Rafael de Santiago

Anotações para a Disciplina de Introdução a Compiladores

Versão de 24 de maio de 2022

Universidade Federal de Santa Catarina

Sumário

CAPÍTULO 1

Introdução

1.1 Revisão

Conjuntos é uma coleção de elementos sem repetição em que a sequência não importa. No Brasil, utilizamos a seguinte notação para enumerar todos os elementos de um conjunto. Na Equação (1.1), é possível visualizar a representação de um conjunto denominado A, formado pelos elementos e_1, e_2, ..., e_n. Devido ao uso da vírgula como separador de decimais, usa-se formalmente o ponto-e-vírgula. Para essa disciplina, podemos utilizar a vírgula como o separador de elementos em um conjunto, desde que utilizados o ponto como separador de decimais[1]. Para dar nome a um conjunto, geralmente utiliza-se uma letra maiúscula ou uma palavra com a inicial em maiúscula.

$$A = \{e_1; e_2; \ldots; e_n\} \tag{1.1}$$

Há duas formas de definir conjuntos. A forma por enumeração por elementos, utiliza notação semelhante a da Equação (1.1). São exemplos de definição de conjuntos por enumeração:

- $N = \{\diamondsuit, \spadesuit, \heartsuit, \clubsuit\}$;

[1] Nas anotações presentes nesse documento, utiliza-se a "notação americana". Para a Equação (1.1) , teria-se $A = \{e_1, e_2, \ldots, e_n\}$.

- $V = \{a, e, i, o, u\}$;
- $G = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta, \theta, \iota, \kappa, \lambda, \mu, \nu, \xi, \pi, \rho, \sigma, \tau, \upsilon, \phi, \chi, \psi, \omega\}$;
- $R = \{-100.9, 12.432, 15.0\}$;
- $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

A forma por descrição de propriedades utiliza-se de uma notação que evidencia a natureza de cada elemento pela descrição de um em um formato genérico. Por exemplo o conjunto D, descrito na Equação (1.2), denota um conjunto com os mesmos elementos em $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

$$D = \{x \in \mathbb{Z} | x \geq 1 \wedge x \leq 10\} \tag{1.2}$$

. Então para que complicar utilizando uma notação não enumerativa? Por dois motivos: por questões de simplicidade, dado a quantidade de elementos; ou para representar conjuntos infinitos, como no exemplo dos inteiros pares $Pares = \{x \in \mathbb{Z} | x \equiv 0(\mod 2)\}$. Para o conjunto dos pares, ainda podemos utilizar uma descrição mais informal, mas que é dependente da conhecimento sobre a linguagem Portuguesa: $Pares = \{x \in \mathbb{Z} |$ *x é inteiro e par* $\}$.

Para denotar a cardinalidade (quantidade de elementos) de um conjunto, utilizamos o símbolo "|". Para os conjuntos apresentados acima, é correto afirmar que:

- $|N| = 4$;
- $|V| = 5$;
- $|R| = 3$;
- $|D| = 10$;
- $|Pares| = \infty$.

A cardinalidade pode ser utilizada para identificar quantos símbolos são necessários para representar um elemento. Por exemplo, $|12, 66| = 5$

Para denotar conjuntos vazios, adota-se duas formas de representação: {} ou $\emptyset$. Utilizando o operador de cardinalidade, têm-se $|\{\}| = |\emptyset| = 0$.

Como principais operações entre conjuntos, pode-se destacar:

- União ($\cup$): união de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cup \{2,4,6,8\} = \{1,2,3,4,5,6,8\}$;
- Intersecção ($\cap$): intersecção de dois conjuntos. Exemplo: $\{1,2,3,4,5\} \cap \{2,4,6,8\} = \{2,4\}$;
- Diferença ($-$ ou $\backslash$): diferença de dois conjuntos. Exemplo $\{1,2,3,4,5\} \backslash \{2,4,6,8\} = \{1,3,5\}$;
- Produto cartesiano ($\times$): Exemplo $\{1,2,3\} \times \{A,B\} = \{(1,A),(2,A),(3,A),(1,B),(2,B),(3,B)\}$;
- Conjunto de partes (ou *power set*): o conjunto de todos os subconjuntos dos elementos de um conjunto. Para o conjunto $A = \{1,2,3\}$ o conjunto das partes seria $2^A = P(A) = \{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$.

Considerando que pode-se formar elementos a partir de um conjunto de elementos, tem-se o fecho de Kleene. Dado um conjunto A, o fecho de Kleene é denotado por A^* e é um conjunto de elementos formados pela concatenação sucessiva dos elementos de A. Considerando o conjunto $B = \{0,1\}$, $B^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$, onde ϵ é um elemento vazio, ou seja, que não utiliza símbolos para sua representação[2]. Além da notação do fecho de Kleene, é comum o uso da soma de Kleene, onde representa-se para o conjunto A através de A^+. Utilizando o conjunto B como exemplo, a soma de Kleene de B seria $B^+ = \{0, 1, 00, 01, 10, 11, 000, \ldots\}$, ou seja, o fecho de Kleene sem o elemento vazio ϵ.

Concatenações sucessivas são semelhantes aos fechos de Kleen, pois formam elementos, mas se estabelece o comprimento de cada novo elemento: $\{0,1\}^2 = \{00, 01, 10, 11\}$.

[2] Acredite, isso será muito útil num futuro breve

Outra forma de concatenação de símbolos pode ser visualizada ao definir dois elementos. Para exemplificar, serão utilizados os elementos $x = 1$ e $y = 2$. Desse modo, $xy = 12$, ou seja, x concatenado a y gera o elemento 12.

Funções são representadas de forma diferente na matemática discreta. Busca-se estabelecer a relação entre um conjunto de domínio (entrada da função) e um contradomínio (resposta da função). A Equação (1.3) exibe a forma como é utilizada para formalizar uma função. Nesse formato, passa-se a natureza da entrada e da saída de um problema. Por exemplo, a função que gera a correspondência entre o domínio dos inteiros positivos em base decimal para base binária seria $f : x \in Z^+ \rightarrow \{0,1\}^{\log_2(|x|+1)}$.

$$\text{nome da funcao} : \text{dominio} \rightarrow \text{contradominio} \tag{1.3}$$

Para representar uma coleção de itens onde a sequência importa e a repetição pode ocorrer, utiliza-se as tuplas. Uma tupla é representada da forma demonstrada na Equação (1.4).

$$A = (e_1, e_2, \ldots, e_n) \tag{1.4}$$

. Um exemplo de uma tupla, pode ser lista de chamada de uma turma ordenada lexicograficamente.

1.2 Linguagens para Teoria da Computação

A Teoria da Computação fornece importante base para a construção de compiladores. De acordo com Sipser (2007), ela se divide em três outras importantes teorias:

- Teoria da computabilidade: estudo das limitações de quais problemas podem ser resolvidos e os que não podem;
- Teoria da complexidade: estuda a dificuldade em resolver problemas;
- Teoria dos autômatos e linguagens: “lida com definições e propriedades dos modelos de computação”.

No contexto de compiladores, a parte da Teoria da Computação mais básica é a "teoria dos autômatos e linguagens".

A teoria dos autômatos e linguagens fornece um ferramental muito importante no reconhecimento de padrões. Ela define máquinas e gramáticas, que dão a base necessária para o reconhecimento de linguagens de programação.

No entanto, existem algumas definições importantes da Teoria da Computação, que devem fazer parte do cotidiano da disciplina. Veremos algumas dessas definições a seguir.

Para Teoria da Computação, toda linguagem deve ter um **alfabeto**. Esse alfabeto é um conjunto finito de símbolos que é utilizado para representar todos os símbolos que podem ser utilizados na linguagem. Imagine uma linguagem escrita apenas com zeros e uns. Qual seria o conjunto alfabeto dessa linguagem? O alfabeto seria $\Sigma = \{0, 1\}$. Para uma linguagem de programação, o alfabeto teria que ser todos os símbolos possíveis na linguagem, incluindo espaço, tabulações, quebras de linha, símbolos de operação e caracteres acentuados.

O alfabeto é utilizado para construir palavras[3] que pertençam a sua linguagem. Desse modo, não podem ser criadas palavras com símbolo que não pertence ao alfabeto da linguagem. Por exemplo, a linguagem dos números inteiros de base binária L_1 possui o alfabeto $\Sigma = \{+, -, 0, 1\}$. Logo, a palavra *-1009* não pertence a essa linguagem. Outra observação importante é que nem todas as palavras que possuem apenas símbolos do alfabeto pertencem a uma linguagem. Como exemplo, imagina-se a linguagem de todas os números binários pares L_2. Essa última linguagem possui o mesmo alfabeto de L_1, mas nem todas as palavras que pertencem a linguagem L_1 pertencem a L_2.

Agora que já se definiu alfabeto e palavra, pode-se definir o que é uma linguagem para Teoria da Computação. Nesse contexto, uma linguagem é um conjunto de palavras. Então as linguagens citadas anteriormente poderiam ser definidas da seguinte forma: $L_1 = \{\ldots, -11, -10, -1, 0, +1, 1, 10, +10, 11, +11, \ldots\}$ e $L_2 = \{\ldots, -100, -10, 0, 10, +10, 100,$

[3] Nesse contexto, palavras, sentenças e cadeias possuem o mesmo significado.

$+100, \ldots\}$.

É comum que as linguagens sejam conjuntos infinitos. Então é mais comum utilizarmos a forma de definição de conjuntos que descreve as propriedades do mesmo. Por exemplo, $L_1 = \{uv | u \in \{\epsilon, -, +\} \wedge v \in \{0,1\}^+\}$ e $L_2 = \{uv0 | u \in \{\epsilon, -, +\} \wedge v \in \{0,1\}^*\}$.

Na Figura 1, pode-se observar um diagrama de Venn com a hierarquia de linguagens, conhecida como Hierarquia de Chomsky. Nessa hierarquia, pode-se observar que todas as linguagens regulares são também livres de contexto. A hierarquia ajuda a identificar a complexidade da linguagem e qual mecanismo deve ser utilizado para realizar a descoberta de cadeias para as mesmas. Quanto mais abrangente o tipo da linguagem na hierarquia, mais difícil é identificar seus padrões. Para o contexto dessa disciplina, as linguagens que mais nos interessam não as regulares e as livres de contexto.

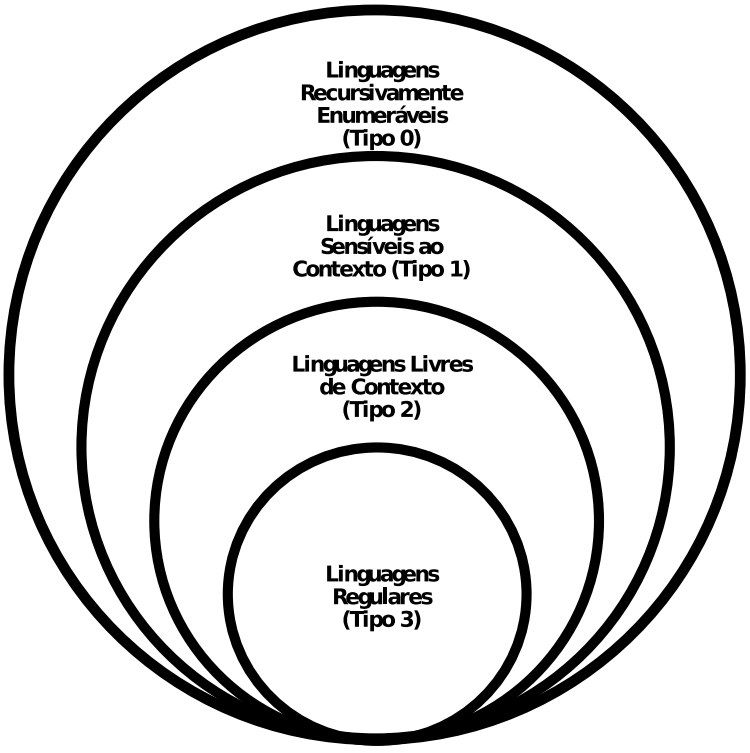


Figura 1 – Hierarquia de Chomsky.

Desafio

O que é um programa para uma linguagem de programação?

( ) uma linguagem

( ) um alfabeto

( ) uma cadeia

( ) uma tupla

( ) um conjunto

1.3 Linguagens de Programação

De acordo com Aho et al. (2008), "linguagens de programação são notações para se descrever computações para pessoas e para máquinas". Todo o software necessita ser escrito em uma linguagem de programação. No entanto, para que possa ser executado, um programa deve ser traduzido no formato que possa ser executado por um processador de um computador. Chamamos os sistemas que fazem essa tradução de *Compiladores* (AHO et al., 2008).

Aho et al. (2008) coloca que podemos definir diferentes formas de traduzir linguagens. As destacadas são:

- Compilador[4]: programa que recebe um programa como entrada (escrito em uma linguagem de programação – linguagem *fonte*) e o traduz para um programa equivalente em outra linguagem (linguagem *objeto*). Relata erros detectados no programa de entrada. Se o programa objeto for um programa em linguagem de máquina, poderá ser executado por um usuário (vide Figura 2(a));

[4] Um programa em código de máquina gerado por um compilador tende a executar mais rapidamente.

- Interpretador: programa que recebe um programa como entrada e executa suas operações diretamente (vide Figura 2(b));
- Compilador híbrido: combina compilação e interpretação. Gera um programa escrito em código intermediário que são interpretados e executados por uma máquina virtual (vide Figura 2(c)).

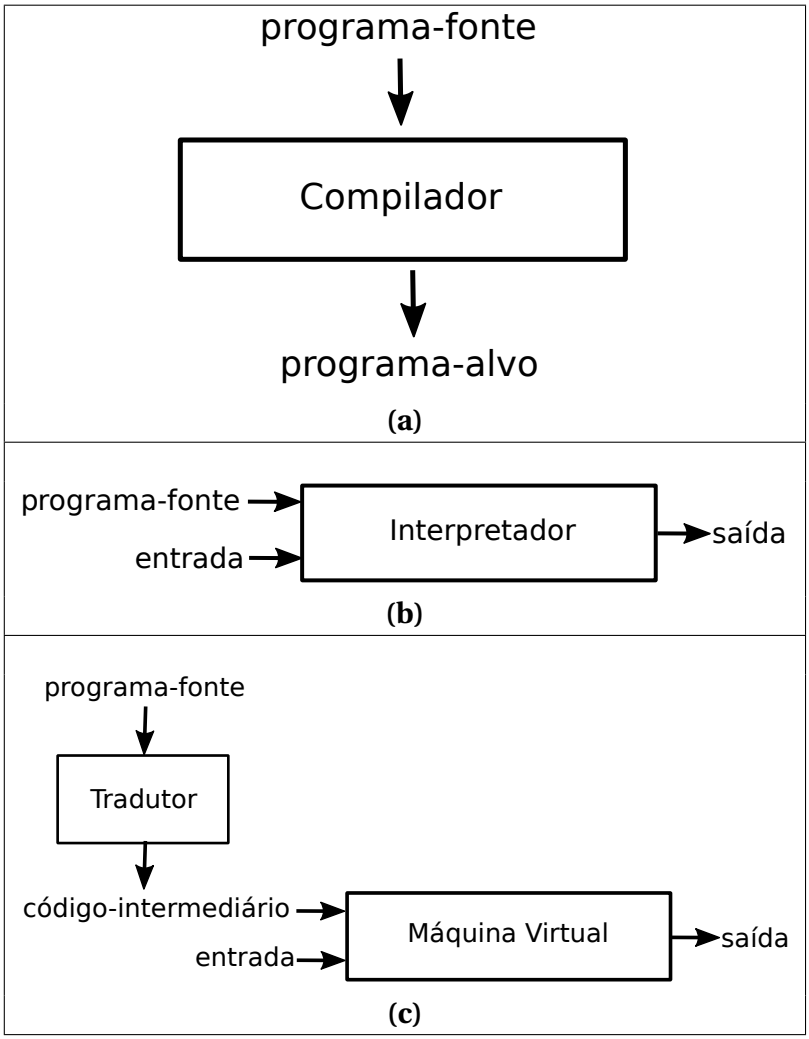


Figura 2 – Diferentes formas de traduzir linguagens. Adaptado de Aho et al. (2008).

Além de um compilador, vários outros programas podem ser necessários para criação de um programa em código-objeto (AHO et al., 2008) (vide Figura 3):

1. Um programa fonte pode ser organizado em arquivos separados (bibliotecas). A tarefa de unir o programa fonte pelos múltiplos arquivos é realizada por um

pré-processador;

2. Logo depois a etapa de pré-processamento, o compilador recebe como entrada o programa fonte modificado e pode produzir como saída uma linguagem simbólica chamada de *assembly* (fácil de gerar saída e fácil de depurar);

3. A linguagem simbólica é processada pelo *montador*, que é um programa que produz código de máquina relocável;

4. Esse código de máquina relocável pode ter de se ligar a outros arquivos objeto relocáveis e a arquivos de biblioteca para formar o código que realmente é executado na máquina. O editor de ligação (*linker*) resolve os endereços de memória externos, e o carregador (*loader*) reúne todos os arquivos objeto executáveis na memória para execução.

A estrutura de um compilador pode ser dividida em duas partes:

- Análise: divide o programa em partes e impõe uma estrutura gramatical sobre elas. Depois, usa essa estrutura para criar uma representação intermediária do programa fonte. Se for detectado que o programa fonte está mal formado ou semanticamente incorreto, então mensagens precisam esclarecer os problemas para que o usuário faça a correção. Esta etapa coleta dados da chamada tabela de símbolos, que e passada adiante junto com a representação intermediária para a parte de síntese;

- Síntese: constrói o programa objeto desejado a partir do que fora produzido pela etapa de análise.

A parte de análise é chamada de *front-end* e a parte de síntese é chamada de *back-end*.

Um compilador também pode ser dividido em fases: Análise Léxica, Análise Sintática, Análise Semântica, Geração de Código Intermediário, Otimização de Código Dependente de Máquina, Geração de Código, Otimizador de Código Independente de máquina.

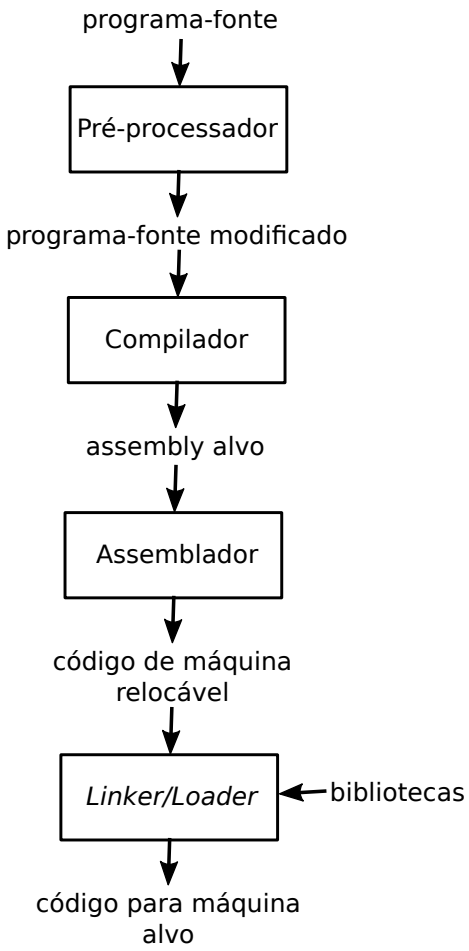


Figura 3 – Programas de um compilador. Adaptado de Aho et al. (2008).

Análise Léxica

A primeira fase de um compilador é chamada de Análise Léxica (ou leitura ou *scanning*). O analisador léxico lê o fluxo de caracteres que compõem o programa fonte e os agrupa em sequências chamadas de *lexemas*. Para cada *lexema*, o analisador gera um token

$$< nome-token, valor-atributo >, \tag{1.5}$$

cujo *nome-token* é um símbolo abstrato usado na análise sintática. O segundo componente é o *valor-atributo* que aponta para uma entrada na tabela de símbolos referente a esse token.

Suponha a sequinte operação de um código fonte

$$position = initial + rate * 60 \tag{1.6}$$

. Após a análise, terá-se a seguinte sequência de tokens

$$< id,1 >< = >< id,2 >< + >< id,3 >< * >< 60 > \tag{1.7}$$

Análise Sintática

A Análise Sintática utiliza os componentes dos tokens para criar uma representação intermediária no formato de uma árvore (estruturas de dados) que mapeia a estrutura gramatical dos tokens. Uma representação típica é a *árvore de sintaxe*, em que cada nodo interior representa uma operação e os filhos do nodo representam os argumentos da operação.

Análise Semântica

Um analisador semântico utiliza a árvore de sintaxe e a tabela de símbolos para verificar a consistência semântica de um programa. Adiciona informação de tipos na árvore sintático abstrata ou na tabela de símbolos para geração de código intermediário. Uma das etapas mais importantes deste processo é a verificação de tipos, processo pelo qual o compilador verifica se o operador possui operandos compatíveis e gera erros caso seja necessário. Conversões também são realizadas nessa fase para permitir que algumas conversões implícitas da linguagem sejam realizadas (atribuir um inteiro a uma variável contínua).

Geração de Código Intermediário

Muitos compiladores geram uma representação intermediária de baixo nível.

Otimização de Código

A etapa de otimização de código realiza modificações no código intermediário independente de máquina, geralmente para torná-lo mais rápido, mas pode ser ajustado a outros objetivos como um consumo menor de energia.

Geração de Código

O gerador de código recebe como entrada a representação intermediária do programa fonte e o mapeia em uma linguagem objeto.

