobre as declarações dos identificadores, e essas informações podem ser consultadas para verificar a correção de cada uso de um identificador.

Esse processo é implementado de forma *dirigida pela sintaxe*: associa-se a cada regra da gramática uma ação (*ação semântica*) a ser executada quando o analisador sintático sinaliza um uso da regra. Essas ações são freqüentemente implementadas como chamadas de *rotinas semânticas*, e podem ser responsáveis por efetuar a análise semântica e a geração de código, pelo menos parcialmente.

Toda vez que o analisador sintático indicar o uso de uma regra associada a uma declaração, a rotina semântica associada a essa regra é chamada para acrescentar à tabela o identificador correspondente, fornecido pelo analisador léxico. Quando uma regra associada a um uso de um identificador for sinalizada, a rotina semântica correspondente será chamada para verificar se o identificador (novamente fornecido pelo analisador léxico) consta da tabela.

Não existe uma fronteira definida entre o que deve ser tratado pelo analisador sintático e o que deve ser tratado pelo analisador semântico, cabendo ao programador do compilador a escolha, segundo suas preferências.

Alguns compiladores incluem a geração de uma representação intermediária para o programa fonte. Uma representação intermediária é um código para uma máquina abstrata e deve ser fácil de produzir e traduzir no programa objeto. Por exemplo, pode ser usada como forma intermediária o código de três endereços (AHO et. al., 1995). O código de três endereços consiste em uma seqüência de instruções, cada uma possuindo no máximo três operandos. Para o comando de atribuição `SOMA:= SOMA + 35` tem-se:

³ a semântica nesse caso engloba apenas uma pequena parte do que vem a ser realmente a semântica de um programa.

```
temp1 := 35  
temp2 := SOMA + temp1  
SOMA := temp2
```

3.6.4 Otimização de código

O processo de otimização de código consiste em melhorar o código intermediário de tal forma que o programa objeto resultante seja mais rápido em tempo de execução. Por exemplo, um algoritmo para geração do código intermediário gera uma instrução para cada operador na árvore sintática, mesmo que exista uma maneira mais otimizada de realizar o mesmo comando. Assim, o código intermediário:

```
temp1 := 35  
temp2 := SOMA + temp1  
SOMA := temp2
```

poderia ser otimizado para:

```
SOMA := SOMA + 35
```

No entanto, não existe nada errado com o algoritmo de geração de código intermediário, desde que o problema pode ser corrigido durante a fase de otimização de código.

3.6.5 Geração de código

A fase final do compilador é a geração do código para o programa objeto, consistindo normalmente de código em linguagem assembly ou de código em linguagem de máquina:

```
MOV AX, [soma] % cópia do conteúdo do endereço de memória correspondente ao rótulo SOMA  
para o registrador AX  
ADD AX, 35 % soma do valor constante 35 ao conteúdo do registrador AX  
MOV [soma], AX % cópia do conteúdo do registrador AX para o endereço de memória  
correspondente ao rótulo "soma"
```

3.6.5.1 Geração de Código e Otimização Dependente de Máquina

Observamos antes que uma representação intermediária do programa fonte deve ser construída durante a fase de análise, para ser usada como base para a geração do programa objeto. Se a forma dessa representação intermediária é bem escolhida, a complexidade do processo de geração de código depende apenas da arquitetura da máquina (real ou virtual) para a qual o código está sendo gerado. Máquinas mais simples oferecem poucas opções e por isso o processo de geração de código é mais direto.

Por exemplo, se uma máquina tem apenas um registrador (acumulador) em que as operações aritméticas são realizadas, e apenas uma instrução para realizar cada operação (uma instrução para soma, uma para produto, ...), existe pouca ou nenhuma possibilidade de variação no código que pode ser gerado. Considere o comando de atribuição:

```
x := a + b * c
```

A primeira operação a ser realizada é o produto de b por c. Seu valor deve ser guardado numa posição temporária, que indicaremos aqui por t1. (Para sistematizar o processo, todos os resultados de operações aritméticas serão armazenados em posições temporárias.) Em seguida, devemos realizar a soma de a com t1, cujo valor será guardado numa posição temporária t2. (Naturalmente, neste caso particular, o valor poderia ser armazenado diretamente em x, mas no caso geral, a temporária é necessária.) Finalmente, o valor de t2 é armazenado em x.

```
t1:=b*c
t2:=a+t1
x:=t2
```

Podemos fazer um gerador de código relativamente simples usando regras como:

1. toda operação aritmética (binária) gera 3 instruções:

instrução	exemplo: b * c
carrega o primeiro operando no acumulador	Load b
usa a instrução correspondente a operação com o segundo operando, deixando o resultado no acumulador	Mult c
armazena o resultado em uma temporária nova	Store t1

2. um comando de atribuição gera sempre duas instruções:

instrução	exemplo: x := t2
carrega o valor da expressão no acumulador	Load t2
armazena o resultado na variável	Store x

O comando de atribuição:

```
x := a + b * c,
```

gera o código:

```
1 Load b      { t1:=b*c }
2 Mult c
3 Store t1
4 Load a      { t2:=a+t1 }
5 Add t1
6 Store t2
7 Load t2     { x:=t2 }
8 Store x
```

Embora correto, este código pode obviamente ser melhorado:

- a instrução 7 é desnecessária e pode ser retirada: copia para o acumulador o valor de t2, que já se encontra lá.
- (após a remoção da instrução 7) a instrução 6 é desnecessária e pode ser retirada: o valor da variável t2 nunca é utilizado.
- (considerando que a soma é comutativa) as instruções 4 e 5 podem ser trocadas por 4' e 5', preparando novas alterações:

```
4' Load t1
```

5' Add a

- As instruções 3 e 4' são desnecessárias e podem ser retiradas (pelas mesmas razões que 6 e 7 acima).

O código final após as transformações é consideravelmente melhor que o original:

1 Load b
2 Mult c
5' Add a
8 Store x

Normalmente, as máquinas oferecem várias instruções (ou variantes de instruções) com características semelhantes, e o gerador deve escolher a mais apropriada entre elas. Como exemplo, vamos examinar o caso da operação de soma. Em geral, podemos observar os seguintes pontos:

- há várias instruções de soma, correspondendo a vários tipos de dados e a vários modos de endereçamento;
- há instruções de soma aplicáveis a casos particulares importantes, como instruções de incremento e decremento: soma com ± 1 ;
- algumas das somas a serem efetuadas não foram especificadas explicitamente pelo programador. Entre essas citamos as somas usadas no cálculo de endereços de variáveis componentes de vetores, matrizes e estruturas situadas em registros de ativação de procedimentos ou funções. Frequentemente, essas somas podem ser incluídas no código de forma implícita através da escolha de modos de endereçamento adequados;
- instruções cuja finalidade principal não é a soma podem mesmo assim efetuar somas. Por exemplo, as instruções que manipulam a pilha de hardware, incrementam ou decrementam o registrador apontador do topo da pilha.

Esses pontos devem ser levados em consideração pelo gerador de código na seleção de instruções. Outro problema que também deve ser tratado é o da escolha do local para guarda dos valores das variáveis definidas pelo usuário e das variáveis temporárias introduzidas pelo compilador. Além da alocação de posições de memória a essas variáveis, é freqüente a disponibilidade de vários registradores de uso geral, que também podem ser usados com essa finalidade. A alocação de registradores, entretanto, não é independente da seleção de instruções, já que muitas instruções usam registradores ou combinações de registradores para operações específicas.

Cabe ao projetista do gerador de código decidir como implementar a geração de código de maneira a fazer bom uso dos recursos disponíveis na máquina. Cabe também ao projetista decidir se a geração do código deve ser feita com cuidado, gerando diretamente código de qualidade aceitável, ou se é preferível usar um esquema mais simples de geração de código, seguido por uma "otimização" do código depois de gerado. Esta otimização do código leva em consideração principalmente as características da máquina alvo, e por isso é normalmente chamada de *otimização dependente de máquina*.

3.6.5.2 Otimização independente de máquina

Algumas transformações feitas no código gerado por um compilador independem da máquina para a qual o código está sendo gerado. Normalmente estas transformações são feitas no código intermediário, pela facilidade de acesso já mencionada anteriormente. Vamos nesta seção apresentar alguns exemplos deste tipo de otimização.

Exemplo 1: *Sub-expressões comuns*. Considere a seqüência de comandos de atribuição da primeira coluna da tabela.

Fonte	código intermediário original	Código intermediário otimizado
w:=(a+b)+c;	t1:=a+b t2:=t1+c w:=t2	t1:=a+b t2:=t1+c w:=t2
x:=(a+b)*d;	t3:=a+b t4:=t3*d x:=t4	t4:=t1*d x:=t4
y:=(a+b)+c;	t5:=a+b t6:=t5+c y:=t6	y:=t2
z:=(a+b)*d+e;	t7:=a+b t8:=t7*d t9:=t8+e z:=t9	t9:=t4+e z:=t9

Claramente, as (sub-)expressões $a+b$, $(a+b)+c$, e $(a+b)*d$ não precisam ser calculadas mais de uma vez. (Isto só é verdade porque os valores de a , b , c e d não se alteram no trecho em questão.) Podemos alterar a representação intermediária correspondente (segunda coluna) para a forma intermediária equivalente otimizada apresentada na terceira coluna.

Exemplo: Retirada de comandos *invariantes de loop*. Considere o trecho de código a seguir:

```
for i:=1 to n do begin
    pi:=3.1416;
    pi4:=pi/4.;
    d[i]:=pi4 * r[i] * r[i];
end;
```

Claramente, os dois primeiros comandos de atribuição podem ser retirados do *loop*, uma vez que seu funcionamento é independente do funcionamento do *loop*. Obteríamos

```
pi:=3.1416;
pi4:=pi/4.;
for i:=1 to n do
    d[i]:=pi4 * r[i] * r[i];
```

que é uma versão "otimizada" do trecho de código anterior. Note, entretanto, que só há uma melhora no tempo de execução se o valor de n for maior que zero. Se $n=0$, o código foi piorado: os dois comandos de atribuição serão sempre executados.

Normalmente, as transformações realizadas no programa durante a otimização são simples: eliminar ou alterar instruções, ou ainda mover instruções para outras posições. A parte mais trabalhosa é a verificar que a transformação pode ser feita. Por exemplo, para eliminar um comando da forma $v:=e$, é preciso verificar que o valor de v calculado neste comando não será usado por nenhum outro comando, e, portanto, examinar toda a parte do programa que poderá ser executada a seguir. Por essa razão, a análise de fluxo de dados (*dataflow analysis*) é uma parte essencial do estudo da otimização, pois visa obter informação sobre o funcionamento do programa, em particular especificando os pontos do programa onde as variáveis recebem valores, e onde os valores são usados.

3.7 FORMAS DE CONSTRUÇÃO DE UM COMPILADOR

Um compilador pode ser composto de várias fases denominadas de passos do compilador.

Dependendo da implementação, certos passos podem ser executados seqüencialmente ou ter execução entrelaçada, enquanto alguns passos podem ser omitidos. Numa compilação em vários passos, a execução de um passo termina antes de iniciar-se a execução dos passos seguintes. Assim, o compilador de dois passos, por exemplo, poderia combinar a análise léxica e análise sintática num primeiro passo e a análise semântica e a geração de código num segundo passo. De outra forma, pode-se utilizar o analisador sintático como módulo principal: para construir a árvore sintática, obtém os *tokens* necessários através de chamadas ao analisador léxico e chama o processo de geração de código para executar a análise semântica e geração de código objeto. Os critérios para escolha da forma de implementação envolvem: memória disponível, tempo de compilação ou tempo de execução, características da linguagem e equipe de desenvolvimento.

A principal vantagem de se construir compiladores de vários passos é a modularização alcançada no projeto e na implementação dos processos que constituem o compilador. A principal desvantagem é o aumento do projeto total, com a necessidade de introdução das linguagens intermediárias.

