



UNIVERSIDADE DO VALE DO ITAJAÍ

Centro de Educação São José
Cursos de Ciência da Computação e
Engenharia de Computação

LINGUAGENS FORMAIS E COMPILADORES

Profa. Joyce Martins

São José / SC, 2002

SUMÁRIO

| | | |
|------------|---|-----------|
| 1 | INTRODUÇÃO À COMPILAÇÃO..... | 1 |
| 1.1 | PROCESSADORES DE LINGUAGENS..... | 1 |
| 1.2 | ESTRUTURA GERAL DE UM COMPILADOR..... | 2 |
| 1.2.1 | ANALISADOR LÉXICO (OU <i>SCANNER</i>) | 3 |
| 1.2.2 | ANALISADOR SINTÁTICO (OU <i>PARSER</i>) | 3 |
| 1.2.3 | ANALISADOR SEMÂNTICO | 4 |
| 1.2.4 | GERAÇÃO DE CÓDIGO INTERMEDIÁRIO | 4 |
| 1.2.5 | OTIMIZAÇÃO DE CÓDIGO | 5 |
| 1.2.6 | GERAÇÃO DE CÓDIGO..... | 5 |
| 1.3 | FORMAS DE CONSTRUÇÃO DE UM COMPILADOR..... | 5 |
| 1.4 | FERRAMENTAS PARA A CONSTRUÇÃO DE COMPILADORES | 6 |
| 2 | CONCEITOS BÁSICOS..... | 7 |
| 3 | LINGUAGENS E SUAS REPRESENTAÇÕES | 10 |
| 3.1 | CONCEITO..... | 10 |
| 3.2 | TIPOS..... | 11 |
| 3.3 | ESPECIFICAÇÃO DE UMA LINGUAGEM..... | 12 |
| 4 | ANALISADOR LÉXICO..... | 16 |
| 4.1 | FUNÇÃO..... | 16 |
| 4.2 | ESPECIFICAÇÃO DE <i>TOKENS</i>: EXPRESSÃO REGULAR..... | 16 |
| 4.3 | RECONHECIMENTO DE <i>TOKENS</i>: AUTÔMATO FINITO..... | 18 |
| 4.4 | IMPLEMENTAÇÃO..... | 29 |
| 5 | ANALISADOR SINTÁTICO | 30 |
| 5.1 | FUNÇÃO..... | 30 |
| 5.2 | ESPECIFICAÇÃO DAS REGRAS SINTÁTICAS: GRAMÁTICA LIVRE DE CONTEXTO | 30 |
| 5.2.1 | NOTAÇÕES..... | 30 |
| 5.2.2 | ÁRVORE DE DERIVAÇÃO OU ÁRVORE SINTÁTICA | 31 |
| 5.2.3 | DERIVAÇÃO MAIS À ESQUERDA E DERIVAÇÃO MAIS À DIREITA | 32 |
| 5.2.4 | SIMPLIFICAÇÕES DE GRAMÁTICAS LIVRES DE CONTEXTO..... | 33 |
| 5.2.5 | CONJUNTOS: <i>FIRST</i> E <i>FOLLOW</i> | 37 |
| 5.3 | AUTÔMATO COM PILHA | 38 |
| 5.4 | TIPOS DE ANALISADORES SINTÁTICOS..... | 39 |
| 5.4.1 | ANALISADOR SINTÁTICO ASCENDENTE (<i>BOTTOM-UP</i>)..... | 39 |
| 5.4.2 | ANALISADOR SINTÁTICO DESCENDENTE (<i>TOP-DOWN</i>) | 40 |
| 5.5 | IMPLEMENTAÇÃO..... | 42 |
| 6 | ANALISADOR SEMÂNTICO E GERADOR DE CÓDIGO INTERMEDIÁRIO | 44 |
| 6.1 | FUNÇÃO..... | 44 |
| 6.2 | ESPECIFICAÇÃO DA SEMÂNTICA DE UMA LINGUAGEM DE PROGRAMAÇÃO..... | 44 |
| 6.3 | DEFINIÇÃO DA MÁQUINA VIRTUAL | 45 |
| 6.4 | DEFINIÇÃO DA TABELA DE SÍMBOLOS | 48 |
| 6.5 | AÇÕES SEMÂNTICAS E GERAÇÃO DE CÓDIGO: DESCRIÇÃO GERAL | 49 |
| | BIBLIOGRAFIA..... | 50 |

1 INTRODUÇÃO À COMPILAÇÃO

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:

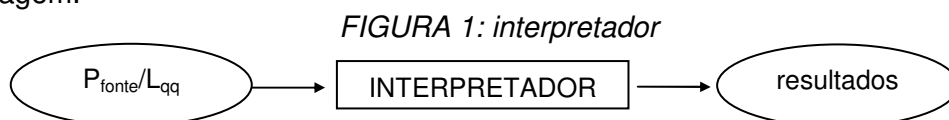
- compiladores para linguagens de programação: um compilador traduz um programa escrito numa *linguagem fonte* em um programa escrito em uma *linguagem objeto*;
- formatadores de texto: um formatador de texto manipula um conjunto de caracteres composto pelo documento a ser formatado e por comandos de 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.

1.1 PROCESSADORES DE LINGUAGENS

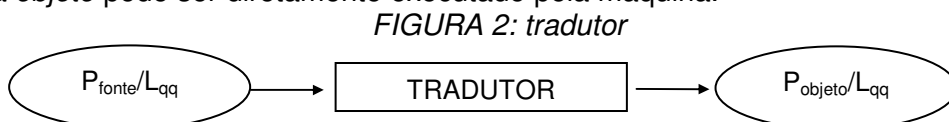
Um **processador** é um programa que permite ao computador “entender” os comandos de alto nível escritos pelos usuários.

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

Um **interpretador** (FIGURA 1) é um programa que aceita como entrada um programa escrito em uma linguagem chamada *linguagem fonte* e executa diretamente as instruções dadas nesta linguagem.



Um **tradutor** (FIGURA 2) 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.



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) esquematiza os três tipos de tradutores.

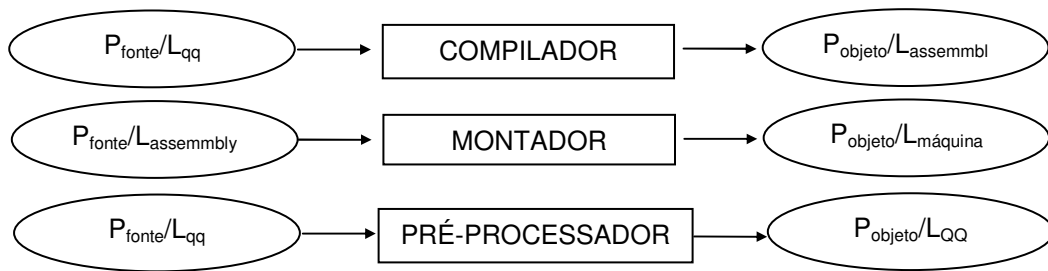


FIGURA 3: tipos de tradutores

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 4) apresenta o exemplo de um sistema de processamento de linguagem:

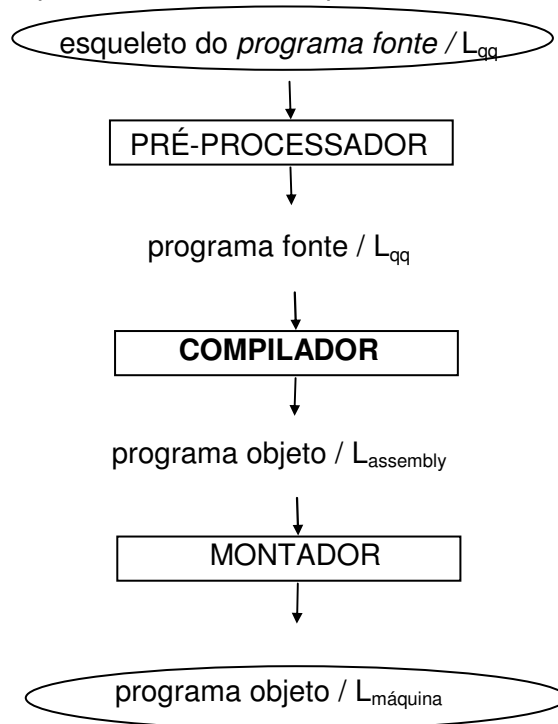


FIGURA 4: um sistema de processamento de linguagem

1.2 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 5 mostra os processos constituintes de um compilador conceitual. Na implementação de um compilador, alguns desses processos podem ser combinados ou um processo pode ser dividido em outros.

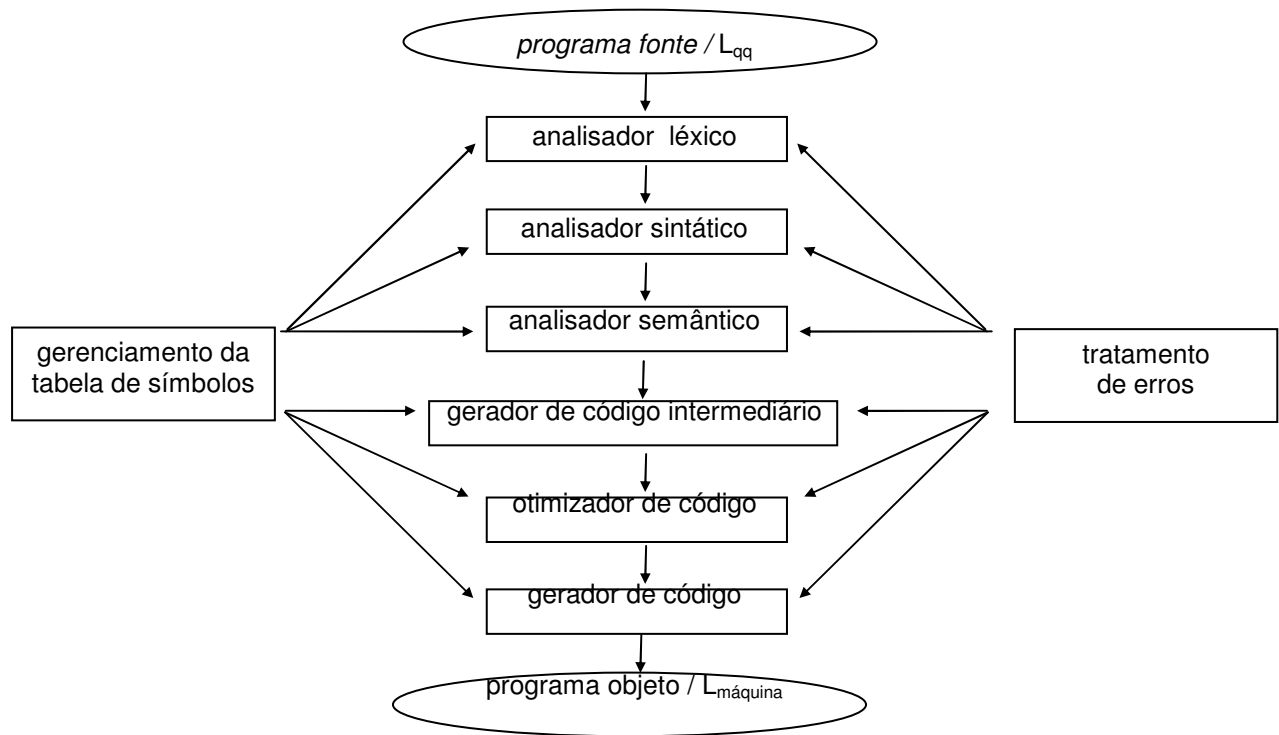


FIGURA 5: estrutura geral de um compilador

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.

1.2.1 Analisador léxico (ou *scanner*)

O **analisador léxico** separa a sequência 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 sequência de caracteres:

SOMA := SOMA + 35

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

| | |
|------|--|
| SOMA | identificador |
| := | símbolo especial (comando de atribuição) |
| SOMA | identificador |
| + | símbolo especial (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, tais como números inteiros com grandeza maior do que pode ser representada, caracteres e símbolos inválidos, etc.

1.2.2 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:

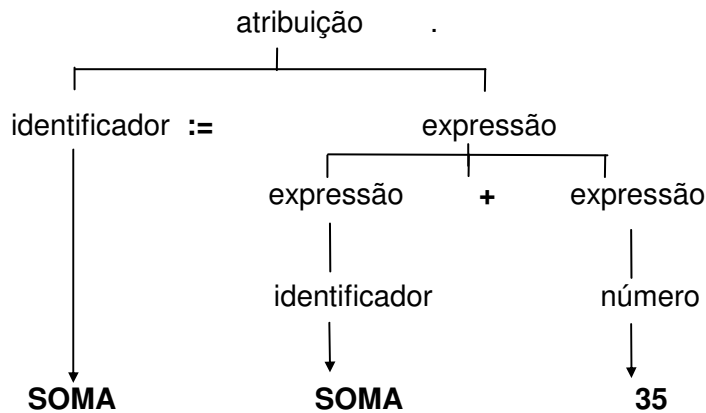


FIGURA 6: árvore de derivação

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.

1.2.3 Analisador semântico

O compilador executa ainda a análise semântica¹. O **analisador semântico** utiliza a árvore sintática determinada pelo analisador sintático para determinar o significado do programa-fonte. Segundo JOSÉ NETO (1987), "ao contrário da sintaxe, que é facilmente formalizável, (...) a semântica, apesar de também poder ser expressa formalmente, exige para isto notações substancialmente mais complexas, de aprendizagem mais difícil e em geral apresentado simbologias bastante carregadas e pouco legíveis. Assim sendo, na grande maioria das linguagens de programação, a semântica tem sido especificada informalmente, via de regra através de textos em linguagem natural". São funções do analisador semântico: criar e manter a tabela de símbolos, identificadores encontrados no programa-fonte; identificar operadores e operandos das expressões; reconhecer e tratar erros semânticos; fazer verificações de compatibilidade de tipo; analisar o escopo e o uso dos identificadores; e 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:

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

1.2.4 Geração de código intermediário

Alguns compiladores incluem a **geração de uma representação intermediária** para o programa fonte. Uma representação intermediária é um código para uma máquina abstrata e

¹ A semântica nesse caso engloba apenas uma pequena parte do que vem a ser realmente a semântica de um programa.

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 sequência de instruções, cada uma possuindo no máximo três operandos. Para o comando de atribuição $SOMA := SOMA + 35$ tem-se:

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

"Um dos grandes objetivos do uso de linguagens intermediárias é o de permitir uma facilidade maior de manipulação com finalidades ligadas à otimização do código gerado" (...) além de "aumentar a portabilidade do programa-objeto, tornando o compilador menos dependente de máquina" (JOSÉ NETO, 1987). A diferença entre o código intermediário e o código objeto é que o primeiro não especifica detalhes dependentes da máquina, tais como registradores, endereçamento de memória, etc.

1.2.5 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.