1.3.1 Ferramentas para Construção de Compiladores

Existem ferramentas que podem auxiliar no processo de criação de um compilador (AHO et al., 2008):

- Geradores de analisadores sintáticos: produzem reconhecedores sintáticos a partir de descrição em uma Gramática Livre de Contexto;

- Geradores de analisadores léxicos: utilizando da descrição em Expressão Regular, produzem analisadores léxicos;

- Mecanismos de tradução dirigida por sintaxe: produzem uma coleções de rotinas para percorrer uma árvore de derivação e gerar código intermediário;

- Geradores de gerador de código: produzem um gerador de código a partir de uma coleção de regras para traduzir para operação da linguagem intermediária na linguagem de máquina para uma máquina alvo;

- Mecanismos de análise de fluxo de dados: “facilitam a coleta de informações sobre como os valores são transmitidos de uma parte do programa para cada uma das outras partes. A análise de fluxo de dados é uma ferramenta essencial para a otimização de código”;

- Conjuntos de ferramentas para a construção de compiladores: oferecem um conjunto integrado de rotinas para a construção de diversas fases de um compilador.

1.4 Lista de Exercícios

1.4.1 Lista de Exercícios sobre Linguagens

Defina os conjuntos para as seguintes linguagens abaixo:

1) Linguagem de todos os números inteiros positivos.

2) Linguagem de todas os números de um a 10.

3) Linguagem de todos os números binários múltiploes de 4.

4) Linguagem de todos de todas as palavras formadas por a's e b's em mesma quantidade.

5) Linguagem de todos de todas as palavras compostas por a's, b's, c's e d's em mesma quantidade e na sequência de a's antes de b's antes de c's antes de d's.

1.4.2 Lista de Exercícios sobre Introdução à Compiladores

1) Diferencie um Compilador de um Interpretador.

2) Cite ao menos dois programas que podem ser utilizados para auxiliar um compilador.

3) Explique as partes de Análise e Síntese de um compilador. Como elas também são chamadas?

4) Cite as fases de um compilador e explique três delas.

CAPÍTULO 2

Análise Léxica

O papel de um analisador léxico é o de ler caracteres de um programa fonte, agrupá-los em lexemas e produzir uma sequência de tokens para cada lexema. O fluxo de tokens é passado para o analisador sintático. O analisador léxico interage com a tabela de símbolos, como quando encontrar identificadores por exemplo. Um analisador léxico também deve ignorar caracteres não imprimíveis quando necessário. Além disso, o analisador léxico é responsável por armazenar número de quebras de linhas e tabulações quando for algo importante para a linguagem do programa fonte. Mensagens de erro léxico devem ser controladas também por este tipo de analisador (AHO et al., 2008).

Aho et al. (2008) divide o processo desempenhado por um analisador léxico em dois:

1. "Escandimento": varredura da entrada sem preocupar-se com a remoção de comentários e a compactação de caracteres com espaço em branco consecutivos;
2. Análise léxica: produz a sequência de tokens.

Importantes termos que são repetidamente mencionados nesse capítulo (AHO et al., 2008):

- Token: é um par constituído de um nome e um valor de atributo opcional. O nome é um símbolo abstrato que representa a unidade léxica (exemplo: identificador,

número, palavras-reservadas, ...).

- Padrão: formato que os lexemas de um token podem assumir.

- Lexema: sequência de caracteres que está de acordo com o padrão determinado para um token.

Exemplo de tokens e lexemas de uma linguagem de programação semelhante ao C e ao Java.

Token	Descrição	Exemplo de Lexemas
if	caracteres i, f	`if`
else	caracteres e, l, s, e	`else`
comparison	<, >, ≤, ≥, == ou ! =	<, >, ≤, ≥, == ou ! =
id	letra seguida de letras ou dígitos	`pi`, `idade`, `telefone1`
number	valor constante numérico	`3.14159`, `7`, `6.02e23`
literal	sequência de símbolos cercados “”	“Hello World”

A Figura 4 ilustra a interação entre o Analisador Léxico e o Analisador Sintático. Geralmente, o Analisador Sintático é quem invoca o Analisador Léxico para reconhecer cada token (chamada *getNextToken*). A Tabela de Símbolos tem um papel fundamental nesse processo. Ela quem deverá armazenar detalhes de identificadores contidos no programa-fonte para posterior uso.

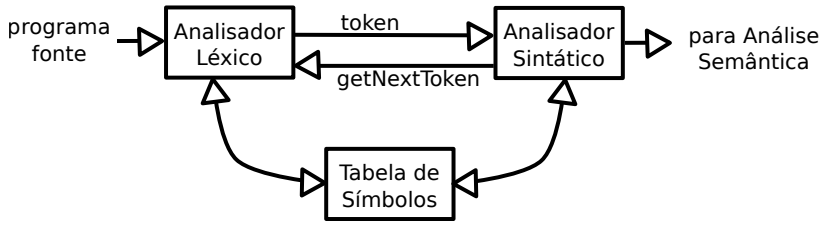


Figura 4 – Interação entre Analisador Léxico e Sintático. Adaptado de Aho et al. (2008).

Tabela de Símbolos

A tabela de símbolos armazena o nome de cada elemento utilizado no programa, como classes, variáveis, métodos, parâmetros entre outros. Por exemplo, ao definir uma classe X, a tabela de símbolos identifica X como uma classe. Quando se cria um objeto da classe X, a tabela auxilia na identificação dos detalhes do objeto (PRICE; TOSCANI, 2001; DELAMARO, 2004).

A tabela de símbolos começa a ser construída na Análise Léxica no momento que os identificadores são reconhecidos. Na primeira vez que o identificador é conhecido, ele é armazenado na tabela. No entanto, a análise léxica não tem como identificar o atributos ou outros detalhes desse identificador. Nesse caso, complementa-se o que for armazenado na tabela na Análise Sintática (PRICE; TOSCANI, 2001).

De acordo com Price e Toscani (2001) uma tabela de símbolos é geralmente implementada através de listas lineares, árvores binárias e tabelas *hash*. Normalmente, são utilizadas tabelas *hash* devido ao melhor desempenho (PRICE; TOSCANI, 2001; AHO et al., 2008).

2.1 Linguagem Regular

A Análise Léxica está relacionada ao reconhecimento de padrões muito simples. Essa simplicidade permite que cada token pode ser definido por uma linguagem que pertence ao conjunto de Linguagens Regulares.

Uma linguagem Regular é aquela em que todas as suas palavras podem ser reconhecidas por um Autômato Finito (AF) ou geradas por uma Gramática Regular (GR). Nesse texto, apenas se dá ênfase aos AFs, por questões de contexto.

2.1.1 Autômatos Finitos

Todo AF M é uma 5-upla $M = (Q, \Sigma, \delta, q_0, F)$, a qual (SIPSER, 2007):

- Q é o conjunto de estados;
- Σ é o conjunto de símbolos do alfabeto;
- $\delta : Q \times \Sigma \rightarrow Q$ é a função de transição[1];
- q_0 é o estado inicial;
- F é o conjunto de estados finais.

Além de suas definições, todo o AF possui:

- Uma fita de entrada, onde são colocados os símbolos da palavra a ser verificada;
- Uma cabeça de leitura, inicialmente posicionada no símbolo mais à esquerda;
- Uma memória que armazena o estado atual do reconhecimento, inicialmente no estado q_0.

O processo de reconhecimento inicia considerando que a palavra de entrada está na fita. A fita divide cada símbolo numa sequência de células justapostas. Considera-se também que a cabeça de leitura esteja na célula mais à esquerda e o estado atual seja q_0. Iterativamente, consulta-se o símbolo que está na celula atualmente visitada pela cabeça de leitura e o estado atual, verificando se há alguma transição possível. Se há, a(s) transição(ões) é(são) feita(s) e move-se a cabeça de leitura para a próxima posição da fita. Além disso, o estado atual é atualizado de acordo com a(s) transição(ões)[2]. Quando nãop houver mais símbolos na fita, consulta-se o estado atual. Se o estado atual for final (percente a F), aceita-se a palavra de entrada, caso contrário, rejeita-se.

2.1.1.1 Representação de Autômatos

Autômatos podem ser representados em duas outras formas, além de sua definição formal. Um AF pode ser descrito na notação em grafos ou por uma tabela de transição.

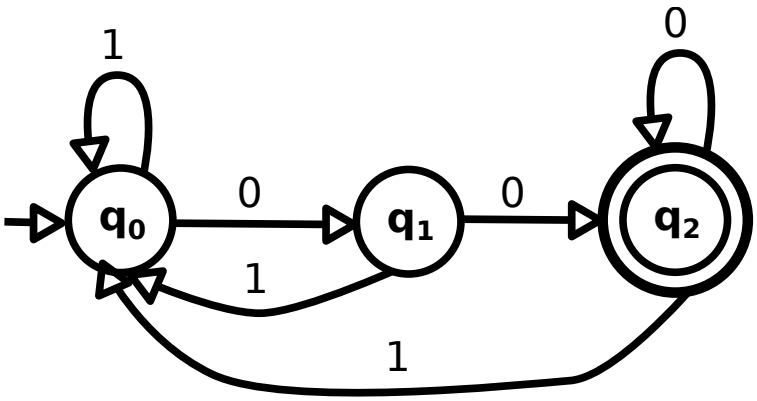
[1] Essa definição é estendida para Autômatos Finitos Não-Determinísticos na Seção 2.1.1.2.

[2] Para o caso de mais de uma transição possível, passasse a ter mais de uma fita e mais de uma memória de estado atual.

Na notação em grafos, representa-se os autômatos através de um grafo orientado rotulado, no qual cada estado é um vértice, e os arcos representa as transições de um estado de origem para o destino. Os rótulos sobre cada arco correspondem ao símbolo do alfabeto requerido para passar do estado de origem para o estado destino. O estado inicial é simbolizado como um vértice com uma flecha vindo do "vazio" para ele, e os estados finais possuem vértices simbolizados visualmente com uma borda dupla ou destacada.

Exemplo de um AF na Notação Gráfica

Considere o AF $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, no qual $\delta\big((q_0,0)\big) \rightarrow q_1$, $\delta\big((q_0,1)\big) \rightarrow q_0$, $\delta\big((q_1,0)\big) \rightarrow q_2$, $\delta\big((q_1,1)\big) \rightarrow q_0$, $\delta\big((q_2,0)\big) \rightarrow q_2$ e $\delta\big((q_2,1)\big) \rightarrow q_0$. Sua representação gráfica pode ser visualizada abaixo



.

Na notação em tabela de transição, cada linha representa um estado e cada coluna representa um símbolo do alfabeto. as celulas representa o(s) estado(s) destino(s) estando no estado da respectiva linha e sobre o símbolo correspondente a respectiva coluna. O estado inicial recebe uma flecha ($\rightarrow$) a sua esquerda, e cada estado final recebe um $*$ a sua esqueda.

Exemplo de um AF na Notação por Tabela de Transição

Considere o AF $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, no qual $\delta\big((q_0,0)\big) \rightarrow q_1$, $\delta\big((q_0,1)\big) \rightarrow q_0$, $\delta\big((q_1,0)\big) \rightarrow q_2$, $\delta\big((q_1,1)\big) \rightarrow q_0$, $\delta\big((q_2,0)\big) \rightarrow q_2$ e $\delta\big((q_2,1)\big) \rightarrow q_0$. A notação em tabela de transição segue o seguinte formato

δ	**0**	**1**
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_2\}$	$\{q_0\}$
$^*q_2$	$\{q_2\}$	$\{q_0\}$

.

2.1.1.2 Autômatos Finitos Determinísticos

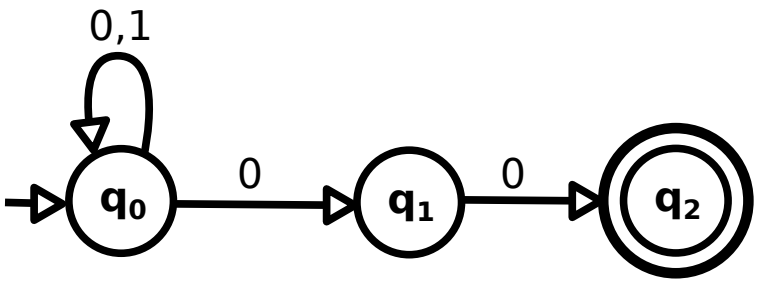
Os Autômatos Finitos Determinísticos (AFD) são aqueles em que para cada símbolo $\alpha \in \Sigma$ a função de transição só possui um contradomínio para cada estado de origem $\delta : (q \in Q, \alpha) \rightarrow p \in Q \cup \{\emptyset\}$. Ou seja, existe apenas uma transição possível a partir de um estado de origem para cada símbolo α.

2.1.1.3 Autômatos Finitos Não-Determinísticos

Os Autômatos Finitos Não-Determinísticos (AFND ou AFN) permitem vários caminhos possíveis para uma transição a partir de um estado e um símbolo específico. Logo, a função de transição passa a ser $\delta : Q \times \Sigma \rightarrow 2^Q$.

Exemplo de um AFND

Considere o AFND $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, no qual $\delta\big((q_0,0)\big) \rightarrow \{q_0, q_1\}$, $\delta\big((q_0,1)\big) \rightarrow \{q_0\}$, $\delta\big((q_1,0)\big) \rightarrow \{q_2\}$, $\delta\big((q_1,1)\big) \rightarrow \{\}$, $\delta\big((q_2,0)\big) \rightarrow \{\}$ e $\delta\big((q_2,1)\big) \rightarrow \{\}$. Sua representação gráfica pode ser visualizada abaixo



.

2.1.2 Determinização

Determinização é o processo de tornar um AFND em um AFD. O Algoritmo 1 descreve o processo de determinização de um AFND (sem transições em ϵ) para um AFD. Esse processo é utilizado para demonstrar a equivalência entre AFNDs e AFDs, ou seja, os

dois resolvem as mesmas linguagens.

Algoritmo 1: Algoritmo de determinização.

Input : um AFND $M = (Q, \Sigma, \delta, q_0, F)$

$F' \leftarrow \{\}$

$Q' \leftarrow \{\{q_0\}\}$

$S \leftarrow Fila()$

$S.queue(\{q_0\})$

while $S.empty() =$ ***false*** **do**

  $q \leftarrow S.dequeue()$

  foreach $\alpha \in \Sigma$ **do**

    $r \leftarrow \bigcup_{p \in q} \delta\big((p, \alpha)\big)$

    define $\delta'\big((q, \alpha)\big) \rightarrow r$

    if $r \neq \{\}$ ***and*** $r \notin Q'$ **then**

      $Q' \leftarrow Q' \cup \{r\}$

      $S.queue(r)$

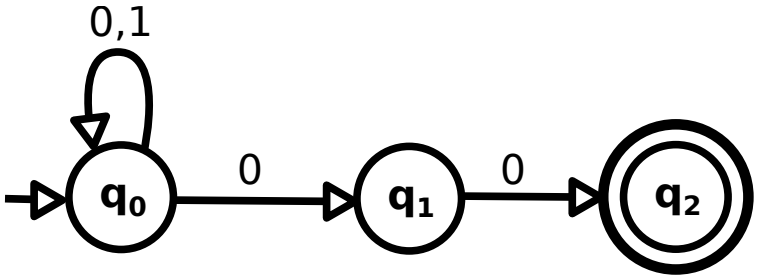
    if $\exists p \in q : p \in F$ **then**

      $F' \leftarrow F' \cup \{q\}$

return $(Q', \Sigma, \delta', \{q_0\}, F')$

Exemplo de determinização utilizando uma tabela de transição

Considere o AFND $M = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_2\})$, no qual $\delta\big((q_0,0)\big) \rightarrow \{q_0, q_1\}$, $\delta\big((q_0,1)\big) \rightarrow \{q_0\}$, $\delta\big((q_1,0)\big) \rightarrow \{q_2\}$, $\delta\big((q_1,1)\big) \rightarrow \{\}$, $\delta\big((q_2,0)\big) \rightarrow \{\}$ e $\delta\big((q_2,1)\big) \rightarrow \{\}$.

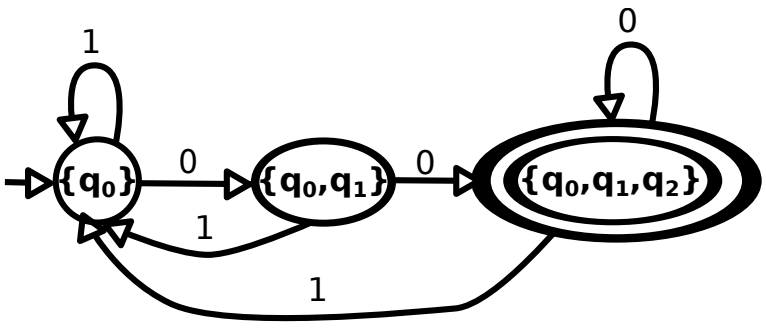


.

A sua determinização pode ser feita iterativamente em uma tabela de transição vazia. Inicia-se com o estado q_0 na tabela e verifica-se para cada símbolo do alfabeto qual a transição possível. Todo estado novo que aparecer (considera-se um conjunto de estados como um estado apenas). Sempre preencher a tabela de transição com todos os destinos definidos pela função de transição do AFND. Abaixo tem uma determinização em tabela de transição para o AFND M.

δ	**0**	**1**
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$
$^*\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_0\}$

. Um exemplo do autômato resultante pode ser visto abaixo



.

2.1.2.1 Determinização AFND-ϵ

A determinização de autômatos AFND-ϵ (não-determinísticos com ϵ-transições) pode ser realizada através do algoritmo sugerido por (AHO et al., 2008). É o procedimento utilizado para provar que todo AFND-ϵ possui um AFD equivalente (que aceita a mesma linguagem). Para compreendê-lo melhor, deve-se antes definir as seguintes funções:

- fecho-$\epsilon(q \in Q)$: conjunto de estados que podem ser atingidos por ϵ-transições a partir de q, incluindo-o;
- fecho-$\epsilon(T \subseteq Q)$: $\bigcup_{p \in T}$fecho-$\epsilon(p)$;
- move($T \subseteq Q$, $\alpha \in \Sigma$): conjunto de estados que são destino de uma transição a partir de $p \in T$ com o símbolo α.

O Algoritmo 2 detalha o procedimento.

Algoritmo 2: Algoritmo de determinização para AFND-ϵ.

Input : um AFND-ϵ $M = (Q, \Sigma, \delta, q_0, F)$

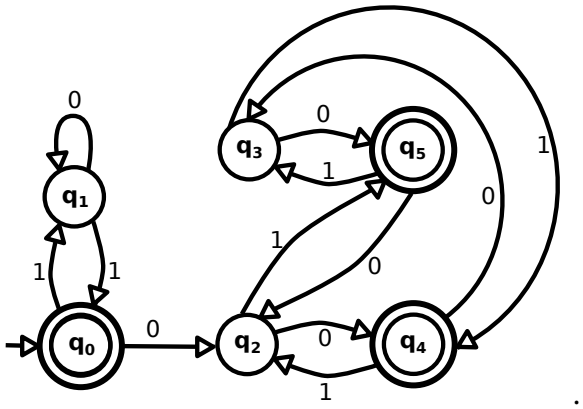
$Q' \leftarrow \{$ fecho-$\epsilon(q_0)$ $\}$
marcados $\leftarrow \{\}$
$q'_0 \leftarrow$ fecho-$\epsilon(q_0)$
while $Q' \neq$ *marcados* **do**
    $T \leftarrow$ selecionar um estado em $Q' \setminus$ marcados
    marcados $\leftarrow$ marcados $\cup\ T$
    foreach $\alpha \in \Sigma$ **do**
        $U \leftarrow$ fecho-ϵ(move(T, α))
        if $U \notin Q'$ **then**
            $Q' \leftarrow Q' \cup \{U\}$
        define $\delta'(T, \alpha) \rightarrow U$
$F' \leftarrow \{\}$
foreach $T \in Q'$ **do**
    foreach $q \in T$ **do**
        if $q \in F$ **then**
            $F' \leftarrow F' \cup \{T\}$
return $(Q', \Sigma, \delta', q'_0, F')$

2.1.3 Minimização

Um autômato finito mínimo é um AFD, sem estados inalcançáveis, sem estados mortos, onde não há estados equivalentes. Um subconjunto de estados $S \subseteq Q$ é equivalente sse para todo estado $q_i \in S$ e todo $\alpha \in \Sigma$, $\delta(q_1, \alpha) = p_1, \delta(q_1, \alpha) = p_2, \ldots, \delta(q_{|S|}, \alpha) = p_{|S|}$ no qual $p_1, p_2, p_{|S|}$ são equivalentes. O procedimento de minimização descrito acima, pode ser visualizado formalmente no Algoritmo **??**.

Exemplo de minimização

Considere o AFND $M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{0,1\}, \delta, q_0, \{q_0, q_4, q_5\})$, no qual $\delta\big((q_0,0)\big) \to q_2$, $\delta\big((q_0,1)\big) \to q_1$, $\delta\big((q_1,0)\big) \to q_1$, $\delta\big((q_1,1)\big) \to q_0$, $\delta\big((q_2,0)\big) \to q_4$, $\delta\big((q_2,1)\big) \to q_5$, $\delta\big((q_3,0)\big) \to q_5$, $\delta\big((q_3,1)\big) \to q_4$, $\delta\big((q_4,0)\big) \to q_3$, $\delta\big((q_4,1)\big) \to q_2$, $\delta\big((q_5,0)\big) \to q_2$ e $\delta\big((q_5,1)\big) \to q_3$.



Inicia-se a minimização com os dois grupos que são "trivialmente não equivalentes". Então divide-se os estados entre finais e não-finais, obtendo-se dois conjuntos suspeitos de serem equivalentes:

$$\{q_1, q_2, q_3\}, \{q_0, q_4, q_5\}. \tag{2.1}$$

Analisando cada estado com os demais de seu grupo, deve-se ver se suas transições incidem para os mesmos conjuntos suspeitos de serem equivalentes. Obtem-se os seguintes novos conjuntos suspeitos de serem equivalentes:

$$\{q_1\}, \{q_2, q_3\}, \{q_0, q_4, q_5\}. \tag{2.2}$$

Em mais uma rodada, obtem-se:

$$\{q_1\}, \{q_2, q_3\}, \{q_0\}, \{q_4, q_5\}. \tag{2.3}$$

Depois dessa última divisão, não encontrou-se mais inequivalências. Logo, os estados q_2 e q_3 unidos, e os estados q_4 e q_5.

