UNISUL - UNIVERSIDADE DO SUL DE SANTA CATARINA

DEPARTAMENTO DE CIÊNCIAS TECNOLÓGICAS

CURSO DE CIÊNCIA DA COMPUTAÇÃO

Prof. Roberto M. Scheffel

Apostila

Linguagens Formais e Autômatos

Sumário

Introdução

Este texto provê os elementos essenciais da disciplina de "**Compiladores I**". O capítulo I situa a área de linguagens formais no contexto da teoria da computação. Em seguida trata de alguns aspectos relacionados à construção de compiladores.

O capítulo II apresenta algumas definições básicas de linguagens formais e uma breve introdução às gramáticas gerativas, que serão base para o restante da apostila.

O capítulo III apresenta a teoria de linguagens regulares e das máquinas de estados finitos, ou autômatos finitos, que serão utilizados como base para a construção de analisadores léxicos.

O capítulo IV apresenta a teoria especifica de gramáticas livres de contexto, que são importantes ferramentas para a construção de analisadores sintáticos de compiladores.

O capítulo V apresenta alguns conceitos relativos a autômatos de pilha, que são utilizados para analisar sentenças geradas pelas gramáticas livres de contexto.

Finalmente, o capítulo VI apresenta o processo de análise sintática propriamente dito.

A partir do capítulo II, ao final de cada capítulo são propostos alguns exercícios de fixação. O exercício cujo enunciado estiver marcado com um asterisco (*) tem sua resolução no final da apostila.

O material apresentado nesta apostila cobre o conteúdo básico da disciplina. Maiores subsídios podem ser encontrados na bibliografia indicada no final da apostila.

I - Teoria da Computação

1. Introdução

Teoria de Linguagens Formais e Teoria de Máquina são tópicos abrangentes que se inserem no estudo da Teoria da Computação em geral. A Teoria da Computação é uma ciência que procura organizar o conhecimento formal relativo aos processo de computação, como complexidade de algoritmos, linguagens formais, problemas intratáveis, etc.

Atualmente, o termo "computar" está associado ao conceito de fazer cálculos ou aritmética, usando para isso máquinas computadoras. Entretanto, a noção de computabilidade pode ser dissociada da implementação física da máquina. Originalmente, a palavra latina *putare* significa "pensar", mas no caso da teoria da computação o sentido mais adequado seria algo como "manipulação de símbolos", o que não é, necessariamente, pensamento. Essa manipulação de símbolos envolvendo, principalmente, letras, números e proposições algébricas permitiria às máquinas computadoras, segundo Alan Turing (1939) realizar qualquer operação formal de que o ser humano seria capaz.

Mesmo assim, alguns filósofos sustentam que os computadores não podem computar (fazer aritmética), porque não compreendem a noção de número (apenas manipulam sinais elétricos). Todavia, pode-se falar, sem entrar no mérito da questão, que os computadores realizam uma "computação perceptível", já que transformam uma entrada em saída como se estivessem realmente computando. Neste caso, o que importa é o efeito final do processo e não a maneira como ele é feito.

2. Algoritmos e Decidibilidade

Os algoritmos surgiram na história da humanidade possivelmente com as tábuas de logaritmos e de números primos, na antigüidade. Certos algoritmos também eram usados para calcular a posição dos planetas e outros auxiliavam a navegação. Pode-se dizer que os computadores surgiram graças aos algoritmos. Por exemplo: Charles Babbage (1820), considerado o "pai da Computação", irritou-se com a maneira mecânica e repetitiva com que os cálculos da posição de estrelas e planetas eram feitos na Royal Astronomical Society, onde trabalhava, e escreveu o artigo "Observations on the Application of Machinery to the Computatiuon of Mathematical Tables", onde propunha que estes algoritmos fossem realizados por máquinas.

Mas foi apenas em 1900 que o matemático Frege chegou à conclusão de que se precisava de uma noção precisa e concisa do conceito de algoritmo. Frege procurou usar a lógica como ferramenta para estabelecer os principais algoritmos de matemática. A notação utilizada realmente era precisa mas verificou-se pouco depois que o trabalho continha uma contradição, que ficou conhecida como paradoxo de Russel.

O Paradoxo de Russel

Muitos problemas até hoje não possuem solução algorítmica, isto é, não podem ser resolvidos por máquinas. Destes problemas, alguns não têm solução conhecida, mas existe um grupo de problemas dos quais sabe-se que não existem algoritmos formais que possam resolvê-los. Em muitos casos, o problema é apresentado como uma simples questão, que deve ser respondida com *sim* ou *não*. Posto o problema desta forma, se existe um algoritmo que sempre responda *sim* ou *não* para qualquer entrada, então o problema é decidível. Em alguns casos, porém, provou-se que não existem nem poderão existir estes algoritmos. Tais problemas são chamados, então, de indecidíveis, ou parcialmente decidíveis, se pelo menos uma das respostas *sim* ou *não* puder ser encontrada.

Gödel, em 1931, provou que a aritmética elementar é indecidível, ou seja, não existe um algoritmo para provar a verdade ou falsidade de qualquer proposição da aritmética. Hilbert, em

1925, propôs vários problemas que se supunha fossem indecidíveis, dentre eles, por exemplo, o de dizer se o polinômio $P(x_1,x_2,...,x_n)=0$ tem solução inteira. Apenas em 1970 Matijacevic provou que não existe nenhum procedimento efetivo para decidir esta questão.

Em termos de linguagem, pode-se dizer que uma linguagem é decidível se dada uma linguagem *L* e uma sentença *w* for possível decidir algoritmicamente se w é uma sentença de L. Para as linguagens não-decidíveis, pode-se esperar que para algum w, a pergunta "$w \in L$?" não será respondida, porque o algoritmo que procura pela resposta não parará.

Um algoritmo é uma descrição finita de um processo de computação; é um texto finito, escrito em um linguagem algorítmica na qual cada sentença tem um significado não-ambíguo.

A palavra *algoritmo* é uma corruptela da *Al-Khuwarizmi*, sobrenome de um matemático persa do século IX. O algoritmo deve poder ser resolvido em tempo finito (deve ter parada garantida). Um *procedimento* não termina necessariamente. Assim, existem problemas que possuem procedimentos para serem resolvidos, mas não possuem algoritmos. Em outras palavras, existem um procedimento de procura de solução, que pode chegar a uma resposta em um tempo finito, ou então pode não chegar a uma resposta em momento algum, ao contrário do algoritmo, que sempre chega à uma resposta (afirmativa ou negativa) num tempo finito.

Decidir se um problema possui um algoritmo ou simplesmente um procedimento é o principal problema indecidível da teoria da computação, conhecido como "Problema da Parada". Outra forma de enunciar o problema da parada é a seguinte: "Dado um programa P e um conjunto de dados d, responda sim se P(d) pára e não caso contrário".

A Tese de Church

Alguns dos mais importantes modelos formais de computabilidade existentes são a Máquina de Turing, as gramáticas do Tipo 0 e as Funções Recursivas Parciais. A Tese de Church (1936) estabelece que: "Embora todas estas caracterizações sejam diferentes na forma, elas caracterizam o mesmo conjunto de funções". No caso, Church provou que era possível definir funções de tradução de representações de um formalismo para qualquer um dos outros, donde se concluiu que tudo o que é expressável em um deles pode ser expresso em qualquer um dos outros. Isto garante que todos os modelos formais de computação têm o mesmo poder expressivo.

Como as Funções Recursivas Parciais são suficientemente inclusivas em relação à computação aparente, e a caracterização da Máquina de Turing demonstra o aspecto mecânico do processamento algorítmico, estes formalismos são aceitos como capazes de expressar qualquer função ou algoritmo que possa ser formalmente expresso de alguma maneira.

3. Introdução a Compiladores

3.1 Processadores de Linguagem

Um processador de linguagem é um programa que permite ao computador "entender" os comandos de alto nível fornecidos pelos usuários.

Existem duas espécies principais de processadores de linguagem: os *interpretadores* e os *tradutores*.

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

Um tradutor é um programa que aceita como entrada um programa escrito em uma linguagem fonte e produz como saída uma outra versão do programa escrito em uma *linguagem objeto*. Muitas vezes a linguagem objeto é a própria linguagem de máquina do computador. Neste caso o programa objeto pode ser diretamente executado pela máquina.

Tradutores são arbitrariamente divididos em montadores e compiladores, os quais traduzem linguagens de baixo nível e de alto nível, respectivamente. Além disso, há os *pré-processadores* que traduzem uma linguagem de alto nível em outra linguagem de alto nível.