1.2.6 Geração de código

A fase final do compilador é a **geração do código** para o programa objeto, consistindo normalmente de código em linguagem de *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"
```

1.3 FORMAS DE CONSTRUÇÃO DE UM COMPILADOR

Dependendo da implementação, os processos apresentados na seção anterior podem ser executados sequencialmente ou ter execução entrelaçada, enquanto alguns podem ser omitidos. Uma compilação pode ser em um ou em vários passos. Entende-se por passo a passagem completa do compilador sobre o programa fonte que está sendo compilado. Em uma 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, equipe de desenvolvimento, disponibilidade de ferramentas de apoio, e prazo 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 (arquivos) intermediárias.

A figura abaixo (FIGURA 7) apresenta a estrutura geral de um compilador:

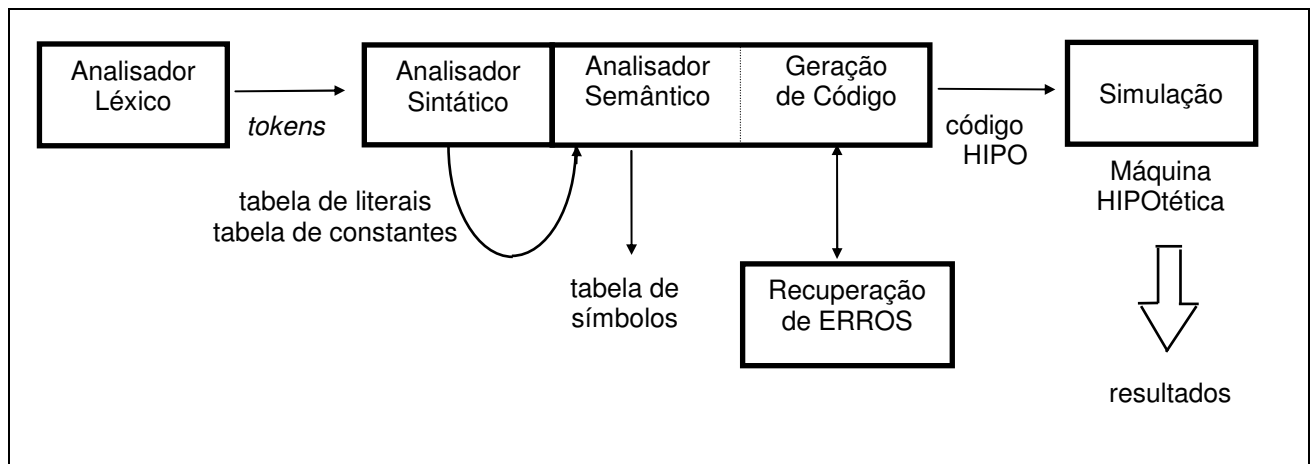


FIGURA 7: estrutura geral de um compilador

1.4 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).

LEITURA COMPLEMENTAR: AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores: princípios, técnicas e ferramentas**. Rio de Janeiro/RJ: LTC, 1995. (capítulo 1: p.1-11); JOSÉ NETO, J. **Introdução à compilação**. Rio de Janeiro/RJ: LTC, 1987 (capítulo 4:116-146). WILHELM, R.; MAURER, D. **Compiler design**. Harlow: Addison-Wesley, 1995 (capítulo 6: 221-233).

2 CONCEITOS BÁSICOS

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 apoia 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.

DEFINIÇÃO nº 1: conjunto

Um conjunto é uma coleção de elementos não repetidos. Segundo MENEZES (1997), a especificação de um conjunto pode ser feita de por extensão, listando seus elementos, ou pode ser feita por compreensão na forma $\{x \mid P(x)\}$, onde P é uma propriedade que deve ser satisfeita pelos elementos que formam o conjunto.

Um conjunto pode ser vazio, finito ou infinito dependendo do número de elementos contidos. Um conjunto que não contém nenhum elemento é chamado de conjunto vazio e é denotado pelo símbolo \emptyset ou $\{\}$.

Um conjunto é finito quando contém um número finito de elementos, podendo ser especificado pela listagem de todos os seus elementos. Assim, tem-se o conjunto finito $C_1 = \{x \mid x \text{ é primo} \wedge x < 11\}$, ou seja, $C_1 = \{2, 3, 5, 7\}$,

Um conjunto é infinito quando contém um número infinito de elementos, podendo ser especificado apenas de forma indireta. São exemplos de conjuntos infinitos: o conjunto dos números naturais (**N**), o conjunto dos números inteiros (**Z**), o conjunto dos números reais (**R**), $C_1 = \{x \mid x \in \mathbf{N} \wedge x \text{ é divisível por 3 e por 7}\}$, ou seja, $C_1 = \{21, 42, 63, 84, \dots\}$; $C_2 = \{x \mid x \in \mathbf{N} \wedge \exists y \in \mathbf{N} \text{ tal que } x = 2y\}$, ou seja, $C_2 = \{0, 2, 4, 6, 8, \dots\}$.

² Os operadores lógicos serão representados da seguinte forma: operador **NÃO** denotado por \neg ; operador **E** denotado por \wedge ; operador **OU** denotado por \vee ; **SE-ENTÃO** denotado por \rightarrow ; **SE-SOMENTE-SE** denotado por \leftrightarrow ; **EXISTE** denotado por \exists .

DEFINIÇÃO nº 2: relações entre elementos e conjuntos

- a) Sejam um conjunto \underline{C} e um elemento \underline{e} , diz-se que \underline{e} *pertence a* \underline{C} ($e \in C$) se \underline{e} é um elemento de \underline{C} , e diz-se que \underline{e} *não pertence a* \underline{C} ($e \notin C$) se \underline{e} não é um elemento de \underline{C} .
- b) Sejam dois conjuntos \underline{C}_1 e \underline{C}_2 , diz-se que \underline{C}_1 *está contido em* \underline{C}_2 ($C_1 \subseteq C_2$) se todo elemento de \underline{C}_1 também é elemento de \underline{C}_2 (\underline{C}_1 é um subconjunto de \underline{C}_2). Pode-se dizer também que \underline{C}_2 *contém* \underline{C}_1 ($C_2 \supseteq C_1$). No entanto, diz-se que \underline{C}_1 *está contido propriamente em* \underline{C}_2 ($C_1 \subset C_2$) se no mínimo um elemento de \underline{C}_2 não for elemento de \underline{C}_1 (\underline{C}_1 é um subconjunto próprio de \underline{C}_2). Pode-se dizer também que \underline{C}_2 *contém propriamente* \underline{C}_1 ($C_2 \supset C_1$).
- c) Sejam dois conjuntos \underline{C}_1 e \underline{C}_2 , diz-se que \underline{C}_1 *é igual a* \underline{C}_2 ($C_1 = C_2$) se \underline{C}_1 possuem os mesmos elementos de \underline{C}_2 , ou seja, se $C_1 \subseteq C_2$ e $C_2 \subseteq C_1$.

DEFINIÇÃO nº 3: operações sobre conjuntos

Sejam dois conjuntos \underline{C}_1 e \underline{C}_2 , as operações definidas sobre conjuntos são:

- a) união ($C_1 \cup C_2$): a união de \underline{C}_1 e \underline{C}_2 é o conjunto $C_3 = \{x \mid x \in C_1 \vee x \in C_2\}$
- b) interseção ($C_1 \cap C_2$): a interseção de \underline{C}_1 e \underline{C}_2 é o conjunto $C_3 = \{x \mid x \in C_1 \wedge x \in C_2\}$
- c) diferença ($C_1 - C_2$): a diferença de \underline{C}_1 e \underline{C}_2 é o conjunto $C_3 = \{x \mid x \in C_1 \wedge x \notin C_2\}$
- d) complemento (C_1'): o complemento de \underline{C}_1 é definido em relação a um conjunto universo (\underline{U}) como o conjunto $C_2 = \{x \mid x \in \underline{U} \wedge x \notin C_1\}$.

DEFINIÇÃO nº 4: algoritmo e procedimento

Um algoritmo é uma descrição finita de um processo de computação, escrito em uma linguagem algorítmica na qual cada sentença tem um significado não ambíguo. Considere a seguinte definição: um número primo é qualquer inteiro positivo divisível apenas pelo um e pelo próprio número. É possível escrever um ALGORITMO para determinar se um número inteiro positivo é ou não um número primo.

Um algoritmo apresenta as seguintes características: possui um conjunto de instruções que podem ser executadas por uma máquina; possui um número finito de passos; tem parada garantida.

Um procedimento é uma seqüência finita de passos que pode ser executada sistematicamente. Um procedimento não termina necessariamente. Assim, existem problemas que possuem procedimentos para serem resolvidos, mas não possuem algoritmos. Considere a seguinte definição: um número perfeito é aquele cuja soma de seus divisores exceto ele é igual ao próprio número. É possível escrever um PROCEDIMENTO para determinar se existe um número perfeito maior que outro número inteiro dado.

DEFINIÇÃO nº 5: 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 do alfabeto, 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.

DEFINIÇÃO nº 6: 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. A sentença vazia, representada por ϵ , é uma sentença constituída por nenhum símbolo.

DEFINIÇÃO nº 7: 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.

DEFINIÇÃO nº 8: prefixo e sufixo de uma sentença

O prefixo de uma sentença é um número qualquer de símbolos tomados do seu início, já um sufixo é um número qualquer de símbolos tomados do seu fim. Assim, *PRIMA* tem como prefixos ϵ , *P*, *PR*, *PRI*, *PRIM* e *PRIMA*, e como sufixos ϵ , *A*, *MA*, *IMA*, *RIMA* e *PRIMA*. Um prefixo ou um sufixo que não seja a própria sentença é chamado de prefixo próprio ou sufixo próprio.

DEFINIÇÃO nº 9: concatenação de sentenças

A concatenação de duas sentenças é a sentença formada pela escrita da primeira seguida da segunda, sendo que o operador de concatenação é a justaposição. Ou seja, se w e x são sentenças, então wx é a sentença resultante da concatenação. Sejam $w = \textit{PRIMA}$ e $x = \textit{VERA}$, a concatenação da sentença w com a sentença x é igual à *PRIMAVERA*. A operação de concatenação tem as seguintes propriedades, supondo w, x, y sentenças:

- a) associativa: $w(xt) = (wx)t$;
- b) elemento neutro: $\epsilon w = w\epsilon = w$

A concatenação sucessiva (ou n -ésima potência) de uma sentença w , denotada por w^n , é a concatenação de w com ela mesma n vezes. Seja $w = \textit{TSE}$, $w^0 = \epsilon$; $w^1 = \textit{TSE}$; $w^2 = \textit{TSETSE}$. Seja $w = \epsilon$, w^0 é indefinida; $w^1 = \epsilon$; $w^2 = \epsilon$; $w^n = \epsilon$.

DEFINIÇÃO nº 10: 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 $V^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$.

LEITURA COMPLEMENTAR: MENEZES, P. F. B. Linguagens Formais e Autômatos. Porto Alegre/RS: II-UFRGS, Sagra Luzzatto, 1997. (capítulo 2: p.15-35).

3 LINGUAGENS E SUAS REPRESENTAÇÕES

3.1 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:

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

$L_2 = \{w \mid w \text{ é uma palíndrome}^4\}$. 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. Reconhecedores ou sistemas geradores são dois tipos de representações finitas. Um reconhecedor é um dispositivo formal usado para verificar se uma determinada sentença pertence ou não à linguagem. São exemplos de reconhecedores 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 reconhecedor e todo sistema gerador pode ser representado por um algoritmo.

³ MOD é uma operação que representa o resto da divisão inteira.

⁴ Uma palíndrome é um palavra que tem a mesma leitura da esquerda para a direita e vice-versa.

3.2 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.

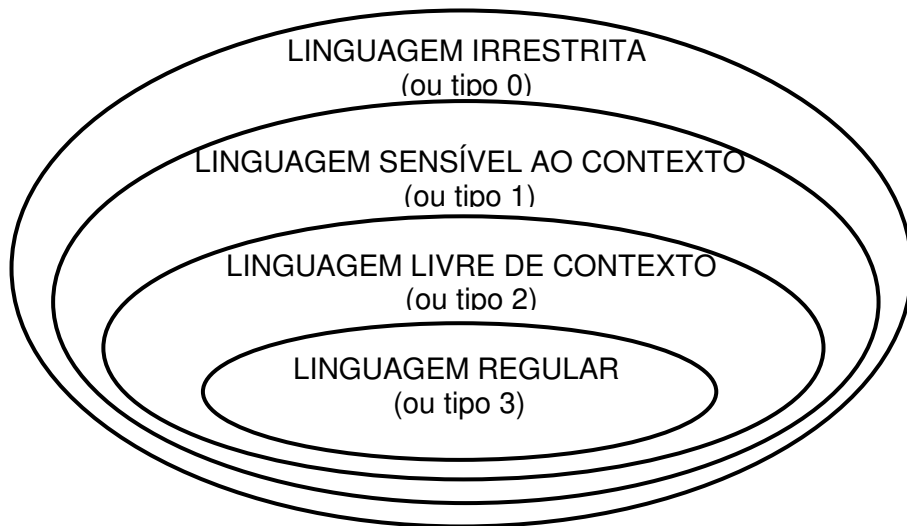


FIGURA 8: hierarquia de Chomsky

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: **analísadores 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*. 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 sensível ao contexto*; 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.

3.3 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.

DEFINIÇÃO nº 1: gramática

Formalmente uma gramática G é definida como sendo uma quádrupla $G = (V_N, V_T, P, S)$, onde: V_N é um conjunto finito de símbolos denominados símbolos não-terminais, usados na descrição da linguagem; V_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 V_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 sequência 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 .

Para facilitar a compreensão, serão adotadas as seguintes convenções notacionais: os símbolos não-terminais serão sempre representados por letras maiúsculas $\{A, B, \dots, T\}$; os símbolos terminais serão sempre representados por letras minúsculas $\{a, b, \dots, t\}$, dígitos, ou caracteres especiais; as sentenças compostas por não-terminais e/ou terminais serão representadas por letras gregas; as sentenças compostas por terminais serão representadas por letras minúsculas $\{u, v, \dots, z\}$.

Abaixo encontra-se um exemplo de uma gramática:

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

$G = (V_N, V_T, P, S)$

onde

$V_N = \{N, D\}$

$V_T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$P = \{ N \rightarrow D N \mid D$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$\begin{array}{l} \} \\ S = N \end{array}$$

DEFINIÇÃO nº 2: derivação e redução

A utilização de gramáticas pode ser formalizada por duas operações de substituição:

- derivação: é a operação que consiste na substituição de uma sentença ou parte dela por outra de acordo com as regras de produção da gramática, no sentido SÍMBOLO INICIAL \Rightarrow SENTENÇA.
- redução: é a operação que consiste na substituição de uma sentença ou parte dela por outro de acordo com as regras de produção da gramática, no sentido SENTENÇA \Leftarrow SÍMBOLO INICIAL.

Sucessivos passos de derivação são definidos da seguinte forma:

- $\alpha \Rightarrow_G \beta$ derivação em um passo ou direta: α deriva diretamente β , se e somente se $\alpha \rightarrow \beta \in P$;
- $\alpha \Rightarrow_G^* \beta$ derivação em zero ou mais passos: α deriva em zero ou mais passos β , se e somente se existirem seqüências $\alpha_1, \alpha_2, \alpha_3 \dots \alpha_n$ tais que $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow \beta$;
- $\alpha \Rightarrow_G^+ \beta$ derivação em um ou mais passos: α deriva em um ou mais passos β , se e somente se for necessário pelo menos um passo na derivação.