2.1.4 Expressões Regulares

Expressões Regulares (ER) podem ser utilizadas para descrever linguagens regulares. ER são representações construídas recursivamente a partir de expressões menores, usando regras bem definidas (AHO et al., 2008).

Cada expressão regular r denota uma linguagem Regular $L(r)$. Seguem algumas regras que definem as expressões regulares para algum alfabeto Σ (AHO et al., 2008):

BASE: Composta por duas regras:

1. ϵ é uma expressão regular e $L(\epsilon) = \{\epsilon\}$;

2. Se $a \in \Sigma$, então $L(a) = \{a\}$.

INDUÇÃO: Composta por quatro partes de indução, por meio das quais as expressões regulares são construídas a partir de partes menores (AHO et al., 2008):

1. $(r)|(s)$ é uma expressão regular que representa a linguagem $L(r) \cup L(s)$[3];

2. $(r)(s)$ é uma expressão regular denotando a linguagem $L(r)L(s)$ (concatenação);

3. $(r)^*$ é uma expressão regular representando a linguagem $\big(L(r)\big)^*$;

4. (r) é uma expressão regular representando a linguagem $L(r)$.

[3] Em algumas referências, usa-se o símbolo + no lugar de |

Exemplos de Expressões Regulares

São exemplos de expressões regulares para um alfabeto $\Sigma = \{0,1\}$:

- $(0|1)(0|1)$ representa a linguagem $\{00,01,10,11\}$;
- 0^* representa a linguagem $\{\epsilon,0,00,000,\ldots\}$;
- $(0|1)^*$ representa a linguagem $\{\epsilon,0,1,00,01,10,11,000,\ldots\}$;
- $(0|1)^*00$ representa a linguagem $\{00,000,100,0000,0100,1000,1100,\ldots\}$;
- $(01)^*$ representa a linguagem $\{\epsilon,01,0101,010101,\ldots\}$.

2.1.4.1 Extensão de Expressões Regulares

As ERs receberam diversas extensões desde suas definições na década de 50[4]. Seguem algumas dessas extensões que podem ser úteis a esta disciplina.

Caracteres:

- .: qualquer símbolo, exceto quebra de linha;
- $\backslash d$: um dígito (0 a 9);
- $\backslash w$: uma letra ou dígito;
- $\backslash s$: um espaço em branco;
- $\backslash D$: um símbolo que não seja um dígito;
- $\backslash W$: um símbolo que não é letra nem dígito;
- $\backslash S$: um símbolo que não é um espaço em branco;
- $\backslash r$: um retorno do carro;
- $\backslash t$: uma tabulação;

[4] Na década de 50, Kleene definiu as Expressões Regulares.

- \n: uma quebra de linha;
- \R: um símbolo que não é retorno do carro;
- \N: um símbolo que não é uma quebra de linha;
- caracteres especiais necessitam ser "escapados": \[, \{, \(, \), \}, \], \., *, \+, \?, \$, \^, \/ e \\.

Quantificadores:

- +: um ou mais;
- ?: nenhum ou um;
- {3}: três vezes;
- {3,}: três ou mais vezes;
- {3,4}: no mínimo três, no máximo quatro.

Classes de caracteres:

- [...]: um dos símbolos entre colchetes;
- [$x-y$]: um símbolo do intervalo entre x e y;
- [^...]: todos os símbolos, exceto os que estão entre os colchetes.

Limitadores:

- ^: início da palavra ou da linha;
- $: fim da palavra ou da linha.

2.1.5 Expressões Regulares para Autômatos Finitos

2.1.5.1 Expressões Regulares para AFND

A conversão de uma ER para um AFND deve inicialmente desmembrar as partes da expressão. As partes básicas são tratadas pela base do procedimento. As regras de indução são para AFNDs maiores (AHO et al., 2008).

Assuma que a princípio, deve-se gerar um AFND $N(r)$ para a expressão regular r. Considera-se que $N(r)$ é um autômato a definir para a expressão regular r. Encontre sempre a operação menos prioritária e aplique a base ou a indução para dividir $N(r)$. Repita o procedimento nas partes geradas até que não haja mais autômatos a definir.
BASE: Para uma expressão ϵ, se constrói o AFND $N(\epsilon)$ como demonstrado na Figura 5. Para um símbolo a do alfabeto Σ, se constrói o AFND $N(a)$ como demonstrado na

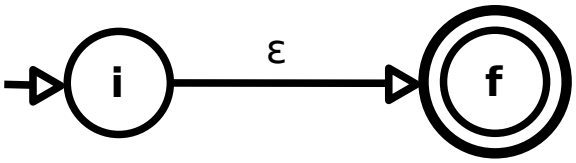


Figura 5 – ER para AFND passo base $r = \epsilon$.

Figura 6.

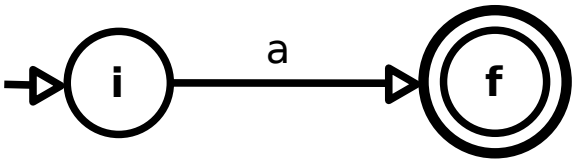


Figura 6 – ER para AFND passo base $r = a$, sendo que $a \in \Sigma$.

INDUÇÃO: Supondo que $N(s)$ e $N(t)$ são AFNDs para ER s e t respectivamente, têm-se uma das 3 opções abaixo:

- Sendo $r = s|t$, construir autômato da Figura 7(a);
- Sendo $r = st$, construir autômato da Figura 7(b);
- Sendo $r = s^*$, construir autômato da Figura 7(c);

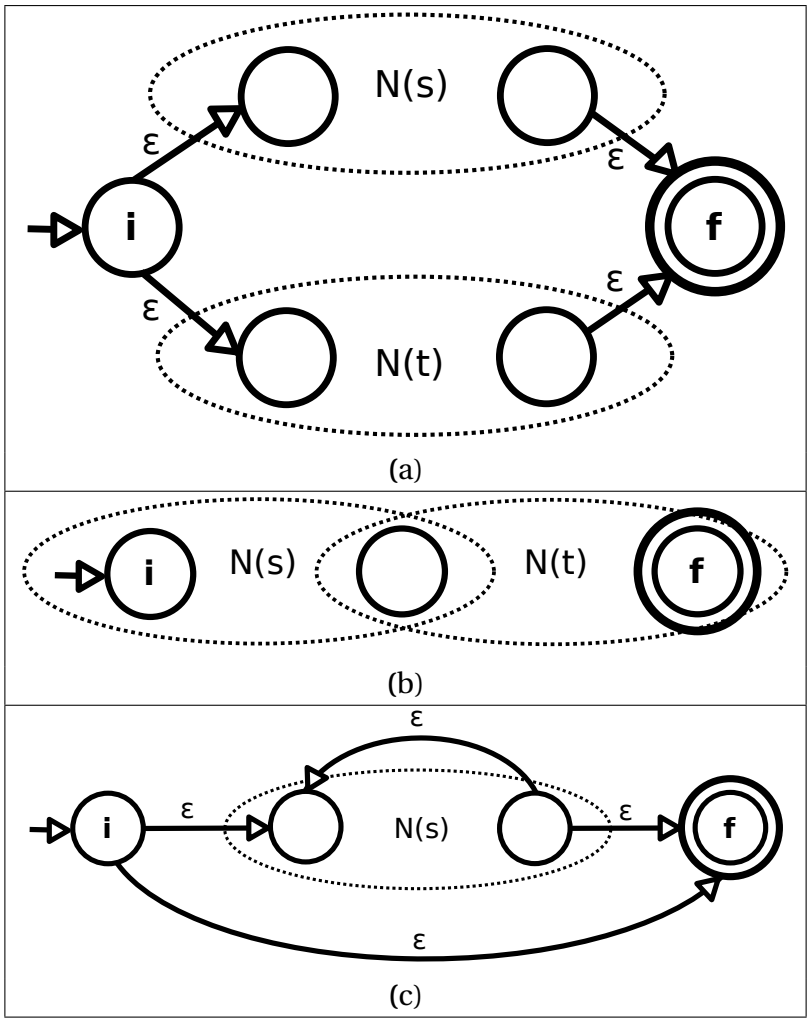


Figura 7 – Regras de substituição de uma ER por um AFND.

2.1.5.2 Expressões Regulares para AFD

Para transformar diretamente uma ER r não estendida para um AFD, deve-se:

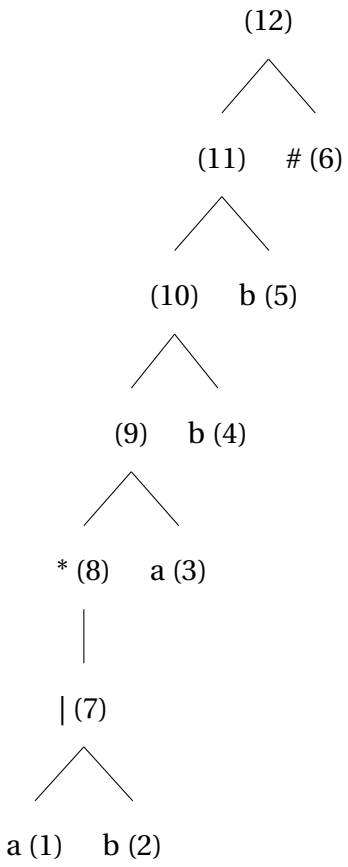
1. Estender r para $(r)\#$;
2. Criar uma árvore sintática para $(r)\#$;
3. Calcular *firstpos*, *lastpos* e *followpos* para cada um dos nodos da árvore;
4. Executar o Algoritmo 3 (algoritmo de conversão ER → AFD).

Detalhes dessa composição são demonstrados nos passos a seguir.

Árvore Sintática para ER

Uma árvore sintática para um ER deve ser construída formando os vértices mais profundos da esquerda para direita. É interessante associar um identificador a cada um dos vértices da árvore. Depois, essa informação será importante.

Para a ER $(a|b)^* abb\#$ teríamos a seguinte árvore sintática abaixo (AHO et al., 2008). Ao lado de cada vértice, um identificador foi colocado entre parenteses para facilitar futuros processos.



Firstpos, *Lastpos* e *Followpos*

Abaixo, há uma explicação de como formar os conjuntos das funções *firstpos*, *lastpos* e *followpos*. Considere que n é um vértice da árvore sintática gerada a partir de uma ER.

- firstpos(n): conjunto de vértices folha de uma árvore sintática que correspondem ao primeiro símbolo de pelo menos uma cadeia da linguagem representada pela sub-ER de n;

- lastpos(n): conjunto de vértices folha de uma árvore sintática que correspondem ao último símbolo de pelo menos uma cadeia da linguagem representada pela sub-ER de n;

- followpos(p):

  - se n é um vértice de concatenação entre um filho esquerdo c_1 e um filho direito c_2, então todas as posições em firstpos(c_2) estão em followpos(i) para cada posição i em lastpos(c_1);

  - se n for um nó de fecho de Kleene (um asterisco) e i é uma posição em lastpos(n), então todas as posições em firstpos(n) estão em followpos(i).

Para o exemplo citado acima, têm-se os seguintes conjuntos para cada um dos vértices:

Vértice	**firstpos**	**lastpos**	**followpos**
1	{1}	{1}	{1,2,3}
2	{2}	{2}	{1,2,3}
3	{3}	{3}	{4}
4	{4}	{4}	{5}
5	{5}	{5}	{6}
6	{6}	{6}	∅
7	{1,2}	{1,2}	∅
8	{1,2}	{1,2}	∅
9	{1,2,3}	{3}	∅
10	{1,2,3}	{4}	∅
11	{1,2,3}	{5}	∅
12	{1,2,3}	{6}	∅

Algoritmo de Conversão

Algoritmo 3: Algoritmo de conversão de ER para AFD.

Input : uma ER r, a árvore sintática n e sua raiz n_0, os conjuntos *firstpos, lastpos* e *followpos*

```
1  Q' ← { firstpos(n_0) }
2  F' ← {}
3  marcado ← {}
4  while Q' ≠ marcado do
5      q ← selecionar um estado em Q' \ marcado
6      marcado ← marcado ∪{q}
7      foreach α ∈ Σ do
8          seja U a união de followpos(p) para todo p ∈ q que corresponde a α
9          if U ≠ {} then
10             if U ∉ Q' then
11                 Q' ← Q' ∪ {U}
12             define δ'(q, α) → U
13             if ∃s ∈ U : s corresponde a um vértice folha # then
14                 F' ← F' ∪ {U}
15 return (Q', Σ, δ', firstpos(n_0), F')
```

Abaixo, executa-se o algoritmo sobre a árvore sintática da ER $(a|b)^* abb$ e os conjuntos *firstpos, lastpos* e *followpos*.

Inicialmente, a linha 1 define que o estado inicial é $\{1,2,3\}$, pois é o que existe em followpos(12). Então, o estado é incluído no conjunto, ou seja, o adicionamos na tabela de transição abaixo.

Está em marcado ?	Q'	**a**	**b**
não	$\{1,2,3\}$	?	?

Depois execula-se o laço da linha 4 pois a condição é verdadeira. Então, $q = \{1,2,3\}$ e q é marcado. Considerando $\alpha = a$, os vértices em q correspondentes a a são 1 e 3, então U =followpos(1) $\cup$ followpos(3)= $\{1,2,3,4\}$. Como o estado $\{1,2,3,4\}$ não possui 6 (vértice correspondente a #) em sua composição, então ele não é final. Considerando $\alpha = b$, o vértice em q correspondentes a b é 2, então U =followpos(3)= $\{1,2,3\}$. Têm-se:

Está em marcado ?	Q'	**a**	**b**
sim	$\{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3\}$
não	$\{1,2,3,4\}$	?	?

Depois, executa-se mais uma nova iteração no laço da linha 4. O único estado não marcado é escolhido como $q = \{1,2,3,4\}$. Considerando $\alpha = a$, os vértices de q que correspondem a a são 1 e 3, então U =followpos(1) $\cup$followpos(3)= $\{1,2,3,4\}$. Como esse estado já foi adicionado a Q', então o mesmo não é adicionado. Ele também não é final, por não ter 6 em sua composição. Considerando $\alpha = b$, os vértices de q que correspondem a b são 2 e 4, então U =followpos(2)$\cup$followpos(4)= $\{1,2,3,5\}$. Como esse estado não contém 6 então também não é final. Ele é adicionado em Q'. Então têm-se:

Está em marcado ?	Q'	**a**	**b**
sim	$\{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3\}$
sim	$\{1,2,3,4\}$	$\{1,2,3,4\}$	$\{1,2,3,5\}$
não	$\{1,2,3,5\}$	?	?

Em uma nova iteração do laço da linha 4, $q = \{1,2,3,5\}$. Considerando $\alpha = a$, os vértices em q correspondentes a a são 1 e 3, então U =followpos(1) $\cup$ followpos(3)= $\{1,2,3,4\}$. Como esse estado já foi adicionado a Q', ele não é adicionado novamente. Como não possui 6 em sua composição, não é final. Considerando $\alpha = b$, os vértices correspondentes a b em q são 2 e 5, então U =followpos(2) $\cup$ followpos(5)= $\{1,2,3,6\}$. Como esse estado não está em Q', ele é adicionado. Como 6 faz parte de sua composição, ele é um estado final. Têm-se:

Está em marcado ?	Q'	**a**	**b**
sim	$\{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3\}$
sim	$\{1,2,3,4\}$	$\{1,2,3,4\}$	$\{1,2,3,5\}$
sim	$\{1,2,3,5\}$	$\{1,2,3,4\}$	$\{1,2,3,6\}$
não	$*\{1,2,3,6\}$	?	?

Em uma nova iteração no laço da linha 4, $q = \{1,2,3,6\}$. Considerando $\alpha = a$, os vértices em q que correspondem a a são 1 e 3, então U =followpos(1)$\cup$followpos(3)= $\{1,2,3,4\}$. Como o estado está em Q' não é mais adicionado. Como não possui 6 em sua composição, não é um estado final. Considerando $\alpha = b$, o vértice em q que correspondem a b é 2, então U =followpos(2)=$\{1,2,3\}$. Como o estado já está em Q', ele não é adiciondo. Como não possui 6 em sua composição, ele não é estado final.

Como não há mais estados a serem analisados, finaliza-se o algoritmo. Note que o primeiro vértice adicionado em Q' (na linha 1) é o estado inicial, então têm-se:

Está em marcado ?	Q'	**a**	**b**
sim	$\rightarrow \{1,2,3\}$	$\{1,2,3,4\}$	$\{1,2,3\}$
sim	$\{1,2,3,4\}$	$\{1,2,3,4\}$	$\{1,2,3,5\}$
sim	$\{1,2,3,5\}$	$\{1,2,3,4\}$	$\{1,2,3,6\}$
sim	$*\{1,2,3,6\}$	{1,2,3,4}	{1,2,3}

O AFD correspondente pode ser visualizado na Figura

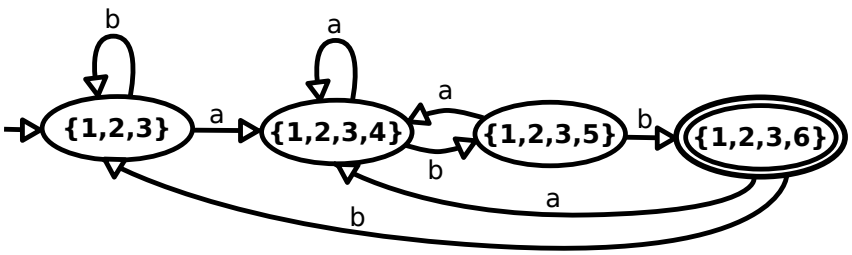


Figura 8 – AFD convertido diretamente da ER $(a|b)^*abb$.

2.2 Projeto de um Analisador Léxico

Um analisador léxico consiste nos seguintes componentes (AHO et al., 2008):

- Uma tabela de transição para o autômato;
- As funções que são passadas diretamente dele para a saída;
- As ações do programa de entrada, que aparecem como trechos de código que serão invocados pelo simulador do autômato.

Estes componentes podem ser visualizados na Figura 9. O autômato simulado executa os passos da tabela de transição. Os *lexemas* são lidos através de dois apontamentos (*lexemeBegin* e *forward*) que auxiliam o autômato a casar o padrão encontrado.

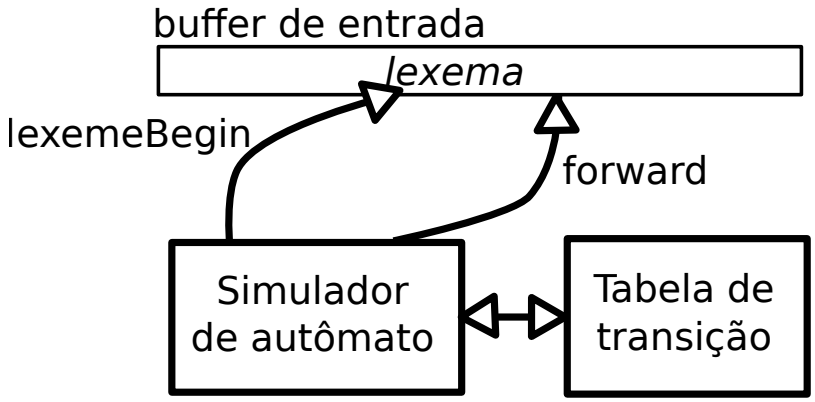


Figura 9 – Uso de Autômato Finito para realizar a Análise Léxica. Adaptado de Aho et al. (2008).

Inicialmente, toma-se as expressões regulares definidas para a linguagem e as convertemos para autômatos. Depois, precisamos de um único autômato. Para isto, todos os autômatos são unidos criando um estado inicial que realiza transições vazias (ϵ) para o estado inicial de cada autômato, conforme demonstrado na Figura 10.

2.2.1 Implementando o Operador *Lookahead*

Algumas vezes, é necessário implementar um operador chamado de *lookahead* quando o padrão pode ser confundido em dois tokens diferentes ou mais. Geralmente formaliza-

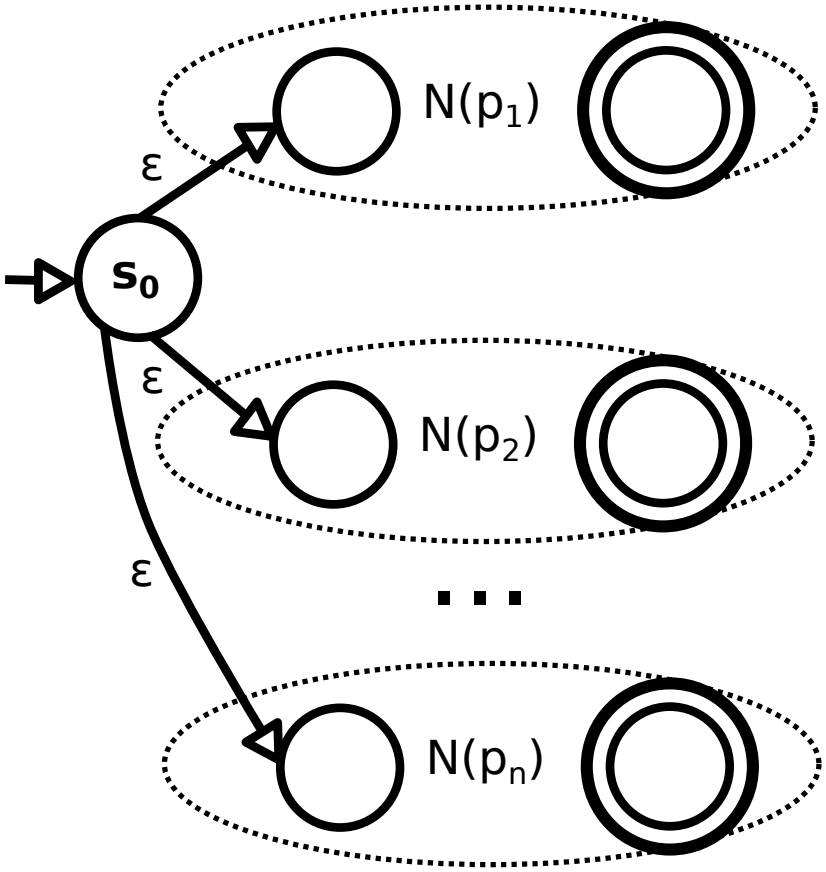


Figura 10 – AFND construído para realizar a Análise Léxica – união dos autômatos. Adaptado de Aho et al. (2008).

se da seguinte forma r_1/r_2, onde r_1 é o padrão que se deseja identificar e o r_2 é o padrão esperado a seguir, ou seja, o *lookahead*. Para construir um autômato com *lookahead* deve-se criar um AF para r_1 e r_2. Depois, une-se o estado "final" q de r_1 ao inicial p de r_2 tal que $\delta(q,\epsilon(/)) \rightarrow p$. Essa transição também é chamada de transição sob o domínio imaginário. q passa a não ser mais estado final após a união (AHO et al., 2008).