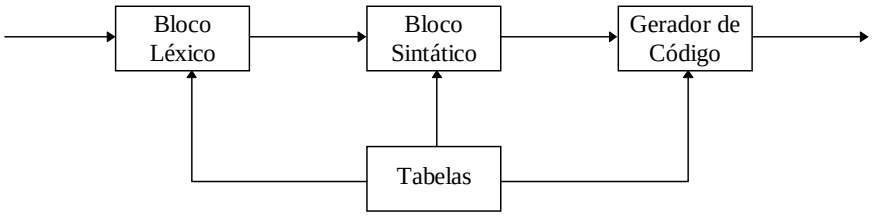
O fundamento matemático da teoria de processamento de linguagens é a teoria de autômatos e linguagens formais.

3.2. Modelo Simples de Compilador

O objetivo de um compilador é traduzir as seqüências de caracteres que representam o programa fonte em instruções de máquina que incorporam a intenção do programador. Esta tarefa é complexa o suficiente para poder ser vista como uma estrutura formada por processos menores interconectados.

Em um modelo simples de compilador o trabalho pode ser feito por uma interconexão serial entre três blocos, chamados: *bloco léxico, bloco sintático* e *gerador de código*.

Os três blocos têm acesso a tabelas de informações globais sobre o programa fonte. Uma destas tabelas é, por exemplo, a *tabela de símbolos*, na qual a informação sobre cada variável ou identificador utilizado no programa fonte é armazenada. A conexão entre estes blocos e tabelas é mostrada na figura seguinte:



3.2.1. Bloco Léxico

A entrada para um compilador é a cadeia de caracteres quer representa o programa fonte. O objetivo do bloco léxico é quebrar esta cadeia de caracteres separando as palavras que nela estão representadas. Por exemplo, a cadeia de entrada poderia ser:

```
RESULTADO:=RESULTADO+15
```

Neste caso, o bloco léxico deve ser capaz de discernir que esta cadeia contém o nome de uma variável `RESULTADO`, seguido de um comando de atribuição `:=`, seguido novamente de `RESULTADO`, seguido de uma operação aritmética `+` e finalizado por uma constante numérica `15`. Assim, os 23 caracteres da cadeia de entrada são transformados em 5 novas entidades. Estas entidades são freqüentemente denominadas de *tokens*.

Um token consiste de uma par ordenado Classe x Valor. A *classe* indica que o token pertence a uma classe de um conjunto finito de classes possíveis e indica a natureza da informação contida em *valor*.

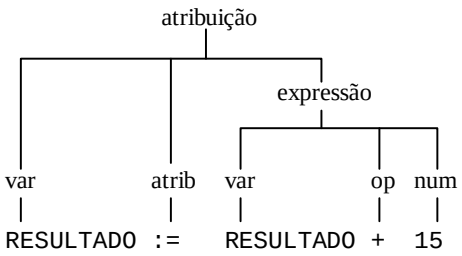
Voltando ao exemplo, a variável `RESULTADO` poderia pertencer à classe *variável* e seu valor seria um ponteiro para a entrada `RESULTADO` na tabela de símbolos. O token `15` poderia pertencer à classe *constante* e ter valor correspondente ao número inteiro 15. Já os tokens `:=` e `+` poderiam pertencer à classes *símbolos especiais*.

Se a tabela de símbolos for vista como um dicionário, então o processo léxico pode ser visto como o ato de agrupar letras em palavras e determinar suas localizações no dicionário.

Outras funções normalmente atribuídas ao bloco léxico são as de ignorar espaços em branco e comentários, além de detectar os erros léxicos.

3.2.2. Bloco Sintático

O bloco sintático agrupa os tokens fornecidos pelo bloco léxico em estruturas sintáticas, verificando se a sintaxe da linguagem foi obedecida. Tomando novamente o exemplo anterior, os 5 tokens gerados pelo bloco léxico poderiam ser agrupados na estrutura sintática *atribuição*, representada por uma *árvore sintática*, da seguinte forma:



Note que a árvore exige que para a atribuição apareça uma variável, seguida de uma atribuição e de uma expressão. Desta forma, uma entrada do tipo

```
15:=RESULTADO+10
```

passaria no bloco léxico, pois as tokens estão escritos da *maneira correta*, porém não passaria no bloco sintático, pois não estão escritos na *ordem correta*.

3.2.3. Gerador de Código

A função do gerador de código é expandir os átomos gerados pelo bloco sintático em uma seqüência de instruções do computador que executam a intenção do programa fonte. Consiste basicamente no *processamento semântico* e na *otimização de código*.

Processamento Semântico

Certas atividades relacionados ao *significado* dos símbolos são freqüentemente classificadas como processamento semântico. Por exemplo, a semântica de um identificador pode incluir o seu tipo, e, se for um array, por exemplo, seu tamanho. O processamento também pode incluir o preenchimento da tabela de símbolos com as propriedades dos identificadores quando estas se tornam conhecidas (quando são declaradas ou implícitas).

Na maioria dos compiladores, as atividades semânticas são realizadas por um bloco separado, denominado *bloco semântico*, que fica entre o bloco sintático e o gerador de código. Algumas operações semânticas usuais são: verificação de compatibilidade de tipo, análise de escopo, verificação de compatibilidade entre declaração e uso de entidades, verificação de correspondência entre parâmetros atuais e formais e verificação de referência não resolvidas.

Otimização de Código

Certas atividades de um compilador que não são estritamente necessárias, mas que permitem obter melhores programas objeto, são freqüentemente referidas como *otimização*. Para alguns compiladores, a otimização é inserida como um bloco entre o bloco sintático e o semântico (se houver). Este bloco pode resolver problemas como, por exemplo, instruções invariantes em laços, rearranjando os símbolos sintáticos antes de passá-los para a análise semântica.

II - Teoria de Linguagens

1. Linguagens Formais

O estudo dos diversos tipos de linguagens aplicam-se a diversas áreas da informática, sempre que for necessário analisar o formato e o significado de uma seqüência de entrada. Dentre as aplicações possíveis, temos: **editor de estruturas**, que analisa as estruturas de um programa fonte sendo editado, tentando minimizar erros; **pretty printers**, que tenta tornar os programas mais legíveis; **verificadores ortográficos**, **verificadores gramaticais, verificadores estáticos, interpretadores, compiladores**, etc.

Serão apresentados nesta seção conceitos gerais sobre linguagens que servirão para fundamentar o estudo de todos os tipos de linguagens que virão a seguir.

1.1. Símbolo

Um símbolo é uma entidade abstrata que não precisa ser definida formalmente, assim como "ponto" e "linha" não são definidos na geometria. Letras e dígitos são exemplos de símbolos freqüentemente usados.

Símbolos são ordenáveis lexicograficamente e, portanto, podem ser comparados quanto à igualdade ou precedência. Por exemplo, tomando as letras do alfabetos, podemos ter a ordenação $a < b < c < ... < z$. A principal utilidade dos símbolos está na possibilidade de usá-los como elementos atômicos em definições de linguagens.

1.2. Cadeia

Uma *cadeia* (ou string, ou palavra) é uma seqüência finita de símbolos justapostos (isto é, sem vírgulas separando os caracteres). Por exemplo, se a, b e c são símbolos, então abcb é um exemplo de cadeia usando utilizando estes símbolos.

O *tamanho* de uma cadeia é o comprimento da seqüência de símbolos que a forma. O tamanho de uma cadeia w será denotado por |w|. Por exemplo, |abcb| = 4. A *cadeia vazia* é denotada por ε, e tem tamanho igual a 0; assim, |ε| = 0.

Um *prefixo* de uma cadeia é um número qualquer de símbolos tomados de seu início, e um *sufixo* é um número qualquer de símbolos tomados de seu fim. Por exemplo, a cadeia abc tem prefixos ε, a, ab, e abc. Seus sufixos são ε, c, bc, abc. Um prefixo ou sufixo que não são a própria cadeia são chamados de *prefixo próprio* e *sufixo próprio*, respectivamente. No exemplo anterior, os prefixos próprios são ε, a e ab; e os sufixos próprios são ε, c e bc.

A *concatenação* de duas cadeias é a cadeia formada pela escrita da primeira cadeia seguida da segunda, sem nenhum espaço no meio. Por exemplo, a concatenação de *compila* e *dores* é *compiladores*. O operador de concatenação é a justaposição. Isto é, se w e x são variáveis que denotam cadeias, então wx é a cadeia formada pela concatenação de w e x. No exemplo acima, se tomarmos w = *compila* e x = *dores*, então temos wx = *compiladores*.