A figura abaixo (FIGURA 3.10) apresenta a estrutura geral de um compilador:

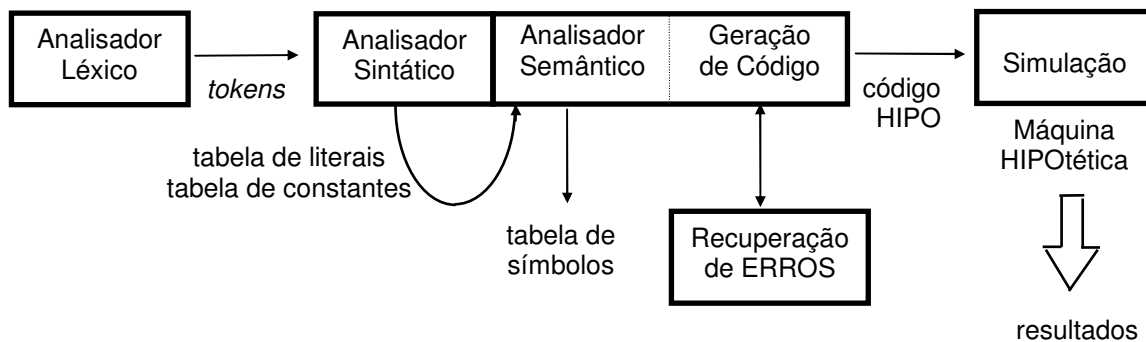


FIGURA 3.10: estrutura geral de um compilador

3.8 FERRAMENTAS PARA A CONSTRUÇÃO DE COMPILADORES

Na construção de compiladores faz-se uso de ferramentas de *software* tais como ambientes de programação, depuradores, gerenciadores de versões, etc. E ainda, foram criadas algumas ferramentas para projeto e geração automática de alguns processos componentes de compiladores. Essas ferramentas são freqüentemente referidas como *compiladores de compiladores*, *geradores de compiladores* ou *sistemas de escrita de tradutores*. Normalmente, são orientados a um modelo particular de linguagem e mais adequados para a construção de compiladores de linguagens similares ao modelo. Segundo AHO et. al. (1995), alguns tipos de ferramentas são:

1. geradores de analisadores léxicos: geram automaticamente analisadores léxicos, normalmente a partir de uma especificação baseada em expressões regulares e uma lista de palavras-chave da linguagem (LEX);
2. geradores de analisadores sintáticos: produzem analisadores sintáticos a partir da especificação de uma gramática livre de contexto (YACC, T-gen, JACK).

4 ANALISADOR LÉXICO

4.1 FUNÇÃO

No processo de compilação, o analisador léxico é responsável pela identificação dos *tokens*, ou seja, das menores unidades de informação que constituem a linguagem em questão. Assim, pode-se dizer que o analisador léxico é responsável pela leitura dos caracteres da entrada, agrupando-os em palavras, que são classificadas em categorias. Estas categorias podem ser, basicamente, as seguintes:

- **palavras reservadas:** palavras que devem aparecer literalmente na linguagem, sem variações. Algumas palavras reservadas da linguagem PASCAL são: BEGIN, END, IF, ELSE.
- **identificadores:** palavras que seguem algumas regras de escrita, porém podem assumir diversos valores. São definidos de forma genérica. Geralmente, as regras de formação de identificadores são as mesmas utilizadas para a formação de palavras reservadas. Nesse caso, é necessário algum mecanismo para decidir quando um *token* forma um identificador ou uma palavra reservada.
- **símbolos especiais:** seqüências de um ou mais símbolos que não podem aparecer em identificadores nem palavras reservadas. São utilizados para composição de expressões aritméticas ou lógicas, comando de atribuição, etc. São exemplos de símbolos especiais: “;” (ponto-e-vírgula), “:” (dois pontos), “:=” (atribuição).
- **constantes:** podem ser valores inteiros, valores reais, caracteres ou literais.
- **comentário:** qualquer cadeia de caracteres iniciando com e terminando com símbolos delimitadores, utilizada na documentação do programa fonte.

Para a construção de um analisador léxico é necessário descrever precisamente a regra de formação (padrão) de *tokens* mais complexos, como palavras reservadas, identificadores, constantes e comentários, usado **expressões regulares**; converter a especificação feita em termos de expressões regulares na implementação de **autômatos finitos** determinísticos mínimos correspondente; implementar os autômatos finitos determinísticos mínimos em uma linguagem de programação.

4.2 RECONHECIMENTO DE *TOKENS*: AUTÔMATO FINITO

Um autômato finito (AF) é o tipo mais simples de reconhecedor de linguagens. Um AF, dentre outras aplicações, pode ser usado como reconhecedor de padrões de processamento de textos e analisador léxico em linguagens de programação.

Um autômato finito pode ser visto como uma máquina de estados. Assume-se que a máquina esteja em um estado inicial quando começa sua operação. O novo estado da máquina é determinado em função do estado corrente e do evento ocorrido. Assim, por exemplo, uma máquina hipotética cujos estados possíveis são repouso, manutenção e atividade, poderia ser representada da seguinte forma (FIGURA 4.1):

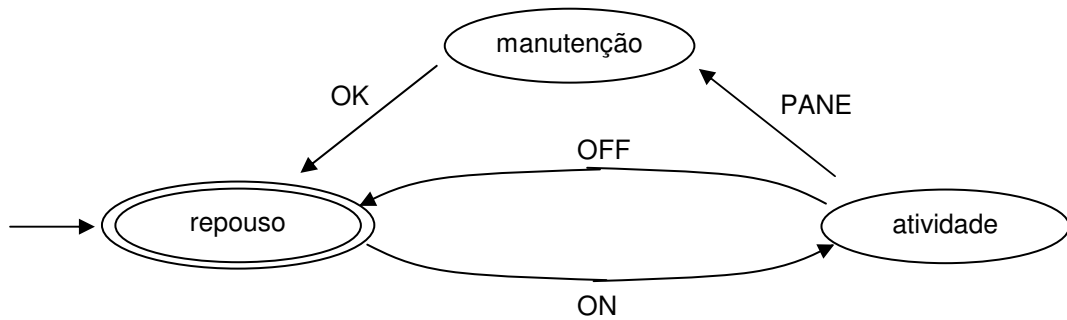


FIGURA 4.1: exemplo de uma máquina de estados (ZILLER, 1997)

Inicialmente a máquina está em repouso. A máquina entra em atividade quando é ligada (evento *ON*) e volta ao estado de repouso quando é automaticamente desligada (evento *OFF*) após a produção de uma determinada peça. Assim, o funcionamento normal consiste numa seqüência de eventos *ON* e *OFF*. No entanto, pode ocorrer uma pane quando a máquina estiver em atividade. Nesse caso, quando da ocorrência do evento *PANE*, a máquina muda para o estado de manutenção, retornando ao estado de repouso após serem realizados os devidos ajustes (evento *OK*).

4.3 IMPLEMENTAÇÃO

Na implementação do analisador léxico deve-se: desconsiderar brancos à esquerda; considerar como marca de final de sentença o primeiro caractere que não pertencer ao alfabeto da seqüência que está sendo reconhecida; eliminar delimitadores e comentários usados por questões de legibilidade pelo programador, os quais são totalmente irrelevantes do ponto de vista de geração de código. O analisador léxico pode ser implementado de forma mista: usa-se a implementação específica de autômatos para o reconhecimento do primeiro caractere e a implementação genérica de autômatos para o reconhecimento do restante da sentença, exceto os símbolos especiais cujo o reconhecimento poderá ser totalmente efetuado de forma específica. Deve-se também implementar estratégias para a recuperação e tratamento de erros léxicos, quais sejam: símbolos que não fazem parte da linguagem em questão bem como seqüências de símbolos que não obedecem às regras de formação dos *tokens* especificados. JOSÉ NETO (1987) apresenta algumas técnicas para recuperação de erros léxicos. AHO et. al. (1995) descreve a técnica *panic-mode*.

5 ANALISADOR SINTÁTICO

5.1 FUNÇÃO

O analisador sintático agrupa os tokens fornecidos pelo analisador léxico em estruturas sintáticas, construindo a árvore sintática correspondente. Para isso, utiliza uma série de regras de sintaxe, que constituem a gramática da linguagem fonte. O analisador sintático tem também por tarefa o reconhecimento de erros sintáticos, que são construções do programa fonte que não estão de acordo com as regras de formação de estruturas sintáticas especificadas através de uma gramática livre de contexto.

5.2 ESPECIFICAÇÃO DAS REGRAS SINTÁTICAS: GRAMÁTICA LIVRE DE CONTEXTO

Segundo PRICE; EDELWEISS (1989), dentro da hierarquia de Chomsky, as gramáticas livres de contexto (GLC) são as mais importantes na área de compiladores e linguagens formais, pois podem especificar a maior parte das construções sintáticas usuais.

5.2.1 Notações

Existem inúmeras notações pelas quais a representação de uma linguagem pode ser especificada. Segundo JOSÉ NETO (1987), "a tais notações dá-se o nome de *metalinguagens*, já que elas próprias são linguagens, através das quais as linguagens são especificadas". A sintaxe de uma linguagem de programação pode ser descrita usando-se as seguintes notações:

- Suponha que uma linguagem de programação só tem variáveis do tipo **inteiro** e que a declaração de variáveis deve ser feita usando a palavra reservada **variáveis** seguida do tipo, de uma lista de identificadores separados por vírgula, e de um ponto e vírgula. Para especificar a sintaxe dessa declaração de variáveis pode-se escrever a seguinte gramática:

EXEMPLO : sintaxe da declaração de variáveis usando a notação de regras de produção

$D \rightarrow \text{variáveis inteiro } L;$
 $L \rightarrow \text{identificador} \mid \text{identificador} , L$

- a notação BNF (Backus-Naur Form), muito usada para a especificação de linguagens livres de contexto, adota a seguinte simbologia:

$\langle x \rangle$ representa um símbolo não-terminal, cujo o nome é dado pela cadeia x de caracteres quaisquer.

$\langle x \rangle ::= \beta$ representa as regras de produção, associando o não-terminal $\langle x \rangle$ à sentença β . O significado de uma regra de produção $\langle x \rangle ::= \beta$ é $\langle x \rangle$ é definido por β .

$|$ separa as diversas regras de produção que estão à direita do símbolo $::=$, desde que o símbolo não-terminal à esquerda seja o mesmo. O significado de $\langle x \rangle ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ é $\langle x \rangle$ é definido por β_1 OU $\langle x \rangle$ é definido por β_2 OU ... $\langle x \rangle$ é definido por β_n .

x ou **X** representa um símbolo terminal, dado pela cadeia **x** ou **X** de caracteres quaisquer e deve ser escrito tal como aparece nas sentenças da linguagem.

A gramática apresentada anteriormente pode ser escrita da seguinte forma usando a notação BNF:

EXEMPLO : sintaxe da declaração de variáveis usando a notação BNF

<declaração de variáveis> ::= **variáveis inteiro** <lista de identificadores> ;
 <lista de identificadores> ::= identificador | identificador , <lista de identificadores>

- diagramas de sintaxe são "uma ferramenta muito cômoda para a documentação e o estudo da sintaxe de linguagens de programação, oferecendo possibilidade de obtenção de reconhecedores eficientes a partir da gramática, mediante um esforço reduzido" (JOSÉ NETO, 1987). Um diagrama de sintaxe apresenta um início e um fim, ligados por um grafo orientado, cujos retângulos representam os símbolos não-terminais e as elipses representam os símbolos terminais. Para ler um diagrama de sintaxe, deve-se seguir as setas, as quais podem eventualmente apresentar caminhos alternativos ou não obrigatórios.

EXEMPLO : a gramática apresentada anteriormente, tem o seguinte diagrama de sintaxe:

declaração de variáveis =

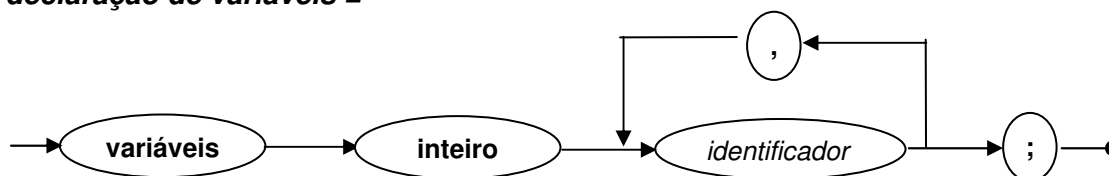


FIGURA 5.1: diagrama de sintaxe

5.2.2 Árvore de derivação ou árvore sintática

Algumas vezes pode ser útil mostrar as derivações de uma GLC através de representações gráficas. Essas representações chamadas de árvores de derivação ou árvores sintáticas impõem estruturas hierárquicas às sentenças das linguagens geradas. A árvore de derivação é a saída lógica da análise sintática, constituindo uma representação intermediária utilizada na análise semântica.