Então o AFND para reconhecer o padrão de aceitação sse:

- o estado atual tem uma transição sob o $\epsilon(/)$;
- existe um caminho a partir do estado inicial para o estado atual correspondente ao padrão r_1;
- existe um caminho a partir do estado inicial para o estado atual correspondente ao padrão r_2;

- o padrão r_1 é o mais longo possível para qualquer palavra satisfazendo as propriedades acima.

2.3 Listas de Exercícios

2.3.1 Autômatos Finitos

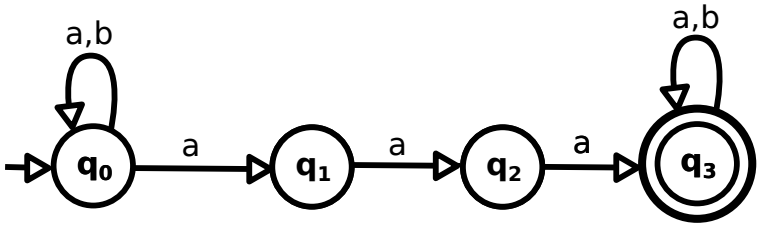
Crie um autômato finito para cada uma das linguagens abaixo. Determine se o autômato que você criou é determinístico (AFD), não-determinístico (AFND) ou não determinístico com ϵ-transições (AFND-ϵ).

- $L_1 = \{w \in \{0,1\}^* | w$ não contém dois 1's consecutivos$\}$
- $L_2 = \{x \in \{0,1\}^* | x$ não possui três 1's consecutivos$\}$
- $L_3 = \{0^m 1^n | m \geq 0, n > 0\}$
- $L_4 = \{w \in \{a,b\}^* |$ o sufixo de w é $aa\}$
- $L_5 = \{w \in \{a,b\}^+ | w = a^n b^m$, sendo que $n \geq 0\}$
- $L_6 = \{w \in \{a,b\}^* | w$ possua aaa como prefixo$\}$
- $L_7 = \{x \in \{a,b,c\}^* | x$ possui abc como subpalavra $\}$
- $L_8 = \{w \in \{a,b\}^* | w = a^n b^m$, sendo que n é impar e m é par$\}$
- $L_9 = \{w \in \{a,b\}^* |$ possui um número ímpar de a e número ímpar de $b\}$
- $L_{10} = \{w \in \{a,b\}^* | w$ possui número par de a e ímpar de b, ou w possui número par de b e ímpar de $a\}$
- $L_{11} = \{x \in \{a,e,i,o,u\}^* | x$ possua aie como subpalavra, e não possua ooo como sufixo$\}$
- $L_{12} = \{a^n b^n | n > 0\}$
- $L_{13} = \{a^n b^{2n} | n > 0\}$

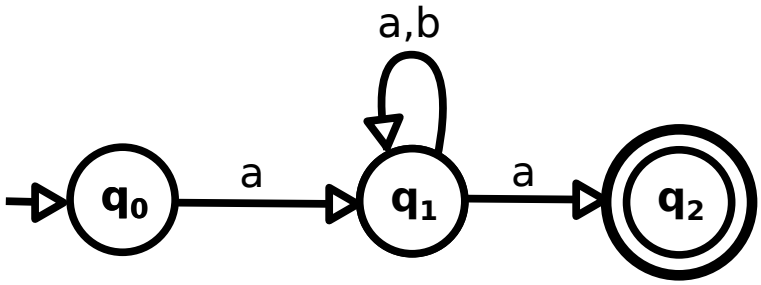
2.3.2 Determinização: AFND → AFD

Dados os AFNDs abaixo, determinize-os a apresente o AFD resultante:

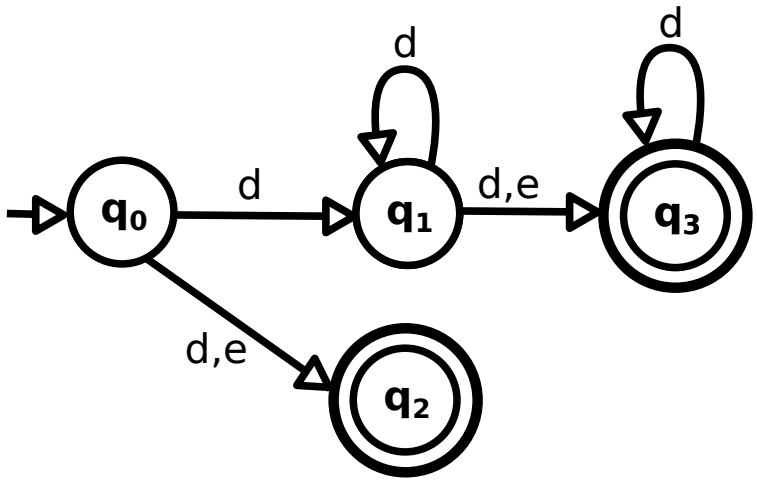
1.



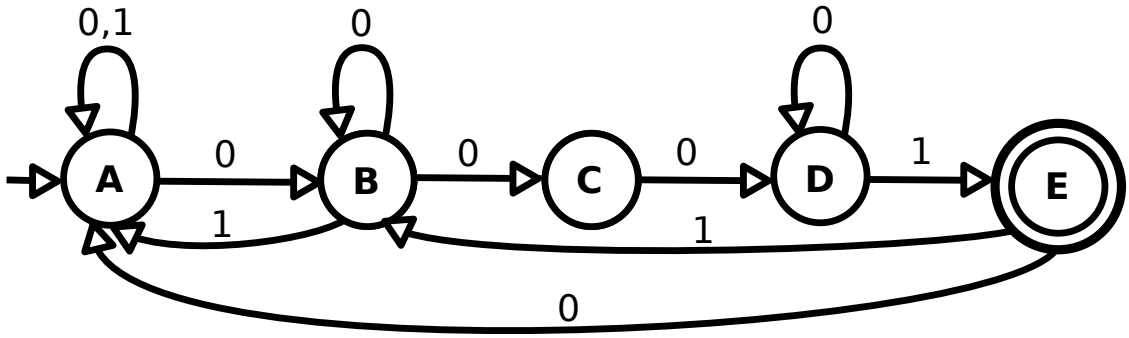
2.



3.



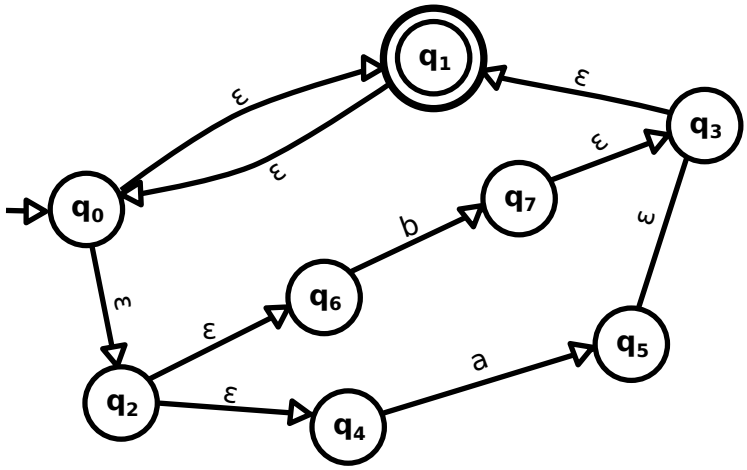
4.



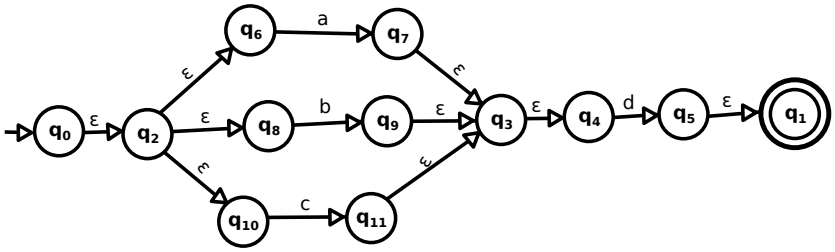
2.3.3 Determinização: AFND-ϵ → AFD

Dados os AFND-ϵ abaixo, determinize-os a apresente o AFD resultante:

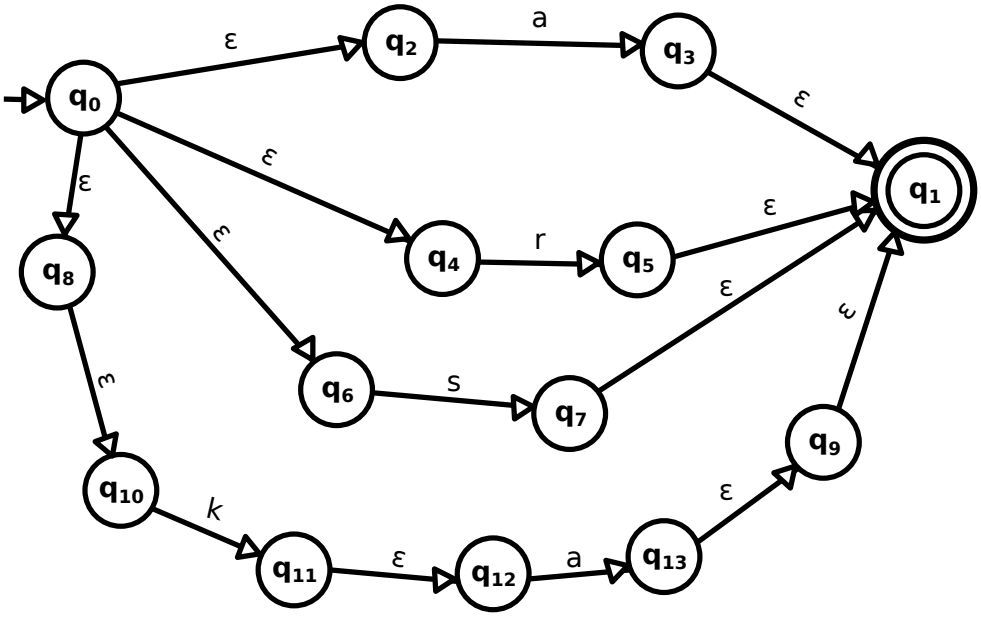
1.



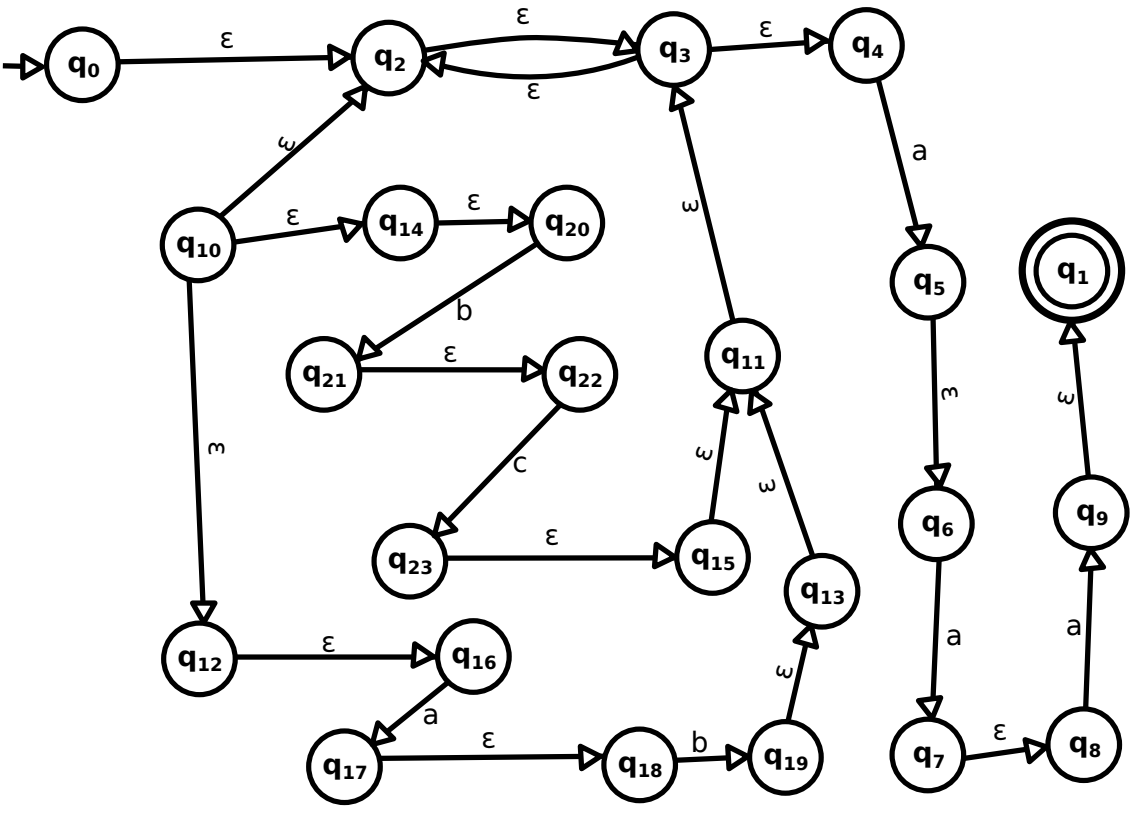
2.



3.



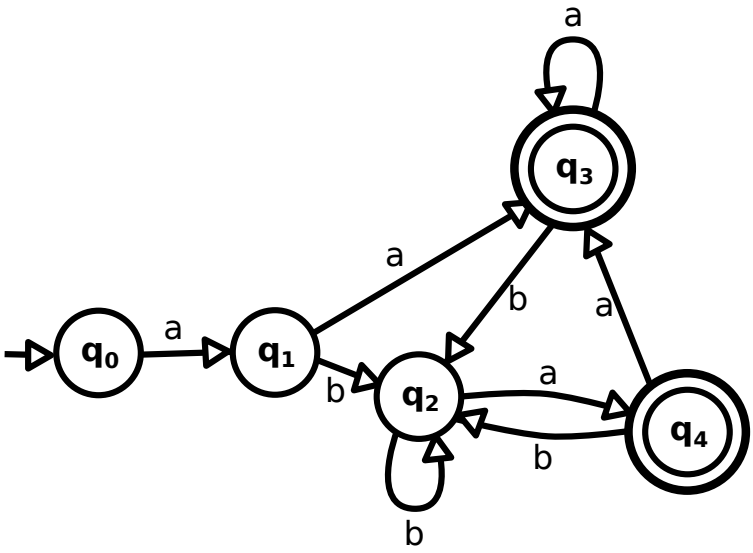
4.



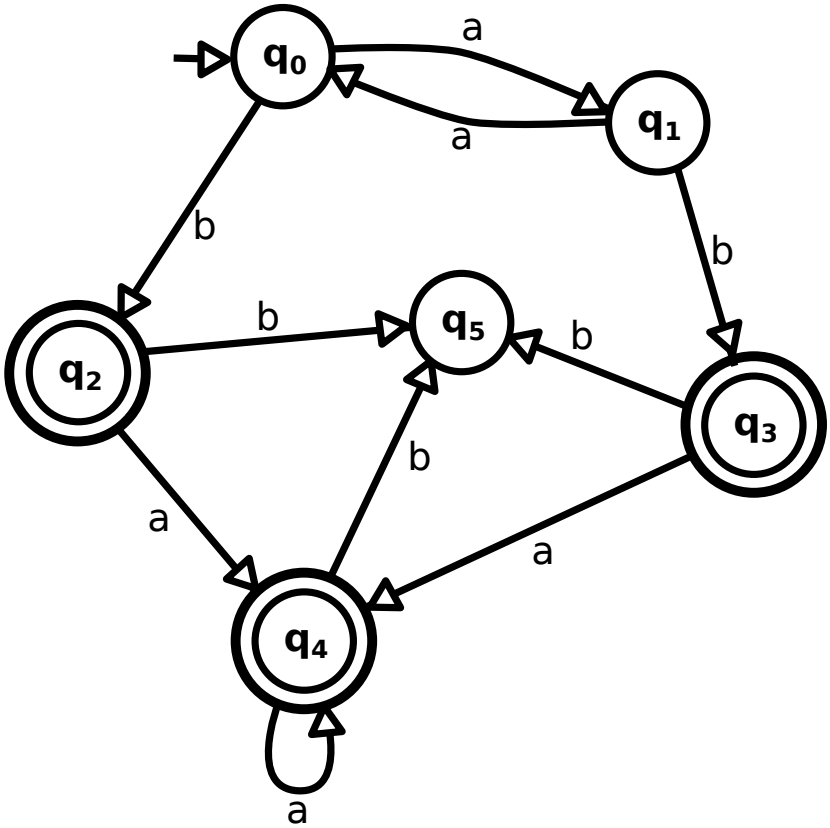
2.3.4 Minimização

Minimize os seguintes Autômatos Finitos:

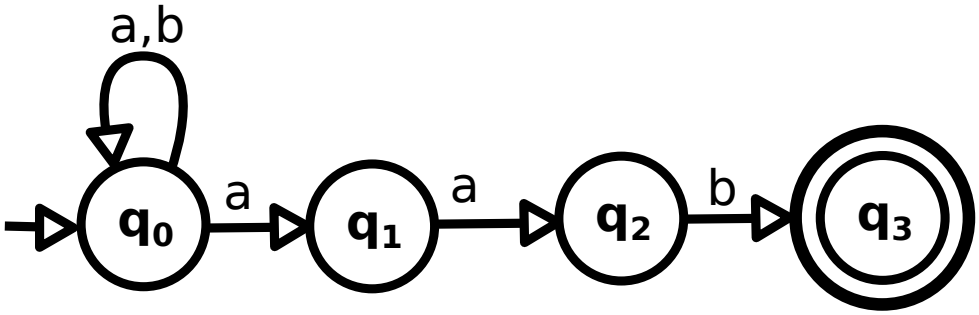
1.



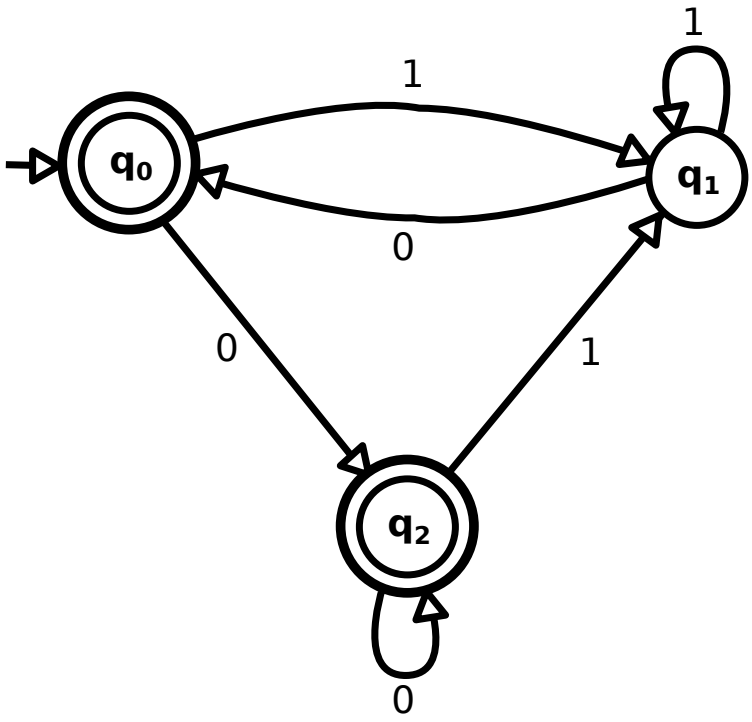
2.



3.



4.



2.3.5 Expressões Regulares

Crie uma Expressão Regular para atender cada um dos padrões abaixo:

1. Nomes de variáveis na linguagem de programação Java;
2. Números em ponto flutuante incluindo a notação científica;
3. Números inteiros hexadecimais;
4. Código de Endereçamento Postal (CEP);
5. Telefones com ou sem DDD, com ou sem prefixo internacional;

6. URLs;

7. Valor monetário com prefixo “R$”;

8. Datas no formato dd/mm/aaaa.

CAPÍTULO 3

Análise Sintática

Para criar linguagens de programação, é necessário estabelecer regras para especificar sua estrutura sintática. Essa estrutura necessita (ou consome) os tokens definidos no processo de análise léxica, considerando-os como elementos indivisíveis (terminais) de uma gramática. De acordo com Aho et al. (2008), há diversos benefícios ao se utilizar uma estrutura de gramática durante um projeto de compiladores:

- Uma gramática provê especificação facilitada;
- Classes gramaticais podem ser utilizadas para construção eficiente de um analisador sintático;
- Gramáticas facilitam a tradução de programas fonte para um código objeto correto e sem erros;
- Uma gramática permite que a construção de uma linguagem fonte seja iterativa, ou seja, recebe posteriores melhorias.

Um analisador sintático recebe uma cadeia de tokens e verifica se essa cadeia pertence a linguagem gerada gramática. Ele gera uma árvore de derivação que será repassada ao restante do *front-end* do compilador (AHO et al., 2008). Entretanto, é muito comum que os demais processos de *front-end* sejam implementados juntamente com o analisador sintático.

Os métodos de análise usados em compiladores são baseadas em estratégias ascendente e descendente. Um métodos ascendente constrói a árvore das folhas até a raiz; um método descendente constrói a árvore da raiz às folhas. Em todas as estratégias, a entrada do analisador sintático é realizada da esqueda para a direita, símbolo por símbolo.