A *n-ésima potência* de uma cadeia x, denotada por x^n, é a concatenação de x com ela mesma n-1 vezes, ou a repetição da cadeia n vezes. Por exemplo, abc^3 = abcabcabc.

Um *alfabeto* é definido simplesmente como um conjunto finito de símbolos. Por exemplo, o alfabeto da língua portuguesa é V = {a, b, c, ..., z}. O alfabeto utilizado para expressar os números naturais é V = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}.

O *fechamento* de um alfabeto V, representado por V^*, é o conjunto de todas as cadeias que podem ser formadas com os símbolos de V, inclusive a cadeia vazia. O *fechamento positivo* de V, denotado por V^+ é definido como $V^* - \{\varepsilon\}$. Ou seja, todas as cadeias formadas com os símbolos de V, exceto a cadeia vazia. Por exemplo, dado o alfabeto V = {0, 1}, então o fechamento de V é dado

por V^* = {ε, 0, 1, 00, 01, 10, 11, 000, 001, ... } e o fechamento positivo de V é dado por $V^+ = V^* - \{\varepsilon\}$ = {0, 1, 00, 01, 10, 11, 000, 001, ... }

1.3. Linguagens

Uma *linguagem* (formal) é um conjunto de cadeias de símbolos tomados de algum alfabeto. Isto é, uma linguagem sobre o alfabeto V é um subconjunto de V^*. Note-se que esta definição é meramente extensional, isto é, não considera os mecanismos formadores da linguagem, mas apenas a sua extensão. Assim, por exemplo, o conjunto de sentenças válidas da língua portuguesa poderia ser definido extensionalmente como um subconjunto de $\{a, b, c,..., z\}^+$.

Uma linguagem é *finita* se suas sentenças formam um conjunto finito. Caso contrário, a linguagem é *infinita*. Uma linguagem infinita precisa ser definida através de uma representação finita. Por exemplo, a linguagem dos números naturais menores que 10 é finita, e pode ser representada, literalmente, como L = {1, 2, 3, 4, 5, 6, 7, 8, 9}. Já a linguagem dos números naturais como um todo não é uma linguagem finita, já que existem infinitos números naturais. Porém, como veremos adiante, existem mecanismos que permitem expressar esta e outras linguagens infinitas através de representações finitas. É o caso das gramáticas e das expressões regulares.

Um *reconhecedor* para uma linguagem é um dispositivo formal usado para verificar se uma determinada sentença pertence ou não a uma determinada linguagem. São exemplos de reconhecedores de linguagens as Máquinas de Turing, o Autômato de Pilha e o Autômato Finito. Cada um destes mecanismos reconhece um conjunto bastante particular de linguagens. Entretanto, os modelos são inclusivos: todas as linguagens reconhecíveis por um Autômato Finito podem ser reconhecidas por uma Autômato de Pilha e todas as linguagens reconhecíveis por um Autômato de Pilha são reconhecíveis por Máquinas de Turing. As recíprocas, entretanto, não são verdadeiras.

Um *sistema gerador* é um dispositivo formal através do qual as sentenças de uma linguagem podem ser sistematicamente geradas. Exemplos de sistemas geradores são as gramáticas gerativas, definidas pela primeira vez por Noam Chomsky, em seus estudos para sistematizar a gramática da língua inglesa.

Todo reconhecedor e todo sistema gerador pode ser representado por algoritmos e/ou procedimentos. Como será visto mais adiante, pode-se garantir que linguagens reconhecidas por autômatos são reconhecidas algoritmicamente. Já o conjunto das linguagens reconhecidas por Máquinas de Turing, que é mais abrangente, inclui linguagens reconhecíveis algoritmicamente e linguagens indecidíveis, que não possuem algoritmos para reconhecimento.

2. Gramáticas

Uma *gramática gerativa* é um instrumento formal capaz de construir (gerar) conjuntos de cadeias de uma determinada linguagem. As gramáticas são instrumentos que facilitam muito a definição das características sintáticas das linguagens. São instrumentos que permitem definir, de forma formal e sistemática, uma representação finita para linguagens infinitas.

Usar um método de definição de conjuntos particular para definir uma linguagem pode ser um passo importante no projeto de um reconhecedor para a linguagem, principalmente se existirem métodos sistemáticos para converter a descrição do conjunto em um programa que processa o conjunto. Como veremos adiante neste curso, certos tipos de gramáticas, que possuem algoritmos para definir se uma sentença pertence ou não à linguagem que geram, são utilizadas para implementar compiladores. Já gramáticas que não permitem a definição de algoritmos não podem ser utilizadas para tal.

Mais adiante serão vistos métodos para transformar gramáticas em máquinas abstratas capazes de reconhecer e traduzir os conjuntos definidos pelas gramáticas. Antes, porém, é necessário entender como conjuntos de sentenças são definidos por gramáticas.

2.1. Definição Formal de Gramática

Uma gramática possui um conjunto finito de *variáveis* (também chamadas de *não-terminais*). A linguagem gerada por uma gramática é definida recursivamente em termos das variáveis e de *símbolos primitivos* chamados *terminais*, que pertencem ao alfabeto da linguagem.

As regras que relacionam variáveis e terminais são chamadas de *produções*, que sempre iniciam num ponto, o *símbolo inicial*, pertencente ao conjunto de variáveis (não-terminais). Uma regra de produção típica estabelece como uma determinada configuração de variáveis e terminais pode ser reescrita, gerando uma nova configuração.

Formalmente, uma gramática é especificada por uma quádrupla (N,T,P,S), onde:

- N é o conjunto de não-terminais, ou variáveis.
- T é o conjunto de terminais, ou alfabeto, sendo que N e T são conjuntos disjuntos.
- P é um conjunto finito de produções, ou regras, sendo que $P \subseteq (N \cup T)^{+} \times (N \cup T)^{*}$.
- S é um símbolo não-terminal inicial, sendo $S \in N$. A partir dele são geradas todas as sentenças da linguagem.

Uma produção (n,p) pode ser escrita n ::= p, para facilitar a leitura. No caso de existir mais de uma alternativa para uma mesma configuração e símbolos do lado esquerdo da regra, como por exemplo:

<substantivo> ::= coisa
<substantivo> ::= cão
<substantivo> ::= pedra

pode-se escrever as opções em uma única produção, separada por |. O exemplo acima ficaria:

<substantivo> ::= coisa | cão | pedra.

O símbolo ::= pode ser lido como “é definido por” e o símbolo | pode ser lido como “ou”.

2.2. Derivação

Uma gramática pode ser usada para gerar uma linguagem através de *reescrita* ou *derivação*. A derivação pode ser definida da seguinte maneira: dada uma sentença a_1ba_2, com a_1 e a_2 sendo cadeias de terminais e não-terminais, se existir uma regra da forma b ::= c, então a sentença inicial pode ser reescrita como a_1ca_2. A derivação, ou reescrita, é denotada da seguinte forma:

$a_1ba_2 \rightarrow a_1ca_2$

Por exemplo, digamos que exista a seguinte seqüência de terminias (letras minúsculas) e não-terminais (letras maiúsculas): “aaaDbbb”, e que exista a produção “D ::= ab” na gramática. Assim, a derivação “aaaDbbb $\rightarrow$ aaaabbbb” é uma derivação válida na gramática dada.

Se $a \rightarrow b$, então se diz que a produz diretamente b. O fecho transitivo e reflexivo da relação $\rightarrow$ é $\rightarrow^{*}$. Portanto, se $a \rightarrow b \rightarrow ... \rightarrow z$, então se diz que a produz z, simplesmente, anotando a produção por $a \rightarrow^{*} z$. O fecho transitivo pode ser definido da seguinte forma:

a) $a \rightarrow^{0} a$

b) $a \rightarrow^{1} b$ se $a \rightarrow b$

c) $a \rightarrow^{n+1} z$ se $a \rightarrow^{n} y$ e $y \rightarrow^{1} z$

d) $a \rightarrow^{*} z$ se $a \rightarrow^{n} z$, para algum $n \geq 0$.