Exemplo:

$E \rightarrow E + T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid a$

Pelas convenções vistas, temos:

símbolos terminais:	{ +, *, (,), a }
símbolos não terminais:	{ E, T, F }
Símbolo inicial:	E
regras:	{ $E \rightarrow E+T$, $E \rightarrow T$, $T \rightarrow T * F$, $T \rightarrow F$, $F \rightarrow (E)$, $F \rightarrow a$ }

Por exemplo, a cadeia $a+a*a$ pertence à linguagem da gramática, por causa da derivação

(1) $E \Rightarrow E+T \Rightarrow T+T \Rightarrow a+T \Rightarrow a+T * F \Rightarrow a+F * F \Rightarrow a+a * F \Rightarrow a+a * a$

Note que outras derivações são possíveis para a mesma cadeia. Por exemplo, temos

$$(2) E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow E+T^*a \Rightarrow E+F^*a \Rightarrow E+a^*a \Rightarrow T+a^*a \Rightarrow F+a^*a \Rightarrow a+a^*a$$

$$(3) E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow E+F^*F \Rightarrow E+a^*F \Rightarrow T+a^*F \Rightarrow T+a^*a \Rightarrow F+a^*a \Rightarrow a+a^*a$$

As derivações (1), (2) e (3) acima são equivalentes, no sentido de que ambas geram a cadeia da mesma maneira, aplicando as mesmas regras aos mesmos símbolos, e se diferenciam apenas pela ordem em que as regras são aplicadas. A maneira de verificar isso é usar uma *árvore de derivação*, que para este caso seria a descrita na figura abaixo:

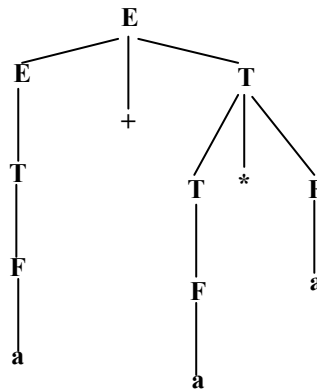


Figura - Árvore de derivação para a+a*a

Num certo sentido, a árvore de derivação caracteriza as gramáticas livres de contexto: a expansão correspondente a cada sub-árvore pode ser feita de forma absolutamente independente das demais sub-árvores. Por essa razão, consideramos equivalentes todas as derivações que correspondem à mesma árvore de derivação. Dois tipos de derivação são especialmente interessantes:

- a *derivação esquerda* (*leftmost derivation*), em que uma regra é sempre aplicada ao primeiro não terminal da cadeia, o que fica *mais à esquerda*;
- a *derivação direita* (*rightmost derivation*), em que uma regra é sempre aplicada ao último não terminal da cadeia, o que fica *mais à direita*.

Para especificar uma derivação de uma cadeia x, podemos, equivalentemente, apresentar uma derivação esquerda de x, uma derivação direita de x, ou uma árvore de derivação de x. A partir de uma dessas três descrições, é possível sempre gerar as outras duas. Por exemplo, dada uma árvore de derivação de x, basta percorrer a árvore em pré-ordem (primeiro a raiz, depois as sub-árvores em ordem, da esquerda para a direita), e aplicar as regras encontradas sempre ao primeiro não terminal, para construir uma derivação esquerda. Para a derivação direita a árvore deve ser percorrida em uma ordem semelhante: primeiro a raiz, depois as sub-árvores em ordem, da direita para a esquerda.

Exemplo (continuação): A derivação (1) é uma derivação esquerda, e a derivação (2) é uma derivação direita. A derivação (3) nem é uma derivação esquerda, nem é uma derivação direita. Dada uma cadeia x, todas as derivações possíveis de x correspondem exatamente à mesma árvore de derivação. Veremos abaixo que isto quer dizer que a gramática anterior não é ambígua.

Uma gramática é ambígua se, para alguma cadeia x, existem duas ou mais árvores de derivação. Podemos mostrar que uma gramática é ambígua mostrando uma cadeia x e duas árvores de derivação distintas de x (ou duas derivações esquerdas distintas, ou duas derivações direitas distintas).

Por outro lado, para mostrar que uma gramática não é ambígua, é necessário mostrar que cada cadeia da linguagem tem exatamente uma árvore de derivação. Isso costuma ser feito pela apresentação de um algoritmo para construção (dada uma cadeia qualquer x da linguagem) da única árvore de derivação de x , de uma forma que torne claro que a árvore assim construída é a única árvore de derivação possível para x .

Exemplo: Considere a gramática

$$S \rightarrow 0 S 1 \mid e$$

A linguagem dessa gramática é formada pelas cadeias da forma $0^n 1^n$, onde $n \geq 0$. Vamos mostrar que essa gramática não é ambígua, mostrando, para cada valor de n , como construir a única derivação de $0^n 1^n$. (No caso, essa derivação é ao mesmo tempo, uma derivação esquerda e uma derivação direita, porque no máximo há um não terminal para ser expandido.)

Para derivar $0^n 1^n$,

"use n vezes a regra $S \rightarrow 0S1$, e uma vez a regra $S \rightarrow e$."

Essa derivação é a única:

- cada aplicação da regra $S \rightarrow 0S1$ introduz um 0 e um 1; as n aplicações introduzem n pares 0, 1.
- a aplicação da regra $S \rightarrow e$ é necessária para obter uma cadeia sem não terminais.

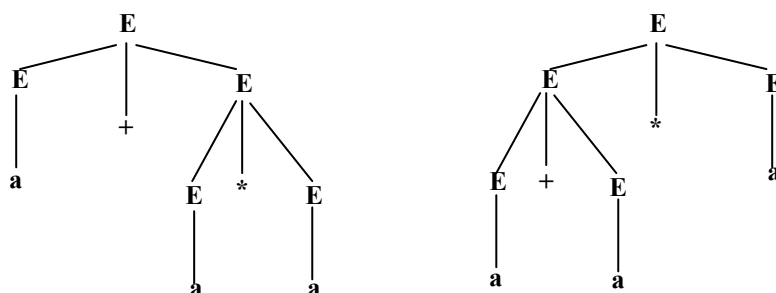
Por exemplo, para $n=3$, temos:

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

Exemplo: A gramática

$$E \rightarrow E+E \mid E^*E \mid (E) \mid a$$

é ambígua. Por exemplo, a cadeia $a+a^*a$ pode ser derivada de duas maneiras diferentes, de acordo com as duas árvores mostradas a seguir:



5.2.3 Análise sintática descendente e ascendente

Os métodos de análise sintática podem ser classificados segundo a maneira pela qual a árvore de derivação da cadeia analisada x é construída:

- nos métodos *descendentes*, a árvore de derivação correspondente a x é construída de *cima para baixo*, ou seja, da raiz (o símbolo inicial S) para as folhas, onde se encontra x .
- nos métodos *ascendentes*, a árvore de derivação correspondente a x é construída de *baixo para cima*, ou seja, das folhas, onde se encontra x , para a raiz, onde se encontra o símbolo inicial S .

Nos métodos descendentes (*top-down*), temos de decidir qual a regra $A \Rightarrow \beta$ a ser aplicada a um nó rotulado por um não terminal A . A *expansão* de A é feita criando nós filhos rotulados com os símbolos de β . Nos métodos ascendentes (*bottom-up*), temos de decidir quando a regra $A \Rightarrow \beta$ deve ser aplicada, e devemos encontrar nós vizinhos rotulados com os símbolos de β . A *redução* pela regra $A \Rightarrow \beta$ consiste em acrescentar à árvore um nó A , cujos filhos são os nós correspondentes aos símbolos de β .

Métodos descendentes e ascendentes constroem a árvore da esquerda para a direita. A razão para isso é que a escolha das regras deve se basear na cadeia a ser gerada, que é lida da esquerda para a direita. (Seria muito estranho um compilador que começasse a partir do fim do programa fonte!).

Exemplo: Considere a gramática

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow a$

e a cadeia $x = a+a*a$, como no exemplo 1. Usando-se um método descendente, a árvore de derivação de x é construída na seqüência especificada.

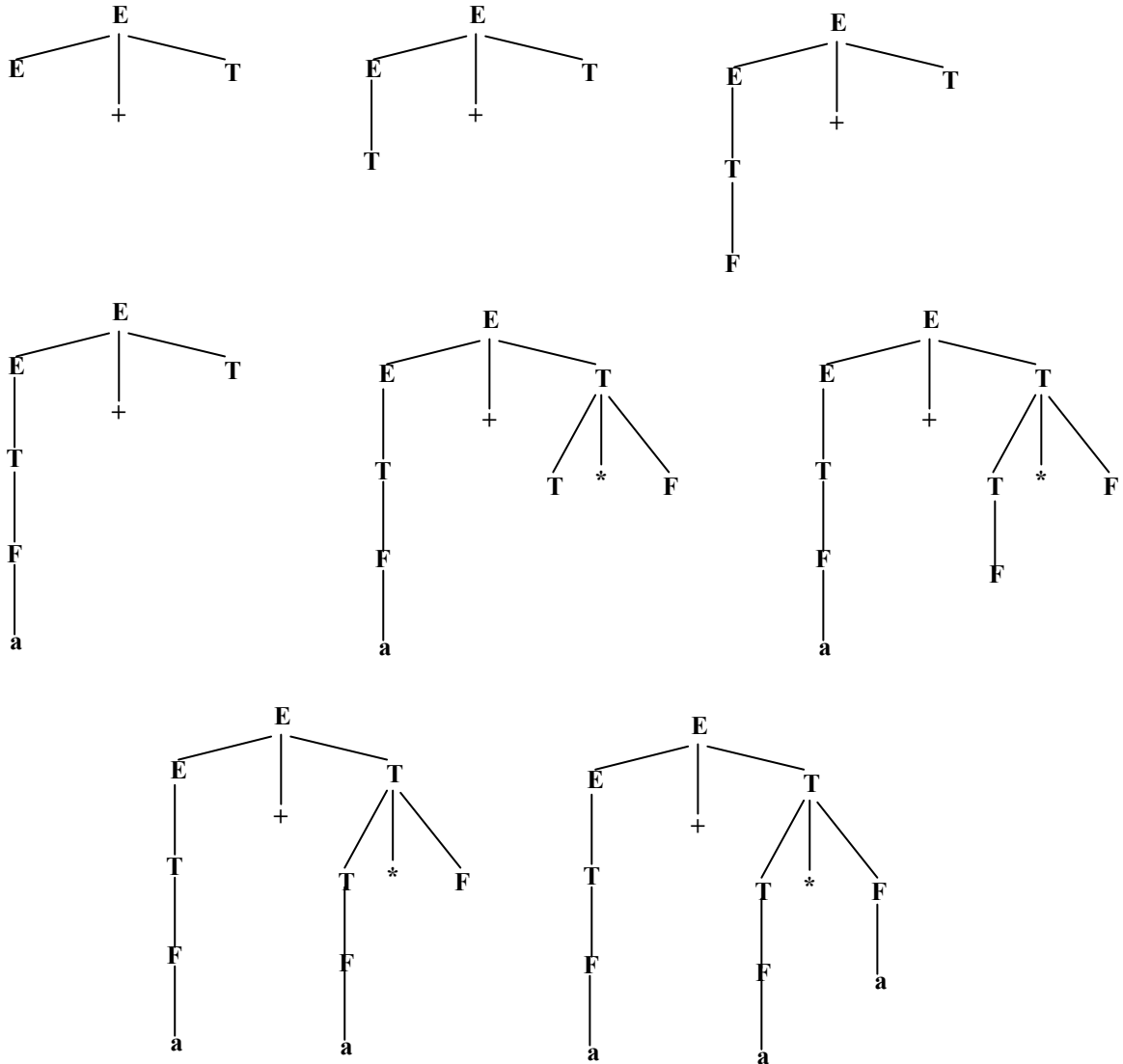


Figura - análise descendente

Note que as regras são consideradas na ordem 1 2 4 6 3 4 6 6, a mesma ordem em que as regras são usadas na derivação esquerda

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

Usando-se um método de análise ascendente, por outro lado, as regras são identificadas na ordem 6 4 2 6 4 6 3 1, e os passos de construção da árvore podem ser vistos na figura abaixo. Neste caso, a ordem das regras corresponde à derivação direita, *invertida*:

$a+a*a \Leftarrow F+a*a \Leftarrow T+a*a \Leftarrow E+a*a \Leftarrow E+F*a \Leftarrow E+T*a \Leftarrow E+T*F \Leftarrow E+T \Leftarrow E$

ou seja,

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

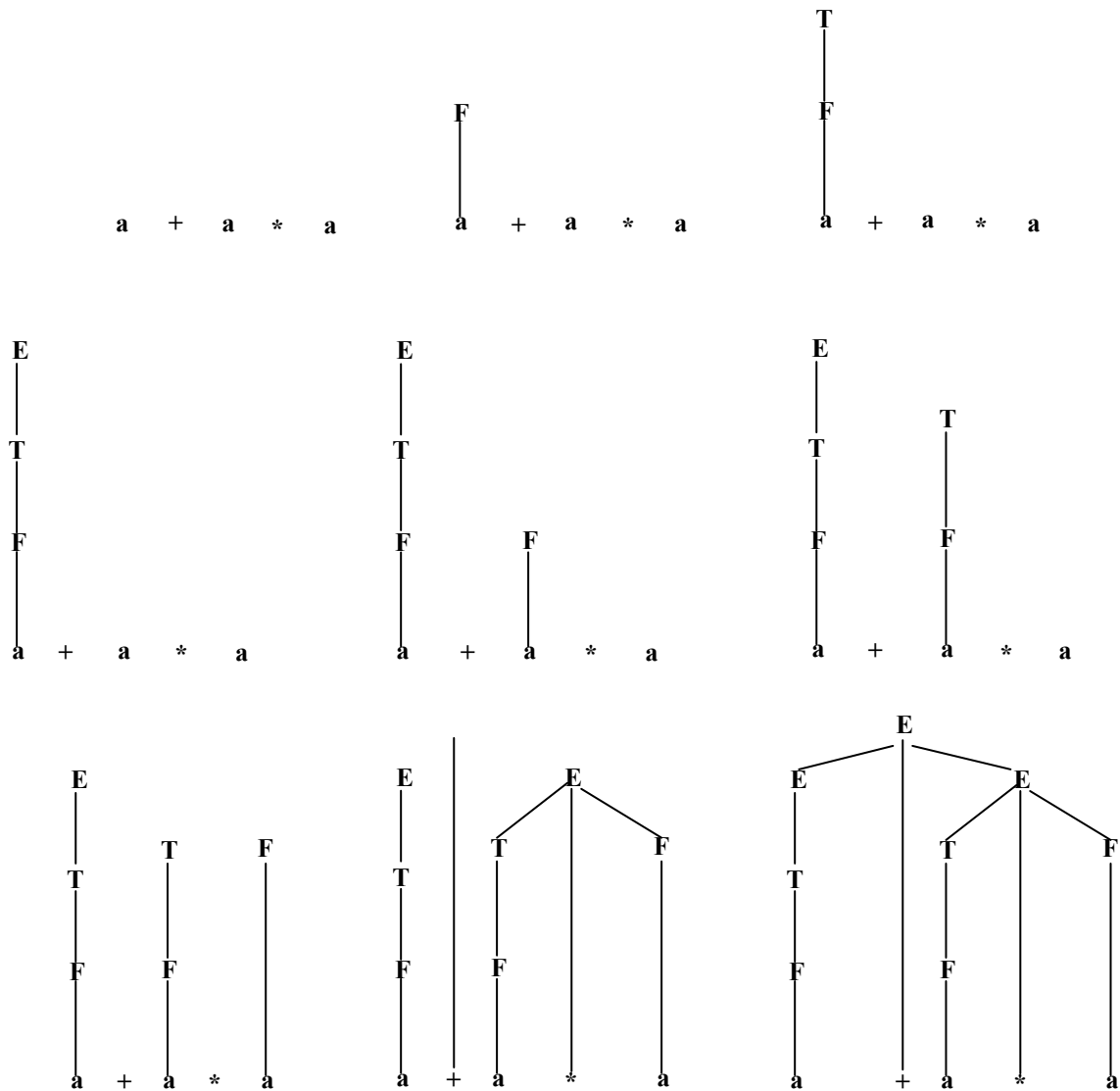


Figura - análise ascendente

Embora a árvore de derivação seja usada para descrever os métodos de análise, na prática ela nunca é efetivamente construída. Às vezes, se necessário, construímos "árvores sintáticas", que guardam alguma semelhança com a árvore de derivação, mas ocupam um espaço de memória significativamente menor. A única estrutura de dados necessária para o processo de análise é uma pilha, que guarda informação sobre os nós da árvore de derivação relevantes, em cada fase do processo. No caso da análise descendente, os nós relevantes são aqueles ainda não expandidos; no caso da análise ascendente, são as raízes das árvores que ainda não foram reunidas em árvores maiores.

5.3 AUTÔMATO DE PILHA

Um autômato com pilha (PDA), também denominado *push down automata*, é um dispositivo formal não-determinístico reconhecedor de linguagens livre de contexto. Um PDA é um modelo natural de um analisador sintático.

5.4 TIPOS DE ANALISADORES SINTÁTICOS

Um analisador sintático (ou *parser*) é um algoritmo capaz de construir uma derivação para qualquer sentença em alguma linguagem baseado em uma gramática. Um *parser* pode também ser visto como um mecanismo para a construção de árvores de derivação.

- **Parse ascendente e parse descendente**

Seja $G = (V_N, V_T, P, S)$ uma GLC com as produções numeradas de 1 a p e x uma sentença tal que exista uma derivação $S \Rightarrow_G^+ x$. **A seqüência formada pelo número das produções utilizadas na derivação $S \Rightarrow_G^+ x$ constitui o *parse* de x em G .** O *parse ascendente* (*bottom-up*) é constituído pela seqüência invertida dos números das produções utilizadas em $S \Rightarrow_{DIR}^+ x$, onde \Rightarrow_{DIR} denota a derivação mais à direita. O *parse descendente* (*top-down*) é constituído pela seqüência dos números das produções utilizadas em $S \Rightarrow_{ESQ}^+ x$, onde \Rightarrow_{ESQ} denota a derivação mais à esquerda.

Abaixo encontram-se o *parse* ascendente e o *parse* descendente de uma sentença x :

EXEMPLO:

Seja G a seguinte gramática GLC:

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

para a sentença $x = id * id$, tem-se:

parse ascendente: $E \Rightarrow_{DIR}^+ x = 64632$, ou seja,
 $E \rightarrow_2 T \rightarrow_3 T * F \rightarrow_6 T * id \rightarrow_4 F * id \rightarrow_6 id * id$

parse descendente: $E \Rightarrow_{ESQ}^+ x = 23466$, ou seja,
 $E \rightarrow_2 T \rightarrow_3 T * F \rightarrow_4 F * F \rightarrow_6 id * F \rightarrow_6 id * id$

Existem duas classes fundamentais de analisadores sintáticos, definidas em função da estratégia utilizada na análise: os **analisadores ascendentes** que procuram chegar ao símbolo inicial da gramática a partir da sentença a ser analisada, olhando a sentença, ou parte dela para decidir qual produção será utilizada na redução; e os **analisadores descendentes** que procuram chegar à sentença a partir do símbolo inicial de G , olhando a sentença ou parte dela para decidir que produção deverá ser usada na derivação.

5.4.1 Analisador sintático ascendente (*bottom-up*)

Um analisador sintático ascendente constrói a redução mais à esquerda da sentença de entrada tentando chegar ao símbolo inicial da gramática. A formulação dos algoritmos de análise sintática ascendente baseia-se em um algoritmo primitivo denominado **algoritmo geral *shift-reduce***. O algoritmo *shift-reduce* utiliza:

- uma pilha sintática, inicialmente vazia;
- um *buffer* de entrada, contendo a sentença a ser analisada;
- uma gramática GLC com as produções numeradas de 1 a p;
- um procedimento de análise que consiste em: transferir (*shift*) os símbolos da entrada um a um para a pilha até que o lado direito de uma produção apareça no topo da pilha. Quando isto ocorrer, o lado direito da produção deve ser trocado (*reduced*) pelo lado esquerdo da produção em questão. Esse processo deve ser repetido até que toda a sentença de entrada tenha sido analisada. Ao final do processo, a sentença estará sintaticamente correta se e somente se a entrada estiver vazia e a pilha sintática contiver apenas o símbolo inicial da gramática. Observa-se que as reduções são prioritárias em relação às transferências.

Pode-se enumerar as seguintes deficiências do algoritmo acima: detecta erro sintático após analisar toda a sentença; pode rejeitar sentenças corretas, visto que nem sempre que o lado direito de uma produção aparece na pilha a ação correta é uma redução. Dessa forma, o método *shift-reduce* pode ser caracterizado como não-determinístico.

O problema de não-determinismo pode ser contornado através do uso da técnica de *back-track*. Contudo, o uso dessa técnica inviabiliza o método na prática em função do tempo e do espaço requeridos para implementação. Na prática, as técnicas ascendentes de análise sintática superam as deficiências do algoritmo *shift-reduce* por serem determinísticas, ou seja, em qualquer situação existe somente uma ação a ser efetuada; e por detectarem erros sintáticos no momento da ocorrência.

Existem várias classes de analisadores sintáticos ascendentes. No entanto, apenas duas classes têm utilização prática:

- a classe de **analisadores LR (*left-to-right*)** tem como principais técnicas os métodos SLR(1), LALR(1) e LR(1) em ordem crescente no sentido de força de abrangência da gramática e complexidade de implementação. Esses analisadores são chamados LR porque analisam as sentenças da esquerda para a direita (*left-to-right*) e constroem uma árvore de derivação mais à direita na ordem inversa. Além do procedimento de análise do algoritmo *shift-reduce*, um analisador LR compõe-se também de uma tabela de análise sintática (ou tabela de *parsing*) específica para cada técnica e para cada gramática. Os analisadores LR são mais gerais que os outros analisadores ascendentes e que a maioria dos descendentes sem *back-track*. No entanto, a construção da tabela de *parsing*, bem como o espaço necessário para armazená-la, são complexos.
- a classe de **analisadores de precedência** compreende os métodos de precedência simples, precedência estendida e precedência de operadores. Esses analisadores também se baseiam no algoritmo *shift-reduce* acrescido de relações de precedência entre os símbolos da gramática, as quais definem de forma determinística a ação a ser efetuada em uma dada situação.

5.4.2 Analisador sintático descendente (*top-down*)

Um analisador sintático descendente constrói a derivação mais à esquerda da sentença de entrada a partir do símbolo inicial da gramática. Como os analisadores ascendentes, também os analisadores descendentes podem ser implementados com ou sem *back-track*. No entanto, embora as implementações com *back-track* permitam a análise de um número maior de gramáticas (GLC), o uso dessas técnicas dificulta a recuperação de erros e causa problemas na análise semântica e geração de código, além de aumentar o tempo necessário para a análise. Já as implementações sem *back-track* limitam o conjunto de gramáticas que podem ser analisadas

mas superam as deficiências das anteriores. Assim sendo, para uma gramática GLC poder ser analisada por um analisador sintático descendente sem *back-track*, deve satisfazer as seguintes condições:

- não possuir recursão à esquerda;
- ser determinística (estar fatorada), isto é, se $X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ são as produções para o não-terminal X , então $\text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) \cap \dots \cap \text{FIRST}(\alpha_n) = \emptyset$
- para todo $X \in V_N$, tal que $X \Rightarrow^* \varepsilon$, $\text{FIRST}(X) \cap \text{FOLLOW}(X) = \emptyset$.

Serão estudadas duas técnicas de análise sintática descendente: análise descendente recursiva e análise preditiva LL(1). A técnica de análise descendente recursiva consiste na construção de um conjunto de procedimentos, normalmente recursivos, um para cada símbolo não-terminal da gramática em questão. É uma técnica simples que aceita uma classe maior de gramáticas considerando que a última condição pode ser relaxada, no entanto, exige a implementação de procedimentos específicos para cada gramática. Assim, por exemplo, para a gramática:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \varepsilon \\ T &\rightarrow (E) \mid \text{id} \end{aligned}$$

5.5 A IMPLEMENTAÇÃO DA ANÁLISE *SHIFT-REDUCE* ATRAVÉS DE PILHAS

A maneira conveniente de implementar o esquema de *shift-reduce* é através de uma **pilha** que conterá os símbolos da gramática e de um *buffer* que conterá a sentença a ser reconhecida.

Tem-se ainda a tabela de precedência de operadores(tabela de parsing) que determinará as relações de precedência entre os operadores e será consultada para determinar o curso de ação a tomar em cada passo.

Convencionando que $\$$ é o símbolo inicial da pilha e, ainda, marca o final da sentença a ser reconhecida, e que w é a sentença a ser reconhecida teremos inicialmente:

PILHA: $\$$ e BUFFER: $w\$$

O analisador opera “passando”(shifting) zero ou mais símbolos de entrada para a pilha até que um *handle* se encontre no topo da pilha. Então o analisador reduz aquele *handle* ao apropriado lado esquerdo da produção. Esse ciclo é repetido até um erro ser detectado ou chegar-se ao símbolo inicial da gramática e o buffer estiver vazio (neste caso a sentença é aceita), ou seja:

PILHA: $\$S$ e BUFFER: $\$$

Existem quatro possíveis cursos de ação no analisador *shift-reduce*, quais sejam: SHIFT, REDUCE, ACEITAÇÃO E ERRO.

Devemos observar que gramáticas ambíguas não podem ser reconhecidas com analisadores *shift-reduce*, mas podemos adaptá-las, eliminando ambigüidades.

Chamamos a classe de gramáticas que podem ser reconhecidas através de um analisador *shift-reduce* de gramática LR. Dentro dessa classe, particularizamos um grupo menos mas bastante importante, cujos reconhedores podem ser construídos de uma forma bem menos complexa, os analisadores de gramáticas de precedência de operadores. Apresentaremos a seguir as duas categorias.