Na Figura 11, há um panorama de como o analisador sintático interage com os demais componentes de um compilador.

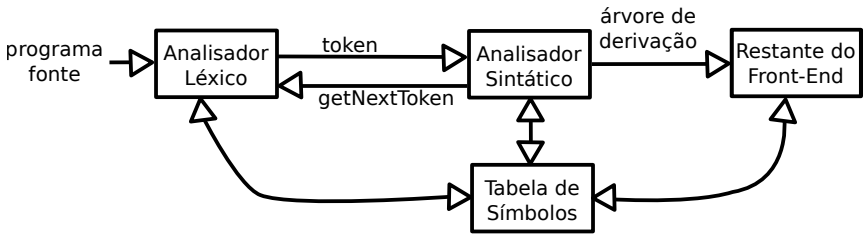


Figura 11 – Posição do analisador sintático no modelo de compilador de Aho et al. (2008).

3.1 Tratamento de Erro de Sintaxe

Erros mais comuns na escrita de programas (AHO et al., 2008):

- Erros léxicos (ortografia errada de identificadores, palavras-chave ou operadores);
- Erros sintáticos incluem ponto-e-vírgula faltante ou mal colocado, ausência de um dos delimitadores de escopo, comandos incompletos ou com erros de formação;
- Erros semânticos incluem divergências entre tipos e operadores;
- Erros lógicos incluem o raciocínio incorreto.

Como estratégicas de recuperação de erros sintáticos, tem-se (AHO et al., 2008):

- Recuperação em modo pânico: ao detectar o erro, o analisador descarta o símbolo da entrada de cada vez até que um conjunto de tokens de sincronismo[1] seja encontrado. Esta recuperação é simples e não se preocupa com a busca de erros adicionais.

- Recuperação em nível de frase: ao detectar o erro, o analisador sintático pode realizar a correção local sobre o restante da entrada. Um exemplo seria a substituição de uma vírgula por um ponto-e-vírgula, exclusão de ponto-e-vírgula desnecessário ou a inserção de um ponto e vírgula. Deve-se ter cuidado com as substituições para evitar que se provoque uma repetição infinita (inserir símbolos a frente da entrada corrente).

- Produções de erro: nesta estratégia, pode-se estender a gramática da linguagem para inserir produções que geram construções erradas, antecipando erros mais comuns.

- Correção global: nesta estratégia, o analisador sintático auxilia na escolha de uma sequência mínima de mudanças a fim de obter uma correção com um custo global menor. Estes métodos são muito custosos e tem interesse meramente teórico.

3.2 Linguagem Livre de Contexto

As Linguagens Livres de Contexto (LLC) pertencem a um conjunto mais amplo que as Linguagens Regulares (LR). Toda Linguagem Regular é Livre de Contexto, mas o oposto não é verdade. As linguagens que são LLC mas não são regulares, diz-se que são linguagens estritamente Livres de Contexto (SIPSER, 2007).

As LLCs surgiram com a tentativa mal sucedida na década de 50 de representar a linguagem natural. A ideia era representar cada parte de estruturas de frases e sentenças

[1] Tokens de sincronismo são delimitadores como o ponto-e-vírgula para as linguagens C e Java.

(sujeito, objeto, verbo, adjetivos, ...). No entanto, a linguagem natural é muito complexa para ser representada inclusive pelas LLC (SIPSER, 2007).

Toda a LLC pode ser gerada por uma Gramática Livre de Contexto (GLC) e podem ser reconhecidas por um Autômato de Pilha (AP), que pode ser determinístico ou não (SIPSER, 2007).

No contexto de compiladores, as LLC são utilizadas na Análise Sintática. Logo, sua função aqui é definir se os tokens de um código fonte estão corretamente posicionados de acordo com as convenções da linguagem. Diferentemente do que foi trabalhado na Análise Léxica, na qual os Autôamtos Finitos tomam boa parte das representações, na Análise Sintática são as Gramáticas Livres de Contexto que são as representações mais utilizadas.

3.2.1 Gramática Livre de Contexto

Toda GLC é uma 4-upla $G = (V, \Sigma, R, S)$, na qual (SIPSER, 2007):

- V é o conjunto finito de todas as **variáveis** ou **não-terminais**;
- Σ é o conjunto finito, disjunto de V de símbolos do alfabeto, aqui chamados também de **terminais**;
- R é o conjunto de regras de produção, sendo que $\alpha \rightarrow \beta \in R$, na qual $\alpha \in V$ e $\beta \in (V \cup \Sigma)^*$;
- $S \in V$ não-terminal inicial.

Se diz que uma GLC gera as palavras da linguagem. As regras em R especificam que o não-terminal (α) que está à esquerda do símbolo "$\rightarrow$" pode ser substituído pela parte direita da produção (β). Então, uma gramática estabelece um conjunto de regras de substituição a partir do não-terminal inicial, que somente é concluído quando todos os símbolos forem terminais. Cada substituição é chamada de derivação.

A literatura quando trata de GLCs normalmente define que os não-terminais são representados apenas por letras maiúsculas e os terminais por letras minúsculas. Note que isso é uma convenção que não pode ser aplicada ao projeto de compiladores. De qualquer modo, inicia-se aqui com essa convenção por questões de facilidade de representatividade.

Veja o seguinte exemplo de uma GLC abaixo:

$$\begin{aligned} A &\rightarrow aAb|B \\ B &\rightarrow ab|\epsilon \end{aligned} \tag{3.1}$$

. A GLC é definida formalmente por $G = (V, \Sigma, R, A)$, na qual $V = \{A, B\}$, $\Sigma = \{a, b\}$ e $R = \{A \rightarrow aAb, A \rightarrow B, B \rightarrow ab, B \rightarrow \epsilon\}$. Para gerar a palavra *aaabbb*, a seguinte sequência de derivações é necessária:

A

aAb

$aaAbb$

$aaBbb$

$aaabbb$

Diz-se então que $A \rightarrow^4 aaabbb$, ou seja, foram necessárias 4 derivações para se atingir a palavra. Como tal sequência de derivações fora encontrada, diz-se que a palavra *aaabbb* pertence à linguagem que a GLC G gera. Tente descobrir que linguagem é essa!

Embora funcione bem para as definições mais teóricas relacionadas a Linguagens Formais, a notação de letras minúsculas para terminais e letras maiúsculas para não-terminais pode ser confusa para representar GLCs de linguagens de programação. Por esse motivo, em alguns momentos, será usada uma notação alternativa: letras em negrito representam os **terminais** e não-negrito os *não-terminais*.

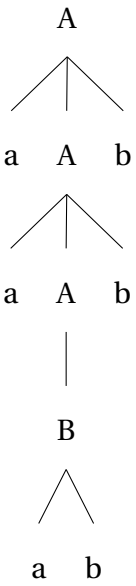
Nas anotações na lousa, o professor deverá representar os não terminais de forma semelhante à notação utilizada na Forma Normal Backus, colocando-os entre os símbolos

< e >.

3.2.1.1 Árvores de Derivação

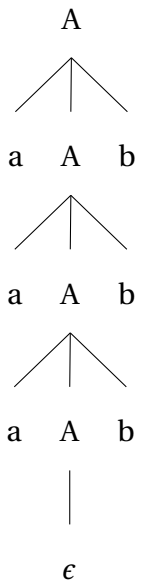
Uma árvore de derivação representa o processo de derivação. Cada nodo da árvore representa ou um símbolo terminal ou um não-terminal. Na raíz, encontra-se o não terminal inicial da gramática. A cada nível i, têm-se o resultado da derivação do nível $i-1$ utilizando uma das regras de produção (do conjunto R da gramática). Dois nodos da árvore são conectados por um arco para representar uma derivação $\alpha \rightarrow \beta$. O nodo de origem é o não-terminal α que foi substituído e um nodo de destino para cada um dos símbolos gerados em β. Então, sempre que se deseja representar uma derivação, deve-se representar todos os terminais e não-terminais que substituem o não-terminal à esquerda da derivação. Os terminais são sempre nodos folhas da árvore.

O processo de derivação da palavra $aaabbb$ pela gramática na Equação (3.1) gera a seguinte árvore de derivação:



Uma característica importante de um processo de derivação é que pode haver várias árvores para chegar a uma mesma palavra. Veja que isso acontece para a palavra acima, na qual duas árvores de derivação podem ser utilizadas para representar a derivação da palavra $aaabbb$ pela gramática na Equação (3.1). Chama-se isso de ambiguidade e diz-se que a referida gramática é ambígua. Além da árvore exibida anteriormente, é

possível chegar à palavra a partir de outra árvore de derivação:



No contexto de compiladores, a ambiguidade é indesejada, pois não é possível determinar qual sequência de derivações seguir com as informações disponíveis no momento da análise da palavra de entrada. Por isso, a eliminação de ambiguidade deve ser considerada no projeto de um compilador.

Eliminando Ambiguidade

Nessa seção, busca-se exemplificar uma maneira de tratar ambiguidade no projeto de linguagens retirada do livro do Aho et al. (2008). Considere a seguinte gramática

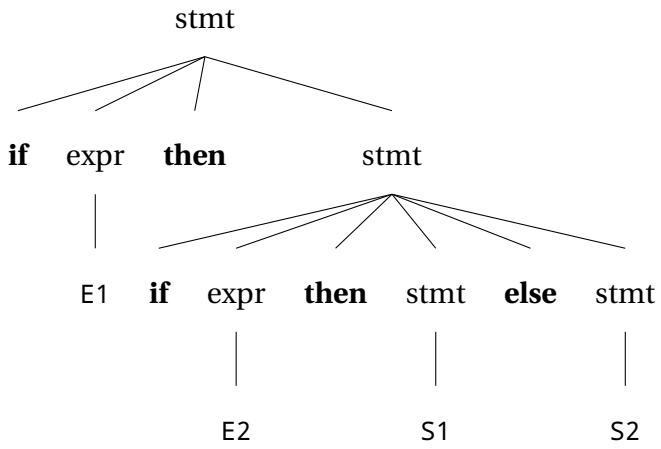
$$\begin{aligned} stmt \rightarrow & \\ & \textbf{if } expr \textbf{ then } stmt \mid \\ & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \mid \\ & \textbf{other} \end{aligned} \tag{3.2}$$

. Considere a seguinte palavra para gerada pela gramática acima

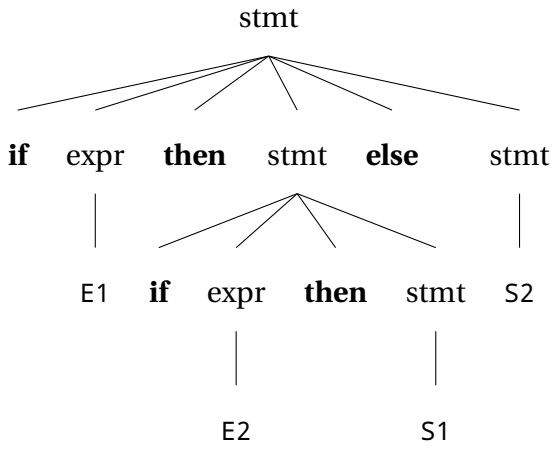
```
if E1 then if E2 then S1 else S2
```

, assumindo que `E1` e `E2` são expressões e `S1` e `S2` são comandos (*statements*). A ambiguidade pode ser visualizada em duas árvores de derivação (AHO et al., 2008):

Primeira árvore de derivação:



Segunda árvore de derivação:



Como pretende-se usar as árvores de derivação para realizar outras operações da compilação (como geração de código ou interpretação) as duas árvores terão significados diferentes quando se busca a definição de uma estrutura corretamente subordinada (aninhada).

Em linguagens de programação como esta, onde o escopo dos desvios condicionais não estão claros, uma ideia de tratar é considerar que cada **else** deve casar com o **then** mais próximo. Reescrevendo a gramática de forma nao ambígua, obtém-se (AHO et al.,

2008):

$$\begin{aligned} stmt &\rightarrow matched\_stmt \mid open\_stmt \\ matched\_stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ matched\_stmt\ \textbf{else}\ matched\_stmt \mid \textbf{other} \\ open\_stmt &\rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt \mid \textbf{if}\ expr\ \textbf{then}\ matched\_stmt\ \textbf{else}\ open\_stmt \end{aligned} \tag{3.3}$$

.

Expressando Precedência em Árvores de Derivação

Observe a seguinte GLC para representar algumas operações aritméticas

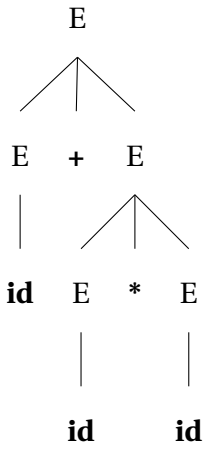
$$E \rightarrow E{+}\ E \mid E * E \mid -E \mid (E) \mid \textbf{id} \tag{3.4}$$

. Para a palavra

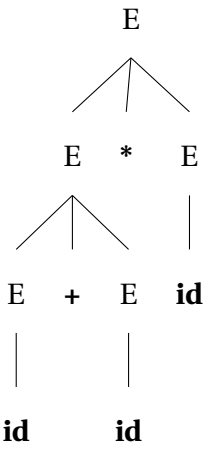
```
id + id * id
```

, é possível gerar duas árvores de derivação.

Primeira árvore de derivação:



Segunda árvore de derivação:



Como já foi dito, a ambiguidade é ruim para um compilador determinar qual caminho seguir no processo de compilação. Além disso, note que a estrutura da árvore não fixa em que nível se encontram os operadores de soma (+) e multiplicação (*) de acordo

com uma mesma palavra. Pode-se tratar a precedência dos operadores da linguagem juntamente com a remoção da ambiguidade.

Sabe-se que as multiplicações devem ser realizadas primeiras do que a soma. Então a árvore que corresponde a correta precedência seria a primeira, representando

```
id + ( id * id )
```

.

Uma gramática não-ambígua que considera as precedências dos operadores da gramática da Equação (3.4) é exibida na Equação 3.5:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow -F \mid G \\ G &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{3.5}$$

3.2.1.2 Forma Normal Chomsky

De acordo com Sipser (2007), uma das maneiras mais simples para tornar uma gramática simplificada é convertendo-a para a Forma Normal Chomsky.

O algoritmo de conversão de qualquer GLC para a Forma Normal Chomsky se encontra abaixo (SIPSER, 2007). Pretende-se explicar o procedimento de transformação utilizando a gramática abaixo:

Veja o exemplo abaixo da seguinte GLC:

$$\begin{aligned} S &\rightarrow ASA \mid aB \\ A &\rightarrow B \mid S \\ B &\rightarrow b \mid \epsilon \end{aligned} \tag{3.6}$$

O algoritmo de transformação com exemplo segue:

1. Deve-se remover as produções em ϵ. Para isso, considere todos os não-terminais da linguagem com produção diretamente para ϵ. Os chamaremos de anuláveis.

Depois, deve-se criar novas produções considerando que os não-terminais anuláveis poderiam ser vazios. Ao final, elimina-se todas as produções em ϵ. No Exemplo, obtém-se a GLC da Equação (3.7).

$$\begin{aligned} S &\rightarrow ASA \mid aB \mid AS \mid SA \mid S \mid a \\ A &\rightarrow B \mid S \\ B &\rightarrow b \end{aligned} \tag{3.7}$$

2. Considera-se produção unitária toda aquela que substituir um não-terminal por um não-terminal apenas. Substitui-se as produções unitárias do tipo $X \rightarrow Y$ fazendo X produzir as produções de Y. No caso de $X \rightarrow X$, apenas remove-se a produção. No Exemplo, obtém-se a GLC da Equação (3.8).

$$\begin{aligned} S &\rightarrow ASA \mid aB \mid AS \mid SA \mid a \\ A &\rightarrow b \mid ASA \mid aB \mid AS \mid SA \mid a \\ B &\rightarrow b \end{aligned} \tag{3.8}$$

3. Todas as produções com o lado direito contendo dois ou mais símbolos devem substituir os terminais por um não-terminal que apenas o produz. No Exemplo, obtém-se a GLC da Equação (3.9).

$$\begin{aligned} S &\rightarrow ASA \mid XB \mid AS \mid SA \mid a \\ A &\rightarrow b \mid ASA \mid XB \mid AS \mid SA \mid a \\ B &\rightarrow b \\ X &\rightarrow a \end{aligned} \tag{3.9}$$

4. Todas as produções com o lado direito contendo três ou mais símbolos devem ter seu sufixo sendo produzido por outros não-terminais. Esses não-terminais também devem respeitar que cada produção deva ter no máximo dois símbolos no lado direito. Então, deve-se aplicar essa etapa recursivamente. No Exemplo,

obtém-se a GLC da Equação (3.10).

$$\begin{aligned} S &\rightarrow AA_1 \mid XB \mid AS \mid SA \mid a \\ A &\rightarrow b \mid AA_1 \mid XB \mid AS \mid SA \mid a \\ B &\rightarrow b \\ X &\rightarrow a \\ A_1 &\rightarrow SA \end{aligned} \tag{3.10}$$

3.2.1.3 Eliminação de Recursão à Esquerda

É dito recursão à esquerda quando um não-terminal A pode ser substituído por uma produção com um não-terminal A à esquerda. Ela pode ser direta ou indireta:

- Uma recursão direta acontece quando há uma produção do tipo $A \rightarrow A\alpha$, na qual α é qualquer concatenação de terminais e/ou não-terminais incluindo ϵ;
- Uma recursão indireta ocorre quando não se produz diretamente o mesmo não-terminal à esquerda diretamente, mas depois de uma ou mais derivações.

Veja o exemplo abaixo de uma GLC com recursões à esquerda:

$$\begin{aligned} S &\rightarrow Aa \mid Sa \mid b \\ A &\rightarrow Ac \mid Sd \end{aligned} \tag{3.11}$$

. Há recursões diretas à esquerda nas produções $S \rightarrow Sa$ e $A \rightarrow Ac$. Há recursão indireta à esquerda se considerarmos as derivações de $S \rightarrow Aa$, e depois $A \rightarrow Sd$. Outra indireta é partir de uma dericação $A \rightarrow Sd$ e depois $S \rightarrow Aa$. Portanto, para eliminar recursões à esquerda, deve-se tratar as recursões diretas e indiretas à esquerda.

O algoritmo que elimina as recursões à esquerda direta e indireta demanda que a gramática de entrada esteja não tenha produções unitárias e produções em ϵ. Se a GLC possui essas características, pode-se aplicar a Forma Normal Chomsky (Seção 3.2.1.2)

para torná-la adequada para o algoritmo de eliminação de recursões à esquerda. O algoritmo de eliminação à esquerda se encontra no Algoritmo 4.

Algoritmo 4: Algoritmo de eliminação de recursões à esquerda.

Input : uma gramática $G = (V, \Sigma, R, S)$ livre de produções unitárias e produções em ϵ

1 determine os não-terminais em V em uma ordem, aqui chamada de $A_1, A_2, \ldots, A_n$, sendo que $A_i \in V$ para todo $i \in \{1, 2, \ldots, n\}$

2 **for** $i \leftarrow 1$ ***to*** n **do**

3 **for** $j \leftarrow 1$ ***to*** $i - 1$ **do**

  `// Eliminar recursões à esquerda indiretas`

4 substitua cada produção $A_i \rightarrow A_j\gamma$ pelas produções $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\ldots|\delta_k\gamma$, nas quais $A_j \rightarrow \delta_1|\delta_2|\ldots|\delta_k$

  `// Eliminar recursões à esquerda diretas`

5 caso A_i tenha ao menos uma recursão direta à esquerda e considerando as suas produções $A_i \rightarrow A_i\alpha_1|A_i\alpha_2|\ldots|A_i\alpha_l|\beta_1|\beta_2|\ldots|\beta_m$, crie um novo não terminal A'_i e substitua as produções de A_i por $A_i \rightarrow \beta_1|\beta_2|\ldots|\beta_m|\beta_1A'_i|\beta_2A'_i|\ldots|\beta_mA'_i$. Depois adicione as produções do novo não-terminal $A'_i \rightarrow \alpha_1|\alpha 2|\ldots|\alpha_l|\alpha_1A'_i|\alpha 2A'_i|\ldots|\alpha_lA'_i$

6 **return** *A gramática G alterada*

Desafio

Tente eliminar a recursão à esquerda da seguinte gramática:

$$\begin{aligned} A &\rightarrow BC \mid a \\ B &\rightarrow CB \mid Bb \mid f \\ C &\rightarrow DA \mid c \\ D &\rightarrow AC \mid Ce \mid d \end{aligned} \tag{3.12}$$

.

3.2.1.4 Fatoração à Esquerda

É uma transformação da gramática útil em uma gramática adequada para um reconhecedor sintático preditivo ou descendente. Quando a escolha entre duas ou mais produções iniciam com a mesma forma setencial, pode-se reescrever essas produções para adiar a decisão até que tenha-se lido uma cadeia de entrada longa o suficiente para se tomar a decisão correta (AHO et al., 2008).

Considere uma gramática no seguinte formato:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m \tag{3.13}$$

. Observe que, com uma entrada α, não se sabe como expandir o não terminal A; não se sabe se assumir $\alpha\beta_1$, $\alpha\beta_2$, ...ou $\alpha\beta_n$. Para adiar a decisão, realiza-se a fatoração à esquerda especificada no Algoritmo 5.

Algoritmo 5: Algoritmo de fatoração à esquerda.

Input : uma gramática $G = (V, \Sigma, R, S)$

```
1  foreach A ∈ V do
2      while Existir um prefixo comum para duas ou mais produções em A do
3          α ← encontrar prefixo α mais longo comum a duas ou mais alternativas de
              produção em A
4          if a ≠ ϵ then
5              V ← V ∪ {A'}
6              foreach A → αβ_i ∈ R do
7                  R ← R\{A → αβ_i}
8                  R ← R ∪ {A' → β_i}
9              R ← R ∪ {A → αA'}
10 return A gramática G alterada
```

Depois de fatorada, a gramática apresentada na Equação (3.13) ficará da seguinte forma

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_m \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned} \tag{3.14}$$

.