Seja, por exemplo, a seguinte gramática, onde as regras são numeradas para acompanhar o desenvolvimento da derivação:

1: <S> ::= <X>+<X>=
2,3: <X> ::= 1<X> | 1
4: <X>+1 ::= <X>1+
5: <X>1 ::= 1<X>
6: <X>+= ::= <X>
7: 1<X> ::= <X>1

A derivação de uma sentença a partir desta gramática consiste em uma seqüência de aplicações das produções a partir do símbolo inicial. Por exemplo:

<S> → <X>+<X>= [1]
→ 1<X>+<X>= [2 no primeiro <X>]
→ 1<X>+1= [3 no segundo <X>]
→ 1<X>1+= [4]
→ 11<X>+= [5]
→ 11<X> [6]
→ 111 [3]

O fato das regras terem sido aplicadas praticamente em seqüência foi mera coincidência. A princípio, qualquer regra de derivação pode ser aplicada a qualquer ponto de sentença desde que a configuração de símbolos do lado esquerdo da regra apareça na sentença sendo derivada.

Outro exemplo é a seguinte gramática:

<S> ::= 1<A> | 0<S> | 1
<A> ::= 0<A> | 1<S>

A sentença 100101 é gerada pela seguinte seqüência de derivações:

<S> → 1<A> → 10<A> → 100<A> → 1001<S> → 10010<S> → 100101

Originalmente, a motivação do uso de gramáticas foi o de descrever estruturas sintáticas da língua natural. Poderia-se escrever regras (produções) como:

<frase> ::= <artigo> <substantivo> <verbo> <artigo> <substantivo>. |
<artigo> <substantivo> <verbo>.
<verbo> ::= dorme | escuta | ama
<substantivo>::= gato | rádio | rato
<artigo> ::= o

Desta gramática simples pode-se obter derivações como:

<frase> → o <substantivo> <verbo> <artigo> <substantivo>.
→ o gato <verbo> <artigo> <substantivo>.
→ o gato ama <artigo> <substantivo>.
→ o gato ama o <substantivo>.
→ o gato ama o rato.

Entretanto a mesma gramática também permite derivar “o gato dorme o rato”, frase que não faz sentido. Para evitar a geração de tais frases, pode-se usar uma gramática sensível ao contexto, estabelecendo que <verbo> só pode ser reescrito para “dorme” se for sucedido pelo ponto final da frase:

<frase> ::= <artigo> <substantivo> <verbo> <artigo> <substantivo>. |
<artigo> <substantivo> <verbo>.
<verbo>. ::= dorme.
<verbo> <artigo> ::= escuta <artigo> | ama <artigo>
<substantivo>::= gato | rádio | rato
<artigo> ::= o

Para esta gramática, é impossível derivar as sentenças “o gato dorme o rato”, “o gato ama” e “o rato escuta”. O verbo dorme só pode ser derivado se for sucedido por um ponto final e os verbos escuta e ama só podem ser derivados se forem sucedidos por um artigo. Porém ainda é possível derivar uma sentença do tipo “o rádio ama o rato”. O leitor é convidado a modificar a gramática acima para não aceitar sentenças desse tipo (dica: permitir que o substantivo “rádio” apareça somente no final da sentença).

2.3. Notação

Como convenção notacional, pode-se admitir que símbolos não terminais serão sempre representados por letras maiúsculas, e terminais por minúsculas. Assim, uma regra como <a> ::= <a>b | b poderia ser escrita como A ::= Ab | b. Além disso, os conjuntos T e N podem ficar subentendidos e não precisam ser expressos sempre.

Pode-se ainda arbitrar que o símbolo inicial S será sempre o não-terminal que aparecer primeiro do lado esquerdo da primeira produção da gramática. Assim, bastaria listar as regras de produção para se ter a definição completa da gramática. Por exemplo, a gramática

S ::= ABSc | ε
BA ::= AB
Bc ::= bc
Ab ::= ab
Bb := bb
Aa ::= aa

Tem como símbolo inicial S, o conjunto de terminais é {a, b, c} e o conjunto de não-terminais (variáveis) é {A, B, S}. As produções estão detalhadas. Tem-se assim a definição completa da gramática.

3. Linguagens Definidas por Gramáticas

A linguagem definida por uma gramática consiste no conjunto de cadeias de terminais que esta gramática pode potencialmente gerar. Este conjunto será denotado por L(G). A linguagem definida por uma gramática G=(N,T,P,S) é dada por:

$$L(G) = \{ \alpha \mid \alpha \in T^* \wedge S \rightarrow^* \alpha \}$$

Em outras palavras, uma cadeia pertence a L(G) se e somente se ela consiste somente de terminais (pode ser a cadeia vazia ε), e pode ser produzida a partir de S em 0 ou mais derivações.

Uma cadeia em $(T \cup N)^*$ é chamada de *forma sentencial* se ela pode ser produzida a partir de S (mas ainda contém não-terminais). Assim, pode-se caracterizar L(G) como o conjunto de todas as formas sentenciais que contém apenas terminais, e foram geradas a partir do símbolo inicial de G. Desta forma, “aaaBbbC” é uma forma sentencial, pois contém variáveis (B e C). Já “aabb” é uma sentença, pois contém apenas terminais.

4. Tipos de Gramáticas

Impondo restrições na forma das produções, pode-se identificar quatro tipos diferentes de gramáticas. Esta classificação foi feita por Chomsky e é conhecida como *hierarquia de Chomsky*. Os tipo de gramática definidos nesta hierarquia são:

a) Tipo 0: Não há restrição na forma das produções. Este é o tipo mais geral. Exemplo:
S ::= ABS | ab Ba ::= aB | a Aa ::= aa | a | ε

b) Tipo 1: Seja G=(N,T,P,S). Se:

- O símbolo inicial S não aparece no lado direito de nenhuma produção, e

- para cada produção $\sigma_1 ::= \sigma_2$, é verdade que $|\sigma_1| \leq |\sigma_2|$ (com exceção da regra S ::= ε),

então se diz que G é uma gramática do tipo 1, ou *Gramática Sensível ao Contexto*.
Exemplo:
S ::= aBC | aABC A ::= aABC | aBC CB ::= BC aB ::= ab
bB ::= bb bC ::= bc cC ::= cc

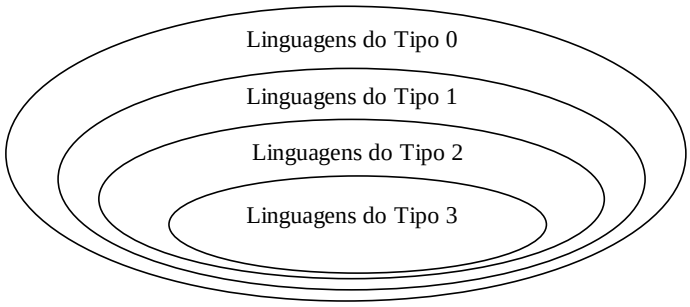
c) Tipo 2: Uma gramática G=(N,T,P,S) é do tipo 2, ou uma *Gramática Livre de Contexto*, se cada produção é livre de contexto, ou seja, cada produção é da forma
$A ::= \sigma$, com $A \in N$ e $\sigma \in (T \cup N)^*$.
Exemplo:
S ::= aSb | A A ::= cAd | e

O tipo 2 não é definido como restrição do tipo 1, porque se permite produções da forma A ::= ε, com $A \neq S$. Também se permite que o símbolo inicial S apareça no lado direito das produções. Entretanto, existe um teorema que prova que a partir de uma gramática de tipo 2 se pode construir outra gramática equivalente, de tipo 2, que satisfaz as restrições do tipo 1.

d) Tipo 3. Uma gramática G é de tipo 3, ou regular, se cada produção é da forma
A ::= aB, A ::= a ou A ::= ε, onde A e B são não-terminais e a é um terminal.
Exemplo:
S ::= aS | bB B ::= bB | bC C ::= cC | ε

5. Tipos de Linguagens

Uma linguagem L é de tipo i se existe uma gramática G de tipo i tal que L = L(G), para i igual a 0, 1, 2 ou 3. Pode-se ver que cada linguagem de tipo 3 é também de tipo 2, cada linguagem de tipo 2 é também de tipo 1, e cada linguagem de tipo 1 é também de tipo 0. Estas inclusões são estritas, isto é, existem linguagens de tipo 0 que não são de tipo 1, existem linguagens de tipo 1 que não são do tipo 2 e existem linguagens do tipo 2 que não são do tipo 3. Graficamente esta inclusão pode ser representada por:



Pode-se relacionar cada tipo de gramática com uma máquina reconhecedora da seguinte maneira: a gramática de tipo 0 gera linguagens reconhecíveis por máquinas de Turing; a gramática de tipo 2 gera linguagens reconhecidas por autômatos de pilha; a gramática de tipo 3 gera linguagens reconhecidas por autômatos finitos.