Esses mesmos conceitos se aplicam de forma análoga à redução, denotada por \Leftarrow . Para a gramática do EXEMPLO 1, tem-se:

derivação em um passo: $N \Rightarrow_G D \quad N \Rightarrow_G D D \Rightarrow_G 1 D \Rightarrow_G 1 2$

derivação em zero ou mais passos: $N \Rightarrow_G^* N, N \Rightarrow_G^* 1 2$

derivação em um ou mais passos: $N \Rightarrow_G^+ 1 2$

redução em um passo: $1 2 \Leftarrow_G 1 D \Leftarrow_G D D \Leftarrow_G D N \Leftarrow_G N$

redução em zero ou mais passos: $1 2 \Leftarrow_G^* N$

redução em um ou mais passos: $1 2 \Leftarrow_G^+ N$

A derivação é a operação adequada à geração de sentenças, enquanto redução é a operação adequada ao reconhecimento de sentenças.

Do exposto anteriormente, pode-se definir formalmente uma linguagem gerada por uma gramática $G = (V_N, V_T, P, S)$, denotada por $L(G)$ como sendo: $L(G) = \{x \mid x \in V_T^* \text{ e } S \Rightarrow_G^+ x\}$. Em outras palavras, uma linguagem L é definida pelo conjunto de sentenças, compostas apenas por símbolos terminais, que podem ser derivadas a partir do símbolo inicial da gramática que a representa.

DEFINIÇÃO nº 3: equivalência entre gramáticas

Diz-se que duas gramáticas G_1 e G_2 são equivalentes, denotada por $G_1 \equiv G_2$, se e somente se $L(G_1) = L(G_2)$. As gramáticas G_1 e G_2 apresentadas no exemplo abaixo são equivalentes visto que a linguagem gerada por G_1 é igual a linguagem gerada por G_2 .

EXEMPLO 2: sejam G_1 e G_2 gramáticas definidas por:

| G_1 | G_2 |
|---|---|
| $V_N = \{S, A\}$ | $V_N = \{S\}$ |
| $V_T = \{a, b\}$ | $V_T = \{a, b\}$ |
| $P = \{ \quad S \rightarrow a A \mid \varepsilon$ | $P = \{ \quad S \rightarrow a b S \mid \varepsilon$ |
| $\quad A \rightarrow b S$ | $\quad \}$ |
| $\quad \}$ | $S = S$ |
| $S = S$ | |

Visto que a linguagem gerada por G_1 é a mesma gerada por G_2 ($L = \{x \mid x \in (\{a, b\}^* \wedge x \text{ segue o padrão de formação } (a b)^n, \text{ onde } n \geq 0)\}$), pode-se afirmar que $G_1 \equiv G_2$.

DEFINIÇÃO n° 4: tipos de gramáticas

Segundo a hierarquia de Chomsky, impondo restrições na forma das regras de produção, pode-se identificar quatro tipos diferentes de gramáticas. O tipo mais simples é a **gramática regular** (GR) que pode ser utilizada para especificar a parte léxica de uma linguagem de programação. Uma gramática regular à direita (ou à esquerda) é uma quádrupla $G = (V_N, V_T, P, S)$, onde: $P = \{A \rightarrow w B \mid A \in V_N, w \in V_T^*, B \in (V_N \cup \{\epsilon\})\}$, ou seja, toda regra de produção tem a seguinte forma:

$$A \rightarrow w B \text{ - à direita} \quad (A \rightarrow B w \text{ - à esquerda})$$

ou $A \rightarrow w$

Em outras palavras, uma gramática regular admite apenas regras de produção constituídas por seqüências de terminais seguidas (precedidas) ou não por apenas um não-terminal.

EXEMPLO 3: a linguagem $L = \{x \mid x \in \{a, b\}^+ \wedge x \text{ começa com } a \text{ e possui zero ou mais seqüências de } ba\}$ é gerada pelas seguintes gramáticas regulares:

| | | | |
|-------------------------------------|------------------------------|-----------------------------------|----------------------------|
| G_1 | G_2 | G_3 | G_4 |
| $A \rightarrow a B$ | $A \rightarrow A b a \mid a$ | $A \rightarrow a B$ | $A \rightarrow B a \mid a$ |
| $B \rightarrow b a B \mid \epsilon$ | | $B \rightarrow b C \mid \epsilon$ | $B \rightarrow A b$ |
| | | $C \rightarrow a B$ | |

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

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 o 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 4: 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:

$$S \rightarrow a S b \mid \epsilon$$

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

As linguagens sensíveis ao contexto são definidas por **gramáticas sensíveis ao contexto** (GSC). Uma gramática sensível ao contexto é uma quádrupla $G = (V_N, V_T, P, S)$, onde: $P = \{\alpha \rightarrow \beta \mid \alpha \text{ é uma sentença em } (V_N \cup V_T)^+ \text{ com no mínimo um não-terminal, } \beta \text{ é uma sentença em } (V_N \cup V_T)^*, |\alpha| \leq |\beta| \text{ exceto para } S \rightarrow \epsilon, \text{ sendo que } S \text{ não pode estar presente no lado direito de nenhuma produção}\}$. Em outras palavras, uma gramática sensível ao contexto admite regras de produção contendo não-terminais e terminais tanto do lado esquerdo como do lado de direito desde que a cada derivação o tamanho da sentença derivada não diminua, exceto quando a sentença vazia é gerada. Segundo MENEZES (1997), "o termo 'sensível ao contexto' deriva do fato de que o lado esquerdo das produções da gramática pode ser uma sentença de terminais e não-terminais, definindo um 'contexto' de derivação".

EXEMPLO 5: 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:

$$S \rightarrow a S B C \mid a B C$$

$$\begin{aligned}BC &\rightarrow CB \\CB &\rightarrow BC \\B &\rightarrow b \\C &\rightarrow c\end{aligned}$$

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 $G = (V_N, V_T, P, S)$ 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.

LEITURA COMPLEMENTAR: HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to automata theory, languages and computation**. Reading: Addison-Wesley, 1979. (capítulo 9: p.217-228).

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 ESPECIFICAÇÃO DE *TOKENS*: EXPRESSÃO REGULAR

Uma expressão regular (ER) é constituída de expressões regulares mais simples usando-se regras de definição para as operações de concatenação e união. A seguir estão as regras que definem expressões regulares sobre um alfabeto V :

1. \emptyset é uma expressão que denota a linguagem vazia.
2. ϵ é uma expressão regular que denota a linguagem contendo a sentença vazia, ou seja, o conjunto $\{\epsilon\}$.
3. para cada símbolo $v \in V$, v é uma expressão regular que denota a linguagem unitária contendo a sentença v , ou seja, o conjunto $\{v\}$.
4. se r e s são expressões regulares que denotam as linguagens R e S , respectivamente, então:
 - a) $r | s$: é uma expressão regular que denota $R \cup S$ (união)
 - b) rs : é uma expressão regular que denota RS (concatenação)
 - c) r^* : é uma expressão regular que denota R^* (concatenação sucessiva)
 - d) r : é uma expressão regular que denota R .

Na escrita de expressões regulares⁵ pode-se omitir muitos parênteses se for assumido que a concatenação sucessiva (*) tem maior precedência que a concatenação e a união (|), e que a concatenação tem maior precedência que a união (|). Assim, por exemplo, a expressão regular $((0(1^*)) | 0)$ pode ser escrita como $01^* | 0$. Além disso, pode-se aplicar simplificações abreviando expressões como:

1. rr^* para r^+ , a qual significa "uma ou mais ocorrências de" r
2. $r^+ | \epsilon$ para r^* , a qual significa "zero ou mais ocorrências de" r
3. $r | \epsilon$ para $r^?$, a qual significa "zero ou uma ocorrência de" r .

Seja $V = \{0, 1\}$. Tem-se as expressões regulares denotando as seguintes linguagens regulares:

| ER | LINGUAGEM DENOTADA |
|-------------|--|
| 0 | a sentença 0 , ou seja, o conjunto $\{0\}$ |
| $0 1$ | as sentenças 0 e 1 , ou seja, o conjunto $\{0, 1\}$ |
| 00 | a sentença 00 , ou seja, o conjunto $\{00\}$ |
| 0^* | todas as sentenças constituídas por zero ou mais 0 s, ou seja, o conjunto $\{\epsilon, 0, 00, 000, 0000, \dots\}$ |
| $(0 1)^*$ | todas as sentenças constituídas por zero ou mais 0 s e/ou 1 s, ou seja, o conjunto $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, 111, 001, \dots\}$. Outra ER para essa linguagem é $(0^* 1^*)^*$ |
| $1 1^* 0$ | a sentença 1 e todas as sentenças que iniciam com zero ou mais 1 s e que terminam com um 0 , ou seja, $\{1, 0, 10, 110, 1110, 11110, \dots\}$ |

Duas expressões regulares podem ser equivalentes, ou seja, denotar a mesma linguagem. AHO et. al. (1995) apresenta leis algébricas para manipular expressões regulares em formas equivalentes (TABELA 1).

| AXIOMA | DESCRIÇÃO |
|--|---|
| $r s = s r$ | a união () é comutativa |
| $r (s t) = (r s) t$ | a união () é associativa |
| $(rs)t = r(st)$ | a concatenação é associativa |
| $r(s t) = rs rt$ \wedge $(s t)r = sr tr$ | a concatenação se distribui sobre a união () |
| $\epsilon r = r$ \wedge $r \epsilon = r$ | ϵ é o elemento neutro da concatenação |
| $r^* = (r \epsilon)^*$ | relação entre ϵ e a concatenação sucessiva |
| $r^{**} = r^*$ | a concatenação sucessiva é idempotente |

TABELA 1: leis algébricas de expressões regulares

Segundo AHO et. al. (1995), por conveniência de notação, pode-se desejar dar nomes a expressões regulares, bem como definir outras expressões usando esses nomes como se fossem símbolos. Assim, se V for um alfabeto de símbolos básicos, então uma definição regular é uma sequência de definições da forma:

$$\begin{aligned} d_1 &\Rightarrow r_1 \\ d_2 &\Rightarrow r_2 \\ &\dots \\ d_n &\Rightarrow r_n \end{aligned}$$

onde cada d_i é um nome distinto e cada r_i uma expressão sobre os símbolos em $V \cup \{d_1, d_2, \dots, d_{n-1}\}$, isto é, os símbolos de V e os nomes das expressões previamente definidos. Assim, considere o seguinte exemplo: seja $V = \{A, B, C, \dots, Z, a, b, c, \dots, z\}$, tem-se a seguinte definição regular:

$$\text{maiúscula} \Rightarrow A | B | C | \dots | Z$$

⁵ Os caracteres (,) e | são usados na especificação das expressões regulares. São portanto chamados de **meta-caracteres** não fazendo parte da linguagem que está sendo definida. Caso os meta-caracteres coincidam com os caracteres pertencentes ao alfabeto, deve-se diferenciá-los de alguma forma.

minúscula $\Rightarrow a | b | c | \dots | z$

nome_próprio \Rightarrow maiúscula minúscula⁺

Nem todas as linguagens podem ser descritas por expressões regulares, como por exemplo as linguagens de programação. Além disso, expressões regulares não podem ser usadas para especificar parênteses balanceados, construções aninhadas ou sentenças repetidas tais como seja $V = \{a, b, c\}$, $L = \{wcw \mid w \text{ é uma sentença de } a \text{ e } b\}$. "As expressões regulares podem ser usadas para denotar somente um número fixo de repetições ou um número não especificado de repetições de uma dada construção. Dois números arbitrários não podem ser comparados a fim de se verificar se são os mesmos" (AHO et. al., 1995). Assim, a linguagem constituída por sentenças na forma $nHa_1a_2\dots a_n$ onde o número de caracteres seguintes ao H precisa ser igual ao número decimal n não pode ser especificada por uma ER.

4.3 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 9):

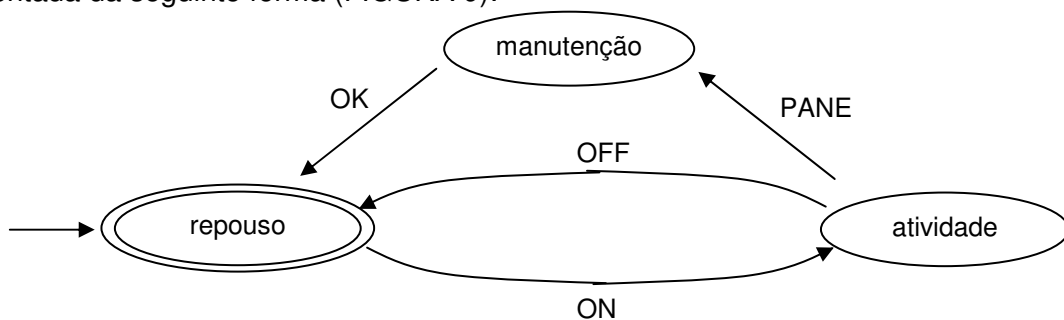


FIGURA 9: exemplo de uma máquina de estados (ZILLER, 1997)

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*).

DEFINIÇÃO nº 1: estado

Um conceito fundamental para autômatos finitos é o conceito de estado. Um estado é uma situação particular no processo de reconhecimento de uma sentença. No exemplo anterior, tem-se os seguintes estados: repouso, atividade e manutenção.

DEFINIÇÃO nº 2: transição de estado

Uma transição de estado é uma mudança de um estado para outro. A máquina da FIGURA 9 só pode estar em um dos três estados em um determinado momento. O ato de ligar ou desligar, por exemplo, faz com que a máquina mude de estado.

DEFINIÇÃO n° 3: autômato finito

Um autômato finito é uma máquina de estados finitos capaz de observar apenas um símbolo de cada vez. Assume-se que a máquina esteja no estado inicial quando começa sua operação. A entrada para um autômato finito é uma seqüência de símbolos de um conjunto finito chamado de alfabeto de entrada, que representa o conjunto de símbolos que o autômato consegue processar. Quando um símbolo de entrada é lido, o autômato muda seu estado em função apenas do estado corrente e do símbolo de entrada, ou seja, ocorre uma transição de estado. Alguns dos estados do autômato são denominados de estados finais.

O autômato finito pode decidir se uma sentença pertence ou não a uma determinada linguagem da seguinte forma:

- se uma seqüência de símbolos faz com que o autômato pare em um estado final após processar o último símbolo, então a seqüência é aceita, isto é, pertence à linguagem em questão;
- se ao final da análise da seqüência de entrada, o autômato pára em um estado que não é final, então a seqüência é rejeitada, isto é, não pertence à linguagem;
- se para um símbolo da seqüência não existe transição, o autômato pára e a seqüência é rejeitada, isto é, não pertence à linguagem.

A partir do que foi colocado acima, segundo MENEZES (1997), pode-se afirmar que um autômato finito sempre pára ao processar qualquer entrada visto que qualquer sentença a ser reconhecida é finita. Ainda segundo MENEZES (1997), um "autômato finito não possui memória de trabalho" de forma que "para armazenar as informações passadas necessárias ao processamento, deve-se usar o conceito de estado".

DEFINIÇÃO n° 4: função de transição (ou função de mapeamento)

A operação de um autômato finito é descrita matematicamente por uma função δ , chamada função de transição (ou função de mapeamento). A função δ determina o estado e_{novo} em função do estado e_{velho} e do símbolo de entrada x . Simbolicamente, $\delta(e_{\text{velho}}, x) = e_{\text{novo}}$.

DEFINIÇÃO n° 5: formas de representação

Um autômato finito pode ser representado formalmente (com prova de teoremas), através de diagramas de transição ou com tabelas de transição.

Formalmente, um AF é um sistema formal $M = (K, \Sigma, \delta, e_0, F)$, onde: K é um conjunto finito não-vazio de estados; Σ é o alfabeto de símbolos de entrada; δ é a função de transição; e_0 é o estado inicial ($e_0 \in K$); F é o conjunto de estados finais ($F \subseteq K$).

Um diagrama de transição para um autômato finito M é um grafo direcionado e rotulado. Os nodos representam os estados e fisicamente são representados por círculos, sendo que o estado inicial é diferenciado por uma seta com rótulo "início" e os estados finais são representados por círculos duplos. As arestas representam as transições, sendo que entre dois estados e_0 e e_1 existirá uma aresta direcionada de e_0 para e_1 com rótulo x considerando que: $x \in \Sigma \wedge \exists \delta(e_0, x) = e_1$ definida em M .

Pode-se também representar um autômato finito de forma tabular, por uma tabela de transição. Em uma tabela de transição as linhas representam os estados (o inicial é indicado por uma seta e os finais por asteriscos), as colunas representam os símbolos de entrada e o conteúdo da posição (e_0, x) será igual a e_1 caso exista $\delta(e_0, x) = e_1$, caso contrário será indefinida.

A máquina hipotética do exemplo anterior foi representada através de um diagrama de estados (FIGURA 9). Também poderia ser definida das seguintes formas:

- definição formal: $M_{\text{máquina}} = (K, \Sigma, \delta, e_0, F)$
 $K = \{\text{repouso, atividade, manutenção}\}$
 $\Sigma = \{\text{ON, OFF, PANE, OK}\}$
 $\delta = \{\delta(\text{repouso, ON}) = \text{atividade}$
 $\delta(\text{atividade, OFF}) = \text{repouso}$
 $\delta(\text{atividade, PANE}) = \text{manutenção}$
 $\delta(\text{manutenção, OK}) = \text{repouso}\}$
 $e_0 = \text{repouso}$
 $F = \text{repouso}$

- tabela de transição:

| δ | ON | OFF | PANE | OK |
|-------------------------|-----------|---------|------------|---------|
| \rightarrow^* repouso | atividade | - | - | - |
| atividade | - | repouso | manutenção | - |
| manutenção | - | - | - | repouso |

Os autômatos finitos classificam-se em: autômatos finitos determinísticos (AFD), autômatos finitos não-determinísticos (AFN), autômatos finitos com movimento vazio (AFNε).

DEFINIÇÃO nº 6: autômato finito determinístico

Um autômato finito determinístico (AFD) é um sistema de transições com número de estados e número de transições finitos. Formalmente, um AFD é um sistema formal $M = (K, \Sigma, \delta, e_0, F)$, onde:

- K conjunto finito não-vazio de estados;
- Σ alfabeto de símbolos de entrada;
- δ função de transição, definida em $K \times \Sigma \rightarrow K$. Para $\delta(e_{\text{velho}}, x) = e_{\text{novo}}$, a interpretação de δ é: "se M está no estado e_{velho} e o próximo símbolo de entrada é x , então x deve ser reconhecido e M deve ir para o estado e_{novo} ".
- e_0 estado inicial ($e_0 \in K$);
- F conjunto de estados finais ($F \subseteq K$).

Em um autômato finito determinístico, para cada estado, através do reconhecimento de um símbolo, o controle passa para **um único** outro estado.

Abaixo encontram-se dois exemplos de autômatos finitos determinísticos:

EXEMPLO 1: $L = \{x \mid x \in \{a, b\}^* \wedge |x| \geq 1 \wedge \text{o número de as é ímpar}\}$

diagrama de transição:

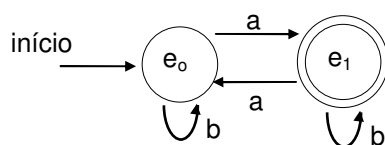


tabela de transição:

| δ | a | b |
|---------------------|-------|-------|
| \rightarrow e_0 | e_1 | e_0 |
| $*$ e_1 | e_0 | e_1 |

EXEMPLO 2: $L = \{x \mid x \in \{0, 1\}^* \wedge |x| \geq 0 \wedge x \text{ é definida pela ER: } (0^* 1^* 0^*)\}$

diagrama de transição:

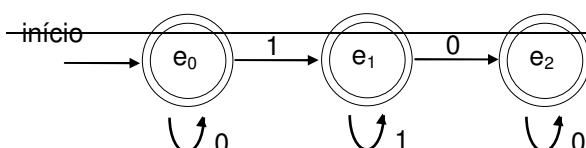


tabela de transição:

| δ | 0 | 1 |
|-----------------|-------|-------|
| * \rightarrow | e_0 | e_1 |

| * | e_1 | e_2 | e_1 |
|---|-------|-------|-------|
| * | e_2 | e_2 | - |

DEFINIÇÃO nº 7: autômato finito não-determinístico

Se uma determinada entrada, a partir de um certo estado pode levar, alternativamente, a mais de um estado seguinte, então o autômato deixa de ser determinístico. Em outras palavras, o símbolo de entrada e o estado atual não mais determinam qual será o estado seguinte.

Um autômato finito não-determinístico (AFN) é um sistema formal $M = (K, \Sigma, \delta, e_0, F)$, onde:

- K conjunto finito não-vazio de estados;
- Σ alfabeto de símbolos de entrada;
- δ função de transição, definida em $K \times \Sigma \rightarrow e(K)$, $e(K)$ é um subconjunto de K . Para $\delta(e_{\text{velho}}, x) = e_{\text{novo1}}, e_{\text{novo2}}, \dots, e_{\text{novo n}}$, a interpretação de δ é: "se M está no estado e_{velho} e o próximo símbolo de entrada é x , então x deve ser reconhecido e M pode ir tanto para o estado e_{novo1} , como para o estado e_{novo2} , ..., como para o estado $e_{\text{novo n}}$ ".
- e_0 estado inicial ($e_0 \in K$);
- F conjunto de estados finais ($F \subseteq K$).

Abaixo encontra-se um exemplo de autômato finito não-determinístico:

EXEMPLO 3: $L = \{x \mid x \in \{0, 1\}^* \wedge |x| \geq 2 \wedge x \text{ é definida pela ER: } (a \mid b)^* abb\}$
 diagrama de transição:

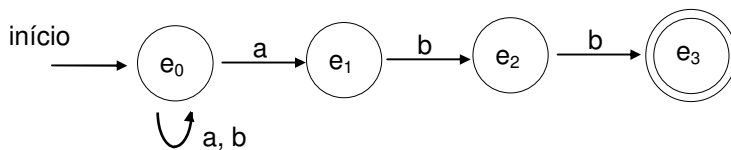


tabela de transição:

| δ | a | b |
|---------------|------------|-------|
| \rightarrow | e_0, e_1 | e_0 |
| e_1 | - | e_2 |
| e_2 | - | e_3 |
| * | e_3 | - |

DEFINIÇÃO nº 8: autômato finito com movimento vazio

Pode-se generalizar o modelo de autômato finito não-determinístico incluindo transições com entrada vazia (ϵ), as quais, segundo MENEZES (1997), podem ser interpretadas como um não determinismo interno ao autômato.

Um autômato finito com movimento vazio (AFN ϵ) é uma quintupla $M = (K, \Sigma, \delta, q_0, F)$ com todos os componentes de um AFN, mas com a função de transição definida sobre $K \times (\Sigma \cup \{\epsilon\}) \rightarrow e(K)$. Na representação da tabela de transições, usa-se uma coluna para o movimento vazio.

O AFN ϵ da figura seguinte aceita sentenças com tamanho maior ou igual a zero constituídas por qualquer número de 0s, seguidos por qualquer número de 1s seguidos, seguidos por qualquer número de 2s:

EXEMPLO 4: $L = \{x \mid x \in \{0, 1, 2\}^* \wedge |x| \geq 0 \wedge x \text{ é definida pela ER: } 0^* 1^* 2^*\}$
 diagrama de transição:

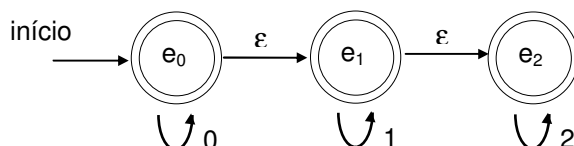


tabela de transição:

| δ | 0 | 1 | 2 | ϵ |
|-----------------|-------|---|---|------------|
| * \rightarrow | e_0 | - | - | e_1 |

* e_1 || - | e_1 | - | e_2 * e_2 || - | - | e_2 | -

DEFINIÇÃO nº 9: equivalência entre um AFN em um AFD

Apesar dos AFN representarem mais adequadamente algumas linguagens regulares, a implementação dos mesmos é mais complexa, exigindo algoritmos com *backtracking*. Assim, é mais conveniente encontrar um AFD equivalente.

Seja L um conjunto de sentenças aceitas por um AFN, então existe um AFD que aceita L. Seja $M = (K, \Sigma, \delta, e_0, F)$ um AFN e $M' = (K', \Sigma', \delta', e_0', F')$ um AFD tal que $M \equiv M'$, onde:

- K' conjunto finito de todas as combinações, sem repetições, de estados de K denotadas por $[e_0 e_1 \dots e_n]$ onde $e_i \in K$. Isto é, cada estado de M' é formado por um conjunto de estados de M. A ordem dos elementos não identifica outros estados, ou seja, $[e_0 e_1] = [e_1 e_0]$;
- Σ' alfabeto de símbolos de entrada, tal que $\Sigma' = \Sigma$;
- δ' função de transição, definida em $K' \times \Sigma' \rightarrow K'$, tal que $\delta'([e_{velho1} \dots e_{velho n}], x) = [e_{novo1} \dots e_{novo n}]$ se e somente se $\delta'([e_{velho1} \dots e_{velho n}], x) = \{e_{novo1} \dots e_{novo n}\}$. Isto é, se $([e_{velho1} \dots e_{velho n}] \in K') \underline{\text{e}} (\delta(e_{velho1}, x) = e_{novo1}, \dots, e_{novo i}) \underline{\text{e}} (\delta(e_{velho2}, x) = e_{novo i+1}, \dots, e_{novo k}) \underline{\text{e}} (\delta(e_{velho n}, x) = e_{novo k+1}, \dots, e_{novo n})$ são as transições de M, então $([e_{velho1} \dots e_{velho n}] \in K') \underline{\text{e}} ([e_{novo1}, \dots, e_{novo i}, e_{novo i+1}, \dots, e_{novo k}, e_{novo k+1}, \dots, e_{novo n}])$ será um estado de M' e M' conterá a transição: $\delta'([e_{velho1} \dots e_{velho n}], x) = [e_{novo1} \dots e_{novo n}]$;
- e_0' estado inicial ($[e_0] \in K'$);
- F' conjunto de estados $[e_0 e_1 \dots e_n]$ ($F' \subseteq K'$) tal que algum estado componente e_i é um estado final em M, ou seja, $e_i \in F$.

O algoritmo nº 1 abaixo apresenta os passos a serem seguidos para construir um AFD a partir de um AFN.

ALGORITMO nº 1: transformação de um AFN em um AFD

ENTRADA: um AFN $M = (K, \Sigma, \delta, e_0, F)$

SAÍDA: um AFD $M' = (K', \Sigma, \delta', e_0', F')$, tal que $M' \equiv M$

PASSO 1: determinar a primeira linha da tabela de transições do AFD como sendo o conjunto unitário contendo apenas o estado inicial do AFN.

PASSO 2: verificar para cada estado do AFD se as transições foram determinadas, ou seja, identificar se o rótulo R, possivelmente os rótulos de vários estados do AFN representando a união dos mesmos, de cada transição já é usado como rótulo de algum estado (em alguma linha) da tabela de transição do AFD. Em caso negativo, criar uma nova linha contendo aquele rótulo R e determinar, para cada símbolo de entrada, o conjunto resultante da união dos vários estados que compõem R. Repetir o passo 2 até que todas as transições tenham sido determinadas.

PASSO 3: determinar o estado inicial do AFD como sendo a primeira linha da tabela de transições. Determinar o(s) estado(s) final(is) do AFD como sendo todos os estados (todas as linhas) rotulados com conjuntos que contenham pelo menos um estado final de AFN.

EXEMPLO 5: os passos para transformar em AFD o AFN definido através da tabela de transição são:

PASSO 1:

tabela de transição (AFN):

| δ | a | b |
|----------|---|---|
|----------|---|---|

| | | | |
|---------------|-------|------------|-------|
| \rightarrow | e_0 | e_0, e_1 | e_0 |
| | e_1 | - | e_2 |

| | | | |
|---|-------|---|-------|
| | e_2 | - | e_3 |
| * | e_3 | - | - |

tabela de transição resultante (AFD):

| | | | |
|---------------|----------|------------|-------|
| | δ | a | b |
| \rightarrow | e_0 | e_0, e_1 | e_0 |
| | | | |
| | | | |

PASSO 2:

- a) o rótulo $\{e_0\}$ já é usado como rótulo de estado do AFD, mas o rótulo $\{e_0, e_1\}$ ainda não é usado. Assim, deve ser criada uma nova linha rotulada com $\{e_0, e_1\}$, sendo que para o símbolo a a união dos estados e_0 e e_1 resulta no conjunto $\{e_0, e_1\}$ e para o símbolo b a união dos estados e_0 e e_1 resulta no conjunto $\{e_0, e_2\}$. Dessa forma, a nova tabela de transição é:

tabela de transição resultante (AFD):

| | | | |
|---------------|----------------|----------------|----------------|
| | δ | a | b |