3.3 Análise Sintática Descendente

Também conhecida como Análise Top-Down, os métodos de análise sintática descendente constroem a cadeia da raiz para as folhas, criando os nós da árvore em pré-ordem (busca em profundidade, produzindo o nodo mais à esquerda) (AHO et al., 2008). O objetivo desses métodos é encontrar a derivação mais à esquerda para a cadeia de entrada ao construir uma árvore sintática a partir de uma raiz.

Esta seção está organizada da seguinte forma. Primeiro serão discutidos como devem ser compostos os conjuntos FIRST e FOLLOW, que são requisitos para construir analisadores descendentes. Depois, as Gramáticas LL(1) são apresentadas como um tipo particular de gramáticas livres de contexto que podem ser analisadas pelos analisadores descendentes aqui apresentados. Em seguida, a tabela utilizadas nos analisadores é apresentada. Depois, dois métodos são apresentados, o de descida recursiva e o sem recursão.

3.3.1 FIRST e FOLLOW

FIRST e FOLLOW são duas funções que auxiliam na construção de analisadores descendentes e ascendentes. Durante uma análise descendente, essas funções ajudam a identificar qual produção aplicar, com base no próximo símbolo de entrada.

$FIRST(\alpha)$ é o conjunto de símbolos terminais que iniciam as cadeias derivadas de α, sendo que α é qualquer cadeia de símbolos pertencentes a gramática (terminais ou não-terminais).

Para se computar FIRST(X), deve-se utilizar as seguintes regras (AHO et al., 2008):

1. Se X é um símbolo terminal, então FIRST(X) = {X};
2. Se X é um símbolo não-terminal e $X \rightarrow Y_1 Y_2 \ldots Y_k$ é uma produção para um $k \geq 1$, então acrescente a a FIRST(X) se, para algum i, a estiver em FIRST (Y_i) e ϵ estiver

em todos os FIRST $(Y_1), \ldots,$ FIRST(Y_{i-1})[2]. Se ϵ estiver em todos os Y_j para todo $j \in \{1,2,\ldots,k\}$, então adicionar ϵ a FIRST(X);

3. Se $X \rightarrow^* \epsilon$ é uma produção, então adicionar ϵ a FIRST(X).

Para se computar FOLLOW(A), utiliza-se as seguintes regras (AHO et al., 2008):

1. Colocar \$ em FOLLOW(S), onde S é o não-terminal inicial da gramática e \$ é o marcador de fim da entrada ou do arquivo;
2. Se houver uma produção $A \rightarrow \alpha B \beta$, então tudo em FIRST($\beta$), exceto ϵ está em FOLLOW(B);
3. Se houver uma produção $A \rightarrow \alpha B$, então inclua FOLLOW(A) a FOLLOW(B);
4. Se houver uma produção $A \rightarrow \alpha B \beta$, na qual FIRST($\beta$) contém ϵ, então inclua FOLLOW(A) a FOLLOW(B).

Como exemplo, imagine a seguinte GLC

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{3.15}$$

. Teria-se os seguintes conjuntos para cada não-terminal:

Tabela 1 – Conjuntos FIRST e FOLLOW para cada não terminal da GLC da Equação (3.15)

não-terminal	**FIRST(·)**	**FOLLOW(·)**
E	{(, **id**}	{), \$}
E'	{+, ϵ}	{), \$}
T	{(, **id**}	{+,), \$}
T'	{∗, ϵ}	{+,), \$}
F	{(, **id**}	{+, ∗,), \$}

.

[2] Ou seja, todas as possibilidades de prefixo.

3.3.2 Gramáticas LL(1)

Os analisadores sintáticos preditivos, ou seja, os analisadores de descida recursiva que não precisam de retrocesso podem ser construídos para uma classe de gramáticas chamada LL(1). O primeiro L significa que a palavra de entrada foi lida da esquerda para a direita. O segundo L representa derivação mais à esquerda. O número 1 indica o uso de um símbolo à frente na entrada, utilizado em cada passo da decisão de análise. Nenhuma gramática LL(1) é ambígua ou possui recursão à esquerda (AHO et al., 2008).

Uma gramática G é LL(1) se e somente se sempre que $A \rightarrow \alpha|\beta$ forem duas produções distintas de G, as seguintes condições forem verdadeiras (AHO et al., 2008):

- para um terminal x, tanto α quanto β não derivam cadeias começando com x (com x à esquerda)[3];
- no máximo um dos dois, α ou β, pode derivar a cadeia vazia;
- Se $\beta \rightarrow^* \epsilon$ então α não deriva nenhuma cadeia começando com um terminal FOLLOW(A). De modo semelhante, se $\alpha \rightarrow^* \epsilon$, então β não deriva qualquer cadeia começando com um terminal em FOLLOW(A)[4].

3.3.3 Tabela de Análise Preditiva

A tabela para o reconhecedor sintático preditivo é um arranjo bidimensional $M[A, a]$, onde A é um não-terminal e a é um terminal da gramática ou o símbolo $ (marcador de fim da entrada). A ideia da tabela é a de uma produção $A \rightarrow \beta$ é escolhida se o próximo símbolo da entrada a estiver em FIRST (β). Para construir essa tabela, dada uma gramática G, para cada produção $A \rightarrow \alpha$ de G, executar os seguintes passos (AHO et al., 2008):

1. Para cada terminal x em FIRST(α), inclua $A \rightarrow \alpha$ em $M[A, x]$;

[3] O procedimento de fatoração auxilia nesse problema.
[4] Geraria uma indecisão do que tratar.

2. Se ϵ pertence a FIRST(α), inclua $A \rightarrow \alpha$ em $M[A, b]$ para cada terminal b em FOLLOW(A). Se ϵ pertence a FIRST(α) e \$ pertence a FOLLOW(A), acrescente também $A \rightarrow \alpha$ a $M[A, \$]$.

Se não houver produção alguma em $M[A, a]$ depois de realizar os passos supracitados, então define-se $M[A, a]$ como erro.

Para toda gramática LL(1), cada entrada na tabela de análise identifica no máximo uma produção ou sinaliza um erro. Para o caso de gramáticas que não podem ser transformadas em LL(1) equivalente, M pode ter algumas entradas com múltiplas definições (AHO et al., 2008).

Como exemplo, imagine a seguinte GLC

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{3.16}$$

. Considera-se os seguintes conjuntos FIRST e FOLLOW para cada não terminal da gramática

não-terminal	**FIRST(·)**	**FOLLOW(·)**
E	{(, **id**}	{), \$}
E'	{+, ϵ}	{), \$}
T	{(, **id**}	{+,), \$}
T'	{∗, ϵ}	{+,), \$}
F	{(, **id**}	{+, ∗,), \$}

. Teria-se a seguinte tabela M:

não-terminal	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (E)$		

3.3.4 Análise Sintática de Descida Recursiva

Na análise sintática de descida recursiva, cada não-terminal é representado por um procedimento. A execução inicia-se por ativar o símbolo inicial da gramática. Este método é exemplificado a seguir (AHO et al., 2008) no Algoritmo 6. De acordo com Aho et al. (2008), uma gramática recursiva à esquerda provocará um *loop* infinito.

Algoritmo 6: Procedimento típico no tratamento de um não terminal A em um analisador descendente.

```
1  Function A():
2      Selecionar uma produção A → X1 X2 ... Xk
3      for i = 1 to k do
           // Se Xi é um não-terminal
4          if Xi ∈ V then
5              ativar o procedimento Xi()
6          else
               // Se Xi casar com o símbolo de entrada
7              if Xi for igual ao símbolo de entrada a then
8                  avance na entrada para o próximo símbolo terminal
9              else
                   /* ocorreu um erro - tratá-lo                    */
```

O procedimento demonstrado acima, não ativa retrocesso. Para permitir o retrocesso, deve-se:

1. deve-se tentar todas as alternativas de produção da linha 2;

2. o erro depois da linha 9 pode não ser uma falha definitiva. Deve-se esperar as outras derivações possíveis;

3. criar apontador para demonstrar onde estava quando se atinje a linha 2 (retrocesso).

3.3.5 Analisador Preditivo sem Recursão

Um analisador sintático preditivo pode ser utilizado em recursão, mantendo uma pilha. O analisador simula a derivação mais à esquerda. Esse algoritmo pode ser visualizado abaixo (Algoritmo 7), recebendo como entrada uma cadeia w e uma tabela M de análise

para uma gramática $G = (V, \Sigma, R, X)$.

Algoritmo 7: Reconhecimento preditivo dirigido por tabela.

Input : uma cadeia w e uma tabela M de análise para a gramática G

```
1  ip deve apontar para o primeiro símbolo em w
   // a é o símbolo da posição ip em w
2  a ← w_ip
3  S ← Pilha()
   /* Considere que X é o não-terminal inicial da gramática. Ele é empilhado
      na pilha S                                                          */
4  S.push($)
5  S.push(X)
6  while S.top() ≠ $ do
7      X ← S.top()
8      if X = a then
9          S.pop()
10         avançar ip
11         a ← w_ip
12     else
13         if X ∈ Σ then
14             erro()
15         else
               // Se M[X,a] possui uma entrada não mapeada
16             if M[X,a] é uma entrada de erro then
17                 erro()
18             else
19                 if M[X,a] = X → Y_1 Y_2 ... Y_k then
20                     imprime a produção X → Y_1 Y_2 ... Y_k
21                     S.pop()
22                     foreach Y ∈ (Y_k, Y_{k-1}, ..., Y_2, Y_1) do
23                         S.push(Y)
```

Casamento	**Pilha**	**Entrada**	**Ação**
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	imprime $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	imprime $T \rightarrow FT'$
	$\mathbf{id}T'E'\$$	$id + id * id\$$	imprime $F \rightarrow \mathbf{id}$
id	$T'E'\$$	$+id * id\$$	casa **id**
continue ...			

3.3.5.1 Recuperação de Erros

No livro Aho et al. (2008) (página 146), há sugestão de dois tipos de recuperação de erros utilizando a pilha de um analisador sintático preditivo por tabela, mas as técnicas podem ser aplicadas em analisadores de descida recursiva. A primeira delas refere-se ao modo pânico. A segunda refere-se à recuperação em nível de frase.

A recuperação em modo pânico baseia-se na ideia de ignorar símbolos de entrada até encontrar um token no conjunto de tokens de sincronismo. Sua eficácia depende da escolha do conjunto de sincronismo. Cada conjunto deve ser escolhido de modo que o analisador se recupere rapidamente de erros que ocorrem na prática. Algumas heurísticas são descritas livro Aho et al. (2008) (página 146).

Em uma recuperação a nível de frase, preenche-se as lacunas da tabela de análise com apontadores para rotinas de erros que podem gerar avisos ou inserir/remover símbolos de entrada. Um dos problemas que acontecer é este processo desencadear uma repetição infinita[5].

[5] Repetição infinita nesse contexto é causada geralmente quando a rotina de erro desempilha um símbolo na entrada.

Desafio

Crie um analisador recursivo descendente para a seguinte gramática LL(1):

```
<program>       <-   <block>

<block>         <-   <var_decl>

                   | <lst_procedure> statement

<lst_procedure> <-  <procedure> <lst_procedure>

                   | \epsilon

<procedure> <-  function ident ( <var_decl> )

                <statement_lst>

<var_decl> <-  var ident <lst_ident>

              | \epsilon ;

<lst_ident> <-   , ident <lst_ident>

               | \epsilon

<statement> <-   ident = <expression> ;

                | begin <statement_lst> end

                | if <condition> then <statement_lst>

                | while <condition> do <statement_lst>

<statement_lst> <-  <statement_lst> <statement_lst>

                    | \epsilon

<condition> <-  <expression> <operator> <expression>

<operator> <-  = | < | <= | > | >= | !=

<expression> <- <term> + <term>

<term> <-  <factor> * <factor>

<factor> <- ident | num | ( <expression> )
```

3.4 Análise Sintática Ascendente

A análise sintática ascendente (Bottom-Up) corresponde a construção da árvore de derivação de uma cadeia de entrada das folhas para a raiz da árvore.

3.4.1 Analisador Sintático Shift-Reduce (Empilha-Reduz)

O primeiro analisador ascendente a ser visitado aqui é o Shift-Reduce. Ele consegue realizar análise sintática em gramáticas de um tipo particular de gramáticas LR. É um analisador não-determinístico.

3.4.1.1 Gramática LR

Chama-se LR é toda a gramática LR(k) na qual $k \in \{0,1\}$. Em tal denominação, "L" vem de escandimento da esquerda pra a direita, o "R" representa a construção das derivações mais à direita ao reverso (redução) e o "k" os símbolos a frente no fluxo de entrada que auxiliam nas decisões de análise. Na prática, $k = 0$ ou $k = 1$ são suficientes.

3.4.1.2 Reduções

Para compreender um analisador sintática Shift-Reduce, é necessário compreender um conceito requisito chamado de redução. A análise ascendente pode ser idealizada como um procedimento de reduzir uma cadeia w para o símbolo inicial da gramática. Em cada passo da redução, a subcadeia específica que possui seu lado direito casando com uma produção é substituída pelo não-terminal na cabeça dessa produção (prefixo). Essa redução é o processo inverso da derivação (AHO et al., 2008).

Reduções a partir da palavra $id * id$ utilizando a gramática

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \tag{3.17}$$

ficaria da seguinte forma:

$$\text{id} * \text{id}, \text{F} * \text{id}, \text{T} * \text{id}, \text{T} * \text{F}, \text{T}, \text{E}$$

3.4.1.3 Handle

Um handle[6] é a subpalavra mais à esquerda da produção que será substituída no processo de redução por um não-terminal.

Para a redução utilizada anteriormente

$$\text{id} * \text{id}, \text{F} * \text{id}, \text{T} * \text{id}, \text{T} * \text{F}, \text{T}, \text{E},$$

os seguintes handles existem durante o processo

Forma Sentencial à Direita	**Handle**	**Produção Utilizada na Redução**
$\text{id}_1 * \text{id}_2$	id_1	$F \rightarrow id$
$F * \text{id}_2$	F	$T \rightarrow F$
$T * \text{id}_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$E \rightarrow T * F$

.

3.4.1.4 Algoritmo de Shift-Reduce (Empilha-Reduz)

O analisador sintático shift-reduce é uma forma de análise ascendente que utiliza uma pilha com os símbolos da gramática. O último símbolo da pilha (presente na base) é o símbolo \$. O mesmo símbolo também deve estar do extremo direito da cadeia de entrada w. Inicialmente, a pilha contém apenas o símbolos \$ e a entrada é w\$.

O processo é realizado da esquerda para a direita da cadeia de entrada. Nele, o analizador sintático transfere zero ou mais símbolos da entrada para a pilha, até que uma cadeia β de símbolos da gramática presente no topo da pilha possa ser reduzida para o lado esquerdo da produção apropriada. Repete-se esse ciclo até que se detecte um erro ou até que a pilha contenha apenas o símbolo inicial da gramática.

Pode-se dividir o processo em quatro operações possíveis (AHO et al., 2008; PRICE; TOSCANI, 2001):

[6] Se a gramática for ambígua pode haver mais de um handle.

- Empilha (transfere ou shift): transfere o próximo símbolo da entrada para o topo da pilha;
- Reduz (reduce): substitui o *handle* do topo da pilha pelo não-terminal à esquerda da produção correspondente;
- Aceita (accept): anuncia o término bem-sucedido da análise;
- Erro (error): na ocorrência de erros, chamar rotina de recuperação de erro.

Considerando a gramática presente na Equação (3.17) e a palavra de entrada "id_1 * id_2", a tabela abaixo demonstra as configurações de um analisador Shift-Reduce.

Pilha	**Entrada**	**Operação**
$	id_1 * id_2 $	Empilha
$ id_1	* id_2 $	Reduz $F \rightarrow id$
$ F	* id_2 $	Reduz $T \rightarrow F$
$ T	* id_2 $	Empilha
$ T *	id_2 $	Empilha
$ T * id_2	$	Reduz $F \rightarrow id$
$ T * F	$	Reduz $T \rightarrow T * F$
$ T	$	Reduz $E \rightarrow T$
$ E	$	Aceita

Um algoritmo de Shift-Reduce pode trazer entrar numa indecisão se empilha ou reduz.

3.4.1.5 Conflitos Durante a Análise Sintática Shift-Reduce

O Shift-Reduce não pode ser aplicado a algumas gramáticas livres de contexto. Geralmente, o analisador não é capaz de decidir se empilha ou reduz (*conflito shift-reduce*), ou não consegue decidir qual das reuções disponíveis deve ser aplicada (*conflito reduce-reduce*). Essas gramáticas são chamadas de não-LR.

Um exemplo pode ser visualizado abaixo, ilustrando um conflito shift-reduce. Considere a gramática

$$\begin{aligned} &< stmt > \rightarrow if < expr > then < expr > \\ &|if < expr > then < expr > else < stmt > \\ &|other \end{aligned} \tag{3.18}$$

. Imaginando que têm-se esta configuração na pilha e na entrada

Pilha	**Entrada**
$\ldots if < expr > then < stmt >$	$else \ldots \$$

, não se poderia decidir se Empilha o $else$ ou se reduz $if < expr > then < stmt >$ a um $< stmt >$ utilizando a produção $< stmt > \rightarrow if < expr > then < expr >$.

3.4.2 Análise Sintática SLR

Essa seção apresenta o analisador sintático LR mais simples (Simple LR, ou SLR), que é um analisador sintático ascendente.

Os analisadores LR são controlados por uma tabela muito parecida com a utilizada para os analisadores LL não-recursivos. Essa tabela será descrita a seguir.

O reconhecedores LR são atraentes devido a serem:

- capazes de reconhecer praticamente todas as construções sintáticas definidas por gramáticas livres de contexto da maioria das linguagens de programação;
- método de análise shift-reduce sem retrocesso implementado com o mesmo grau de eficiência que outros métodos shift-reduce;
- capazes de detectar erros sintáticos tão logo ele aparece na entrada;
- capazes de reconhecer uma “superclasse” de gramáticas das reconhecidas por métodos preditivos ou LL.

A principal desvantagem está relacionada a dificuldade de implementação sem apoio de ferramenta geradora, pois demanda processo de programação trabalhoso.

3.4.2.1 Automato LR(0)

Para identificar quando empilhar ou quando reduzir, um analisador sintático LR decide sobre as ações shift-reduce mantendo estados para acompanhar onde se encontra a análise. Os estados representam os conjuntos de "itens". Uma produção $A \rightarrow XYZ$ gera os seguintes itens

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

. Um item indica quanto de uma produção já foi analisada em determinado ponto no processo de reconhecimento sintático.

Para criar um autômato LR(0) são utilizados os Algoritmos 8 e 10. Esse autômato será utilizado para definir as tabelas para os analisadores LR e SLR.

Dada uma gramática $G = (V, \Sigma, R, S)$, o automato LR(0) é uma quintupla $(E, \Pi, \delta : S \rightarrow S \cup \{ACEITAR\}, F, I_0)$, na qual E é um conjunto de estados, $\Pi = \Sigma \cup V \cup \{\$\}$ é o conjunto de símbolos da gramática, uma função de transição δ, um estado que entra em aceitação na transição com \$, e I_0 é o estado inicial. Cada estado em S é chamado aqui de conjunto fechado de itens (ou conjunto de itens).

Para criar esse autômato, deve-se executar o procedimento presente no Algoritmo

10. Esse algoritmo usa as funções CLOSURE (Algoritmo 8) e GOTO (Algoritmo 9).

Algoritmo 8: CLOSURE

Input : gramática $G = (V, \Sigma, R, S)$, um conjunto de itens I

$J \leftarrow I$
repeat
  foreach $A \rightarrow \alpha \cdot B\beta \in J$ **do**
    foreach $B \rightarrow \gamma \in R$ **do**
      if $B \rightarrow \cdot\gamma \notin J$ **then**
        $J \leftarrow J \cup \{B \rightarrow \cdot\gamma\}$
until *nenhum seja adicionado a J em um passo do laço*
return J

Algoritmo 9: GOTO

Input : uma gramática estendida G', um conjunto de itens I, um símbolo X da gramática

$I' \leftarrow$ selecionar subconjunto de items em I tais que $[A \rightarrow \alpha \cdot X\beta]$
if $I' \neq \{\}$ **then**
  $I'' \leftarrow \{\}$
  foreach $[A \rightarrow \alpha \cdot X\beta] \in I'$ **do**
    $I'' \leftarrow I'' \cup \{[A \rightarrow \alpha X \cdot \beta]\}$
  $I''' \leftarrow$ CLOSURE(G', I'')
  return I'''
else
  return $\{\}$

Algoritmo 10: Construção completa do autômato LR(0)

Input : gramática $G = (V, \Sigma, R, S)$

```
// Cria-se primeiramente a gramática estendida
```

$R' \leftarrow R$
$V' \leftarrow V \cup \{S'\}$
$R' \leftarrow R' \cup \{S' \rightarrow S\}$
$G' \leftarrow (V', \Sigma, R', S')$

```
// Definindo símbolos que ativam transições no autômato
```

$\Pi \leftarrow V \cup \Sigma$

```
// Definindo estado inicial
```

$I_0 \leftarrow \text{CLOSURE}(G', \{[S' \rightarrow \cdot S]\})$

```
// Construindo função de transição
```

$E \leftarrow \{I_0\}$
$C \leftarrow \{I_0\}$

```
// Criando novos estados e transições
```

repeat
  $C' \leftarrow \{\}$
  foreach $I \in C$ **do**
    foreach $X \in \Pi$ **do**
      $I' \leftarrow \text{GOTO}(G', I, X)$
      if $I' \notin E$ **then**
        $C' \leftarrow C' \cup \{I'\}$
        $E \leftarrow E \cup \{I'\}$
      definir $\delta\big((I, X)\big) \rightarrow I'$
  $C \leftarrow C'$
until $C = \{\}$

```
// Definindo transição de aceitação
```

$\Pi \leftarrow \Pi \cup \{\$\}$
F é o conjunto de itens que contém $[S' \rightarrow S\cdot]$
definir $\delta(F, \$) \rightarrow ACEITAR$
return (E, Π, δ, F, I_0)

Desafio

Para compreender como o autômato finito, inicialmente, monte um autômato LR(0) para a gramática

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned} \tag{3.19}$$

. Depois, crie uma pilha e empilhe o estado inicial do autômato. Considere que você possui duas variáveis que devem ser atualizadas constantemente. Uma delas corresponde aos símbolos da gramática sendo tratados atualmente, a qual chama-se nesse exemplo de Símbolos. A outra, chamada de Entrada, corresponde a palavra de entrada no estado atual, ou seja, sem os símbolos que já foram consumidos da esqueda para a direita.
Como exemplo, tente reconhecer a palavra de entrada $id * id\$$. Utilize as seguintes ações para isso:

1. Se o estado no topo da pilha possui transição para a palavra de entrada, consuma o símbolo da Entrada e coloque-a à direita da variável símbolo. Empilhe o estado resultante da transição;

2. Se o estado no topo da pilha não casar com a esqueda da Entrada, desempilhe-o até que haja uma redução possível, casando o conteúdo de Símbolo por uma das produções indicadas nos itens do estado atual.