5.5.1 O Analisador de gramáticas de precedência de operadores

As gramáticas de precedências de operadores se caracterizam por:

- Não gerarem sentença vazia ϵ ;
- O lado direito das produções das gramáticas não contém dois não-terminais adjacentes;
- Os operadores têm precedência uns sobre os outros.

Exemplo:

A gramática $G = (\{E\}, \{id, +, -, *, /, (,)\}, P, E)$,

onde $P: E \Rightarrow E+E \mid E-E \mid E * E \mid E/E \mid E-E \mid id$ é uma gramática de precedência de operadores.

A mesma gramática com P :

$E \Rightarrow EAE \mid (E) \mid -E \mid id$

$A \Rightarrow + \mid - \mid * \mid /$ **não é** uma gramática de precedência de operadores.

Na gramática do exemplo anterior, a precedência dos operadores é dada pela seguinte tabela:

	Id	+	-	/	*	()	\$
Id	E3	>	>	>	>	E3	>	>
+	<	>	>	<	<	<	>	>
-	<	>	>	<	<	<	>	>
/	<	>	>	>	>	<	>	>
*	<	>	>	>	>	<	>	>
(<	<	<	<	<	<	=	
)	E3	>	>	>	>	E3	>	>
\$	<	<	<	<	<	<	E2	E1

O Algoritmo de análise funciona da seguinte forma:

- O reconhecimento é feito da esquerda para a direita;
- Quando a relação de precedência entre o símbolo do topo da pilha e a posição atual da sentença for $<$ ou $=$ é feito um *shift* (empilha símbolo corrente do *buffer* na pilha);
- Quando a relação de precedência entre o símbolo do topo da pilha e a posição atual da sentença for $>$ é feito um *reduce* na pilha. Nesse momento deve ser feita uma verificação dos símbolos que circundam o *handle* para verificar se há erros.

UMA descrição em pseudo-linguagem desse algoritmo pode ser encontrada na pg. 206 do Aho/Ullman.

Nos casos em que nenhuma relação de precedência têm-se situações de erro (e1:"**falta operando**", e2:"**parênteses não balanceados**", e3:"**falta operador**" e e4:"**falta parênteses da direita**". Nesses casos, uma rotina deveria ser chamada para emitir uma mensagem de erro e agir na pilha e no buffer para que a análise possa continuar.

Nos casos em que nenhuma relação de precedência tem-se situações de erro (e1:"falta operando", e2:"parênteses não balanceados", e3:"falta operador" e e4:"falta parênteses da direita".

Existem duas situações de detecção de erros, quais sejam:

- Se não existe relação de precedência determinada na tabela entre o terminal da pilha e o terminal corrente no buffer. No exemplo anterior, e1, e2, e3, e4;
- Numa situação de *reduce*, se não existe lado direito nas produções igual ao *handle* que está na pilha. Para o exemplo dado anteriormente, teríamos basicamente as seguintes situações de erros em *reduce* possíveis:
- Se +, -, * ou / são reduzidos, verifica-se se aparecem terminais em ambos os lados. Se não, diagnostica-se um erro do tipo “falta um operando”;
- Se id é reduzido, verifica-se se aparece um não terminal em algum dos lados. Se aparecer diagnostica-se um erro do tipo “falta operador”;
- Se () é reduzido, verifica-se se aparece um não-terminal entre os parênteses. Se não houver, diagnostica-se um erro do tipo “**nenhum expressão entre os parênteses**”. Ainda nesse caso, verifica-se se aparece um não terminal aparece do lado externo aos parênteses. Se aparecer, diagnostica-se um erro do tipo “falta operador”.

Exemplo:

Para a sentença incorreta **id +)**, as ações seriam as seguintes (símbolo da pilha na primeira coluna e símbolo do buffer na primeira linha):

- PILHA: \$, BUFFER: id +)\$. Assim, \$ < id, logo SHIFT id;
- PILHA: \$id, BUFFER: +)\$\$. Assim, id > +, logo uma redução deve ser feita. Como temos o *handle* id, podemos fazer REDUCE id para E;
- PILHA: \$E, BUFFER: +) \$. Assim, \$ < +, logo SHIFT +;
- PILHA: \$E+; BUFFER:)\$. Assim, + >), logo uma redução deve ser feita. Entretanto não temos *handle* na pilha, por isso não podemos fazer o REDUCE (E + E para E). Por isso tem-se uma situação de erro de “falta de operando”. A mensagem deveria ser emitida e a redução é feita da mesma maneira para continuar o processo;
- PILHA: \$E, BUFFER:)\$. Não existe na tabela precedência entre \$ e), logo chama-se a rotina de erro E2;
- Agora temos a configuração final do Analisador: PILHA: \$E BUFFER: \$

Exercícios de autômatos de pilha:

- E \Rightarrow E + E
- E \Rightarrow E - E
- E \Rightarrow E * E
- E \Rightarrow E / E
- E \Rightarrow (E)

$E \Rightarrow -E$

$E \Rightarrow id$

- a) $id1 * id2 + id3 * (id4 + id5)$
- b) $(id1 + id2) * (id3 - id4)$
- c) $(id1 - id2 * (id3 + id4)) / id5$
- d) $id1 + id2 / id3 * id4$
- e) $id1 + id2 * (id3/id4)$
- f) $id1 / (id2 - id3) * id4$
- g) $id1 * (id2+id3)$
- h) $id1 + id2 * id3 + (id4* id5)$
- i) $id1 * id2 * id3 * (id4 + id5)$
- j) $id1 * (id2 + id3 * (id4 + id5))$
- k) $id1 * (+id2)$
- l) $id1 + ((id2*id3) - id4)$

5.6 SIMPLIFICAÇÕES DE GRAMÁTICAS LIVRES DE CONTEXTO

Uma GLC usada na representação de uma linguagem de programação eventualmente deve apresentar determinadas características para poder ser aplicada na implementação de determinados métodos de análise sintática. Assim, uma GLC: não pode possuir símbolos inúteis; deve ser ϵ -livre; não pode possuir produções unitárias; não pode ser ambígua; deve estar fatorada, ou seja, deve ser determinística; não pode possuir recursão à esquerda.

Existem várias maneiras de restringir as produções de uma gramática livre de contexto sem reduzir seu poder expressivo. Se L é uma linguagem livre de contexto não-vazia, então L pode ser gerada por uma gramática livre de contexto G com as seguintes propriedades :

- a) Cada variável e cada terminal de G aparecem na derivação de alguma palavra de L .
- b) Não há produções da forma $A ::= B$, onde A e B são variáveis.

Além disso, se ϵ não está em L , então não há necessidade de produções da forma $A ::= \epsilon$.

5.6.1 Símbolos Inúteis ou Improdutivos

A exclusão de símbolos inúteis (não-usados na geração de palavras de terminais) é realizada excluindo as produções que fazem referência a estes símbolos, bem como os próprios símbolos inúteis. Não é necessária qualquer modificação adicional nas produções da gramática. O algoritmo apresentado é dividido em **duas etapas**, como segue :

a) **Qualquer variável gera palavra (sentença) de terminais**. O algoritmo gera um novo conjunto de variáveis, analisando as produções da gramática a partir de terminais gerados. Inicialmente, considera todas as variáveis que geram terminais diretamente (exemplo : $A \rightarrow a$). A seguir, sucessivamente são adicionadas as variáveis que geram palavras de terminais indiretamente (exemplo : $B \rightarrow Ab$);

b) **Qualquer símbolo é atingível a partir do símbolo inicial**. Após a execução da etapa acima, o algoritmo analisa as produções da gramática a partir do símbolo inicial. Inicialmente, considera exclusivamente o símbolo inicial. Após, sucessivamente as produções da gramática são aplicadas e os símbolos referenciados adicionados aos novos conjuntos.

Pode-se eliminar os símbolos inúteis de uma gramática sem prejudicar seu potencial expressivo. Um símbolo X é útil se existe uma derivação $S \rightarrow^* \alpha X \beta \rightarrow^* w$, para algum w , α e β , onde w é uma cadeia de T e α e β são cadeias quaisquer de variáveis e terminais. Caso contrário, o símbolo X é inútil. Tanto terminais quanto não-terminais podem ser úteis ou inúteis. Se o símbolo inicial for inútil, então a linguagem definida pela gramática é vazia.

Há dois tipos de símbolos inúteis :

- 1 - aqueles que não geram nenhuma cadeia de terminais e;
- 2 - aqueles que jamais são gerados a partir de S .

O primeiro caso corresponde aos símbolos improdutivos (ou *mortos*), e o segundo caso corresponde aos símbolos inalcançáveis. Um terminal sempre é produtivo, mas pode ser inalcançável. Já o não-terminal pode ser tanto improdutivo quanto inalcançável, mas o símbolo inicial sempre é alcançável. A seguir serão vistos algoritmos para eliminar tanto o primeiro quanto o segundo tipo de símbolos inúteis.

O algoritmo para eliminação dos símbolos improdutivos é baseado na idéia de que se um não-terminal A tem uma produção consistindo apenas de símbolos produtivos, então o próprio A é produtivo.

Algoritmo: Eliminação de Símbolos Improdutivos

Entrada : Uma GLC $G=(N,T,P,S)$

Saída : Uma GLC $G'=(N',T,P',S)$, sem símbolos improdutivos.

$SP := T \cup \{\epsilon\}$

Repita

$Q := \{X \mid X \in N \text{ e } X \notin SP \text{ e (existe pelo menos uma produção } X ::= X_1X_2, \dots, X_n \text{ tal que } X_1 \in SP, X_2 \in SP, \dots, X_n \in SP)\};$

$SP := SP \cup Q;$

Até $Q = \emptyset;$

$N' = SP \cap N;$

Se $S \in SP$ Então

$P' := \{p \mid p \in P \text{ e todos os símbolos de } p \text{ pertencem a } SP\}$

Senão " $L(G) = \emptyset$ ";

$P' := \emptyset;$

Fim Se

No seguinte exemplo, para facilitar o acompanhamento do algoritmo, os símbolos do conjunto SP serão sublinhados a cada passo.

Seja G a seguinte gramática :

$S ::= \underline{A}SB \mid BSA \mid SS \mid aS \mid \epsilon$

$A ::= \underline{A}BS \mid B$

$B ::= BSSA \mid A$

Inicialmente $SP = \{a\}$, e são marcadas as produções :

$S ::= \underline{a}S$

$S ::= \underline{\epsilon}$

Uma vez que o lado direito da produção $S ::= \epsilon$ está completamente marcado, deve-se marcar todas as ocorrências de S . Como S é o único novo símbolo a ser marcado, então $Q := \{S\}$. As marcações resultam:

$$\begin{aligned} \underline{S} &::= \underline{A}\underline{S}B \mid B\underline{S}A \mid \underline{S}\underline{S} \mid \underline{a}S \mid \underline{\epsilon} \\ \underline{A} &::= \underline{A}B\underline{S} \mid B \\ \underline{B} &::= \underline{B}\underline{S}\underline{S}A \mid A \end{aligned}$$

Neste momento $SP = \{a, S\}$. Agora $Q := \emptyset$, e assim a repetição do algoritmo termina com $SP = \{a, S\}$, os únicos símbolos produtivos.

Para simplificar a gramática, os símbolos improdutivos podem ser removidos, bem como todas as produções de P que contenham estes símbolos. Assim, a gramática simplificada para o exemplo seria:

$$S ::= SS \mid aS \mid \epsilon$$

🔴 **Dicas para a eliminação de símbolos inúteis :**

- 1º Todo **T** (terminal) é produtivo;
- 2º Marcar todos os **T** (terminais);
- 3º Marcar todos os **NT** (não-terminais) que possuem produções totalmente marcadas;
- 4º Marcar nas outras produções os **NT** que já foram marcados;
- 5º Eliminar os estados e as produções não marcadas.

Obs : Se o inicial for inútil, a linguagem (L) é vazia.

Obs II : Quando inútil, o símbolo não encaminha a finalização de uma sentença.

O segundo tipo de símbolos inúteis são aqueles que jamais são gerados a partir de S . Estes símbolos, chamados de inalcançáveis, são eliminados pelo seguinte algoritmo:

Algoritmo: Eliminação de Símbolos Inalcançáveis

Entrada : Uma GLC $G=(N,T,P,S)$

Saída : Uma GLC $G'=(N',T,P',S)$, sem símbolos inalcançáveis.

$SA := \{S\}$

$M := \{S\}$ "o conjunto de símbolos marcados".