| \rightarrow | $\{e_0\}$ | $\{e_0, e_1\}$ | $\{e_0\}$ |
| | $\{e_0, e_1\}$ | $\{e_0, e_1\}$ | $\{e_0, e_2\}$ |
| | | | |

- b) o rótulo $\{e_0, e_1\}$ já é usado como rótulo de estado do AFD, mas o rótulo $\{e_0, e_2\}$ ainda não. Assim, deve ser criada uma nova linha rotulada com $\{e_0, e_2\}$, sendo que para o símbolo a a união dos estados e_0 e e_2 resulta no conjunto $\{e_0, e_1\}$ e para o símbolo b a união dos estados e_0 e e_2 resulta no conjunto $\{e_0, e_3\}$. Dessa forma, a nova tabela de transição é:

tabela de transição resultante (AFD):

| | | | |
|---------------|----------------|----------------|----------------|
| | δ | a | b |
| \rightarrow | $\{e_0\}$ | $\{e_0, e_1\}$ | $\{e_0\}$ |
| | $\{e_0, e_1\}$ | $\{e_0, e_1\}$ | $\{e_0, e_2\}$ |
| | $\{e_0, e_2\}$ | $\{e_0, e_1\}$ | $\{e_0, e_3\}$ |

- c) o rótulo $\{e_0, e_1\}$ já é usado como rótulo de estado do AFD, mas o rótulo $\{e_0, e_3\}$ ainda não. Assim, deve ser criada uma nova linha rotulada com $\{e_0, e_3\}$, sendo que para o símbolo a a união dos estados e_0 e e_3 resulta no conjunto $\{e_0, e_1\}$ e para o símbolo b a união dos estados e_0 e e_3 resulta no conjunto $\{e_0\}$. Dessa forma, a nova tabela de transição é:

tabela de transição resultante (AFD):

| | | | |
|---------------|----------------|----------------|----------------|
| | δ | a | b |
| \rightarrow | $\{e_0\}$ | $\{e_0, e_1\}$ | $\{e_0\}$ |
| | $\{e_0, e_1\}$ | $\{e_0, e_1\}$ | $\{e_0, e_2\}$ |
| | $\{e_0, e_2\}$ | $\{e_0, e_1\}$ | $\{e_0, e_3\}$ |
| | $\{e_0, e_3\}$ | $\{e_0, e_1\}$ | $\{e_0\}$ |

- d) tanto o rótulo $\{e_0, e_1\}$ como o rótulo $\{e_0\}$ já são usados como rótulos de estados do AFD.

PASSO 3: o estado inicial do AFD é a primeira linha da tabela de transições, ou seja, o estado $\{e_0\}$. O estado final do AFD é(são) aquele(s) que contém o(s) estado(s) final(is) do AFN, ou seja, o estado $\{e_0, e_3\}$. Tem-se o seguinte AFD:

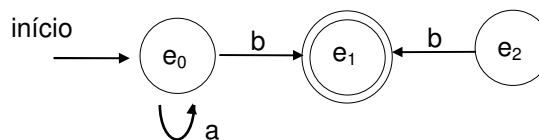
tabela de transição:

| δ' | a | b |
|------------------------------------|------------------------------------|------------------------------------|
| \rightarrow {e ₀ } | {e ₀ , e ₁ } | {e ₀ } |
| {e ₀ , e ₁ } | {e ₀ , e ₁ } | {e ₀ , e ₂ } |
| {e ₀ , e ₂ } | {e ₀ , e ₁ } | {e ₀ , e ₃ } |
| * | {e ₀ , e ₁ } | {e ₀ } |

DEFINIÇÃO n° 10: autômato finito determinístico mínimo

Segundo MENEZES (1997), o objetivo de transformar um AFD em um AFD mínimo (AFDm) é gerar um autômato finito equivalente com o menor número de estados possíveis implicando diretamente na implementação. Para uma determinada linguagem "o autômato finito mínimo é único. Assim, dois autômatos distintos que aceitam a mesma linguagem ao serem minimizados geram o mesmo autômato mínimo, diferenciando-se, eventualmente, na identificação dos estados" (MENEZES, 1997).

Um autômato finito determinístico $M = (K, \Sigma, \delta, e_0, F)$ para ser transformado em um AFDm não pode possuir **estados inacessíveis**, ou seja, não podem existir estados que nunca serão atingidos a partir do estado inicial; e não pode possuir **transições indefinidas**, ou seja, a partir de qualquer estado com qualquer símbolo do alfabeto de entrada não podem existir transições não previstas. No AFD abaixo, o estado e_2 é um estado inacessível e o estado e_1 não possui transições previstas com os símbolos a e b .



Para reduzir o número de estados de um autômato finito deve-se determinar os estados equivalentes. Dois estados são ditos equivalentes se e somente se para qualquer seqüência de símbolos resultam simultaneamente em estados finais ou estados não-finais também equivalentes entre si.

O algoritmo n° 2 abaixo apresenta os passos a serem seguidos para construir um AFDm a partir de um AFD.

ALGORITMO n° 2: transformação de um AFD em um AFDm

ENTRADA: um AFD $M = (K, \Sigma, \delta, e_0, F)$

SAÍDA: um AFDm $M' = (K', \Sigma, \delta', e_0', F')$, tal que $M' \equiv M$

PASSO 1: **eliminar os estados inacessíveis** da seguinte forma:

- marcar primeiramente o estado inicial;
- para cada estado e_i marcado, marcar todos os estados e_k que podem ser alcançados por uma transição a partir de e_i ;
- eliminar os estados não marcados.

PASSO 2: **eliminar as transições indefinidas** da seguinte forma:

- incluir um novo estado não-final Φ com transições para ele mesmo com todos os símbolos do alfabeto de entrada;
- trocar todas as transições não previstas por uma transição para o novo estado criado.

PASSO 3: **determinar os estados equivalentes**, segundo MENEZES (1997), da seguinte forma:

- construir uma tabela relacionando os estados distintos, onde cada par de estados ocorre

somente uma vez;

| | | | | |
|--------|-------|-------|-----|-----------|
| e_1 | | | | |
| ... | | | | |
| e_n | | | | |
| Φ | | | | |
| | e_0 | e_1 | ... | e_{n-1} |

- b) marcar todos os pares do tipo {estado final, estado não-final};
- c) para cada par $\{e_i, e_k\}$ não marcado e para cada símbolo $x \in \Sigma$, supor que $\delta(e_i, x) = e_r$ e $\delta(e_k, x) = e_s$. Tem-se as seguintes possibilidades:
- se $e_r = e_s$, então e_i é equivalente a e_k para o símbolo x e não deve ser marcado;
 - se $e_r \neq e_s$ e $\{e_r, e_s\}$ não está marcado, então incluir $\{e_i, e_k\}$ em uma lista a partir de $\{e_r, e_s\}$ para posterior análise;
 - se $e_r \neq e_s$ e $\{e_r, e_s\}$ está marcado, então e_i não é equivalente a e_k e $\{e_i, e_k\}$ deve ser marcado. Caso $\{e_i, e_k\}$ seja o primeiro de uma lista de pares de estados, então todos os pares da lista devem ser marcados (e recursivamente todos os pares das listas cujos primeiros elementos foram marcados).

Ou, segundo PRICE; EDELWEISS (1989), da seguinte forma:

- a) dividir K , o conjunto de estados, em duas classes de equivalência, uma contendo os estados finais e outra contendo todos os demais estados de K ;
- b) dividir sucessivamente as classes obtidas até que nenhuma nova classe possa ser obtida. Deve-se considerar nesse passo que um conjunto de estados $e_0, e_1 \dots e_j$ está em uma mesma classe de equivalência se todas as transições possíveis a partir de cada um destes estados levam o autômato aos estados $e_i, e_{i+1} \dots e_n$ estando estes últimos em uma mesma classe de equivalência.

PASSO 4: eliminar os estados inúteis/mortos (estados que não são finais e que a partir deles nenhum estado final pode ser alcançado) da seguinte forma:

- a) marcar inicialmente os estados finais;
- b) marcar todos os estados e_k que alcançam um estado marcado e_i por uma transição com qualquer símbolo de entrada;
- c) eliminar os estados não marcados, deixando indefinidas as transições que levam a um estado eliminado.

PASSO 5: construir $M' = (K', \Sigma, \delta', e_0', F')$, onde:

- K' conjunto finito de estados equivalentes obtidos;
- Σ' alfabeto de símbolos de entrada, tal que $\Sigma' = \Sigma$;
- δ' função de transição, definida em $K' \times \Sigma' \rightarrow K'$, tal que $\delta'(\{p\}, x) = \{q\} \leftrightarrow \delta(p_i, x) = q_i$ é uma transição de M e p_i e q_i são elementos de $\{p\}$ e $\{q\}$, respectivamente, onde $\{p\}$ e $\{q\}$ são conjuntos de estados equivalentes;
- e_0' estado equivalente que contém o estado inicial e_0 ;
- F' conjunto de estados equivalentes $\{e_0, e_1 \dots e_n\}$ ($F' \subseteq K'$) tal que algum estado componente e_i é um estado final em M , ou seja, $e_i \in F$.

EXEMPLO 6: os passos para transformar em AFDm o AFD definido através da tabela de transição são:

PASSO 1: eliminar os estados inacessíveis

tabela de transição:

| δ | a | b | acessível |
|-----------------------|-------|-------|-----------|
| \rightarrow^* e_0 | e_6 | e_1 | x |

| | | | |
|---------|-------|-------|---|
| e_1 | e_5 | e_4 | x |
| e_2 | e_2 | e_6 | x |
| * e_3 | e_0 | e_7 | |
| e_4 | e_4 | e_0 | x |

| | | | | |
|---|-------|-------|-------|-----|
| | e_5 | e_1 | e_2 | x |
| * | e_6 | e_6 | e_5 | x |
| | e_7 | e_7 | e_3 | |

tabela de transição resultante: (após a eliminação dos estados inacessíveis)

| δ | a | b |
|-----------------------|-------|-------|
| \rightarrow^* e_0 | e_6 | e_1 |
| e_1 | e_5 | e_4 |
| e_2 | e_2 | e_6 |
| e_4 | e_4 | e_0 |
| e_5 | e_1 | e_2 |
| * e_6 | e_6 | e_5 |

PASSO 2: eliminar as transições indefinidas: todas as transições estão definidas, logo não é necessário incluir o novo estado não-final Φ .

PASSO 3: determinar os estados equivalentes:

a) construir a tabela e marcar os pares de estados do tipo {estado final, estado não-final};

| | | | | | |
|-------|-------|-------|-------|-------|-------|
| e_1 | x | | | | |
| e_2 | x | | | | |
| e_4 | x | | | | |
| e_5 | x | | | | |
| e_6 | | x | x | x | x |
| | e_0 | e_1 | e_2 | e_4 | e_5 |

b) análise de cada par não marcado:

$\{e_0, e_6\} \Rightarrow$ $\delta(e_0, a) = e_6$; $\delta(e_6, a) = e_6$ como $e_6 = e_6$ não deve ser marcado
 $\delta(e_0, b) = e_1$; $\delta(e_6, b) = e_5$ como $\{e_1, e_5\}$ não está marcado, $\{e_0, e_6\}$ deve ser incluído na lista a partir de $\{e_1, e_5\}$ para posterior análise.

$\{e_1, e_2\} \Rightarrow$ $\delta(e_1, a) = e_5$; $\delta(e_2, a) = e_2$ como $\{e_2, e_5\}$ não está marcado, $\{e_1, e_2\}$ deve ser incluído na lista a partir de $\{e_2, e_5\}$ para posterior análise
 $\delta(e_1, b) = e_4$; $\delta(e_2, b) = e_6$ como $\{e_4, e_6\}$ está marcado, $\{e_1, e_2\}$ deve ser marcado.

$\{e_1, e_4\} \Rightarrow$ $\delta(e_1, a) = e_5$; $\delta(e_4, a) = e_4$ como $\{e_4, e_5\}$ não está marcado, $\{e_1, e_4\}$ deve ser incluído na lista a partir de $\{e_4, e_5\}$ para posterior análise
 $\delta(e_1, b) = e_4$; $\delta(e_4, b) = e_0$ como $\{e_0, e_4\}$ está marcado, $\{e_1, e_4\}$ deve ser marcado.

$\{e_1, e_5\} \Rightarrow$ $\delta(e_1, a) = e_5$; $\delta(e_5, a) = e_1$ como $\{e_1, e_5\}$ são os estados que estão sendo analisados não devem ser marcados
 $\delta(e_1, b) = e_4$; $\delta(e_5, b) = e_2$ como $\{e_2, e_4\}$ não está marcado, $\{e_1, e_5\}$ deve ser incluído na lista a partir de $\{e_2, e_4\}$ para posterior análise.

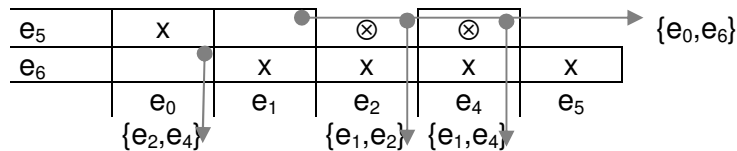
$\{e_2, e_4\} \Rightarrow$ $\delta(e_2, a) = e_2$; $\delta(e_4, a) = e_4$ como $\{e_2, e_4\}$ são os estados que estão sendo analisados não devem ser marcados
 $\delta(e_2, b) = e_6$; $\delta(e_4, b) = e_0$ como $\{e_0, e_6\}$ não está marcado, $\{e_2, e_4\}$ deve ser incluído na lista a partir de $\{e_0, e_6\}$ para posterior análise.

$\{e_2, e_5\} \Rightarrow$ $\delta(e_2, a) = e_2$; $\delta(e_5, a) = e_1$ como $\{e_1, e_2\}$ está marcado, $\{e_2, e_5\}$ deve ser marcado;
 como $\{e_1, e_2\}$ está na lista a partir de $\{e_2, e_5\}$, $\{e_1, e_2\}$ deve ser marcado.

$\{e_4, e_5\} \Rightarrow$ $\delta(e_4, a) = e_4$; $\delta(e_5, a) = e_1$ como $\{e_1, e_4\}$ está marcado, $\{e_4, e_5\}$ deve ser marcado.

| | | | | |
|-------|-----|-----------|--|--|
| e_1 | x | | | |
| e_2 | x | \otimes | | |
| e_4 | x | \otimes | | |

$\{e_1, e_5\}$



PASSO 4: eliminar os estados inúteis/mortos: todos os estados são úteis/vivos.

PASSO 5: construir $M' = (K', \Sigma, \delta', e_0', F')$, denominando $\{e_0, e_6\} = A$, $\{e_1, e_5\} = B$ e $\{e_2, e_4\} = C$, tem-se o seguinte AFDm:

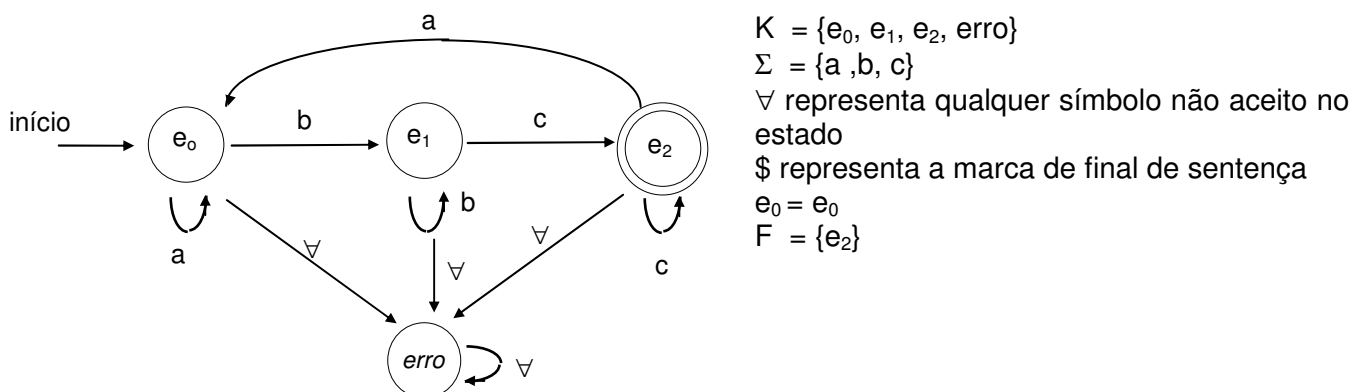
tabela de transição:

| δ' | a | b |
|-------------------|---|---|
| \rightarrow^* A | A | B |
| B | B | C |
| C | C | A |

DEFINIÇÃO nº 11: implementação de autômatos finitos

Existem duas formas básicas para implementação de autômatos finitos: implementação específica e implementação geral (ou genérica). Na exemplificação das formas de implementação de autômatos finitos, considere o AFDm abaixo.

EXEMPLO 7: Seja $M = (K, \Sigma, \delta, e_0, F)$ um AFDm definido pelo seguinte diagrama de transição:



A **implementação específica** consiste em representar cada estado do autômato finito através de um conjunto de instruções, ou seja, consiste em programar cada estado do autômato finito. Assim, por exemplo, os dois algoritmos abaixo implementam de forma específica o autômato anterior:

ALGORITMO implementação_específica {1}

VARIÁVEIS

CARACTER: símbolo

INÍCIO