3.4.2.2 Tabela SLR

Para construir uma tabela SLR, deve-se utilizar o Algoritmo 11. Para se ter um analisador SLR precisa-se do algoritmo de análise sintática LR, presente na Seção 3.4.3.

Toda tabela de análise LR consiste em duas partes principais. A parte ACTION mapeia ações da análise. A parte GOTO define a transição. A função ACTION recebe

como argumentos um estado i e um terminal a (que pode ser o delimitador \$). O valor de ACTION[i, a] pode ter um dos seguintes formatos:

- Transfere ou *Shift* j, onde j é um estado: a ação tomada pelo analisador sintático transfere a entrada a para a pilha, mas usa o estado j para representar a;
- Reduzir ou *Reduce* $A \rightarrow \beta$: a ação do analisador sintático reduz β no topo da pilha para A;
- Aceitar ou *Accept*: o analisador sintático aceita a entrada e termina a análise;
- Erro ou *Error*: O analisador sintático descobre um erro em sua entrada e passa o controle para o recuperador de erros.

3.4.3 Algoritmos de Análise Sintática LR

Um analisador sintático LR é composto por uma entrada w, uma saída, uma pilha, o algoritmo de análise sintática e uma tabela constituída de duas partes: ACTION e GOTO. Apenas a tabela muda de um analisador LR para outro. Concatena-se o símbolo \$ à direita da entrada w para que o analisador possa compreender o final da análise (AHO et al., 2008).

O Algoritmo 12 descreve o algoritmo LR. Inicialmente, o analisador sintático possui w\$ no *buffer* de entrada e 0 na pilha (estado inicial). Para que o algoritmo possa ser executado, uma tabela LR deve ser utilizada.

3.5 Análise Sintática LR Mais Poderosos

Nesta seção, veremos dois analisadores que operam com o *lookahead*, ou seja, símbolo não lido ainda na entrada (AHO et al., 2008):

- "LR canônico" ou "LR": usa a tabela construída a partir do conjunto de itens denominado LR(1);

Algoritmo 11: Construção de tabela SLR

Input : uma gramática $G = (V, \Sigma, R, S)$

```
// Obter autômato LR(0)
```

$(E, \Pi, \delta, F, I_0) \leftarrow$ executar Algoritmo 10 passando G como parâmetro

Assuma $E' = (I_0, I_1, \ldots)$ seja uma lista com os estados (ou conjunto de itens) em E, na qual I_0 seja o estado inicial

foreach $I_i \in E$ **do**

  foreach $A \rightarrow \alpha \cdot a\beta \in I_i$ **do**

    `// para a condição a seguir,` a `deve ser um terminal`

    if $A \rightarrow \alpha \cdot a\beta \in I_i \wedge a \in \Sigma \wedge \delta\big((I_i, a)\big) = I_j$ **then**

      ACTION$[i, a] \leftarrow$ *Shift* j

    else

      `/*` S `e` S' `são os não-terminais inicial da gramática` G `e de sua extensão` G' `respectivamente;` `*/`

      if $S' \rightarrow S\cdot \in I_i$ **then**

        ACTION$[i, \$\,] \leftarrow$ *Accept*

      else

        if $A \rightarrow \alpha\cdot \in I_i$ **then**

          forall $a \in$ *FOLLOW(A)* **do**

            ACTION$[i, a] \leftarrow$ *Reduce* $A \rightarrow \alpha$

    `/* Se qualquer ação de conflito resultar das regras anteriores, dizemos que a gramática não é SLR(1). Não se produz o analisador neste caso. */`

  forall $A \in V$ **do**

    if *a transição* $\delta(I_i, A) \rightarrow I_j$ *for* ***definida*** **then**

      GOTO$[i, A] \leftarrow j$

todas as regras não definidas pelo procedimento acima, caracterizam erro no reconhecimento

o estado inicial do analisador é construído a partir do conjunto de itens contendo $S' \rightarrow \cdot S$

- "Look Ahead LR" ou LALR: tabela construída a partir dos conjuntos de itens LR(0) e possui menos estados que analisadores típicos LR(1). Incorpora-se símbolos *lookahead* aos itens LR(0), permitindo trabalhar com mais gramáticas que o SLR. Não possui tabelas ACTION e GOTO maiores que o SLR.

3.5.1 Construindo Conjuntos de Itens LR(1)

A nova forma geral de um item agora é uma dupla $[A \rightarrow \alpha \cdot \beta, a]$, a qual $A \rightarrow \alpha\beta$ é uma produção e a é um terminal (ou marcador de final de entrada \$) *lookahead*. Em um

Algoritmo 12: Análise LR

```
   Input :uma cadeia w, a tabela LR com ACTION e GOTO
1  seja a o primeiro símbolo de w$
   // Pilha para os estados da tabela LR
2  P ← Pilha()
3  P.push(0)
4  while true do
5      s ← P.top()
6      if ACTION[s, a] = Shift t then
7          P.push(t)
8          seja a o próximo símbolo da entrada
9      else
10         if ACTION[s, a] = Reduce A → β then
11             for i ← 1 to |β| do
12                 P.pop()
13             t ← P.top()
14             P.push(GOTO[t, A])
15             imprima a produção A → β
16         else
17             if ACTION[s, a] = Accept then
18                 pare
                   // análise concluída
19             else
20                 chame uma rotina de recuperação de erros
```

item da forma $[A \rightarrow \alpha\cdot, a]$, é requerido uma redução segunda produção $A \rightarrow \alpha$ com o próximo símbolo da entrada sendo a. Os métodos CLOSURE (Algoritmo 13), GOTO (Algoritmo 14) e Items (Algoritmo 15) sofrem modificações.

Algoritmo 13: Closure LR(1)

```
   Input :um conjunto de itens I, uma gramática estendida G'
1  repeat
2      foreach [A → α · Bβ, a] ∈ I do
3          foreach produção B → γ ∈ G' do
4              foreach terminal b ∈FIRST(βa) do
5                  I ← I ∪ {[B → ·γ, b] no conjunto}
6  until não conseguir adicionar mais itens em I
7  return I
```

Algoritmo 14: Goto LR(1)

Input : um conjunto de itens I, não-terminal X

$J \leftarrow \{\}$
foreach *item* $[A \rightarrow \alpha \cdot X\beta, a] \in I$ **do**
    $J \leftarrow J \cup [A \rightarrow \alpha X \cdot \beta, a]$
return CLOSURE(J)

Algoritmo 15: Items LR(1)

Input : uma gramática estendida G'

$C \leftarrow$ CLOSURE($[S' \rightarrow \cdot S], \$$)
repeat
    foreach *conjunto de itens* $I \in C$ **do**
        foreach *símbolo* X *da gramática* G' **do**
            if *GOTO(*I*,* X*)*$\neq \{\}$ ***and*** *GOTO(*I*,* X*)*$\notin C$ **then**
                $C \leftarrow C \cup$GOTO(I, X)
until *não haja mais conjuntos nde itens para serem incluídos em* C
return C

3.5.2 Tabela de Análise LR(1) Canônica

O Algoritmo 16 trata da construção da tabela para o analisador LR(1) Canônico.

3.5.3 Algoritmos para Tabelas LALR

São dois algoritmos para construção das tabelas de análise LALR. O primeiro demanda muito tempo e espaço computacional (Algoritmo 17). O segundo é mais eficiente (Algoritmos 18 e 19).

Para o algoritmo eficiente, precisamos de dois processos básicos: (*i*) descobrir *lookaheads* propagados ou espontâneos; e (*ii*) com as bases LALR(1), gera-se a tabela de análise LALR(1) como se os itens fossem conjuntos de itens LR(1) (AHO et al., 2008).

3.6 Listas de Exercícios

3.6.1 Gramáticas Livres de Contexto

Crie uma Gramática Livre de Contexto para cada uma das linguagens abaixo:

Algoritmo 16: Construção da tabela LR(1) Canônica

Input : uma gramática estendida G'

Construa a coleção de conjuntos de itens $C' = \{I_0, I_1, \ldots I_n\}$ para G'
foreach $i \in \{0, 1, \ldots, n\}$ **do**
    if $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ ***and*** $GOTO(I_i, a) = I_j$ **then**
        `// a é um terminal`
        ACTION$[i,a] \leftarrow$ *Shift* j
    else
        if $[A \rightarrow \alpha\cdot, a] \in I_i$ ***and*** $A \neq S'$ **then**
            ACTION$[i,a] \leftarrow$ *Reduce* $A \rightarrow \alpha$
        else
            if $[S' \rightarrow S\cdot, \$] \in I_i$ **then**
                ACTION$[i,\$] \leftarrow$ *Accept*
            else
                `/* Se qualquer conflito resultar das regras anteriores,`
                `dizemos que a gramática não é LR(1). Então, o algoritmo`
                `falha. */`
foreach $GOTO(I_i, A) = I_j$ **do**
    GOTO$[i, A] \leftarrow j$
`// Todas entradas não definidas na tabela devem gerar erro.`
O estado inicial do analisador sintático corresponde ao conjunto de itens LR(1) que contém o item $[S' \rightarrow S, \$]$

Algoritmo 17: Construção da tabela LALR simples

Input : uma gramática estendida G'

Construa a coleção de conjuntos de itens LR(1) $C = \{I_0, I_1, \ldots I_n\}$
Para cada conjunto de mesmo núcleo em I, uní-los de acordo com o procedimento descrito em Aho et al. (2008) na página 171.
Considere que $C = \{J_0, J_1, \ldots J_m\}$ seja o conjunto resultante. Utilizar o Algoritmo 16 para construir as ações.
A tabela GOTO é construída a partir do mesmo algoritmo. Para todo estado unido K resultante de uma união, GOTO$[J,X] = K$

- $L_1 = \{a^x b^y | x = y\}$
- $L_2 = \{a^x b^y | x \leq y\}$
- $L_3 = \{a^x b^y | x \geq y\}$
- $L_4 = \{a^x b^x | x \geq 1\} \cup \{a^x b^{2x} | x \geq 1\}$
- $L_5 = \{a^x b^y c^x | x \geq 0 \wedge y \geq 0\}$

Algoritmo 18: Construção da tabela LALR eficiente: determinando *lookaheads*

Input :A base K de um conjunto I de itens LR(0) e um símbolo X da gramática

1 **foreach** *item* $A \to \alpha \cdot \beta$ *em* K **do**
2 $J \leftarrow$ CLOSURE($\{[A \to \alpha \cdot \beta, \#]\}$)
3 **if** $[B \to \gamma \cdot X\delta, a] \in J$ ***and*** $a \neq \#$ **then**
  `// conclui que o` *lookahead* a `é gerado espontaneamente para o item` $B \to \gamma X \cdot \delta$ `em GOTO(`I`,` X`).`
4 **if** $[B \to \gamma \cdot X\delta, \#] \in J$ **then**
  `// conclui que os` *lookaheads* `se propagam de` $A \to \alpha \cdot \beta$ `em` I `para` $B \to \gamma X \cdot \delta$ `em GOTO(`I`,` X`).`

Algoritmo 19: Construção da tabela LALR eficiente: bases de itens da coleção LALR(1) para G'

Input :Uma gramática estendida G'

1 Construa as bases dos conjuntos LR(0) para G (construir os conjuntos de itens LR(0)).
2 Aplique o Algoritmo 18 à base de cada conjunto de itens LR(0) e símbolo X da gramática para determinar quais *lookaheads* são gerados espontaneamente e quais são propagados para os itens de base em GOTO(I, X).
3 Inicie uma tabela que dê os *lookaheads* associados para cada item em cada conjunto de itens. Inicialmente, cada item tem associado a ele apenas os *lookaheads* que determinamos no passo da linha 2 como sendo gerados espontaneamente.
4 Faça passadas repetidas sobre os itens base em todos os conjuntos. Quando visita-se o item i, identifica-se os itens de base os quais i propaga seus *lookaheads*, usando as formulações tabuladas na linha 2. O conjunto corrente de *lookaheads* para i é acrescentado aos que já estão associados a cada um dos itens aos quais i propaga seus *lookaheads*. Continua-se fazendo passadas pelos itens de base até que não seja mais possível propagar novos *lookaheads*.

- $L_6 = \{a^x b^y c^z | x, y, z \geq 0 \wedge x = y + z\}$

3.6.2 Forma Normal Chomsky

Transforme as seguintes gramáticas em gramáticas na Forma Normal Chmonsky:

1.

$$S \to aSa|bSb|a|b|aa|bb|\epsilon$$

2.

$$S \rightarrow bA|aB$$

$$A \rightarrow bAA|aS|a$$

$$B \rightarrow aBB|bS|b|\epsilon$$

3.

$$S \rightarrow 0A0|1B1|BB$$

$$A \rightarrow C$$

$$B \rightarrow S|AC$$

$$C \rightarrow S|\epsilon$$

4.

$$S \rightarrow aAa|bBb|\epsilon|AABaC$$

$$A \rightarrow C|a$$

$$B \rightarrow C|b$$

$$C \rightarrow CD|\epsilon$$

$$D \rightarrow A|B|ab$$

3.6.3 Eliminação de Recursão à Esquerda

Crie gramáticas isentas de recursão à esquerda a partir das seguintes GLCs:

1.

$$S \rightarrow Sa|Sb|a|b|aa|bb$$

2.

$$S \rightarrow SA|SB|a$$

$$A \rightarrow AA|Ba$$

$$B \rightarrow BB|bS|bA|b$$

3.

$$S \rightarrow Aba$$

$$A \rightarrow Aaab$$

$$B \rightarrow Sa|Bc|b$$

4.

$$A \rightarrow BC|a$$

$$B \rightarrow CB|Bb|b$$

$$C \rightarrow DA|c$$

$$D \rightarrow AC|Ce|d$$

5.

$$S \rightarrow A0|B1|SB$$

$$A \rightarrow A0|1$$

$$B \rightarrow A1|c$$

$$C \rightarrow Sx|x$$

6.

$$S \rightarrow Aa|Bb|Sa$$

$$A \rightarrow CD|a$$

$$B \rightarrow CD|b$$

$$C \rightarrow CAB|BD|f$$

$$D \rightarrow A|CB|ab|\epsilon$$

3.6.4 Fatoração à Esquerda

Execute a fatoração à esquerda para as gramáticas a seguir:

1.

$$A \rightarrow abcD|aY$$

$$Y \rightarrow x$$

2.

$$A \rightarrow XYD|XYE$$

$$Y \rightarrow x$$

$$D \rightarrow d$$

$$E \rightarrow e$$

3.

$$S \rightarrow abcS|abdS|aA$$

$$A \rightarrow aS|aY$$

$$Y \rightarrow x$$

3.6.5 Gramáticas LL(1)

Construa uma tabela de análise para um analisador descendente preditivo para cada uma das gramáticas abaixo:

1.

$$P \rightarrow KVC$$

$$K \rightarrow cK|\epsilon$$

$$V \rightarrow vV|F$$

$$F \rightarrow fP;|\epsilon$$

$$C \rightarrow bVCe|com;C|\epsilon$$

CAPÍTULO 4

Análise Semântica

Uma definição dirigida por sintaxe (SDD - *Syntax-Directed Definition*) é uma gramática livre de contexto acrescida de atributos e regras. Os atributos são associados às produções. Se X é um símbolo e a um atributo de X, então escreve-se $X.a$ para denotar o atributo a do símbolo X (AHO et al., 2008).

Os atributos para os não-terminais podem ser herdados ou sintetizados (AHO et al., 2008):

- Um atributo sintetizado para um não-terminal A em um nó N da árvore de derivação é definido por uma regra semântica associada à produção naquele nó. Observe que a produção precisa ter A como sua cabeça. Um atributo sintetizado no nó N é definido apenas em termos dos valores dos atributos dos filhos de N e do próprio N;

- Um atributo herdado para um não terminal B em um nó N da árvore de derivação é definido por uma regra semântica associada à produção no pai de N. Observe que a produção precisa ter B como um símbolo em seu corpo. Um atributo herdado no nó N é definido apenas em termos dos valores dos atributos do pai de N, do próprio N e dos irmãos de N.

A SDD da Tabela 2 possui apenas atributos sintetizados. Essa tipo de SDD é chamada de S-atribuída.

Tabela 2 – SDD com atributos sintetizados apenas.

#	Produção	Regras Semânticas
1	$L \rightarrow E\mathbf{n}$	L.val = E.val
2	$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
3	$E \rightarrow T$	E.val = T.val
4	$T \rightarrow T_1 * F$	T.val = T_1.val * F.val
5	$T \rightarrow F$	T.val = F.val
6	$F \rightarrow (E)$	F.val = E.val
7	$F \rightarrow$ **digit**	F.val = **digit**.lexema

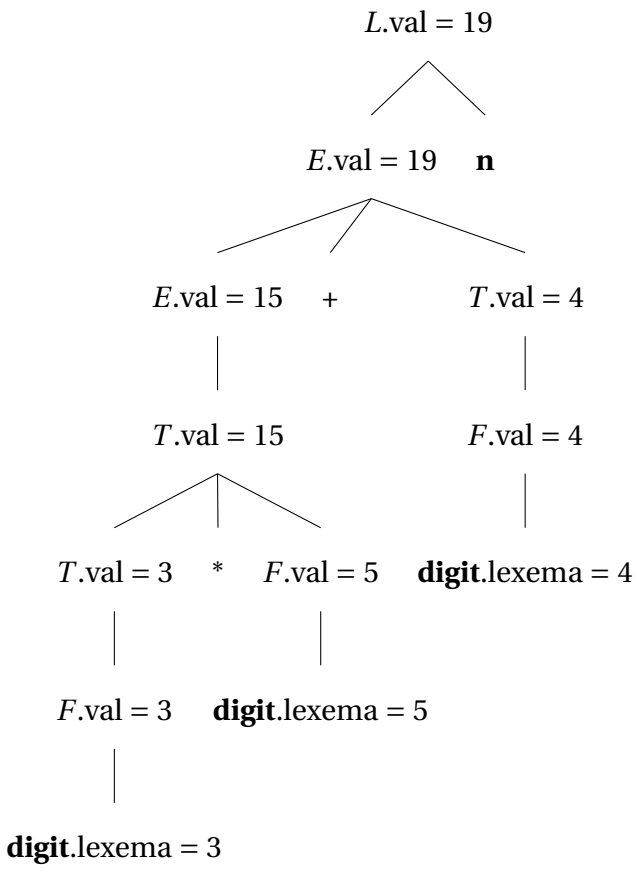
Os atributos herdados são úteis quando a árvore de derivação não casa com a sintaxe do código-fonte. A Tabela 3 demonstra um exemplo de SDD de quando a gramática foi construída para ser reconhecidade sintaticamente e não traduzida.

Tabela 3 – SDD com atributos herdados apenas.

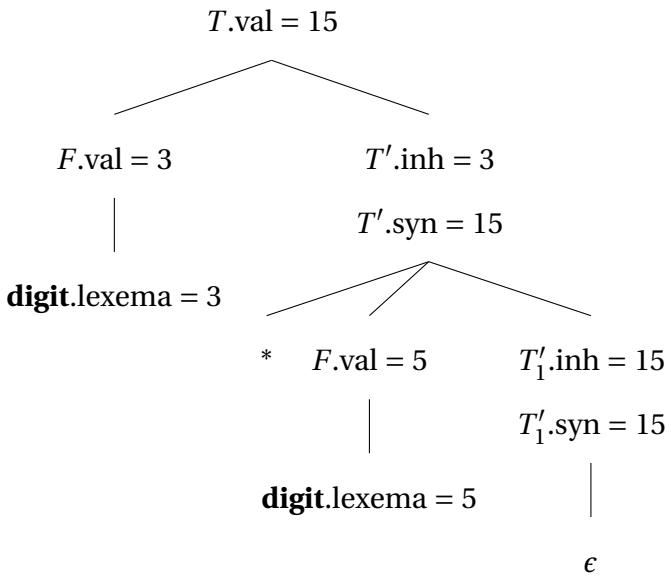
#	Produção	Regras Semânticas
1	$T \rightarrow FT'$	T'.inh = F.val
		T.val = T'.syn
2	$T' \rightarrow *FT'_1$	T'_1.inh = T'.inh × F.val
		T'.syn = T'_1.syn
3	$T' \rightarrow \epsilon$	T'.syn = T'.inh
4	$F \rightarrow$ **digit**	F.val = **digit**.lexema

4.1 Árvore de Derivação Anotada

Uma árvore de derivação anotada exibe os valores dos atributos estabelecidos na SDD. A seguinte árvore de derivação anotada demonstra a derivação da palavra $3 * 5 + 4\mathbf{n}$ na SDD da Tabela 2.



Árvore de derivação anotada para a palavra $3 * 5$ usando a SDD da Tabela 3.