Não há uma classe de reconhecedores para apenas linguagens de tipo 1, porque qualquer máquina capaz de reconhecer linguagens de tipo 1 é poderosa o suficiente para reconhecer linguagens do tipo 0. A principal razão para identificar as linguagens de tipo 1 como um tipo separado é porque toda a linguagem do tipo 1 é decidível, isto é, se G=(N,T,P,S) é do tipo 1, então existe um algoritmo tal que, para qualquer sentença w, responda "sim" se $w \in L(G)$ e "não" caso contrário.

Assim, pode-se dizer que as linguagens do tipo 1 são reconhecíveis por Máquinas de Turing, sendo decidíveis, ou seja, a máquina sempre vai parar num tempo finito com uma resposta, afirmativa ou negativa, sobre a sentença analisada. Já as linguagens do tipo 0 também podem ser

analisadas por Máquinas de Turing. O problema é que podem existir linguagens, ou sentenças dentro de linguagens, para as quais a máquina não pare. Ou seja, a máquina ficará analisando a sentença indefinidamente, sem chegar a nenhuma resposta.

EXERCÍCIOS

1. Uma palíndrome é uma cadeia que pode ser lida da mesmo maneira da direita para a esquerda e da esquerda para a direita. Exemplos: a, ama, osso, asddsa. Defina uma gramática para a linguagem das palíndromes (conjunto de todas as palíndromes sobre o alfabeto {a, b, c, ... , z}.

* 2. Dada a gramática (livre de contexto):

S ::= aSba | ε

mostre três cadeias de L(G) e a derivação de cada uma delas.

3. Dado o conjunto de terminais (alfabeto) T={a, b, c}, defina as gramáticas para as seguintes linguagens:

* a) Sentenças que começam e terminam com a.

* b) Sentenças que tenham tamanho ímpar.

c) Sentenças em que todos os a's apareçam consecutivos.

* d) Sentenças na forma $1^n0^n2^n$, com n > 0.

e) Sentenças na forma $a^nb^mc^na^n$, n > 0 e m > 0.

* f) Sentenças na forma $a^nb^mc^n$, com n > 0 e m ≥ 0.

g) Sentenças na forma $a^mb^nc^n$, com m ≥ 0 e n > 0.

h) Sentenças em que o número de a's seja par.

i) Sentenças na forma $a^nb^{2n}c^m$, n > 0 e m > 2.

4. Defina uma gramática que gere as expressões numéricas com o seguinte alfabeto: dígitos (0 a 9), sinais (+, -, *, /) e parênteses (e). Teste sua definição mostrando a derivação das seguintes sentenças:

a) (5+5)/(4+1)

b) 4*(7/9)

c) ((5)/((4+5)*(1+3)))

5. Escreva uma gramática para cada uma das linguagens abaixo:

* a) $\{ a^{2i+1} b^{i+3} \mid i \geq 0 \} \cup \{a^{i+4} b^{3i} \} \mid i \geq 0\}$

* b) $\{a^ib^k \mid k > 0, i > k \}$

c) $\{ a^i b^j c^j d^i e^3 \mid i, j \geq 0 \}$

d) $\{ a^i b^j c^k d^l \mid i, j, k, l > 0, i \neq l, \ j > k\}$

6. Indique o tipo de cada gramática:

( )	S ::= aS | b | ε		
( )	S ::= AAA | aA	AA ::= AbcAA | ε	A ::= a | b
( )	S ::= aB | b	B ::= aB | b	
( )	S ::= (S) | S + S	(S+S) ::= [id + id]	
( )	S := ABc | abc	A := ab | ε	B ::= bc | b

7. Construa expressões regulares para as cadeias de 0s e 1s descritas abaixo:

* a) com número par de 1s e 0s.

b) com número ímpar de ocorrências do padrão 00.

* c) com pelo menos duas ocorrências do padrão 101.

d) com qualquer cadeia de entrada.

e) com nenhuma cadeia de entrada.

f) as cadeias 0110 e 1001.

e) todas as cadeias que começam com 01 e terminam com 10.

g) todas as cadeias que contenham exatamente quatro 1s.

h) a cadeia nula e 001.

III - Linguagens Regulares e Autômatos Finitos

1. Linguagens Regulares

As linguagens regulares constituem um conjunto de linguagens decidíveis bastante simples e com propriedades bem definidas e compreendidas. Essas linguagens podem ser reconhecidas por autômatos finitos e são facilmente descritas por expressões simples, chamadas *expressões regulares* (E.R.) Antes de introduzir o mecanismo das E.R.s, será feita uma revisão das operações de concatenação e fechamento de conjuntos, já que são estas as operações básicas para a definição de linguagens regulares.

1.1. Operações de Concatenação e Fechamento

Seja C um conjunto finito de símbolos (alfabeto), e sejam L, L_1 e L_2 subconjuntos de C^* (linguagens sobre o alfabeto C). A concatenação de L_1 e L_2, denotada por L_1L_2 é o conjunto $\{xy \mid x \in L_1 \text{ e } y \in L_2\}$. Em outras palavras, as cadeias da linguagem L_1L_2 são formadas por uma cadeia de L_1 concatenada a uma cadeia em L_2, nesta ordem, incluindo aí todas as combinações possíveis.

Define-se $L^0 = \{\varepsilon\}$ e $L^n = LL^{n-1}$ para $n \geq 1$. O *fecho de Kleene* (ou simplesmente *fecho*) de uma linguagem L, denotado por L^*, é o conjunto:

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$

e o *fecho positivo* da linguagem L, denotado por L^+, é o conjunto:

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \ldots$$

Em outras palavras, L^* denota as cadeias construídas pela concatenação de qualquer número de cadeias tomadas de L. O conjunto L^+ é semelhante, mas neste caso, as cadeias de zero palavras, cuja concatenação é definida como ε, são excluídas. Note-se, porém, que L^+ contém ε se e somente se L contém ε. Esta definição difere da definição do fechamento de alfabetos, onde A+ era definido como $A^* - \{\varepsilon\}$. Note-se que no caso de linguagens, podem ocorre dois casos:

a) Se $\varepsilon \in L$, então $L^+ = L^*$.

b) Se $\varepsilon \notin L$, então $L^+ = L^* - \{\varepsilon\}$.

Exemplos:

1. Seja $L_1 = \{10, 1\}$ e $L_2 = \{0011, 11\}$. Então $L_1L_2 = \{100011, 1011, 10011, 111\}$

2. $\{10, 11\}^* = \{\varepsilon, 10, 11, 1010, 1011, 1110, 1111, \ldots\}$

1.2. Definição de Expressões Regulares

Seja A um alfabeto. As expressões regulares sobre o alfabeto A e os conjuntos (linguagens) que elas denotam são definidas indutivamente como segue:

a) ∅ é uma expressão que denota o conjunto vazio.

b) ε é uma expressão regular que denota o conjunto $\{\varepsilon\}$.

c) Para cada símbolo em a em A, a é uma expressão regular que denota o conjunto $\{a\}$.

d) Se r e s são expressões regulares que denotam os conjuntos R e S, respectivamente, então (r+s), (rs), e (r^*) são expressões regulares que denotam os conjuntos $R \cup S$, RS e R^*, respectivamente.

Quando se escreve expressões regulares, pode-se omitir muitos parênteses se for assumido que o símbolo * tem precedência maior que a concatenação e +, e que a concatenação tem precedência maior que +. Por exemplo, $((0(1^*))+0)$ pode ser escrita como 01^*+0. Além disso, pode-se aplicar simplificaçõs abreviando expressões como rr^* para r^+.

As expressões regulares possuem as seguintes propriedades algébricas:

AXIOMA	DESCRIÇÃO
$r + s = s + r$	+ é comutativo.
$r + (s + t) = (r + s) + t$	+ é associativo.
$(rs)t = r(st)$	a concatenação é associativa.
$r(s + t) = rs + rt$ $(s + t)r = sr + tr$	a concatenação é distibutiva sobre +.
$\varepsilon r = r$ $r\varepsilon = r$	ε é o elemento neutro (identidade) da concatenação.
$r^* = (r + \varepsilon)^*$	relação entre ε e *.
$r^{**} = r^*$	* é idempotente.

Exemplos:

* fechamento
(ab)* = abababab....