```

e0:  LEIA (símbolo)
      SE símbolo = 'a' ENTÃO
        GOTO e0
      SENÃO
        SE símbolo ≠ 'b' ENTÃO
          GOTO erro
        FIMSE
      FIMSE

e1:  LEIA (símbolo)
      SE símbolo = 'b' ENTÃO
        GOTO e1
      SENÃO
        SE símbolo ≠ 'c' ENTÃO
          GOTO erro
        FIMSE
      FIMSE

```

```

e2:  LEIA (símbolo)
      SE símbolo = '$' ENTÃO
        ESCRIVA ('sentença reconhecida')
        GOTO fim
      SENÃO
        SE símbolo = 'c' ENTÃO
          GOTO e2
        SENÃO
          SE símbolo = 'a' ENTÃO
            GOTO e0
          FIMSE
        FIMSE
      FIMSE

erro:  ENQUANTO símbolo ≠ '$' FAÇA
        LEIA (símbolo)
        FIMENQUANTO
        ESCRIVA ('sentença não reconhecida')

fim:
FIM

```

ALGORITMO implementação_específica {2}

VARIÁVEIS

CARACTER: símbolo

INTEIRO: estado

INÍCIO

estado ← 0

REPITA

```

{ e0 }  ENQUANTO estado = 0 FAÇA
          LEIA (símbolo)
          ESCOLHA símbolo
            'a': estado ← 0
            'b': estado ← 1
          SENÃO estado ← 3
          FIMESCOLHA
        FIMENQUANTO

{ e1 }  ENQUANTO estado = 1 FAÇA
          LEIA (símbolo)
          ESCOLHA símbolo
            'b': estado ← 1
            'c': estado ← 2
          SENÃO estado ← 3
          FIMESCOLHA
        FIMENQUANTO

```

```

{ e2 }  ENQUANTO (estado = 2) E (símbolo ≠ '$')
          FAÇA
            LEIA (símbolo)
            ESCOLHA símbolo
              'a': estado ← 0
              'c': estado ← 2
              '$': estado ← 2
            SENÃO estado ← 3
            FIMESCOLHA
          FIMENQUANTO


{ erro } ENQUANTO (estado = 3) E (símbolo ≠ '$')
          FAÇA
            LEIA (símbolo)
          FIMENQUANTO
        ATÉ símbolo = '$'

SE estado = 2 ENTÃO
  ESCRIVA ('sentença reconhecida')
SENÃO
  ESCRIVA ('sentença não reconhecida')
FIMSE
FIM

```

A **implementação genérica** requer uma tabela de transições e um vetor de estados finais e consiste na especificação de um procedimento para interpretar a tabela de transições do autômato finito em função da sequência a ser analisada. Assim, para o AFDm anterior, tem-se:

tabela de transição:

|  | δ | a | b | c | \forall |
|---|----------|---|---|---|-----------|
| e_0 | 0 | 0 | 1 | 3 | 3 |
| e_1 | 1 | 3 | 1 | 2 | 3 |
| e_2 | 2 | 0 | 3 | 2 | 3 |
| erro | 3 | 3 | 3 | 3 | 3 |

vetor de estados finais:

| | |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 0 |

EF [estado] = 0

estado \notin F

EF [estado] = 1

estado \in F**ALGORITMO implementação_genérica**

VARIÁVEIS

CARACTER: símbolo

INTEIRO: estado

MATRIZ: tabela

VETOR: EF

INÍCIO

{ inicialização da tabela de transição e do vetor de estados finais }

estado \leftarrow 0 { estado inicial }

LEIA (símbolo)

ENQUANTO símbolo \neq '\$' FAÇAestado \leftarrow tabela [estado,símbolo]

LEIA (símbolo)

FIMENQUANTO

SE EF [estado] = 1 ENTÃO

ESCREVA ('*sentença reconhecida*')

SENÃO

ESCREVA ('*sentença não reconhecida*')

FIMSE

FIM

4.4 IMPLEMENTAÇÃO

Na implementação do analisador léxico deve-se: desconsiderar brancos à esquerda; considerar como marca de final de sentença o primeiro caracter que não pertencer ao alfabeto da sequência 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 caracter 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*.

LEITURA COMPLEMENTAR: AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores: princípios, técnicas e ferramentas.** Rio de Janeiro/RJ: LTC, 1995. (capítulo 3: p.38-66); JOSÉ NETO, J. **Introdução à compilação.** Rio de Janeiro/RJ: LTC, 1987 (capítulo 4:116-146; capítulo 5: 158-163).

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.

DEFINIÇÃO nº 1: gramática livre de contexto

Uma gramática livre de contexto (GLC) é definida como sendo uma quádrupla $G = (V_N, V_T, P, S)$, onde: V_N é um conjunto finito de símbolos não-terminais; V_T é um conjunto finito de símbolos terminais; P é conjunto finito de regras de produção (ou regras gramaticais) cada uma da forma $P = \{A \rightarrow \alpha \mid A \in V_N, \alpha \text{ é uma sentença em } (V_N \cup V_T)^*\}$; S é o símbolo inicial da gramática pertencente a V_N . Uma GLC admite apenas regras de produção cujo o lado esquerdo contém exatamente um não-terminal e cujo o lado direito contém qualquer combinação de terminais e não-terminais, incluindo a sentença vazia.

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:

- a notação das regras de produção apresentada no Capítulo 3. Assim, por exemplo, 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 1: 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 2: sintaxe da declaração de variáveis usando a notação BNF

$\langle \text{declaração de variáveis} \rangle ::= \text{variáveis inteiro} \langle \text{lista de identificadores} \rangle ;$

$\langle \text{lista de identificadores} \rangle ::= \text{identificador} \mid \text{identificador} , \langle \text{lista de identificadores} \rangle$

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

declaração de variáveis =

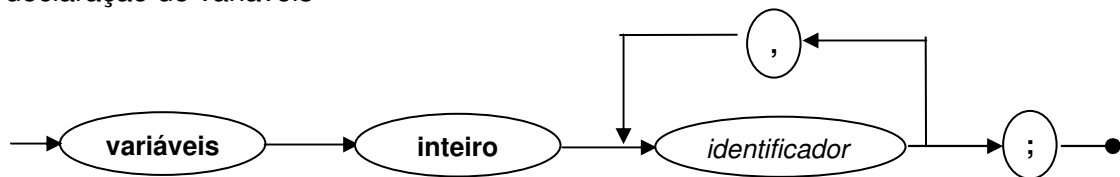


FIGURA 10: diagrama de sintaxe

Além dessas notações apresentadas, pode-se citar a notação de Wirth, as expressões regulares estendidas, as gramáticas de dois níveis, entre outras, como alternativas a serem usadas na especificação de linguagens de programação.

5.2.2 Árvore de derivação ou árvore sintática

Algumas vezes pode ser útil mostrar as derivações de uma GLC através 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.

DEFINIÇÃO nº 2: árvore de derivação (ou árvore sintática)

Seja $G = (V_N, V_T, P, S)$ uma GLC. Uma árvore é uma árvore de derivação para G se:

- todo nó n tem um rótulo r tal que $r \in V_N \cup V_T^*$, ou seja, todo nó de uma árvore de derivação é rotulado ou com terminal ou com não-terminal ou mesmo com a sentença vazia;
- o rótulo do nó raiz é o símbolo inicial S ;
- se um nó n com rótulo r tem um ou mais descendentes, então $r \in V_N$, ou seja, os nós não-folha são rotulados apenas com símbolos de V_N ;
- se um nó n tem rótulo R e os nós n_1, n_2, \dots, n_k são descendentes de n , da esquerda para a direita, com rótulos X_1, X_2, \dots, X_k , respectivamente, então $R \rightarrow X_1 X_2 \dots X_k$ é uma produção em P ;
- se um nó n tem rótulo R e o nó n_1 é descendente de n com rótulo ε , então $R \rightarrow \varepsilon$ é uma produção em P .

Considere, como exemplo, a gramática especificada anteriormente que define a declaração de variáveis. Uma árvore de derivação para a sentença “variáveis inteiro idade, altura;” gerada por esta gramática, poderia ser:

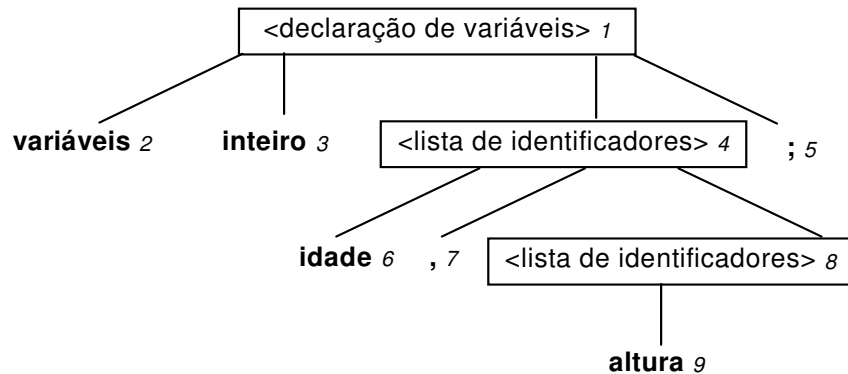


FIGURA 11: árvore de derivação

Os nós da árvore de derivação anterior foram numerados para facilitar a explicação: os nós não-folha são 1, 4 e 8. O nó 1 tem o rótulo <declaração de variáveis> e seus nós descendentes, da esquerda para a direita têm rótulos **variáveis**, **inteiro**, <lista de identificadores> e ;. Observa-se que <declaração de variáveis> ::= **variáveis inteiro** <lista de identificadores> ; é uma produção da gramática em questão. Da mesma forma, para os demais nós não-folha e seus nós descendentes existem produções correspondentes na gramática. Assim, as condições para que esta árvore seja uma árvore de derivação para G foram cumpridas.

Segundo HOPCROFT; ULLMAN (1979), pode-se estender a ordem “da esquerda para a direita” dos descendentes para produzir uma ordem da esquerda para a direita de todas as folhas. A isso denomina-se de limite de uma árvore de derivação. No exemplo acima, a concatenação da esquerda para a direita nas folhas da árvore produziria a seguinte sequência: 2, 3, 6, 7, 9, 5, ou seja, o limite da árvore de derivação do exemplo é a sentença “variáveis inteiro idade, altura;”. PRICE; EDELWEISS (1989) também define profundidade de uma árvore de derivação como o comprimento do maior caminho entre a raiz e um nó terminal. Assim, a árvore da figura 11 tem profundidade 3.

5.2.3 Derivação mais à esquerda e derivação mais à direita

Uma árvore de derivação ignora a ordem em que os símbolos são substituídos. Considere, por exemplo, a gramática abaixo que define as expressões aritméticas:

```

<expressão> → <expressão> + <expressão>
              | <expressão> * <expressão>
              | - <expressão>
              | ( <expressão> )
              | número
    
```

A sentença “- (123 * 456)” pode ser derivada de duas maneiras diferentes:

$\langle \text{expressão} \rangle \Rightarrow_G \langle \text{expressão} \rangle \Rightarrow_G (\langle \text{expressão} \rangle) \Rightarrow_G (\langle \text{expressão} \rangle * \langle \text{expressão} \rangle) \Rightarrow_G (\text{número} * \langle \text{expressão} \rangle) \Rightarrow_G (\text{número} * \text{número})$, ou seja, - (123 * 456)

ou

$\langle \text{expressão} \rangle \Rightarrow_G \langle \text{expressão} \rangle \Rightarrow_G (\langle \text{expressão} \rangle) \Rightarrow_G (\langle \text{expressão} \rangle * \langle \text{expressão} \rangle) \Rightarrow_G (\langle \text{expressão} \rangle * \text{número}) \Rightarrow_G (\text{número} * \text{número})$, ou seja, - (123 * 456)

Essas duas derivações correspondem à mesma árvore de derivação da FIGURA 12.

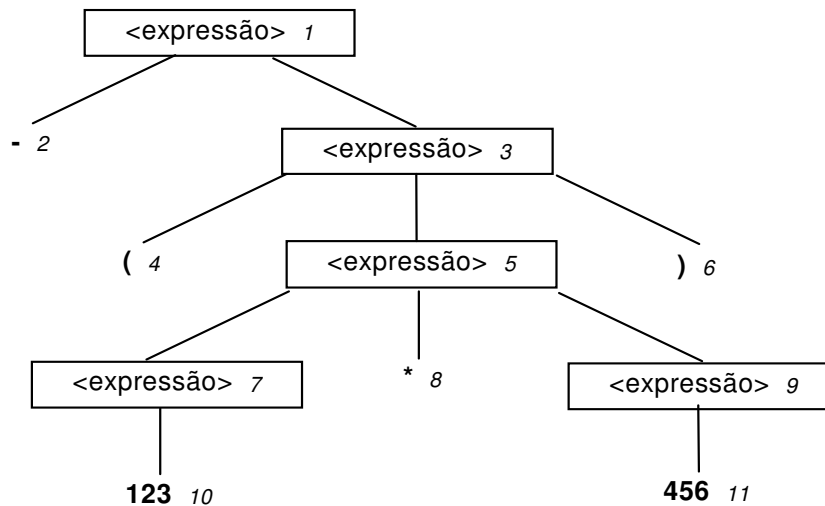


FIGURA 12: árvore de derivação

DEFINIÇÃO nº 3: derivação mais à esquerda e derivação mais à direita

Se a cada passo na produção de uma derivação é aplicado o não-terminal mais à esquerda, então a derivação é chamada derivação mais à esquerda. Similarmente, uma derivação onde a cada passo o não-terminal mais à direita é substituído, é chamada de derivação mais à direita. A primeira derivação mostrada anteriormente para a sentença “- (123 * 456)” é a derivação mais à esquerda, enquanto a segunda é a derivação mais à direita.

Observa-se que uma sentença pode ter várias derivações mais à esquerda e várias derivações mais à direita, já que pode haver mais de uma árvore de derivação para a sentença. Entretanto, para cada árvore de derivação, apenas uma derivação mais à esquerda e uma derivação mais à direita podem ser obtidas.

5.2.4 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 simples; não pode ser ambígua; deve estar fatorada, ou seja, deve ser determinística; não pode possuir recursão à esquerda.

DEFINIÇÃO nº 4: símbolos inúteis

Um símbolo terminal ou não-terminal é inútil em uma GLC se não aparecer na derivação de nenhuma sentença. Existe dois tipos de símbolos inúteis: aqueles que não aparecem em nenhuma forma sentencial da gramática, ou seja, jamais são gerados a partir do símbolo inicial; e aqueles que não geram nenhuma sentença de símbolos terminais. Aos primeiros dá-se o nome de *símbolos inalcançáveis*, aos segundos dá-se o nome de *símbolos inférteis* ou improdutivos. Observa-se que se o símbolo inicial for inútil, então a linguagem definida pela gramática é vazia.

Abaixo encontra-se um exemplo de uma gramática com símbolos inúteis:

EXEMPLO 4:

$$S \rightarrow AB \mid C$$

$$A \rightarrow a A \mid \varepsilon$$

$$B \rightarrow b B \mid \varepsilon$$

$$C \rightarrow c C$$

(o símbolo terminal c e o símbolo não-terminal C são inúteis)

DEFINIÇÃO nº 5: GLC ε -livre

As produções cujo o lado direito contém apenas a sentença vazia são chamadas de ε -produções. Uma GLC é dita ε -livre quando não possui ε -produções ou quando possui uma única ε -produção, $S \rightarrow \varepsilon$, onde S é o símbolo inicial da gramática e S não aparece do lado direito de nenhuma regra de produção.

DEFINIÇÃO nº 6: produções simples (ou produções unitárias)

Uma produção da forma $A \rightarrow \alpha$ em uma GLC é uma produção simples se α é um não-terminal. Um caso especial de produção simples é a produção $A \rightarrow A$, também chamada *produção circular*. Tal produção pode ser removida imediatamente sem afetar a capacidade de geração da gramática.

Abaixo encontra-se um exemplo de uma gramática com produções simples:

EXEMPLO 5:

$$S \rightarrow A B \mid C$$

$$A \rightarrow a A \mid \varepsilon$$

$$B \rightarrow b B \mid \varepsilon$$

$$C \rightarrow c C \mid c$$

DEFINIÇÃO nº 7: gramática ambígua

Uma GLC é ambígua quando, para alguma sentença da linguagem gerada, existe mais de uma árvore de derivação.

Abaixo encontram-se exemplos de gramáticas ambíguas e as gramáticas não ambíguas equivalentes:

EXEMPLO 6:

G_1 ambígua

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

G_2 não ambígua $\equiv G_1$

$$E \rightarrow id \ S \ E \mid id$$

$$S \rightarrow + \mid *$$

G_3 ambígua

$$\begin{aligned} \text{<comando>} &\rightarrow \text{IF <expressão> THEN <comando> ELSE <comando>} \\ &\quad \mid \text{IF <expressão> THEN <comando>} \\ &\quad \mid \text{<outro>} \end{aligned}$$

onde <outro> representa outro comando qualquer.

G_4 não ambígua $\equiv G_3$: por convenção o **ELSE** é sempre associado ao último **IF** mais próximo ainda não associado. Assim:

$$\text{<comando>} \rightarrow \text{<associado>} \mid \text{<não associado>}$$

$$\begin{aligned} \text{<associado>} &\rightarrow \text{IF <expressão> THEN <associado> ELSE <associado>} \\ &\quad \mid \text{<outro>} \end{aligned}$$

$$\begin{aligned} \text{<não associado>} &\rightarrow \text{IF <expressão> THEN <comando>} \\ &\quad \mid \text{IF <expressão> THEN <associado> ELSE <não associado>} \end{aligned}$$

Segundo FURTADO (1992), a gramática não ambígua equivalente não é utilizada por tornar complexa a análise semântica e geração de código.

DEFINIÇÃO n° 8: gramática fatorada (ou determinística)

Uma GLC está fatorada se não possui produções para um mesmo não-terminal no lado esquerdo cujo lado direito inicie com o mesmo conjunto de símbolos ou com símbolos que derivam seqüências que iniciem com o mesmo conjunto de símbolos.

Abaixo encontram-se exemplos de gramáticas não fatoradas e as gramáticas fatoradas equivalentes:

EXEMPLO 7:

G_1 não fatorada

$$A \rightarrow a B \mid a C \mid d A$$

$$B \rightarrow b B \mid b$$

$$C \rightarrow c C \mid c$$

G_2 fatorada $\equiv G_1$

$$A \rightarrow a A' \mid d A$$

$$A' \rightarrow B \mid C$$

$$B \rightarrow b B'$$

$$B' \rightarrow B \mid \varepsilon$$

$$C \rightarrow c C'$$

$$C' \rightarrow C \mid \varepsilon$$

G_3 não fatorada

$$A \rightarrow B \mid C$$

$$B \rightarrow a b B \mid a b$$

$$C \rightarrow a c C \mid a c$$

G_4 fatorada $\equiv G_3$

$$A \rightarrow a A'$$

$$A' \rightarrow b B' \mid c C'$$

$$B \rightarrow a b B'$$

$$B' \rightarrow B \mid \varepsilon$$

$$C \rightarrow a c C'$$

$$C' \rightarrow C \mid \varepsilon$$
DEFINIÇÃO n° 9: recursão à esquerda

Um não-terminal A em uma GLC é recursivo se $A \Rightarrow_G^+ \alpha A \beta$, para α e $\beta \in V^*$. 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 possui recursão à esquerda direta se P contém pelo menos uma produção da forma $A \rightarrow A \alpha$. Uma GLC possui recursão à esquerda indireta se existe em G uma derivação da forma $A \Rightarrow^n A \beta$, para algum $n \geq 2$.

Abaixo encontram-se exemplos de gramáticas com recursão à esquerda direta e indireta:

EXEMPLO 8:

G_1 recursão à esquerda direta

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

G_2 recursão à esquerda indireta

$$S \rightarrow A b \mid B c$$

$$A \rightarrow S a \mid a$$

$$B \rightarrow b B \mid \varepsilon$$

Para obter as características descritas acima, pode-se aplicar simplificações em uma GLC. No entanto, é importante salientar que a simplificação efetuada não deve alterar a linguagem gerada. Serão apresentados algoritmos para efetuar as seguintes simplificações: eliminação de símbolos inúteis; eliminação do não determinismo e eliminação de recursão à esquerda. Para eliminar ambigüidade não existe algoritmo. As demais transformações podem ser encontradas na bibliografia.

O algoritmo n° 1 abaixo apresenta os passos a serem seguidos para eliminação dos símbolos inúteis.

ALGORITMO n° 1: eliminação de símbolos inúteis

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: uma GLC $G = (V_N', V_T', P', S')$ sem símbolos inúteis

PASSO 1: **eliminar os símbolos inférteis** da seguinte forma:

$i \leftarrow 0$

$V_N[i] \leftarrow \{ \}$

REPITA

$i \leftarrow i + 1$

$V_N[i] \leftarrow V_N[i - 1] \cup \{A \mid A \rightarrow \alpha \in P, \alpha \in (V_N[i - 1] \cup V_T)^*\}$

ATÉ $V_N[i] = V_N[i - 1]$

$V_{N1} \leftarrow V_N[i]$ {conjunto de símbolos não-terminais férteis}

P_1 {possui as mesmas regras de produção de P , exceto aquelas cujos símbolos não-terminais não pertencem a V_{N1} }

PASSO 2: **eliminar os símbolos inalcançáveis** da seguinte forma:

$i \leftarrow 0$

$V_T[i] \leftarrow \{ \}$

$V_N[i] \leftarrow \{S\}$

REPITA

$i \leftarrow i + 1$

$V_N[i] \leftarrow V_N[i - 1] \cup \{X \mid A \rightarrow \alpha X \beta \in P_1, \alpha \text{ e } \beta \in (V_{N1} \cup V_T)^*, A \in V_N[i - 1]\}$

$V_T[i] \leftarrow V_T[i - 1] \cup \{x \mid A \rightarrow \alpha x \beta \in P_1, \alpha \text{ e } \beta \in (V_{N1} \cup V_T)^*, A \in V_N[i - 1]\}$

ATÉ $(V_N[i] = V_N[i - 1]) \text{ E } (V_T[i] = V_T[i - 1])$

$V_N' \leftarrow V_{N1} \cap V_N[i]$ {conjunto de símbolos não-terminais férteis e alcançáveis}

$V_T' \leftarrow V_T \cap V_T[i]$ {conjunto de símbolos terminais alcançáveis}

P' {possui as mesmas regras de produção de P_1 , exceto aquelas cujos símbolos não pertencem a $V_N' \cup V_T'$ }

$S' \leftarrow S$

O algoritmo nº 2 abaixo apresenta os passos a serem seguidos para eliminação do não determinismo.

ALGORITMO nº 2: fatoração de GLC (ou eliminação do não determinismo)

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: uma GLC $G = (V_N, V_T, P', S)$ fatorada

PASSO 1: **eliminar não determinismo direto** da seguinte forma: substituir as regras de produção na forma

$A \rightarrow \alpha \beta \mid \alpha \gamma$

por

$A \rightarrow \alpha A'$

$A' \rightarrow \beta \mid \gamma$

PASSO 2: **eliminar não determinismo indireto** da seguinte forma: transformar em não determinismo direto através de derivações sucessivas e então aplicar o PASSO 1. P' possui as regras de produção anteriormente especificadas que não possuem não determinismo e as regras de produção modificadas pela aplicação do algoritmo.

O algoritmo nº 3 abaixo apresenta os passos a serem seguidos para eliminação da recursão à esquerda.

ALGORITMO nº 3: eliminação da recursão à esquerda

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: uma GLC $G = (V_N, V_T, P', S)$ sem recursão à esquerda

PASSO 1: **eliminar a recursão à esquerda direta** da seguinte forma: substituir as regras de produção na forma

$A \rightarrow A \alpha_1 \mid \dots \mid A \alpha_n \mid \beta_1 \mid \dots \mid \beta_m$, onde nenhum β_i começa com A

por

$A \rightarrow \beta_1 A' \mid \dots \mid \beta_m A'$

$A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_n A' \mid \varepsilon$, onde A' é um novo não-terminal

PASSO 2: **eliminar a recursão à esquerda indireta** da seguinte forma:

ordenar os não-terminais de G em uma ordem qualquer (A_1, \dots, A_n)

PARA I DE 1 ATÉ n FAÇA

PARA J DE 1 ATÉ $i - 1$ FAÇA

substituir as regras de produção na forma

$A_i \rightarrow A_j \gamma$

por

$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$, onde $\alpha_1, \dots, \alpha_k$ são os lados direitos das regras de produção A_j , ou seja, $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$

FIMPARA

eliminar as recursões à esquerda diretas das A_i produções aplicando o PASSO 1

FIMPARA

5.2.5 Conjuntos: *first e follow*

Os conjuntos *FIRST* e *FOLLOW* estão associados aos símbolos terminais e não-terminais de uma gramática G e permitem construir a tabela a ser usada na análise sintática preditiva.

Seja α uma sentença qualquer gerada por uma gramática G . $\text{FIRST}(\alpha)$ será o conjunto de símbolos terminais que podem iniciar α , ou seja, que podem aparecer na posição mais à esquerda de α ou de seqüências derivadas (direta ou indiretamente) de α . Além disso, se $\alpha = \varepsilon$ ou $\alpha \Rightarrow^+ \varepsilon$ então $\varepsilon \in \text{FIRST}(\alpha)$. Para calcular o *FIRST*, pode-se aplicar o algoritmo a seguir.

ALGORITMO nº 4: cálculo do conjunto *first*

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: conjunto *first* para todos terminais e não-terminais de G

PASSO 1: $\forall^6 X \mid X \in (V_N \cup V_T)$

REPITA

SE $(X \in V_T)$ ENTÃO

$\text{FIRST}(X) = \{X\}$

SE $(X \in V_N)$ ENTÃO

SE $(X \rightarrow a \alpha \in P)$ ENTÃO

$\text{FIRST}(X) = \text{FIRST}(X) \cup \{a\}$

SE $(X \rightarrow \varepsilon \in P)$ ENTÃO

$\text{FIRST}(X) = \text{FIRST}(X) \cup \{\varepsilon\}$

SE $(X \rightarrow Y_1 \dots Y_n \in P)$ E $(\forall Y_i, \text{ com } 1 \leq i < (n - 1) \mid Y_i \in V_N \wedge \varepsilon \in \text{FIRST}(Y_i))$ ENTÃO

$\text{FIRST}(X) = \text{FIRST}(X) \cup (\text{FIRST}(Y_i) - \{\varepsilon\}) \cup \text{FIRST}(Y_n)$

em outras palavras: coloque $\text{FIRST}(Y_1)$ exceto ε em $\text{FIRST}(X)$, se $\varepsilon \in \text{FIRST}(Y_1)$ então coloque $\text{FIRST}(Y_2)$ exceto ε em $\text{FIRST}(X)$, se $\varepsilon \in \text{FIRST}(Y_2)$ então coloque $\text{FIRST}(Y_3)$ exceto ε em $\text{FIRST}(X)$, e assim sucessivamente até Y_n .

⁶ O símbolo \forall é lido como: *para todo*.

(Y_2) então ... coloque $FIRST(Y_n)$ em $FIRST(X)$

SE $(X \rightarrow \alpha_1 \mid \dots \mid \alpha_n)$ são todas as produções com lado esquerdo X ENTÃO

$FIRST(X) = FIRST(\alpha_1) \cup \dots \cup FIRST(\alpha_n)$

ATÉ $(FIRST(X))$ estar completo, isto é, não ser alterado)

Seja X um símbolo não-terminal da gramática G . $FOLLOW(X)$ será o conjunto de símbolos terminais que podem aparecer imediatamente após X em alguma sentença de G , ou seja, o conjunto de símbolos terminais x tais que exista uma derivação da forma $S \Rightarrow^+ \alpha X x \beta$. Durante a derivação pode ter existido símbolos não-terminais entre X e x . Neste caso, os mesmos derivaram ϵ e foram eliminados. Para calcular o $FOLLOW$, pode-se aplicar o algoritmo nº 5.

ALGORITMO nº 5: cálculo do conjunto *follow*

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: conjunto *follow* para todos os não-terminais de G

PASSO 1: $FOLLOW(S) = FOLLOW(S) \cup \{\$, \}$, onde $\$$ é marca de final de sentença e S o símbolo inicial da gramática

PASSO 2: $\forall p \mid p \in P$

REPITA

SE $(X \rightarrow \alpha Y \beta \in P)$ E $(\beta \neq \epsilon)$ ENTÃO

$FOLLOW(Y) = FIRST(\beta) - \{\epsilon\}$

ATÉ analisar todas as regras de produção

PASSO 3: $\forall X \mid X \in V_N$

REPITA

REPITA

SE $(X \rightarrow \alpha Y \in P)$ ENTÃO

$FOLLOW(Y) = FOLLOW(Y) \cup FOLLOW(X)$

ATÉ analisar todas as regras de produção

REPITA

SE $(X \rightarrow \alpha Y \beta \in P)$ E $(\epsilon \in FIRST(\beta))$ ENTÃO

$FOLLOW(Y) = FOLLOW(Y) \cup FOLLOW(X)$

ATÉ analisar todas as regras de produção

ATÉ $(FOLLOW(X))$ estar completo, isto é, não ser alterado)

Observa-se que o conjunto *FIRST* é definido para sentenças de símbolos, enquanto que o conjunto *FOLLOW* é definido apenas para não-terminais. Além disso, os elementos de um conjunto *FIRST* pertencem ao conjunto $V_T \cup \{\epsilon\}$ e os elementos de um conjunto *FOLLOW* pertencem ao conjunto $V_T \cup \{\$, \}$.

5.3 AUTÔMATO COM 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.

LEITURA COMPLEMENTAR: MENEZES, P. F. B. *Linguagens Formais e Autômatos*. Porto Alegre/RS: II-UFRGS, Sagra Luzzatto, 1997. 168 (capítulo 3: 112-121).

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.

DEFINIÇÃO nº 10: *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 9:

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 $X \Rightarrow^* \epsilon$, $\text{FIRST}(X) \cap \text{FOLLOW}(X) = \emptyset$.

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:

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \varepsilon$$

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

o analisador sintático descendente recursivo correspondente é composto pelos seguintes procedimentos:

ALGORITMO descendente_recursivo

INÍCIO

 análise_léxica (sentença)
 $token \leftarrow \text{próximo}(\text{sentença})$
 E (t)

FIM

PROCEDIMENTO E (VAR token: STRING)

INÍCIO

 T (token)
 Elinha (token)

FIM

PROCEDIMENTO Elinha (VAR token: STRING)

INÍCIO

 SE $token = '+'$ ENTÃO
 $token \leftarrow \text{próximo}(\text{sentença})$
 T (token)
 Elinha (token)

FIMSE

FIM

PROCEDIMENTO T (VAR token: STRING)

INÍCIO

 SE $token = '('$ ENTÃO
 $token \leftarrow \text{próximo}(token)$
 E (token)
 SE $token = ')'$ ENTÃO
 $token \leftarrow \text{próximo}(\text{sentença})$
 SENÃO erro sintático
 FIMSE
 SENÃO
 SE $token = id$ ENTÃO
 $token \leftarrow \text{próximo}(\text{sentença})$
 SENÃO erro sintático
 FIMSE

FIMSE

FIM

O analisador sintático preditivo LL(1) é uma maneira eficiente de implementar um analisador descendente recursivo. O analisador preditivo utiliza:

- uma pilha sintática, usada para simular a recursividade, inicializada com \$ e o símbolo inicial da gramática em questão;
- um *buffer* de entrada, contendo a sentença a ser analisada seguida por uma marca de final de sentença (\$);
- uma gramática GLC com as produções numeradas de 1 a p;
- uma tabela de análise sintática (tabela de *parsing*), contendo as ações a serem efetuadas. É uma matriz TP [X, a] onde $X \in V_N$ e $a \in V_T$, ou seja, cada linha de TP é rotulada com um símbolo não-terminal e cada coluna de TP é rotulada com um símbolo terminal de G mais a marca de final de sentença;
- um procedimento de análise sintática que consiste em determinar, a partir do elemento do topo da pilha sintática (X) e do próximo símbolo de entrada (a), a ação a ser efetuada que poderá ser:
 - SE $(X = a)$ E $(a = \$)$ ENTÃO **fim de análise**. A ação consiste em encerrar a análise sintática.
 - SE $(X = a)$ E $(a \neq \$)$ ENTÃO **reconhecimento sintático do símbolo de entrada**. A ação consiste em retirar X do topo da pilha sintática e a do *buffer* de entrada.
 - SE $(X \neq a)$ E $(X \in V_T)$ ENTÃO **ocorrência de erro sintático**. A ação consiste em encerrar a análise sintática ou em ativar as rotinas de recuperação de erro para que a situação seja contornada e a análise possa continuar.

SE $(X \in V_N)$ ENTÃO **derivação pela produção p_n** ou **indicação de erro sinático**. A ação

consiste em consultar a tabela de *parsing* TP [X, a] e, no primeiro caso, substituir X, o elemento do topo da pilha sintática, pelos símbolos que compõem o lado direito da produção p_n de tal modo que o símbolo mais à esquerda fique no topo da pilha, ou, no segundo caso, ativar as rotinas de recuperação de erro para que a situação seja contornada e a análise possa continuar.

O processo de determinar e efetuar as ações sintáticas deverá ser repetido até que X, o elemento do topo da pilha sintática, seja igual a \$ ou quando um erro for encontrado, nas implementações sem recuperação de erro.

O termo preditivo deve-se ao fato de que a pilha sintática sempre contém a descrição do restante da sentença, isto é, a pilha “prevê” a parte da sentença que deve estar na entrada para que sentença esteja correta.

ALGORITMO nº 6: analisador sintático preditivo LL(1)

ENTRADA: lista de *tokens*, resultante do processo de análise léxica

SAÍDA: código de erro, indicando se análise foi ou não efetuada com sucesso

PASSO 1:

```

empilha (pilha_sintática, '$')
empilha (pilha_sintática, S)
inserir_no_fim (lista_tokens, '$')
código_erro ← 0
a ← retira (lista_tokens, 1)
X ← desempilha (pilha_sintática)
ENQUANTO (X ≠ '$') E (código_erro = 0) FAÇA
    SE (é_terminal (X)) ENTÃO
        SE X = a ENTÃO
            a ← retira (lista_tokens, 1)
            X ← desempilha (pilha_sintática)
        SENÃO SE X = ε ENTÃO
            X ← desempilha (pilha_sintática)
        SENÃO
            código_erro ← -1
        FIMSE
    FIMSE
    SENÃO
        SE existe_produção (X, a) ENTÃO
            número_produção ← tabela_parsing [X, a]
            produção ← gramática (número_produção)
            empilha (pilha_sintática, produção)
            X ← desempilha (pilha_sintática)
        SENÃO
            código_erro ← -1
        FIMSE
    FIMSE
FIMENQUANTO

```

5.5 IMPLEMENTAÇÃO

Para a construção de um analisador sintático são necessários basicamente três passos: definir a gramática para a linguagem; escolher uma técnica de análise sintática e construir a tabela de *parsing*, conforme o caso; implementar o algoritmo de análise sintática numa linguagem de programação.

A idéia geral para a construção da tabela de *parsing* é a seguinte: SE $(X \rightarrow \alpha \in P)$ E $(a \in \text{FIRST}(\alpha))$ ENTÃO se X está no topo da pilha sintática e a é o próximo símbolo da entrada, deve-se derivar X, usando a produção α ($X \rightarrow \alpha$). No entanto, SE $(\alpha = \varepsilon)$ OU $(\alpha \Rightarrow^* \varepsilon)$ ENTÃO desde que ε nunca aparece no *buffer* de entrada, deve-se derivar X, usando a produção α ($X \rightarrow \alpha$), caso $(a \in \text{FOLLOW}(X))$. A tabela de *parsing* de analisadores preditivos LL(1) é construída pela aplicação do seguinte algoritmo.

ALGORITMO nº 7: construção da tabela de *parsing*

ENTRADA: uma GLC $G = (V_N, V_T, P, S)$

SAÍDA: tabela de *parsing* (TP) para o analisador sintático preditivo LL(1)

PASSO 1: numerar de 1 a p as regras de produção da gramática

PASSO 2: $\forall p \mid p \in P$

PARA cada produção $(X \rightarrow \alpha \in P)$ FAÇA

$\forall a \mid (a \in \text{FIRST}(\alpha))$, exceto ε , coloque o número da produção $X \rightarrow \alpha$ em TP (X, a)

SE $(\varepsilon \in \text{FIRST}(\alpha))$ ENTÃO

$\forall b \mid (b \in \text{FOLLOW}(X))$, coloque o número da produção $X \rightarrow \alpha$ em TP (X,

b)

FIMSE

FIMPARA

PASSO 3: as posições em TP que ficaram indefinidas representam situações de erro e podem ser preenchidas com as indicações adequadas.

A tabela de *parsing* de analisadores preditivos deve possuir a propriedade de que, para cada entrada da tabela, exista no máximo o número de uma produção, tornando a análise da sentença de entrada determinística.

LEITURA COMPLEMENTAR: AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores: princípios, técnicas e ferramentas.** Rio de Janeiro/RJ: LTC, 1995. (capítulo 4: p.72-111); JOSÉ NETO, J. **Introdução à compilação.** Rio de Janeiro/RJ: LTC, 1987. (capítulo 5).

6 ANALISADOR SEMÂNTICO E GERADOR DE CÓDIGO INTERMEDIÁRIO

6.1 FUNÇÃO

O próximo passo no processo de compilação consiste em **determinar o significado** de cada construção sintática e então **traduzir o programa** escrito na linguagem fonte para o código intermediário da máquina virtual. Basicamente, os *tokens* são identificados, a árvore sintática correspondente é construída e então percorrida, avaliando-se cada nodo para determinar a **semântica** correspondente. O processo pode ser esquematizado pela figura abaixo:



Na determinação do significado do programa fonte, o analisador semântico: verifica a coerência da declaração e do uso de identificadores; cria e mantém a tabela de símbolos; detecta e trata erros semânticos.

6.2 ESPECIFICAÇÃO DA SEMÂNTICA DE UMA LINGUAGEM DE PROGRAMAÇÃO

Segundo JOSÉ NETO (1987), "ao contrário da sintaxe, que é facilmente formalizável, (...) a semântica, apesar de também poder ser expressa formalmente, exige para isto notações substancialmente mais complexas, de aprendizagem mais difícil". A semântica de linguagens de programação pode ser especificada: informalmente, através de textos em linguagem natural; formalmente, através de métodos formais tais como gramática de atributos, semântica operacional, semântica de ações, semântica axiomática ou semântica denotacional; ou de maneira semi-formal através de ações semânticas embutidas na gramática da linguagem em questão.

Com o uso de ações semânticas, tem-se a **tradução dirigida pela sintaxe** onde a **geração do código intermediário** é feita por um conjunto de rotinas, responsáveis pela determinação do significado de cada instrução do programa fonte. As ações semânticas, parte do compilador que interpreta o significado de um programa, executam esta interpretação baseada na estrutura sintática do programa. Isto é, o analisador sintático ativa a execução das ações semânticas sempre que forem atingidos determinados estados do reconhecimento ou sempre que determinadas transições ocorrerem durante a análise do programa fonte.

As ações semânticas são normalmente associadas com regras de produção da gramática ou sub-árvores de uma árvore sintática. A idéia é embutir na gramática símbolos de ação que indicarão verificações de ordem semântica a serem efetuadas e instruções a serem geradas no processo de compilação. No entanto, a tradução dirigida pela sintaxe pode ser implementada em um único passo através da avaliação das ações semânticas sem construção explicitamente de uma árvore sintática. Faz-se uma linearização da árvore. As duas principais vantagens dessa abordagem são: o compilador fica mais simples desde que nenhuma árvore é construída portanto não é necessário caminhar; o espaço de memória requerido é menor para processar um programa desde que nenhuma árvore sintática completa é explicitamente construída.

A presença de ações semânticas nas regras de produção da gramática implicam na chamada das rotinas semânticas correspondentes quando o programa fonte é analisado. *Que e quando* as rotinas semânticas serão chamadas são explicitamente projetadas para cada

construção da linguagem. Por exemplo, o comando **if** pode ser definido pela seguinte regra de produção:

`<comando if> → if <expressão> #StartIF then <lista comandos> endif #FinishIF`

ou seja, essa produção especifica que duas rotinas semânticas (**#StartIF** e **#FinishIF**) devem ser apropriadamente chamadas quando da análise sintática da `<expressão>` e do **endif** no comando.

Nem todos os símbolos sintáticos (terminais e não-terminais) têm ações semânticas associadas, desde que não possuem informações semânticas associadas. Contudo, algumas produções podem incluir mais do que uma ação semântica. O uso de várias ações semânticas é comum em estruturas de controle (case, if, loop, etc.) desde que é necessário gera código em mais de um ponto do programa fonte.

Embora as rotinas semânticas não chamem outras rotinas semânticas explicitamente, se comunicam implicitamente umas com as outras utilizando para tanto "registros semânticos" recebidos como parâmetros (ou variáveis locais ao analisador sintático). Elas usam informações sobre um *token* fornecidas pelo **analisador léxico** para construir o registro semântico apropriado. O processamento pode incluir geração de código, registro de informações em uma tabela de símbolos, busca de identificadores e atributos associados na tabela de símbolos, verificação dos tipos dos argumentos para operandos para determinar o tipo do resultado, bem como construção de um novo registro semântico.

6.3 DEFINIÇÃO DA MÁQUINA VIRTUAL

No projeto do analisador semântico e gerador de código, deve-se decidir se alguma representação intermediária será usada ou se será gerado código de máquina. As vantagens do uso de representação intermediária estão basicamente relacionadas à geração de código independente da máquina alvo, facilitando a otimização e possibilitando a portabilidade de código. Gerar código de máquina diretamente diminui o *overhead* de um passo extra para a tradução da representação interna no código objeto, e permite a construção de um modelo de compilação de um-passo conceitualmente mais simples. Assim, representações intermediárias são de real valor se otimização e portabilidade são questões importantes. Se essas questões não são importantes, a simplicidade de geração de código de máquina é preferível.

Uma grande variedade de formas de representação intermediária tem sido usada, por várias razões, na história de compiladores. A mais simples é a notação polonesa. "Originalmente utilizada na compilação de expressões, foi estendida posteriormente para abranger construções não aritméticas. Utiliza o conceito de pilha, onde operandos são armazenados até que um operador force sua manipulação, operação e desempilhamento. O compilador que gera código em notação polonesa necessita de um bom suporte do ambiente de execução para a realização das operações correspondentes aos operadores da notação polonesa" (JOSÉ NETO, 1987). Há duas variantes para a notação polonesa: a notação **prefix**, onde os operadores aparecem antes dos operandos; e a notação **postfix**, onde os operadores aparecem depois dos operandos.

Outro tipo de representação intermediária usado é chamado de **código de 3 endereços** que é código *assembler* generalizado para uma máquina virtual de 3 endereços, onde cada instrução consiste de um operador e 3 endereços, sendo 2 para operandos e um para o armazenamento do resultado. Pode ser representada com tripas ou quadras.

O compilador da linguagem que está sendo definida deverá gerar código intermediário para uma máquina virtual usando o conjunto de instruções especificado abaixo. Esse conjunto de instruções deve ser passível de simulação em qualquer linguagem de programação. A máquina virtual compõe-se dos seguintes elementos:

- uma **área de instruções**: uma região em disco ou em memória onde estão armazenadas as

instruções a serem executadas;

- uma **pilha de dados**: uma área de alocação onde os dados manipulados pelas instruções são armazenados;
- um **conjunto de registradores**: sendo um **ponteiro**, inicializado com 1 (um), para determinar a próxima instrução a ser executada; um **topo**, inicializado com 0 (zero), para determinar o topo da pilha de dados.
- um **conjunto de instruções**: cada instrução na máquina hipotética tem a seguinte forma geral (CÓDIGO, parâmetro), sendo:

| CÓDIGO | parâmetro | comentário |
|---------------------------------|--------------|---|
| aritméticas | | |
| ADD | 0 | executar operação aritmética soma |
| DIV | 0 | executar operação aritmética divisão |
| MUL | 0 | executar operação aritmética multiplicação |
| SUB | 0 | executar operação aritmética subtração |
| memória (variáveis, constantes) | | |
| ALB | deslocamento | alocar espaço na pilha de dados, para variáveis do tipo lógico, igual ao deslocamento passado como parâmetro |
| ALI | deslocamento | alocar espaço na pilha de dados, para variáveis do tipo inteiro, igual ao deslocamento passado como parâmetro |
| ALR | deslocamento | alocar espaço na pilha de dados, para variáveis do tipo real, igual ao deslocamento passado como parâmetro |
| ALS | deslocamento | alocar espaço na pilha de dados, para variáveis do tipo literal, igual ao deslocamento passado como parâmetro |
| LDB | constante | carregar na pilha de dados a constante lógica passada como parâmetro |
| LDI | constante | carregar na pilha de dados a constante inteira passada como parâmetro |
| LDR | constante | carregar na pilha de dados a constante real passada como parâmetro |
| LDS | constante | carregar na pilha de dados a constante literal passada como parâmetro |
| LDV | endereço | carregar na pilha de dados o valor armazenado no endereço passado como parâmetro |
| STR | endereço | armazenar conteúdo do topo da pilha de dados no endereço passado como parâmetro |
| lógica | | |
| AND | 0 | executar operação lógica E |
| NOT | 0 | executar operação lógica NÃO |
| OR | 0 | executar operação lógica OU |
| relacional | | |
| BGE | 0 | executar operação relacional maior ou igual |
| BGR | 0 | executar operação relacional maior que |
| DIF | 0 | executar operação relacional diferente |
| EQL | 0 | executar operação relacional igual |
| SME | 0 | executar operação relacional menor ou igual |
| SMR | 0 | executar operação relacional menor que |
| desvio / controle | | |
| JMF | endereço | desviar quando for falso para a instrução do endereço passado como parâmetro |
| JMP | endereço | desviar para a instrução do endereço passado como parâmetro |
| JMT | endereço | desviar quando for verdadeiro para a instrução do endereço passado como parâmetro |
| STP | 0 | finalizar a execução |
| entrada / saída de dados | | |
| REA | tipo | ler dados do tipo passado como parâmetro, sendo que 1 indica um dado do tipo inteiro, 2 indica um dado do tipo real, 3 indica um dado do tipo literal |
| WRT | 0 | Escrever dados |
| instrução: ADD, 0 | | $\text{pilha}[\text{topo} - 1] \leftarrow \text{pilha}[\text{topo} - 1] + \text{pilha}[\text{topo}]$ $\text{topo} \leftarrow \text{topo} + 1$ |

ponteiro ← ponteiro + 1

instrução: ALB, deslocamento

PARA i DE (topo + 1) ATÉ (topo + deslocamento)
FAÇA pilha [i] := FALSE
FIMPARA
topo ← topo + deslocamento
ponteiro ← ponteiro + 1

instrução: ALI, deslocamento

PARA i DE (topo + 1) ATÉ (topo + deslocamento)
FAÇA pilha [i] := 0
FIMPARA
topo ← topo + deslocamento
ponteiro ← ponteiro + 1

instrução: ALR, deslocamento

PARA i DE (topo + 1) ATÉ (topo + deslocamento)
FAÇA pilha [i] := 0.0
FIMPARA
topo ← topo + deslocamento
ponteiro ← ponteiro + 1

instrução: ALS, deslocamento

PARA i DE (topo + 1) ATÉ (topo + deslocamento)
FAÇA pilha [i] := ''
FIMPARA
topo ← topo + deslocamento
ponteiro ← ponteiro + 1

instrução: AND, 0

pilha [topo - 1] ← (pilha [topo - 1]) E (pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: BGE, 0

pilha [topo - 1] ← (pilha [topo - 1] ≥ pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: BGR, 0

pilha [topo - 1] ← (pilha [topo - 1] > pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: DIF, 0

pilha [topo - 1] ← (pilha [topo - 1] ≠ pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: DIV, 0

SE pilha [topo] = 0
ENTÃO ESCRIVA ('*RUNTIME error: divisão por 0*')
HALT
FIMSE
pilha [topo - 1] ← pilha [topo - 1] / pilha [topo]
topo ← topo - 1

ponteiro ← ponteiro + 1

instrução: EQL, 0

pilha [topo - 1] ← (pilha [topo - 1] = pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: JMF, endereço

SE (pilha [topo] = FALSE)

ENTÃO ponteiro ← endereço
SENÃO ponteiro ← ponteiro + 1
FIMSE
topo ← topo - 1

instrução: JMP, endereço

ponteiro ← endereço

instrução: JMT, endereço

SE (pilha [topo] = TRUE)
ENTÃO ponteiro ← endereço
SENÃO ponteiro ← ponteiro + 1
FIMSE
topo ← topo - 1

instrução: LDV, endereço

topo ← topo + 1
pilha [topo] ← pilha [endereço]
ponteiro ← ponteiro + 1

instrução: LDB, constante

topo ← topo + 1
pilha [topo] ← constante
ponteiro ← ponteiro + 1

instrução: LDI, constante

topo ← topo + 1
pilha [topo] ← constante
ponteiro ← ponteiro + 1

instrução: LDR, constante

topo ← topo + 1
pilha [topo] ← constante
ponteiro ← ponteiro + 1

instrução: LDS, constante

topo ← topo + 1
pilha [topo] ← constante
ponteiro ← ponteiro + 1

instrução: MUL, 0

pilha [topo - 1] ← pilha [topo - 1] * pilha [topo]
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: NOT, 0

pilha [topo] ← NAO (pilha [topo])
ponteiro ← ponteiro + 1

instrução: OR, 0

pilha [topo - 1] ← (pilha [topo - 1]) OU (pilha [topo])
topo ← topo - 1
ponteiro ← ponteiro + 1

instrução: REA, tipo

topo ← topo + 1
LEIA (pilha [topo])
ponteiro ← ponteiro + 1
SE tipo (pilha [topo]) ≠ tipo
ENTÃO ESCRIVA ('*RUNTIME error*')
HALT
FIMSE

instrução: SME, 0

pilha [topo - 1] ← (pilha [topo - 1] ≤ pilha [topo])
topo ← topo - 1

ponteiro \leftarrow ponteiro + 1

instrução: SMR, 0

pilha [topo - 1] \leftarrow (pilha [topo - 1] < pilha [topo])

topo \leftarrow topo - 1

ponteiro \leftarrow ponteiro + 1

instrução: STR, endereço

pilha [endereço] \leftarrow pilha [topo]

topo \leftarrow topo - 1

ponteiro \leftarrow ponteiro + 1

instrução: STP, 0

HALT

instrução: SUB, 0

pilha [topo-1] \leftarrow pilha [topo-1] - pilha [topo]

topo \leftarrow topo - 1

ponteiro \leftarrow ponteiro + 1

instrução: WRT, 0

ESCREVA (pilha [topo])

topo \leftarrow topo - 1

ponteiro \leftarrow ponteiro + 1

6.4 DEFINIÇÃO DA TABELA DE SÍMBOLOS

A tabela de símbolos armazena informações (atributos) relativas aos identificadores encontrados no programa fonte durante o processo de compilação. Desde que esses atributos são essencialmente uma representação do significado semântico dos identificadores com os quais estão associados, uma tabela de símbolos é algumas vezes chamada de dicionário. A tabela de símbolos é um componente necessário de um compilador porque a definição dos identificadores aparece em um único lugar no programa (declaração), enquanto o identificador pode ser usado em vários lugares. A cada vez que é usado, a tabela de símbolos fornece acesso a informações sobre o identificador armazenadas quando sua declaração foi processada.

A organização da tabela de símbolos depende da estrutura da linguagem que está sendo projetada. As linhas da tabela de símbolos têm, no mínimo, as seguintes informações⁷:

| NOME | CATEGORIA - pode ser: | ATRIBUTO - depende da categoria: |
|--------------------------------|---|---|
| <i>identificador analisado</i> | 0: identificador de programa | nada |
| | 1: identificador de variável inteira | deslocamento (endereço de memória) na pilha de dados |
| | 2: identificador de variável real | deslocamento (endereço de memória) |
| | 3: identificador de variável literal | deslocamento (endereço de memória) |
| | 4: identificador de variável lógica | deslocamento (endereço de memória) |

As operações para manipulação da tabela de símbolos são:

- **criar:** cria uma nova tabela de símbolos vazia.
USO: antes de começar a análise semântica/geração de código.
- **destruir:** remove todas as entradas da tabela de símbolos.
USO: após executar a análise semântica/geração de código.
- **pesquisar:** busca por um identificador na tabela de símbolos. Para isso, verifica a existência do identificador na tabela, indicando a posição em que se encontra.
USO: quando da compilação do programa.
- **inserir:** insere um identificador na tabela de símbolos. Para isso, verifica a existência do identificador na tabela. Em caso afirmativo, indica condição de erro: *identificador já declarado*; em caso negativo, insere a tupla correspondente na tabela de símbolos.
USO: quando da compilação da declaração de variáveis e, em linguagens que incluem abstrações como modularização, classificação, etc., quando da compilação do corpo do programa.
- **alterar:** associa atributos a um identificador da tabela de símbolos. Para isso, verifica a

⁷ para variáveis: além do tipo (categoria) e do endereço (atributo), precisão e tamanho; para parâmetros formais: tipo, mecanismo de passagem; para procedimentos: número de parâmetros, etc.

existência do identificador na tabela. Em caso afirmativo, efetua as alterações pertinentes, em caso negativo, indica condição de erro: *identificador não declarado*.

USO: quando na compilação da declaração de variáveis e, em linguagens que incluem abstrações como modularização, classificação, etc., quando da compilação do corpo do programa.

- **recuperar:** recupera atributos associados a um identificador da tabela de símbolos. Para isso, verifica a existência do identificador na tabela. Em caso afirmativo, recupera as informações correspondentes, em caso negativo, indica condição de erro: *identificador não declarado*.
USO: quando da compilação do corpo do programa.

Para implementar a tabela de símbolos pode-se usar estruturas de dados dos mais variados tipos: lista ordenada, árvores binárias de busca, dicionário, tabela *hash*, etc.

6.5 AÇÕES SEMÂNTICAS E GERAÇÃO DE CÓDIGO: DESCRIÇÃO GERAL

Nos compiladores usuais, compiladores dirigidos pela sintaxe, o analisador sintático ativa a execução das ações semânticas sempre que forem atingidos determinados estados do reconhecimento ou sempre que determinadas transições ocorrerem durante a análise do programa-fonte. Deve-se, então, incluir na gramática símbolos que indicam as **ações semânticas**. A chamada de rotinas semânticas via (símbolos de) ações semânticas 'trabalha' particularmente bem em conjunção com um analisador sintático LL. As ações semânticas são tratadas como outros símbolos da gramática e colocados na pilha sintática juntamente com a regra de produção a qual está associada. O algoritmo de análise sintática deverá ser alterado para que, ao detectar um símbolo de ação semântica no topo da pilha sintática, seja invocada a rotina correspondente (de geração de código ou de verificação). Após a execução da ação, o símbolo deve ser retirado da pilha e a análise deve continuar. Assim, tem-se que a tradução para o código intermediário é controlada pela sintaxe.

Leitura Complementar: AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores:** princípios, técnicas e ferramentas. Rio de Janeiro/RJ: LTC, 1995. (capítulos 5-6: p.120-166; capítulo 8: p.200-221); FISCHER, C.N.; LEBLANC, R.J. **Crafting a compiler.** Menlo Park: The Benjamin/Cummings, 1998 (capítulos 7-8: p.215-286).

BIBLIOGRAFIA

- AHO, A.V.; SETHI, R.; ULLMAN, J.D. **Compiladores: princípios, técnicas e ferramentas**. Rio de Janeiro/RJ: LTC, 1985. 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.
- PRICE, A.M.A.; EDELWEISS, N. **Introdução às linguagens formais**. Porto Alegre/RS: II-UFRGS, 1989. 60 p.
- MENEZES, P. F. B. **Linguagens Formais e Autômatos**. Porto Alegre/RS: II-UFRGS, Sagra Luzzatto, 1997. 168 p.
- WAZLAWICK, R.S. **Linguagens formais e compiladores**: notas de aula. Florianópolis/SC: INE-UFSC, 1992. 83 p.
- WILHELM, R.; MAURER, D. **Compiler design**. Harlow: Addison-Wesley, 1995. 606 p.
- ZILLER, R.M. **Aplicação de autômatos finitos**. In: SEMANA TECNOLÓGICA, 1., Florianópolis, 1997. Anais. Biguaçu, UNIVALI, 1997. p.51-71.