Repita

$M := \{X \mid X \in N \cup T \text{ e } X \notin SA \text{ e existe uma produção } Y ::= \alpha X \beta \text{ e } Y \in SA\};$

$SA := SA \cup M;$

Até $M = \emptyset;$

$N' = SA \cap N;$

$T' = SA \cap T;$

$P' := \{p \mid p \in P \text{ e todos os símbolos de } p \text{ pertencem a } SA\};$

Para exemplificar, seja G :

$$\begin{aligned} S &::= aS \mid SB \mid SS \mid \epsilon \\ A &::= ASB \mid c \\ B &::= b \end{aligned}$$

Inicialmente, $SA = \{S\}$ e $M = \{S\}$. Assim, se obtém :

$$S ::= \underline{aS} \mid \underline{SB} \mid \underline{SS} \mid \varepsilon$$

e a e B são os únicos símbolos recém marcados que não estão em SA . Assim, a M é atribuído o conjunto $\{a, B\}$ e a SA é atribuído $\{S, a, B\}$. Em seguida se obtém :

$$B ::= \underline{b}$$

e assim M passa a ser igual a $\{b\}$ e SA igual a $\{S, a, B, b\}$. Já que M não contém não-terminais, a execução do laço deixará $M = \emptyset$ e SA não será alterado. Assim, S, a, B e b são alcançáveis e A e c são inalcançáveis e suas produções podem ser eliminadas da gramática, resultando :

$$\begin{aligned} S &::= aS \mid SB \mid SS \mid \varepsilon \\ B &::= b \end{aligned}$$

☛ *Dicas para a eliminação de símbolos inalcançáveis :*

Os símbolos inalcançáveis são aqueles que partindo do inicial S , não são alcançados por n transições.

5.6.2 ε - Produções

Conforme foi dito anteriormente, uma **GLC** pode ter produções do tipo $A ::= \varepsilon$. Mas toda **GLC** pode ser transformada em uma **GLC** equivalente sem este tipo de produções (chamadas **ε -produções**), com exceção da produção $S ::= \varepsilon$, se esta existir (ou seja, a cadeia vazia pertence à linguagem). Assim procedendo, é possível mostrar que toda **GLC** pode obedecer à restrição das **GSC** (tipo 1). É o método da eliminação de **ε -produções**, da qual se tratará agora.

A exclusão de **ε -produções** pode determinar modificações diversas nas produções da gramática. O algoritmo é dividido em três etapas, como segue :

a) **Variáveis que constituem produções vazias.** Considera, inicialmente, todas as variáveis que geram diretamente ε (ex: $A \rightarrow \varepsilon$). A seguir, sucessivamente são determinadas as variáveis que indiretamente geram ε (ex: $B \rightarrow A$);

b) **Exclusão de produções vazias.** Inicialmente, são consideradas todas as produções não-vazias. A seguir, cada produção cujo lado direito possui uma variável que gera a palavra vazia, determina uma produção adicional, sem esta variável;

c) **Inclusão de geração da palavra vazia, se necessário.** Se a palavra vazia pertence à linguagem, então é incluída uma produção para gerar a palavra vazia.

O método consiste em determinar, para cada variável A em N , se $A \rightarrow^* \varepsilon$. Se isto for verdade, se diz que a variável A é anulável. Pode-se assim substituir cada produção da forma $B ::= X_1 X_2 \dots X_n$ por todas as produções formadas pela retirada de uma ou mais variáveis X_i anuláveis.

Exemplo :

A ::= BCDe
 B ::= ε | e
 C ::= ε | a
 D ::= b | cC

Neste caso, as variáveis anuláveis são B e C, assim a gramática pode ser transformada na seguinte :

A ::= BCDe | CDe | BDe | De
 B ::= e
 C ::= a
 D ::= b | c | cC

Os não-terminais que derivam a sentença ε são chamados de ε-não-terminais. Um não-terminal A é um ε-não-terminal se existir uma derivação $A \rightarrow^* \epsilon$ em G. Note que ε está em L(G) se e somente se S é um ε-não-terminal. Se G não tem ε-não-terminais, ela é dita ε-livre.

Se G tem ε-produções, então em uma derivação sentencial da forma : $S \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_n$, os tamanhos das sentenças irão variar não-monotonicamente um em relação ao outro, isto é, ora aumentam, ora diminuem. Entretanto, se G não tem **ε-produções**, então :

$$|S| \leq |\alpha_1| \leq |\alpha_2| \leq \dots \leq |\alpha_n|$$

isto é, os tamanhos das sentenças são monotonicamente crescentes. Esta propriedade é útil quando se quer testar se uma dada palavra é ou não gerada por uma **GLC**.

Antes de apresentar um algoritmo para eliminar as **ε-produções**, será apresentado um algoritmo para encontrar os **ε-não-terminais**.

Algoritmo: Encontrar o Conjunto dos ε-não-terminais

Entrada : Uma GLC $G=(N,T,P,S)$

Saída : O conjunto E dos ε-não-terminais.

E := {ε}

Repita

Q := {X | X ∈ N e X ∉ E e existe pelo menos uma produção $X ::= Y_1Y_2\dots Y_n$ tal que $Y_1 \in E, Y_2 \in E, \dots, Y_n \in E$ };

E := E ∪ Q;

Até Q = ∅;

Seja G a seguinte gramática :

S ::= aS | SS | bA
 A ::= BC
 B ::= CC | ab | aAbC
 C ::= ε

Inicialmente temos $E = \{\epsilon\}$.

Quando o laço é executado pela primeira vez, $Q = \{C\}$, e se obtém :

$S ::= aS \mid SS \mid bA$ $A ::= \underline{BC}$ $B ::= \underline{CC} \mid ab \mid aAbC$ $C ::= \epsilon$

Como Q não é vazio, temos $E = \{\epsilon, C\}$ e uma segunda iteração, que deixa $Q = \{B\}$, e obtém-se :

$S ::= aS \mid SS \mid bA$ $A ::= \underline{BC}$ $B ::= \underline{CC} \mid ab \mid aAbC$ $C ::= \epsilon$

Novamente Q não é vazio, temos $E = \{\epsilon, B, C\}$, e numa nova iteração, que faz $Q = \{A\}$

$S ::= aS \mid SS \mid b\underline{A}$ $A ::= \underline{BC}$ $B ::= \underline{CC} \mid ab \mid a\underline{AbC}$ $C ::= \epsilon$

O símbolo **A** é acrescentado a **E**, que passa a ser $E = \{\epsilon, A, B, C\}$. Na próxima iteração não são acrescentados símbolos a **Q**, terminando assim o algoritmo, e temos que os ϵ -não-terminais em **G** são **A**, **B** e **C**.

Para eliminar os ϵ -não-terminais de uma **GLC**, pode-se usar o seguinte algoritmo, que elimina ϵ -não-terminais sem introduzir novos. A estratégia é baseada na seguinte idéia: Seja **A** um ϵ -não-terminal em **G**. Então ele é dividido conceitualmente em dois não-terminais **A'** e **A''**, tal que **A'** gera todas as cadeias não-vazias e **A''** gera apenas ϵ . Agora, do lado direito de cada produção onde **A** aparece uma vez, por exemplo $B ::= \alpha A \beta$, é trocado por duas produções $B ::= \alpha A' \beta$ e $B ::= \alpha A'' \beta$. Já que **A''** só gera ϵ , ele pode ser trocado por ϵ (ou seja, eliminado), deixando $B ::= \alpha \beta$. Depois disso, pode-se usar **A** em lugar de **A'**. A produção final fica: $B ::= \alpha \beta \mid \alpha A \beta$, onde **A** não é mais um ϵ -não-terminal. Se $B ::= \alpha A \beta A \gamma$, então são obtidas quatro combinações possíveis pelas trocas de **A** por ϵ ($\alpha A \beta A \gamma$, $\alpha A \beta \gamma$, $\alpha \beta A \gamma$, $\alpha \beta \gamma$) e assim por diante.

Algoritmo: Eliminar Todos os ϵ -não-terminais

Entrada : Uma GLC $G=(N,T,P,S)$

Saída : Uma GLC $G=(N',T,P',S')$ ϵ -livre.

Construa o conjunto E

$P' := \{p \mid p \in P \text{ e } p \text{ não é } \epsilon\text{-produção}\}$

Repita

Se P' tem uma produção da forma $A ::= \alpha B \beta$, tal que $B \in E$, $\alpha \beta \in (N \cup T)^*$ e $\alpha \beta \neq \epsilon$, então inclua a produção $A ::= \alpha \beta$ em P' ;

Até que nenhuma nova produção possa ser adicionada a P' ;

Se $S \in E$, então

Adicione a P' as produções $S' ::= S \mid \epsilon$;

$N' := N \cup \{S'\}$;

Senão

$S' := S$;

$N' := N$;

Fim Se

*** Dica : Como eliminar as ϵ -produções.**

1º Se o símbolo inicial S possui ϵ -produção, cria a seguinte regra : $S' ::= S \mid \epsilon$

2º Riscar as ϵ -produções de todas as regras

3º marcar os **NT** (não-terminais) que possuem ϵ -produções indiretas, ou seja, muda para um estado onde tenha ϵ -produções.

4º Onde houver **NT** marcados, fazer novas combinações, criando novas produções.

5.6.3 Produções Unitárias

Uma produção na forma $A \rightarrow B$ não adiciona informação alguma em termos de geração de palavras, a não ser que a variável A pode ser substituída por B . Neste caso, se $B \rightarrow \alpha$, então a produção $A \rightarrow B$ pode ser substituída por $A \rightarrow \alpha$.

Uma produção da forma $A ::= \alpha$ em uma **GLC** $G=(N,T,P,S)$ é chamada de *produção unitária* se α é um não-terminal. Um caso especial de produção unitária é a produção $A ::= A$, também chamada *produção circular*. Tal produção pode ser removida imediatamente sem afetar a capacidade de geração da gramática. O algoritmo para eliminar produções unitárias assume que as produções circulares já tenham sido eliminadas anteriormente. Essa eliminação é trivial, pois a identificação de produções circulares é bastante simples.

Algoritmo: Eliminar Produções Unitárias

Entrada : Uma GLC $G=(N,T,P,S)$ sem produções circulares.

Saída : Uma GLC $G=(N,T,P',S)$ sem produções unitárias.

Para toda $A \in N$ faça:

$N_A := \{B \mid A \rightarrow^* B, \text{ com } B \in N\}$;

Fim Para

$P' := \emptyset$;

Para toda produção $B ::= \alpha \in P$ faça:

Se $B ::= \alpha$ não é uma produção unitária, então:

$P' := P' \cup \{A ::= \alpha \mid B \in N_A\}$;

Fim Se

Fim Para

Podem surgir símbolos inalcançáveis depois da aplicação deste algoritmo. Isso se deve ao fato de que o símbolo é substituído por suas produções, e se o mesmo aparecer somente em produções unitárias, o mesmo irá desaparecer do lado direito das produções. Para exemplificar, aplique o algoritmo na seguinte gramática:


```
S ::= aB | aA | A // substitui A pelas suas produções
B ::= aA | B | A // idem com A e B
A ::= C | aaa // idem com C
C ::= ε | cBc
```

Observação : Depois de eliminar produções unitárias deve-se eliminar produções inalcançáveis, e transformar a GLC em GLC ϵ -livre se necessário.

5.6.4 Fatoração

Uma GLC está *fatorada* se ela é determinística, ou seja, se ela não possui produções para um mesmo não-terminal no lado esquerdo cujo lado direito inicie com a mesma cadeia de símbolos ou com símbolos que derivam seqüências que iniciem com a mesma cadeia de símbolos. De acordo com esta definição, a seguinte gramática é não-determinística:

$S ::= aSB \mid aSA$
 $A ::= a$
 $B ::= b$



 fonte do
 não-determinismo

Para fatorar uma GLC, deve-se alterar as produções envolvidas no não-determinismo da seguinte forma:

a) As produções com não-determinismo direto da forma $A ::= \alpha\beta \mid \alpha\gamma$ serão substituídas por:

$$\begin{aligned} A & ::= \alpha A' \\ A' & ::= \beta \mid \gamma \end{aligned}$$

Na gramática acima, temos $\alpha = aS$, assim, com a eliminação do não-determinismo, teríamos a seguinte gramática :

$$\begin{aligned} S & ::= aSS' \\ S' & ::= B \mid A \\ A & ::= a \\ B & ::= b \end{aligned}$$

b) Não-determinismo indireto é retirado através de sua transformação em não-determinismo direto (através de derivações sucessivas) e posterior eliminação segundo o item (a). Tomemos por exemplo a seguinte gramática :

$$\begin{aligned} S & ::= AC \mid BD \\ A & ::= aD \mid cB \\ B & ::= aB \mid dD \\ C & ::= eC \mid eA \\ D & ::= fD \mid AB \end{aligned}$$

Onde o não-determinismo indireto se dá pelo fato de que A e B podem iniciar com o terminal a . Assim, transformaremos primeiro o não-determinismo indireto em não-determinismo direto, substituindo os não-terminais A e B pelas suas produções em S :

$$\begin{aligned} S & ::= aDC \mid cBC \mid aBD \mid dDD \\ A & ::= aD \mid cB \\ B & ::= aB \mid dD \\ C & ::= eC \mid eA \\ D & ::= fD \mid AB \end{aligned}$$

O próximo passo é simples, bastando agora eliminar o não-determinismo direto, segundo a regra do item a, ou seja:

$$\begin{aligned} S & ::= aS' \mid cBC \mid dDD \\ S' & ::= DC \mid BD \\ A & ::= aD \mid cB \\ B & ::= aB \mid dD \\ C & ::= eC' \\ C' & ::= C \mid A \\ D & ::= fD \mid AB \end{aligned}$$

☛ **Dica : Fatoração.**

- 1º Eliminar indeterminismo direto.
- 2º Eliminar indeterminismo indireto.
- 3º Fazer substituições para descobrir indeterminismos indiretos.