- 00 é uma expressão regular que denota a linguagem { 00 }.
- A expressão $(0+1)^*$ denota todas as cadeias de 0s e 1s.
- $(0+1)^*00(0+1)^*$ denota as cadeias de 0s e 1s com pelo menos dois 0s consecutivos.
- $(1+10)^*$ denota as cadeias de 0s e 1s que começam com 1 e não tem 0s consecutivos.
- $(0+1)^*001$ denota as cadeias de 0s e 1s que terminam por 001.
- a^+b^* denota as cadeias que tem qualquer quantidade de a's (mínimo 1) seguidos de qualquer quantidade de b's (possivelmente 0). Exemplo = {a, aaa, aab, aabbbbbb, ab, abbbb, ...}

É importante notar que as linguagens regulares podem ser expressas tanto por expressões regulares como por gramáticas regulares. Assim, qualquer linguagem descrita na forma de uma expressão regular pode ser descrita por uma gramática regular. A seguir é apresentada uma tabela que mostra linguagens regulares representadas por expressões regulares e por gramáticas regulares.

Expressão Regular	Gramática Regular
(0 + 1)*	S ::= 0S | 1S | ε
001*	S ::= 0A A ::= 0B B ::= 1B | ε
$(00)^*(11)^+$	S ::= 0A | 1B A ::= 0S B ::= 1 | 1C C ::= 1B
a(aa)*(b+c)*	S ::= aA A ::= ε | aB | bC | cC B ::= aA C ::= bC | cC | ε

2. Autômatos Finitos

O Autômato Finito (AF) é o tipo mais simples de reconhecedor de linguagens. Ele é usado como reconhecedor de padrões em processamento de textos e também como analisador léxico de linguagens.

Autômatos finitos são usados com vantagens na compilação porquê:

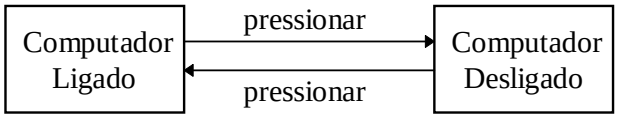
a) Eles têm a capacidade de realizar várias tarefas simples de compilação. Em particular, o projeto do bloco léxico pode ser feito baseado em um autômato finito.

b) Uma vez que a simulação de um AF por um computador requer apenas um pequeno número de operações para processar um símbolo de entrada individual, ela opera rapidamente.

c) A simulação de um AF requer uma quantidade finita de memória.

d) Existem vários teoremas e algoritmos para construir e simplificar autômatos finitos para vários propósitos.

O conceito fundamental do AF é o conceito de *estado*. O estado representa uma condição sobre o modelo representado. Por exemplo, um computador pode ser representado por um modelo com dois estados: *ligado* e *desligado*. Pode-se ainda dizer que o ato de pressionar um botão ocasiona uma mudança de estado, dependendo do estado atual. Esta mudança de estado é denominada *transição*. O AF que representa este modelo pode ser:



Um Autômato Finito é um reconhecedor de Linguagens Regulares. Entende-se por *reconhecedor* de uma linguagem L um dispositivo que tomando uma seqüência w como entrada, responde "sim" se $w \in L$ e "não" caso contrário.

Os Autômatos Finitos classificam-se em:

- Autômatos Finitos Determinísticos (AFD)
- Autômatos Finitos Não-Determinísticos (AFND)

2.1 Autômatos Finitos Determinísticos

Formalmente definimos um AFD como sendo um sistema formal $M = (K, \Sigma, \delta, q_0, F)$, onde:

- $K \rightarrow$ É um conjunto finitos não-vazio de ESTADOS;
- $\Sigma \rightarrow$ É um ALFABETO, finito, de entrada;
- $\delta \rightarrow$ FUNÇÃO DE MAPEAMENTO (ou função de transição), definida em $K \times \Sigma \rightarrow K$;
- $q_0 \rightarrow \in K$, é o ESTADO INICIAL;
- $F \rightarrow \subseteq K$, é o conjunto de ESTADOS FINAIS.

2.1.1. Interpretação de δ

A interpretação de $\delta(q, a) = p$, onde $\{q, p\} \in K$ e $a \in \Sigma$, é de que se o "controle de M" está no estado "q" e o próximo símbolo de entrada é "a", então "a" deve ser reconhecido e o "controle" passa para o próximo estado (no caso, "p"). Note que a função de mapeamento é definida por $K \times \Sigma \rightarrow K$, ou seja, de um estado, através do reconhecimento de um símbolo, o controle passa para UM ÚNICO outro estado. Por isso este tipo de autômato é chamado de DETERMINÍSTICO.

2.1.2. Significado Lógico de um Estado

Logicamente um estado é uma situação particular no processo de reconhecimento de uma sentença. Mais genericamente, um estado representa uma condição sobre a parte da sentença que já foi reconhecida.

2.1.3. Sentenças Aceitas por M

Uma sentença x é aceita (reconhecida) por um AF M = (K, Σ, δ, q_0, F) se, e somente se, $\delta(q_0, x) = p \wedge p \in F$. Em outras palavras, a sentença x é reconhecida se, partindo do estado inicial q_0, através das transições para cada símbolo de x, um estado final é alcançado.

2.1.4. Linguagem Aceita por M

É um conjunto de todas as sentenças aceitas por M. Formalmente, definimos por:

$L(M) = \{x \mid \delta(q_0, x) = p \wedge p \in F\}$.

Ou seja, é o conjunto de todas as sentenças que, partindo do estado inicial, fazem com que o autômato alcance um estado final quando termina de reconhecer a sentença.

Obs.: Todo conjunto aceito por um Autômato Finito é um Conjunto Regular.

2.1.5. Diagrama de Transição

Um diagrama de transição para um AF M é um grafo direcionado e rotulado. Os vértices representam os estados e fisicamente são representados por círculos, sendo que o estado inicial é diferenciado por uma seta com rótulo "início" e os estados finais são representados por círculos duplos. As arestas representam as transições, sendo que entre dois estados "p" e "q" existirá uma aresta direcionada de "p" para "q" com rótulo "a" $(a \in \Sigma) \leftrightarrow \exists\, \delta(p, a) = q$ em M.

2.1.6. Tabela de Transições

É uma representação tabular de um AF

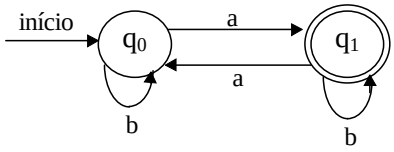
Nesta tabela as linhas representam os estados (o inicial é indicado por uma seta e os finais por asteriscos), as colunas representam os símbolos de entrada e o conteúdo da posição (q, a) será igual a "p" se existir $\delta(q, a) = p$, senão será indefinida.

Exemplos

Nesta seção daremos alguns exemplos de AFD nas formas gráficas e tabulares.

Exemplo 1: AFD que reconheça as sentenças da linguagem $L = \{(a,b)^+$, onde o número de a's é ímpar$\}$.

Forma Gráfica:



Forma Tabular:

	δ	a	b
→	q_0	q_1	q_0
*	q_1	q_0	q_1

É possível interpretar o estado q_0 como a condição lógica "há um número par de a's na senteça" e o estado q_1 como a condição "há um número ímpar de a's na sentença".

Para reconhecer a sentença "abbaba" o autômato realiza as seguintes transições, partindo do estado inicial:

q_0 abbaba
a q_1 bbaba
ab q_1 baba
abb q_1 aba
abba q_0 ba

abbab q_0 a
abbaba q_1

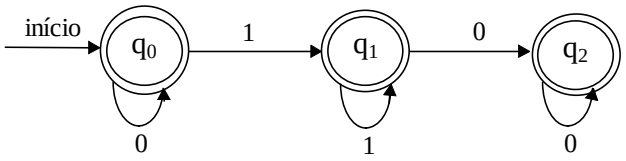
Como a sentença terminou com o autômato num estado final, a mesma é reconhecida, ou seja, faz parte da linguagem definida. Já a sentença "ababaab" leva às seguintes transições do AFD acima:

q_0 ababaab
a q_1 babaab
ab q_1 abaab
aba q_0 baab
abab q_0 aab
ababa q_1 ab
ababaa q_0 b
ababaab q_0

Ao final da sentença o AFD está num estado não-final. Assim, a sentença não faz parte da linguagem.

Exemplo 2: AFD que reconheça as sentenças da linguagem L = {(0,1)*, onde todos os 1's apareçam consecutivos}.

Forma Gráfica:



Forma Tabular:

	δ	0	1
→*	q_0	q_0	q_1
*	q_1	q_2	q_1
*	q_2	q_2	-

Note que neste caso, o não reconhecimento de sentenças com 1's não consecutivos é impossível pois não há transição no estado q2 para o símbolo 1. A sentença "0101" leva às seguintes transições:

q_0 0101
0 q_0 101
01 q_1 01
010 q_2 1 (?)

Quando o AFD, no estado atual, não apresentar transição para um símbolo da entrada, a sentença também é rejeitada.