4.2 Ordens de Avaliação para SDDs

Enquanto uma árvore de derivação anotada mostra os valores dos atributos, um grafo de dependência ajuda a determinar como esses valores podem ser avaliados. Ele representa o fluxo de informações entre as instâncias dos atributos em uma árvore de derivação. Um arco de uma instância de atributo para outra significa que o valor da primeira é necessário para calcular a segunda. Os arcos expressam restrições impostas pelas regras semânticas (AHO et al., 2008). A seguir, há o processo da criação do grafo de dependência em detalhes:

- Para cada nó da árvore de derivação (rotulado pelo símbolo X da gramática), o grafo de dependência possui um nó para cada atributo associado a X.

- Suponha que uma regra semântica associada a uma produção-p defina o valor do atributo sintetizado $A.b$ em termos do valor de $X.c$. Então, o grafo de dependência possui um arco de $X.c$ para $A.b$. Mais precisamente, em todo nó N rotulado com A onde a produção-p é aplicada, crie um arco para o atributo b em N, a partir do atributo c no filho de N correspondendo a essa instância do símbolo X no corpo da produção.

- Suponha que uma regra semântica associada a uma produção-p defina o valor do atributo herdado $B.c$ em termos do valor de $X.a$. Então, o grafo de dependência tem um arco de $X.a$ para $B.c$. Para cada nó N rotulado com B que corresponda a uma ocorrência desse B no corpo da produção-p, crie um arco para o atributo c em N a partir do atributo a no nó M que corresponda a essa ocorrência em X. Observe que M poderia ser o pai ou um irmão de N.

Uma ordenação topológica do grafo é toda a sequência de nós $N_1, N_2, \ldots, N_k$ que possuem um arco de dependência de N_i para N_j, tal que $i < j$. Se houver qualquer ciclo não haverá ordenações topológicas. Na prática, as traduções são implementadas usando-se classes de SDD, pois é muito difícil saber se há ciclos no grafo.

4.2.1 Definições S-Atribuídas

Uma SDD é S-atribuída se todo atributo é sintetizado. Tanto essa classe quanto as L-Atribuídas podem ser implementadas eficientemente em conexão com métodos de análise sintática descendente ou ascendente. Quando isso acontece, pode-se avaliar os atributos em qualquer ordem ascendente dos nodos na árvore de derivação. Avalia-se os atributos em um caminhamento pós-ordem a partir da raiz N:

1. Para cada filho C_i de N, execute a pós-ordem sobre C_i;

2. Avalia atributos associados aos nodos de N.

As definições S-Atribuídas podem ser implementadas por um analisador sintática ascendente visto que utiliza-se um caminhamento pós-ordem.

4.2.2 Definições L-Atribuídas

A outra classe é a de "Definições L-atribuídas". A ideia é que os arcos do grafo de dependência podem ser direcionados da esquerda para direita entre os atributos associados ao corpo de uma produção, mas não da direita para esquerda (por isso o nome L-atribuídas). Mais precisamente, cada atributo precisa ser:

1. Sintetizado; ou

2. Herdado, mas com regras limitadas como segue. Suponha que exista uma produção $A \rightarrow X_1 X_2 \ldots X_n$, e que exista um atributo herdado $X_i.a$, calculado por uma regra associada a essa produção. Então, a regra pode usar apenas:

   a) Os atributos herdados associados à cabeça A.

   b) Os atributos herdados ou sintetizados associados às ocorrências dos símbolos $X_1, X_2, \ldots, X_{i-1}$ localizados à esquerda de X_i.

   c) Os atributos herdados ou sintetizados associados à ocorrência do próprio X_i, desde que não existam ciclos em um grafo de dependência formado pelos atributos dese X_i.

4.3 Construção de Árvores de Sintaxe

Exemplo de uma SDD S-atribuída e a montagem de sua árvore.

#	Produção	Regras Semânticas
1	$E \rightarrow E_1 + T$	E.node = **new** Node(+, E_1.node, T.node)
2	$E \rightarrow E_1 - T$	E.node = **new** Node(−, E_1.node, T.node)
3	$E \rightarrow T$	E.node = T.node
4	$T \rightarrow (E)$	T.node = E.node
5	$T \rightarrow id$	T.node = **new** Leaf(id, id.entry)
6	$T \rightarrow num$	T.node = **new** Leaf(num, num.val)

Exemplo de uma SDD L-atribuída e a montagem de sua árvore.

#	Produção	Regras Semânticas
1	$E \rightarrow TE'$	E.node = E'.syn
		E'.inh = T.node
2	$E' \rightarrow +TE'_1$	E'_1.inh = **new** Node(+, E'.inh, T.node)
		E'.syn = E'_1.syn
3	$E' \rightarrow -TE'_1$	E'_1.inh = **new** Node(−, E'.inh, T.node)
		E'.syn = E'_1.syn
4	$E' \rightarrow \epsilon$	E'.syn = E'.inh
5	$T \rightarrow (E)$	T.node = E.node
6	$T \rightarrow id$	T.node = **new** Leaf(id, id.entry)
7	$T \rightarrow num$	T.node = **new** Leaf(num, num.val)

4.4 Esquemas de Tradução Dirigidos por Sintaxe

Os esquemas de tradução dirigidos por sintaxe (SDT) são uma notação para as definições dirigidas por sintaxe. Ela é uma GLC com fragmentos de programa (ações semânticas) incorporados em suas traduções. Por convenção, as ações aparecem entre chaves (AHO et al., 2008).

Um SDT pode ser implementado construindo-se a árvore de derivação e executando as ações em um caminhamento pré-ordem da esquerda para a direita. É comum não implementar a árvore de derivação durante a implementação da SDT na análise.

Aho et al. (2008) demonstram como implementar duas classes importantes de SDDs:

- GLC é analisável por um LR e a SDD é S-atribuída;
- GLC é analisável por um LL e a SDD é L-atribuída.

A seguir, é demonstrado como uma SDD pode ser convertida em uma SDT com ações que são executadas no momento correto. Durante a análise, uma ação semântica é executada logo depois de todos os símbolos à esquerda tiverem sido casados.

4.4.1 SDTs pós-fixados

Quando é possível analisar a gramática de baixo para cima (Bottom-Up, LR) e a SDD é S-atribuída, o SDT é construído adicionando as ações semânticas no fim de cada produção (extremo direito do corpo da produção). Essas SDTs são chamadas de SDTs pós-fixadas.

Para implementar um STD pós-fixado, Aho et al. (2008) utiliza um analisador sintático baseado em pilha. As implementações devem acontecer no momento em que ocorrem as reduções em um analisador LR. Os atributos de cada símbolo da gramática podem ser colocados na pilha para que sejam facilmente encontrados durante o processo de redução. Esses atributos poderiam estar em registros da própria pilha.

Exemplo de SDT pós-fixado implementando uma calculadora de mesa.

1	$L \rightarrow E\mathbf{n}$	{ print(E.val); }
2	$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val; }
3	$E \rightarrow T$	{ E.val = T.val; }
4	$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val; }
5	$T \rightarrow F$	{ T.val = F.val; }
6	$F \rightarrow (E)$	{ F.val = E.val; }
7	$F \rightarrow$ **digit**	{ F.val = **digit**.lexema; }

Exemplo de SDT acima, em um analisador ascendente usando uma pilha.

1	$L \rightarrow E\mathbf{n}$	{ pilha.pop(); print(pilha.pop()); }
2	$E \rightarrow E_1 + T$	{ x = pilha.pop(); pilha.pop(); y = pilha.pop(); pilha.push(x+y); }
3	$E \rightarrow T$	
4	$T \rightarrow T_1 * F$	{ x = pilha.pop(); pilha.pop(); y = pilha.pop(); pilha.push(x*y); }
5	$T \rightarrow F$	
6	$F \rightarrow (E)$	{ pilha.pop(); x = pilha.pop(); pilha.pop(); pilha.push(x); }
7	$F \rightarrow$ **digit**	

4.4.2 SDTs com Ações Inseridas nas Produções

Em uma SDT com Ações Inseridas nas Produções[1], uma ação semântica pode ser adicionada em qualquer posição dentro do corpo da produção. Ela é processada logo após todos os símbolos da esquerda serem processados. Ao analisar uma produção $B \rightarrow X\{a\}Y$, a ação de a é realizada logo depois de reconhecer X, se X for um terminal, ou todos os terminais derivados de X, se X for um não-terminal. Se o analiador for LR, então efetua-se a ação a assim que X aparecer no topo da pilha sintática. Se o analisador for LL, então efetua-se a ação a imediatamente antes de tentar expandir a ocorrência de Y, se esse for um não-terminal, ou verificar-se Y, se esse for um terminal.

[1] As limitações a um SDT com Ações Inseridas nas Produções podem ser lidas em Aho et al. (2008) na página 209.

De acordo com Aho et al. (2008), qualquer SDT pode ser implementado da seguinte forma:

1. Ignorando-se as ações semânticas, analisa-se a entrada e constrói-se uma árvore de derivação;

2. Examina-se o interior de cada nodo N, como por exemplo o para a produção $A \rightarrow \alpha$. Adicione um nodo filho a N para as ações em α, de modo que os filhos de N da esquerda para a direita tenham a exatamente os símbolos e ações de α.

3. Faz-se um caminho pré-ordem na árvore e realize a ação semântica logo que um nodo marcado com uma ação seja visitado.

4.4.3 Eliminando Recursões à Esquerda

Ao eliminar recursões à esquerda em uma GLC de uma SDT, deve-se tomar alguns cuidados. Segue o procedimento em dois momentos. Antes:

$$\begin{aligned} A &\rightarrow A_1 Y \ \{\mathbf{A.a} = \mathbf{g}(\mathbf{A_1.a}, \mathbf{Y.y})\} \\ A &\rightarrow X \ \{\mathbf{A.a} = \mathbf{f}(\mathbf{X.x})\} \end{aligned} \tag{4.1}$$

. Depois:

$$\begin{aligned} A &\rightarrow X \ \{\mathbf{R.i} = \mathbf{f}(\mathbf{X.x})\} \ R \ \{\mathbf{A.a} = \mathbf{R.s}\} \\ R &\rightarrow Y \ \{\mathbf{R_1.i} = \mathbf{g}(\mathbf{R.i}, \mathbf{Y.y})\} \ R_1 \ \{\mathbf{R.s} = \mathbf{R_1.s}\} \\ R &\rightarrow \epsilon \ \{\mathbf{R.s} = \mathbf{R.i}\} \end{aligned} \tag{4.2}$$

.

4.4.4 SDTs para Definições L-Atribuídas

De acordo com Aho et al. (2008), como em qualquer gramática, a técnica pode ser implementada anexando-se as ações semânticas a uma árvore de derivação executando-as durante o caminhamento pré-ordem. As regras para uma SDD L-atribuída são:

1. Incorpora-se a ação que avalia os atributos herdados para um não-terminal A imediatamente antes dessa ocorrência de A no corpo da produção. Se vários atributos herdados pra A dependerem um do outro no modo não-cíclico, ordena-se a avaliação de modo que aqueles que são necessários primeiro sejam os primeiros calculados.

2. Coloque as ações que avaliam o atributo sintetizado da cabeça de uma produção no fim do corpo dessa produção.

Exemplo de SDD L-atribuída para um comando *while*:

$S \rightarrow$ **while** (C) S_1	L1 = new();
	L2 = new();
	S_1.next = L1;
	C.false = S.next;
	C.true = L2;
	S.code = **label** || L1 || C.code || **label** || L2 || S_1.code;

. Para esse exemplo, os seguintes atributos são utilizados para gerar o código intermidiário correto:

- S.next: esse atributo herdado é um **label** o início do código que deve ser executado quando S terminar;

- S.code: esse atributo sintetizado representa a sequência de passos de código intermediário que implementa um comando S e termina com um desvio para S.next;

- C.true: esse atributo herdado é um **label** para o início do código que precisa ser executado, se C for verdadeiro;

- C.false: esse atributo herdado é um **label** para o início do código que precisa ser executado, se C for falso;

- C.code: esse atributo sintetizado define a sequência de passos de código intermediário que implementa a expressão condicional C e desvia para C.true ou para C.false, dependendo de C ser verdadeiro ou falso.

O SDT equivalente é:

$S \rightarrow$ **while** (	{ L1 = new(); L2 = new(); C.false = S.next; C.true=L2; }
C)	{ S_1.next = L1; }
S_1	{ S.code = **label** || L1 || C.code || **label** || L2 || S_1.code; }

.

Agradecimentos

Agradeço o apoio dos professores e colegas de departamento Jerusa Marchi e Ricardo Azambuja Silveira. Parte desse material foi inspirado no que foi gentilmente cedido por esses professores.

Referências

AHO, A. V. et al. *Compiladores: princípios, técnicas e ferramentas*. Porto Alegre, Brazil: Pearson, 2008. Citado 35 vezes nas páginas 9, 10, 12, 14, 17, 18, 19, 26, 29, 32, 34, 39, 40, 49, 50, 55, 57, 62, 63, 64, 65, 66, 67, 70, 72, 73, 80, 83, 84, 89, 92, 94, 95, 96 e 97.

DELAMARO, M. E. *Como Construir um Compilador: utilizando ferramenta Java*. São Paulo, Brazil: Novatec Editora Ltda, 2004. Citado 4 vezes nas páginas 19, 105, 108 e 111.

PRICE, A.; TOSCANI, S. S. *Compiladores: princípios, técnicas e ferramentas*. Porto Alegre, Brazil: Sagra Luzzato, 2001. Citado 2 vezes nas páginas 19 e 73.

SIPSER, M. *Introdução à Teoria da Computação*. São Paulo, Brazil: Cengage Learning, 2007. Citado 5 vezes nas páginas 6, 19, 51, 52 e 58.

APÊNDICE A

JavaCC

JavaCC é um gerador de analisador sintático (DELAMARO, 2004). Realizada a especificação de uma linguagem de programação de acordo com as convenções do JavaCC, pode-se executá-lo para gerar os códigos-fonte do analisador sintático na linguagem Java.

Apesar do JavaCC facilitar o árduo trabalho de construir um analisador sintático para uma linguagem de programação, o programador deve ainda implementar diversos outros aspectos, caso o objetivo final seja gerar um compilador/interpretador. Um deles é definir a tabela de símbolos.

As definições principais da linguagem devem ficar em um arquivo na extensão ".jj", mas é possível importar pacotes e classes Java para auxiliar no processo.

A.1 A seção OPTIONS

O Java CC permite fornecer argumentos para configurar o analisador sintático a ser gerado. Esses argumentos podem ser definidos na seção "OPTIONS". São alguns exemplos (DELAMARO, 2004):

- LOOKAHEAD = *n*: estabelece o valor global de *lookahead* como *n*;

- STATIC = true/false: indica se o analisador sintático gerado terá seus membros estáticos ou não. Com uma classe não estática, podem ser criados varios objetos da classe do analisador sintático;
- DEBUG_PARSER = true/false: indica se deve ser gerado código que emite mensagens sobre o funcionamento do analisador sintático, como métodos chamados e não-terminais reconhecidos;
- DEBUG_TOKEN_MANAGER = true / false: idem para o analisador léxico;
- UNICODE_INPUT = true /false: indica se o analisador léxico deve aceitar código UNICODE;
- IGNORE_CASE = true/false: indica se deve ignorar diferença entre letras minúsculas e maiúsculas.

Exemplo de uso:

```
1 options {
2     STATIC = false;
3 }
```

A.2 Definição de Tokens

Uma seção de definição de tokens começa com a palavra “TOKEN” seguida de dois pontos e o escopo. Podem ser abertas inúmeras seções “TOKEN”. Em cada uma delas, pode-se declarar uma lista de tokens separados pelo símbolo “|”. Cada token é definido entre os símbolos “<” e “>”. Entre esses símbolos, deve-se identificar o nome do token, seguido por um símbolo “:” e a definição regular.

```
1 TOKEN:
2 {
3     < CLASS: "class">
```

```
    |
    < constante_string: "\""(~["\"","\n", "\r"])* "\"" >
}
```

Nas definições léxicas, pode-se inserir padrões que serão ignorados pelo seu compilador/interpretador. Para isso, usa-se seções “SKIP”. Abaixo encontram-se os exemplos para ignorar comentários de linha

```
SKIP:
{
                "/*": multilinecomment
}
<multilinecomment> SKIP:
{
        "*/": DEFAULT | <~[]>
}
```

e comentários de múltiplas linhas

```
SKIP:
{
        "//": singlelinecomment
}
<singlelinecomment> SKIP:
{
        <["\n", "\r"]> :DEFAULT | <~[]>
}
```

.

O programador deve definir a classe que deverá abrigar a análise sintática. A classe deve estar entra “PARSER_BEGIN(·)” e “PARSER_END(·)” e o nome da classe deve aparecer entre os parênteses.

```
1  PARSER_BEGIN(MinhaClasseSintatica)
2  import java.io.*;
3
4  public class MinhaClasseSintatica{
5          ...
6          public static void main(String[] args) throws ParseException{
7              ...
8          }
9          ...
10 }
11 PARSER_END(MinhaClasseSintatica)
```

A.3 A Classe *Token*

Quando ocorre o casamento de uma cadeia com um dos tokens definidos na linguagem especificada no arquivo ".jj", produz-se um objeto *Token* com os seguintes atributos (DELAMARO, 2004):

- *int kind*: define o tipo do token reconhecido, com os nomes dos tokens definidos no arquivo ".jj" e gerados no arquivo "...Constants.java";
- *int beginLine, beginColumn, endLine, endColumn*: indicam linha e coluna dentro do arquivo de entrada onde se inicia determinado token;
- *String image*: a cadeia que corresponde ao token;
- *Token next*: referência do objeto correspondente ao próximo token reconhecido;
- *Token specialToken*: apontador referente ao último token especial reconhecido (definido em uma seção "SPECIAL_TOKEN".

A.4 Erros Léxicos

A variável “countLexError” conta quantos erros léxicos ocorreram.

```
1 TOKEN_MGR_DECLS:
2 {
3         int countLexError = 0;
4         public int foundLexError(){
5                 return countLexError;
6         }
7 }
```

Quando for identificado um token incorreto a variável é incrementada, como no exemplo abaixo:

```
1  SPECIAL_TOKEN:
2  {
3          <INVALID_TOKEN: (~[
4                                                  "0"-"9", "a"-"z",
                                                      "A"-"Z", "\"", "(",
                                                      ")",
5                                                  "[", "]", "{", "}", ";",
                                                      ",", ".", "=", ">",
6                                                  "<", "!", "+", "-", "*",
                                                      "/", "%", " ", "\t",
7                                                  "\n", "\r", "\f"
8                                          ])
9          >
10         {
11                 System.err.println(     "Linha "+
                       input_stream.getEndLine()+
```

```
12                                                          " - palavra
                                                              invÃ¡lida
                                                              encontrada:
                                                              "+image);
13              countLexError++;
14      }
15      |
16      <INVALID_CONSTANT:
17      "\"" (~["\n", "\r", "\""])* ["\n", "\r"]>
18      {
19              System.err.println(     "Linha "+
                    input_stream.getEndLine()+ " - palavra constante tem
                    uma quebra de linha: "+image);
20              countLexError++;
21      }
22  }
```

A.5 Imprimindo os Tokens de um Analisador Léxico

Adicione ao arquivo ".jj" a seguinte função para adicionar que na análise léxica, exiba-se todos os tokens do programa de entrada. Substitua <???> pelo nome da classe principal do compilador/interpretador.

```
1  JAVACODE void program()
2  {
3      Token t;
4      do{
5          t = getNextToken();
6          Token st = t;
```

```
7         while ( st.specialToken != null)
8             st = st.specialToken;
9         do{
10            if ( Menosshort )
11                System.out.println( st.image + "   " +
                                                          im(st.kind) +
                      " " +
12                                    st.kind);
13            else
14                System.out.println( "Line: " + st.beginLine +
15                                    " Column: " + st.beginColumn +
16                                    " " + st.image +
17                                    "   " + im(st.kind) + "   "+t.kind);
18            st = st.next;
19        }while (st != t.next);
20    }while (t.kind != <???>Constants.EOF);
21 }
```

A.6 Análise Sintática

O JavaCC implementa a linguagem através da técnica de análise descendente recursiva, na qual cada não-terminal é implementado por um método (DELAMARO, 2004).

Quando há possíveis produções a serem seguidas, o analisador sintático produzido pelo JavaCC pega a primeira que casar com a entrada. O programador pode alterar esse comportamento utilizando o comando LOOKAHEAD.

APÊNDICE B

Linguagem X+++

B.1 Analisador Sintático

B.1.1 Símbolos Ignoráveis

- Espaço vazio;
- Tabulação;
- Quebra de linha;
- Retorno do carro;

B.1.2 Palavras Reservadas

- **break**
- **class**
- **constructor**
- **else**
- **extends**

- **for**
- **if**
- **int**
- **new**
- **print**
- **read**
- **return**
- **string**
- **super**

B.1.3 Comentários

- Única linha: **//Meu comentário!**
- Múltiplas linhas: **/*Meu comentário em múltiplas linhas. Meu comentário em múltiplas linhas. Meu comentário em múltiplas linhas. Meu comentário em múltiplas linhas. Meu comentário em múltiplas linhas. Meu comentário em múltiplas linhas. */**

B.1.4 Constantes

- inteiros na base binária, por exemplo: **01011b** ou **01011B**;
- inteiros na base octal, por exemplo: **7541o** ou **7541O**;
- inteiros na base decimal, por exemplo: **2019**;
- inteiros na base hexadecimal, por exemplo: **10ABAC0h** ou **10ABAC0H** ou **10abac0H** ou **10abac0h**;

- texto ou sequência de símbolos entre aspas duplas, por exemplo: **"Esse é 1º exemplo de um texto!!!"**
- constante nula: **null**.

B.1.5 Identificadores

- Nomes de variáveis, métodos e classes, define-se que deve-se começar com uma letra, seguido de letras ou números nenhuma ou várias vezes. As letras podem ser maiúsculas ou minúsculas.

B.1.6 Símbolos especiais

- Gerais: "(", ")", "{", "}", "[", "]", ";", ",", ".";
- Atribuição: =
- Comparações: >, <, <=, >=, ==, ! =
- Operadores aritméticos: +, −, /, ∗, %[1]

[1] Resto da divisão.