Obs : fazer 1º os diretos e depois os indiretos.

5.6.5 Eliminação de Recursão à Esquerda

Um não-terminal A , em uma GLC $G=(N,T,P,S)$ é *recursivo* se $A \rightarrow \alpha A \beta$, para $\alpha \in (N \cup T)^*$.

Se $\alpha = \varepsilon$, então A é *recursivo à esquerda*; se $\beta = \varepsilon$, então A é *recursivo à direita*. Esta recursividade pode ser direta ou indireta.

Uma gramática com pelo menos um não-terminal recursivo à esquerda ou à direita é uma *gramática recursiva à esquerda* ou *à direita*, respectivamente.

Uma GLC $G=(N,T,P,S)$ possui *recursão à esquerda direta* se P contém pelo menos uma produção da forma $A ::= A\alpha$.

Uma GLC $G=(N,T,P,S)$ possui *recursão à esquerda indireta* se existe em G uma derivação da forma $A \rightarrow^n A\beta$, para algum $n \geq 2$.

Para eliminar as recursões diretas à esquerda nas produções :

$$A ::= A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \dots \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

onde nenhum β_i começa com A , deve-se substituir estas produções pelas seguintes :

$$\begin{aligned} A & ::= \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A' \\ A' & ::= \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_n A' \mid \varepsilon \end{aligned}$$

onde A' é um novo não-terminal.

A seguir é apresentado um algoritmo para eliminar recursões diretas e indiretas à esquerda :

Algoritmo: Eliminar Recursões à Esquerda

Entrada : Uma GLC $G=(N,T,P,S)$.
 Saída : Uma GLC $G=(N',T,P',S)$ sem recursão à esquerda.

$N' := N$;
 $P' := P$;

Ordene os não-terminais de N' em uma ordem qualquer (por exemplo, $A_1, A_2, A_3, \dots, A_n$);
 Para i de 1 até n , faça:
 Para j de 1 até $i-1$ faça
 Substitua as produções de P' da forma $A_i ::= A_j \gamma$ por produções da forma
 $A_i ::= \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$, onde $\delta_1, \delta_2, \dots, \delta_k$ são os lados direitos das produções
 com lado esquerdo A_j , ou seja, $A_j ::= \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$.
 Fim Para;
 Elimine as recursões diretas das produções de P' com lado esquerdo A_i ;
 Fim Para;

Note que este algoritmo já elimina as recursões à esquerda diretas e indiretas.

☛ **Dica : Eliminação de recursão à esquerda.**

- 1º Detectar recursão à esquerda
- 2º Eliminar a recursão, mantendo as não recursivas na mesma regra com X' no final de cada produção
- 3º Criar um estado ' X ' com produções formadas pelas produções recursivas mais ' X ' no final de cada produção. Adicionar no final a produção ϵ
- 4º Verificar se não existem recursões indiretas

5.7 TIPOS ESPECIAIS DE GLC

- A seguir serão identificados alguns tipos especiais de GLC :

a) Gramática Própria: Uma GLC é própria se :

- a.1) Não possui produções cíclicas (ver definição seguinte);
- a.2) É ϵ -livre;
- a.3) Não possui símbolos inúteis.

b) Gramática Sem Ciclos: $G = (N, T, P, S)$ é uma GLC sem ciclos (ou livre de ciclos) se não existe em G nenhuma derivação da forma $A \rightarrow^+ A$, para todo $A \in N$.

c) Gramática Reduzida: Uma GLC $G=(N,T,P,S)$ é uma GLC reduzida se :

- c.1) $L(G)$ não é vazia;
- c.2) Se $A ::= \alpha \in P$ então $A \neq \alpha$
- c.3) G não possui símbolos inúteis

d) Gramática de Operadores: Uma GLC $G=(N,T,P,S)$ é de operadores se ela não possui produções cujo lado direito contenha não-terminais consecutivos.

e) Gramática Unicamente Inversível: Uma GLC reduzida é unicamente inversível se ela não possui produções com lados direitos iguais.

f) Gramática Linear: Uma GLC $G=(N,T,P,S)$ é linear se todas as suas produções forem da forma $A ::= xBw \mid x$, onde A e B pertencem a N , e x e w pertencem a T^* .

g) Forma Normal de Chomsky: Uma GLC está na forma normal de Chomsky se ela é ϵ -livre, e todas as suas produções (exceto, possivelmente, $S ::= \epsilon$) são da forma:

- g.1) $A ::= BC$, com A, B e $C \in N$, ou
- g.2) $A ::= a$; com $A \in N$ e $a \in T$.

h) Forma Normal de Greinbach: Uma GLC está na forma normal de Greinbach se ela é ϵ -livre e todas as suas produções (exceto, possivelmente, $S ::= \epsilon$) são da forma $A ::= a\alpha$ tal que $a \in T, \alpha \in N^*$ e $A \in N$.

5.8 PRINCIPAIS NOTAÇÕES DE GLC

A **BNF** (*Backus Naur Form*) é uma notação utilizada na especificação formal da sintaxe de linguagens de programação. Esta é a forma de especificação de gramáticas que tem sido utilizada neste contexto. As gramáticas a seguir são exemplos de gramáticas descritas na notação BNF:

$$\langle S \rangle ::= a \langle S \rangle | \epsilon \qquad \langle E \rangle ::= \langle E \rangle + id | id$$

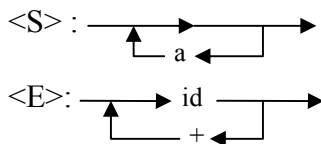
A **BNFE** (Backus Naur Form Extended) é equivalente a BNF, mas permite uma especificação mais compacta da sintaxe de uma linguagem de programação. Os símbolos entre chaves abreviam os fechos das cadeias que eles representam. As mesmas gramáticas descritas acima em BNF podem ser representadas em BNFE da seguinte forma:

$$\langle S \rangle ::= \{a\} \qquad \langle E \rangle ::= id\{+id\}$$

A **RRP** é uma notação de **GLC** onde o lado direito das produções é especificado através de expressões regulares, envolvendo os símbolos de **N** e **T** de uma gramática. As gramáticas dos exemplos acima podem ser descritas em RRP da seguinte forma:

$$\langle S \rangle ::= a^* \qquad \langle E \rangle ::= id(+id)^*$$

O diagrama sintático é uma notação gráfica para GLC's bastante utilizada em análise de linguagem natural e em manuais de linguagens de programação, para descrever graficamente a sintaxe da linguagem. As gramáticas do exemplo acima teriam os seguintes diagramas sintáticos:



5.9 CONJUNTOS FIRST E FOLLOW

5.9.1 Conjunto First

Seja α uma forma sentencial qualquer gerada por **G**. **FIRST(α)** será o conjunto de símbolos terminais que podem iniciar (que podem aparecer na posição mais à esquerda das sentenças derivadas desta forma sentencial) α ou seqüências derivadas (direta ou indiretamente) de α . Além disso, se $\alpha = \epsilon$ ou $\alpha \rightarrow^+ \epsilon$ então $\epsilon \in \text{FIRST}(\alpha)$.

Para calcular **FIRST(X)** para todo $X \in (\mathbf{N} \cup \mathbf{T})^*$, vamos considerar os seguintes casos especiais:

- $X = \epsilon$. Neste caso, $\text{FIRST}(X) = \{\epsilon\}$.
- $X \in \mathbf{T}$. Neste caso, $\text{FIRST}(X) = \{X\}$.
- $X \in \mathbf{N}$. Neste caso, se $X ::= \alpha_1 | \alpha_2 | \dots | \alpha_n$ são todas as produções com lado esquerdo X , então $\text{FIRST}(X) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \dots \cup \text{FIRST}(\alpha_n)$.

O algoritmo genérico do cálculo do conjunto FIRST para uma cadeia qualquer é recursivo e utiliza as definições para os casos especiais acima.

Algoritmo: Cálculo do Conjunto First de uma Cadeia

Entrada: Uma cadeia de símbolos $X \in (N \cup T)^*$ de uma gramática $G=(N,T,P,S)$.

Uma tabela **TABFIRST** com o conjunto **FIRST** de cada não-terminal de G .

Saída: O conjunto **FIRST(X)**, denotado por F .

Se $X = \epsilon$ então:

$F := \{\epsilon\}$.

Senão se $X \in T$ então:

$F = \{X\}$

Senão se $X \in N$ então:

Sejam $X ::= \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ as produções com lado esquerdo X , então

$F := \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \dots \cup \text{FIRST}(\alpha_n)$;

Senão

$i := 0$;

$F := \emptyset$;

Repita:

$i := i+1$;

X_i := o i -ésimo símbolo da cadeia X ;

$F := F \cup \text{TABFIRST}(X_i) - \{\epsilon\}$;

Se $X = \epsilon$ então:

$F := F \cup \{\epsilon\}$;

Fim Se

Até $X \in T$ ou $X = \epsilon$ ou $(X \in N \text{ e } \epsilon \notin \text{TABFIRST}(X))$

Fim Se

Para determinar o conjunto FIRST de cada um dos não-terminais de uma gramática, procede-se como segue:

Crie uma tabela **TABFIRST** cujas linhas são rotuladas com os não-terminais da gramática e com uma coluna rotulada como "FIRST", que irá conter os símbolos pertencentes ao FIRST de cada não-terminal. Inicialize cada uma das posições desta tabela com \emptyset (conjunto vazio).

Para cada um dos não-terminais, calcule o seu conjunto FIRST utilizando o algoritmo acima. Repita o passo 2 até que não ocorra mais nenhuma modificação na tabela.

Para exemplificar, considere a gramática:

$S ::= ABS \mid aA$

$A ::= \epsilon \mid a$

$B ::= Bb \mid cd$

Inicialmente constrói-se a tabela:

N	FIRST
S	\emptyset
A	\emptyset
B	\emptyset

Calculando o FIRST de S pelo algoritmo acima, faz-se $\mathbf{FIRST(S) = FIRST(ABS) \cup FIRST(aA)}$. Como $\mathbf{FIRST(A) = \emptyset}$, na tabela, então $\mathbf{FIRST(S)}$ inicialmente é $\emptyset \cup \{a\}$. A tabela fica:

N	FIRST
S	$\{a\}$
A	\emptyset
B	\emptyset

Calculando o $\mathbf{FIRST(A)}$ temos $\{\epsilon, a\}$. Então a tabela fica:

N	FIRST
S	$\{a\}$
A	$\{\epsilon, a\}$
B	\emptyset

Calculando o $\mathbf{FIRST(B)}$ temos $\mathbf{FIRST(B) = FIRST(Bb) \cup FIRST(cd) = \emptyset \cup \{c\}}$. A tabela fica:

N	FIRST
S	$\{a\}$
A	$\{\epsilon, a\}$
B	$\{c\}$

Como a tabela foi modificada, o passo 2 deve ser repetido. Desta vez, ao calcular o \mathbf{FIRST} de S teremos: $\mathbf{FIRST(S) = FIRST(ABS) \cup FIRST(aA) = \{a\} \cup FIRST(BS) \cup \{a\} = \{a\} \cup \{c\} \cup \{a\} = \{a, c\}}$. Logo a tabela fica:

N	FIRST
S	$\{a, c\}$
A	$\{\epsilon, a\}$
B	$\{c\}$

O cálculo do \mathbf{FIRST} de A e B não modifica a tabela. Mas como houve uma modificação no cálculo do \mathbf{FIRST} de S, o passo 2 deve ser novamente executado. Como desta vez não haverá modificações, a última tabela será de fato a definitiva.

5.9.2 Conjunto Follow

$\mathbf{FOLLOW(A)}$ é definido para todo $A \in N$ como sendo o conjunto de símbolos terminais que podem aparecer imediatamente após A em alguma forma sentencial de G. A seguir, é definido um algoritmo que constrói o \mathbf{FOLLOW} de todos os não-terminais simultaneamente.