2.2. Autômatos Finitos Não-Determinísticos

Um AFND é um sistemas formal M = M = (K, Σ, δ, q_0, F) onde:

K, Σ, q_0, F → possuem a mesma definição de AFD

δ → é uma função de mapeamento, definida em K x Σ = ρ(K) (um conjunto de estados), sendo que ρ(K) é um subconjunto de K. Isto equivale a dizer que δ(q, a) = p_1, p_2, ..., p_n. A

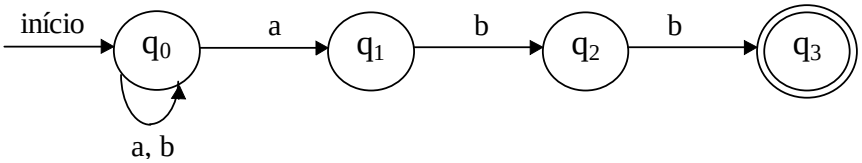
interpretação de δ é que M no estado "q", com o símbolo "a" na entrada pode ir tanto para o estado p_1, como para o estado p_2, ..., como para o estado p_n.

Uma sentença x é aceita por um AFND se $\delta(q_0, x) = P$, e $\{P \cap F\} \neq \varnothing$. Em outras palavras, uma sentença x é aceita se, dentre os estados alcançados, pelo menos um é final.

Na análise de uma sentença, quando um símbolo leva à mais de um estado, pode-se dizer que o AFND se divide em vários, sendo que cada um deles continua o reconhecimento da sentença por uma das possibilidades. Se pelo menos um deles chegar a um estado final no final da sentença, então ela é válida. Se nenhum chegar a um estado final, então a sentença é inválida.

Exemplo: Seja a linguagem L = {(a,b)* abb}. O AFND seria:

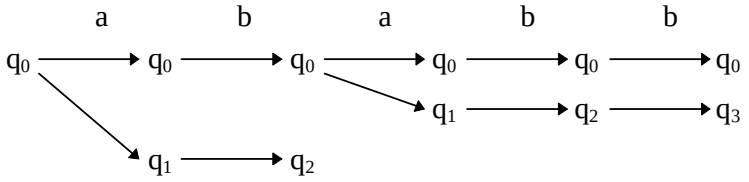
Forma Gráfica:



Forma Tabular:

	δ	a	b
→	q_0	q_0, q_1	q_0
	q_1	-	q_2
	q_2	-	q_3
*	q_3	-	-

Para reconhecer a sentença "ababb", o autômato poderia seguir as seguintes alternativas:



Há três possibilidades no reconhecimento da sentença:

- um caminho termina num estado não-final (q_0);
- um caminho chega a um estado (q_2) onde não há transição definida para o próximo símbolo (a);
- um caminho chega a um estado final (q_3) no final da sentença.

Como há a possibilidade de chegar a um estado final com a sentença, então a mesma é válida, ou seja, deve ser reconhecida como parte da linguagem definida.

2.3. Comparação entre AFD e AFND

Os AFDs e os AFNDs representam a mesma classe de linguagens, ou seja, as linguagens regulares. A tabela abaixo mostra as principais diferenças entre AFD e AFND:

	Vantagens	**Desvantagens**
AFD	Implementação trivial	Não representa claramente algumas L. R.
AFND	Representa mais claramente	Implementação complexa

	algumas L. R.	

2.4. Transformação de AFND para AFD

Como os autômatos finitos representam a mesma classe de linguagens, sejam determinísticos ou não-determinísticos, é de se esperar que haja uma equivalência entre eles. Realmente esta equivalência existe. Assim, para cada AFND é possível construir um AFD equivalente. Nesta seção é apresentado um algoritmo que converte um AFND num AFD equivalente. Assim, é possível utilizar a clareza da representação de um AFND, e para fins de facilidade de implementação, transformá-lo num AFD.

Teorema 3.1: "Seja L um conjunto aceito por um AFND., então $\exists$ um AFD que aceita L."

Prova: Seja M = (K, Σ, δ, q_0, F) um AFND Construa um A. F. D. M' = (K', Σ', δ', q_0', F') como segue:

1 - K' = $\{\rho(K)\} \rightarrow$ isto é, cada estado de M' é formado por um conjunto de estados de M.

2 - q_0' = $[q_0] \rightarrow$ obs.: representaremos um estado $q \in K'$ por [q].

3 - F' = $\{\rho(K) \mid \rho(K) \cap F \neq \varnothing\}$.

4 - $\Sigma' = \Sigma$.

5 - Para cada $\rho(K) \subset K'$, $\delta'(\rho(K), a) = \rho'(K)$, onde $\rho'(K) = \{p \mid$ para algum $q \in \rho(K)$, $\delta(q, a) = p\}$, ou seja:

Se $\rho(K) = [q_1, q_2, ..., q_r] \in K'$ e

se $\delta(q_1, a) = p_1, p_2, ..., p_j$

$\delta(q_2, a) = p_{j+1}, p_{j+2}, ..., p_k$

...

$\delta(q_r, a) = p_i, p_{i+1}, ..., p_n$ são as transições de M,

então $\rho'(K) = [p_1, p_2, ..., p_j, p_{j+1}, ..., p_k, ..., p_i, p_{i+1}, ..., p_n]$ será um estado de M' e M' conterá a transição: $\delta'(\rho(K), a) = \rho'(K)$.

Para concluir a prova do teorema, basta provar que a linguagem de M' é igual à linguagem de M (L(M) = L(M')), prova esta que não é apresentada neste texto.

Exemplo: Tomemos por exemplo o AFND da seção 3, representado na tabela abaixo:

	δ	a	b
$\rightarrow$	q_0	q_0, q_1	q_0
	q_1	-	q_2
	q_2	-	q_3
*	q_3	-	-

Para determinizar o AFND acima, basta definir o AFD M'= (K', Σ', δ', q_0', F'), onde

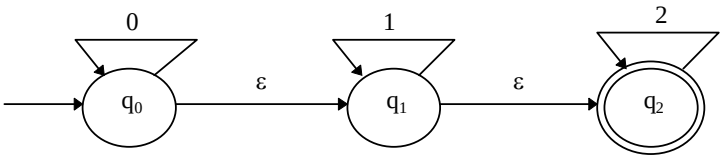
	δ'	a	b
$\rightarrow$	$[q_0]$	$[q_0, q_1]$	$[q_0]$
	$[q_0, q_1]$	$[q_0, q_1]$	$[q_0, q_2]$
	$[q_0, q_2]$	$[q_0, q_1]$	$[q_0, q_3]$
*	$[q_0, q_3]$	$[q_0, q_1]$	$[q_0]$

K' = $\{[q_0], [q_0, q_1], [q_0, q_2], [q_0, q_3]\}$

$\Sigma' = \{a, b\}$ $q_0' = [q_0]$ $F' = \{[q_0, q_3]\}$

2.5. Autômato Finito com ε-Transições

Pode-se estender o modelo de autômato finito não-determinístico para incluir transições com entrada vazia: ε. O AFND da figura seguinte aceita uma linguagem constituída por qualquer número de 0's, seguidos por qualquer número de 1's, seguido por qualquer número de 2's (incluindo a quantidade 0, para cada um dos casos).



Definição:

Um *autômato finito não-determinístico com ε-transições* é uma quíntupla $(K, \Sigma, \delta, q_0, F)$, com todos os componentes de um AFND, mas com a relação de transição δ definida sobre $K \times \Sigma \cup \{\varepsilon\} \rightarrow K$.

Na representação da tabela de transições do AF, é utilizada uma coluna para a ε-transição. No caso do exemplo, a tabela seria a seguinte:

	δ	0	1	2	ε
→	q_0	q_0			q_1
	q_1		q_1		q_2
*	q_2			q_2	

2.5.1. Equivalência Entre AFs Com e Sem ε-Transições

Para um autômato finito com ε-transições, define-se o ε-fechamento de um estado q como sendo o conjunto de estados alcançáveis a partir de q utilizando-se somente ε-transições.