Algoritmo: Cálculo do Conjunto Follow de um Não-Terminal

Entrada: Uma gramática $G=(N,T,P,S)$.

Saída: Uma tabela **TABFOLLOW** com o conjunto **FOLLOW** de cada não-terminal de G .

Para todo $X \in N$, exceto S :

TABFOLLOW(X) := \emptyset ;

Fim Para

TABFOLLOW(S) := $\{\$$; (marca de final de sentença)

Para toda ocorrência de um não-terminal X em uma produção $Y ::= \alpha X \beta \in P$ com $\beta \neq \epsilon$:

TABFOLLOW(X) := $TABFOLLOW(X) \cup FIRST(\beta) - \{\epsilon\}$

Fim Para;

Repita:

Para cada produção da forma $X ::= Y_1 Y_2 \dots Y_n$ faça:

$i ::= n+1$;

Repita:

$i ::= i-1$;

Se $Y_i \in N$ então:

$TABFOLLOW(Y_i) := TABFOLLOW(X) \cup TABFOLLOW(Y_i)$

Fim Se

Até $Y_i \notin N$ ou $\epsilon \notin TABFIRST(Y_i)$

Fim Para

Até que não haja mais modificações na tabela **TABFOLLOW**

Seguem duas observações importantes:

- a) A função **FIRST** é definida para cadeias de símbolos, **FOLLOW** só é definida para não-terminais.
- b) Os elementos de um conjunto **FIRST** pertencem ao conjunto $T \cup \{\epsilon\}$; os elementos de um conjunto **FOLLOW** pertencem ao conjunto $T \cup \{\$\}$.

Como exemplo, será calculado o conjunto **FOLLOW** da gramática utilizada como exemplo no cálculo do **FIRST**, que é:

$S ::= ABS | aA$

$A ::= \epsilon | a$

$B ::= Bb | cd$

cuja tabela **FIRST** é:

N	FIRST
S	{a, c}
A	{ ϵ , a}
B	{c}

O tabela FOLLOW dos não-terminais da gramática é inicializada como:

N	FOLLOW
S	{\\$}
A	\emptyset
B	\emptyset

Para o primeiro passo, temos as ocorrências de $Y ::= \square X \square$ em $S ::= ABS$ e $B ::= Bb$, assim:

$$\begin{aligned} \mathbf{FOLLOW(A)} &= \mathbf{FOLLOW(A)} \cup \mathbf{FIRST(BS)} = \emptyset \cup \{c\} = \{c\} \\ \mathbf{FOLLOW(B)} &= \mathbf{FOLLOW(B)} \cup \mathbf{FIRST(S)} = \emptyset \cup \{a, c\} = \{a, c\} \\ \mathbf{FOLLOW(B)} &= \mathbf{FOLLOW(B)} \cup \mathbf{FIRST(b)} = \{a, c\} \cup \{b\} = \{a, b, c\} \end{aligned}$$

Temos então a tabela FOLLOW intermediária:

N	FOLLOW
S	{\\$}
A	{c}
B	{a, b, c}

No segundo passo, as produções são analisadas da esquerda para a direita. A produção $S ::= ABS$ faz com que:

$$\mathbf{FOLLOW(S)} = \mathbf{FOLLOW(S)} \cup \mathbf{FOLLOW(S)} = \{\$\}$$

Como $\epsilon \notin \mathbf{FIRST(S)}$, a segunda regra termina para esta produção. Para a produção $S ::= aA$, temos:

$$\mathbf{FOLLOW(A)} ::= \mathbf{FOLLOW(A)} \cup \mathbf{FOLLOW(S)} = \{c, \$\}$$

Como $\epsilon \in \mathbf{FIRST(A)}$, o símbolo a esquerda de A é analisado. Como é um terminal (a), o algoritmo pára. Já que não há mais produções que tenham não-terminais a direita, o cálculo da tabela FOLLOW termina. A tabela final fica:

N	FOLLOW
S	{\\$}
A	{c, \\$}
B	{a, b, c}

Lista de Exercícios de Simplificação de GLC's

1 – Eliminação de símbolos inúteis ou improdutivos :

a)

$S ::= ASB \mid BSA \mid SS \mid aS \mid \epsilon$

$A ::= ABS \mid B$

$B ::= BSSA \mid A$

b)

$A ::= aBa \mid cB \mid cC \mid aD$

$B ::= aCb \mid aD \mid ab \mid \epsilon$

$C ::= ab \mid cBd \mid aA$

$D ::= aD \mid dE \mid aD$

$E ::= Ead \mid aD \mid ED$

c)

$S ::= aA$

$A ::= a \mid bB$

$B ::= b \mid dD$

$C ::= cC \mid c$

$D ::= dD$

d)

$S ::= aCD \mid ab \mid bB \mid aaS$

$B ::= bbB \mid Daa \mid a$

$C ::= aCa \mid BCb \mid Ecab$

$E ::= ab \mid Ea \mid Ba$

$D ::= abB \mid ab \mid DD$

e)

$S ::= Abc \mid aBc$

$A ::= aAb \mid AB \mid Abc \mid CD$

$B ::= bBc \mid bC \mid Bc$

$C ::= cCc \mid cC \mid CD$

$D ::= bbD \mid Dbc \mid DD$

$E ::= bEc \mid EC \mid cc$

f)

$S ::= 0A1 \mid 1B0 \mid C$

$A ::= 1A0 \mid AC$

$B ::= 0D1 \mid 01$

$C ::= 1A \mid 0C$

$D ::= 1B0 \mid 10$

g)

$S ::= E * E \mid E + E \mid (E)$

$A ::= id \mid id * E \mid id + E \mid (id)$

$E ::= BS \mid A + E \mid A * E \mid A$

$B ::= id + B \mid id * E$

h)

$S ::= aAc \mid aBc \mid ac$

$A ::= aEd \mid aAb \mid ab$

$B ::= BaD \mid aBb \mid a$

$C ::= aCd \mid af$

$D ::= aDd \mid aD$

$E ::= aEa \mid af$

i)

$S ::= aAb \mid aCd \mid ab$

$A ::= aAb \mid aA$

$B ::= ad \mid aBC$

$C ::= aSa \mid aa$

j)
S ::= ABB | CAC
A ::= a
B ::= Bc | ABB
C ::= bB | a

2 - Eliminação de Símbolos Inalcançáveis : Verifique no exercício 1, a partir da letra 'e', quais são os símbolos inalcançáveis, eliminando-os.

3 – Elimine as ϵ -produções das seguintes gramáticas :

a)
X ::= 0X1 | 1X1 | AB | B1C0
A ::= 1A1 | 00A | ABC | ϵ
B ::= BA | 1B0 | BAC | ϵ
C ::= BCB | 0011 | 1A1 | ϵ

b)
S ::= ABC | aBC | bC
A ::= aAa | ϵ
B ::= BC | bB | ϵ
C ::= CC | cC | ϵ

c)
S ::= 0A1 | 00B | A0B | AC
A ::= 1A0 | 1A | AB | ϵ
B ::= BA | 00B | 11B | ϵ
C ::= 0C1 | 01

d)
S ::= QL | MN | kQk
Q ::= kQk | ϵ
L ::= lllL | ϵ
M ::= mM | ϵ
N ::= nNnn | nnn

e)
S ::= 1AB | 0ABC
A ::= 1A0C | AC | 1 | ϵ
B ::= ACA | 1B | 0
C ::= 1C | C1C | ϵ

f)
S ::= 1B | BCD
B ::= BCB | 01 | ϵ
C ::= C1 | 1 | ϵ
D ::= 1D0 | 10

g)
S ::= aSASb | Saa | AA
A ::= caA | Ac | bca | ϵ

h)
S ::= AB | aS
A ::= bA | BCD
B ::= dB | C | ϵ
C ::= cCc | BD
D ::= CD | d | ϵ

i)
P ::= KL | bKLe
K ::= cK | TV
T ::= tT | ε
V ::= vV | ε
L ::= LC | C
C ::= P | com | ε

4 – Produções Unitárias :

a)
S ::= bS | A
A ::= aA | a

b)
S ::= aSb | A
A ::= aA | B
B ::= bBc | bc

c)
S ::= 1S0 | 0A00 | A | B
A ::= 0 | BC | C | S
B ::= A | 01 | 0S0 | 1
C ::= 1 | 01 | 0S
D ::= D | 01 | ab

d)
S ::= 1A0 | 0B1 | B
A ::= 1B0 | C | 01
B ::= 10B | 01C | D
C ::= 10 | 01 | 0C
D ::= 1D0 | 10 | C

e)
S ::= aA | aB | A
A ::= bC | Bd | B
B ::= aCd | aC | A
C ::= aC | S

f)
S ::= aSa | FbD
A ::= aA | CA | ε
B ::= bB | FE
C ::= cCb | AcA
D ::= Dd | fF | c
E ::= BC | eE | EB
F ::= fF | Dd

5 – Fatoração :

a)
c ::= V = exp | id (E)
V ::= id [E] | id
E ::= exp + E | exp

b)
S ::= bcD | Bcd
B ::= bB | b
D ::= dD | d

c)

S ::= aA | aB

A ::= aA | a

B ::= b

d)

S ::= Ab | ab | baA

A ::= aab | b

e)

S ::= Abc | bBC | bCD

A ::= aBC | aDC

B ::= dCc | dc

C ::= Acd | cd

D ::= aBC | abc | aC

f)

S ::= 1A0 | 1B1

A ::= 1A0 | 100

B ::= C10 | C01

C ::= 11C | 11D

D ::= 10 | 11

g)

S ::= 10D | 11C | 0B

B ::= 1CD | 101B | 01

C ::= 101C | 1B0 | 00

D ::= 00B | 011D | 110

h)

S ::= aBd | acD | bC

B ::= bDc | bCd | ad

D ::= cdD | caB

C ::= cbB | adD

i)

S ::= bcD | Bcd

B ::= bB | b

D ::= dD | d

j)

C ::= V=exp | id(E)

V ::= id[E] | id

E ::= exp,E | exp

6 – Eliminação de recursão à esquerda :

a)

A ::= Abc | Acd | cdB | c

B ::= Bc | dd

b)

S ::= Aab | Bc | ScAb

A ::= Sac | BaA | ab

B ::= Ac | aBb | ab

c)

A ::= Ac | bd | cd

B ::= ac | Bd | Ac

d)

S ::= Aa | Bb | Sac

A ::= Sab | ab

B ::= a | Ba

e)

S ::= A0 | 0B1 | 11C

A ::= 01 | 0BC | S0
B ::= B01 | 01A
C ::= 01 | 10

f)

S ::= 01S | 00A | S10 | A01
A ::= 00D | 01S | A00 | S11 | 0B
B ::= B00 | B11 | C01 | 001
C ::= B10 | C00 | 01C | 01B | 0
D ::= D01 | D00 | 01 | 00

g)

S ::= SaB | Sbc | Acd
A ::= ABc | Acd | Bcd | Cd
B ::= Acd | Bcc | Cdc
C ::= Ac | Cd | dc

h)

S ::= Cab | Ab | b
A ::= Bcd | Ac | ab
B ::= Aca | Bc | Cba | a
C ::= DaC | Cc | c
D ::= ac | CaD

i)

E ::= E+T | E-T | T
T ::= T*F | T/F | F
F ::= F**P | P
P ::= (E) | id | cte

j)

S ::= BaS | ϵ
B ::= SAa | Bb
A ::= Sa | ϵ

BIBLIOGRAFIA

- AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores: princípios, técnicas e ferramentas**. Rio de Janeiro/RJ: LTC, 1995. 344 p.
- FURTADO, O.J.V. **Linguagens formais e compiladores**: notas de aula. Florianópolis/SC: INE-UFSC, 1992. 77 p.
- HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to automata theory, languages and computation**. Reading: Addison-Wesley, 1979. 418 p.
- JOSÉ NETO, J. **Introdução à compilação**. Rio de Janeiro/RJ: LTC, 1987. 222 p.
- MARTINS, Joyce. **Linguagens Formais e Compiladores**. Universidade do Vale do Itajaí – Notas de Aula. São José/SC.
- MENEZES, P. F. B. **Linguagens Formais e Autômatos**. Porto Alegre/RS: II-UFRGS, Sagra Luzzatto, 1997. 168 p.
- PRICE, A.M.A.; EDELWEISS, N. **Introdução às linguagens formais**. Porto Alegre/RS: II-UFRGS, 1989. 60 p.
- WAZLAWICK, R.S. **Linguagens formais e compiladores**: notas de aula. Florianópolis/SC: INE-UFSC, 1992. 83 p.
- ZILLER, R.M. **Aplicação de autômatos finitos**. In: SEMANA TECNOLÓGICA, 1., Florianópolis, 1997. Anais. Biguaçu, UNIVALI, 1997. p.51-71.