Para o AF do exemplo anterior, têm-se os seguintes ε-fechamento dos estados:

ε-fecho(q_0) = $\{q_0, q_1, q_2\}$

ε-fecho(q_1) = $\{q_1, q_2\}$

ε-fecho(q_2) = $\{q_2\}$

Seja $M = (K, \Sigma, \delta, q_0, F)$ um AF com ε-transições, pode-se construir $M' = (K, \Sigma, \delta', q_0, F')$ sem ε-transições, utilizando-se o seguinte algoritmo:

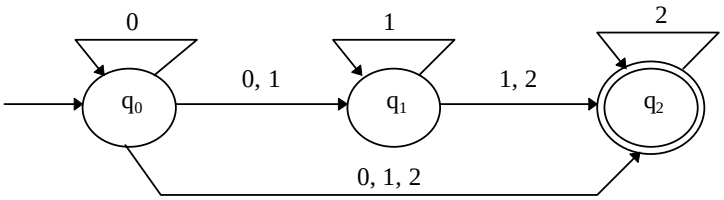
Algoritmo: Eliminação de ε-transições
Entrada: Um AFND com ε-transições $M = (K, \Sigma, \delta, q_0, F)$
Saída: Um AFND com ε-transições $M = (K, \Sigma, \delta', q_0, F')$

F' será definido por $F \cup \{q \in K$ e ε-fecho(q) contém um estado de F.
δ' deve conter todas as transições de t que não são ε-transições, mais aquelas que são obtidas pela composição de transições de δ' com as ε-transições de δ, da seguinte maneira:

Se $(q1, a) = q2 \in \delta'$, e $q3 \in$ ε-fecho(q2) então
    coloque $(q1, a) = q3$ em δ'
Fim Se;
Se $(q1, a) = q2 \in \delta'$, e $q1 \in$ ε-fecho(q3) então
    coloque $(q3, a) = q2$ em δ'
Fim Se;

Fim;

Para o autômato do exemplo anterior, têm-se o seguinte AFND sem ε-transições:



Claramente, para todo AFND com ε-transições, há um AFD equivalente, uma vez que basta eliminar as ε-transições, e em seguida determinizar o AFND resultante, chegando-se ao AFD equivalente.

3. Relação Entre GR e AF

As gramáticas regulares são sistemas geradores das linguagens regulares, enquanto os AFs são sistemas reconhecedores do mesmo conjunto de linguagens. Assim, há uma correspondência entre as GR e os AF. Nesta seção são apresentados algoritmos para determinar o AF para reconhecer a linguagem de um GR, bem como para determinar a GR que gera a linguagem de um dado AF.

Teorema 3.2: "Seja G = (N, T, P, S) uma Gramática Regular, então ∃ um AF M = (K, Σ, δ, q_0, F) | L(M) = L(G)".

Prova: a - Mostar que M existe;

b- Mostrar que L(M) = L(G).

a) Defina M como segue:

1 - K = N ∪ {A}, onde A é um novo símbolo não-terminal;

2 - Σ = T;

3 - q_0 = S;

4 - F = {B | B → ε ∈ P} ∪ {A}, ou seja, todos os estados que produzem ε, juntamente com o novo estado.

5 - Construa δ de acordo com as regras a, b e c:

a) Para cada produção da forma B → a ∈ P, crie a transição δ(B,a) = A;

b) Para cada produção da forma B → aC ∈ P, crie a transição δ(B,a) = C;

c) Para produções da forma B → ε, não é criada nenhuma transição.

d) Para todo a ∈ T, δ(A,a) = - (indefinida)

b) Para mostrar que L(M) = L(G), devemos mostrar que (1) L(G) ⊆ L(M) e (2) L(M) ⊆ L(G). Esta demonstração é fácil considerando-se as produções da GR e a definição das transições de M, sendo omitida neste texto.

Exemplo: Dada a GR abaixo, que gera as sentenças da linguagem {($a^n b^m$), com n par e m > 0}, iremos definir um AF que reconheça as sentenças desta linguagem.

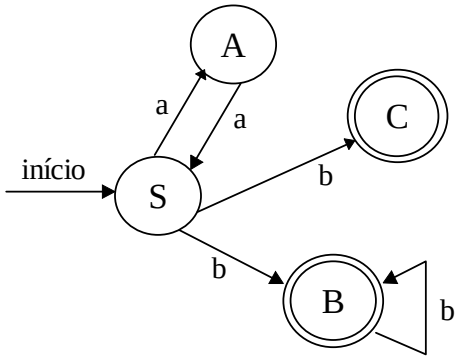
Gramática: S ::= aA | bB | b
A ::= aS
B ::= bB | ε

Autômato:

Primeiramente devemos definir os estados do autômato, com um estado novo, denominado aqui C. Assim, temos que K = {S, A, B, C). O alfabeto é o mesmo, assim Σ = (a, b), o estado inicial é S, e F = {B, C}, pela definição acima. Na forma tabular o autômato fica o seguinte:

	δ	a	b
→	S	A	B, C
	A	S	-
*	B	-	B
*	C	-	-

Na forma gráfica o autômato fica o seguinte:



Teorema 3.3: "Seja um AF M = (K, Σ, δ, q_0, F). Então ∃ uma GR G = (N, T, P, S) | L(G) = L(M)".

Prova: a - Mostra que G existe;

b - Mostrar que L(G) = L(M).

a) Seja M = (K, Σ, δ, q_0, F) um AF. Construa uma G. R. G = (N, T, P, S) como segue:

1 - N = K;

2 - T = Σ;

3 - S = q_0;

4 - Defina P como segue:

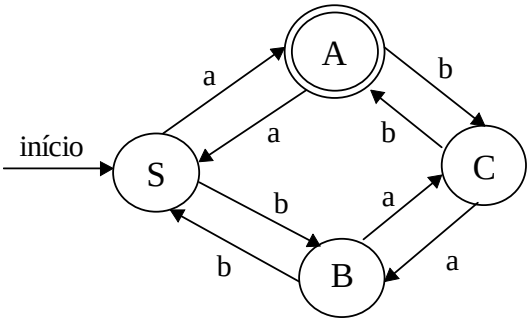
4.a) Se δ(B,a) = C, então adicione B ::= aC em P;

4.b) Se δ(B,a) = C e C ∈ F, então adicione B ::= a em P;

4.c) Se q0 ∈ F, então ε ∈ L(M). Assim a gramática deve ser transformada encontrar uma outra G.R. G1, tal que L(G1) = L(G) ∪ {ε}, e L(G1) = L(M). Senão ε ∉ L(M), e L(G) = L(M).

b) Para mostrar que L(G) = L(M), devemos mostrar que (1) L(M) ⊆ L(G) e (2) L(G) ⊆ L(M). Esta demonstração é análoga à parte (b) da prova do teorema 5.1, sendo omitida neste texto.

Exemplo: Tomemos o seguinte A. F.:



Na forma tabular:

	δ	a	b
→	S	A	B
*	A	S	C
	B	C	S
	C	B	A

A GR definida pelo algoritmo acima resulta em:

S ::= aA | bB | a
A ::= aS | bC
B ::= aC | bS
C ::= aB | bA | b

Pelo algoritmo acima, a GR gerada a partir de um AF nunca apresentará produções do tipo A ::= ε.

4. Minimização de Autômatos Finitos

Definição: Um A. F. D. M = (K, Σ, δ, q_0, F) é MÍNIMO se:

1 - Não possui estados INACESSÍVEIS;

2 - Não possui estados MORTOS;

3 - Não possui estados EQUIVALENTES.

Estados Inacessíveis:

Um estado q ∈ K é inacessível quando não existe w tal que a partir de q_0, q seja alcançado, ou seja, ~∃ w | δ(q0, w) = q, onde w é uma sentença ou parte dela.

Estados Mortos:

Um estado q ∈ K é morto se ele ∉ F e ~∃ w | δ(q, w) = p, onde p ∈ F e w é uma sentença ou parte dela. Ou seja, q é morto se ele não é final e a partir dele nenhum estado final pode ser alcançado.

Estados Equivalentes:

Um conjunto de estados q_1, q_2, ..., q_j são equivalentes entre sí, se eles pertencem a uma mesma CLASSE DE EQUIVALÊNCIA.

Classe de Equivalência (CE):

Um conjunto de estados q_1, q_2, ..., q_j estão em uma mesma CE se δ(q_1, a), δ(q_2, a) ,..., δ(q_j, a), para cada a ∈ Σ, resultam respectivamente nos estados q_i, q_{i+1}, ..., q_n e estes pertencem a uma mesma CE.

4.1. Algoritmo para Construção das Classes de Equivalência

1. Se necessário, crie um estado ϕ para representar as indefinições.
2. Devida K em duas CE, uma contendo F e outra contendo K-F.
3. Divida as CE existentes, formando novas CE (de acordo com a definição - lei de formação das CE), até que nenhuma nova CE seja formada.

4.2. Algoritmo para Construção do Autômato Finito Mínimo

Entrada: Um AFD $M = (K, \Sigma, \delta, q_0, F)$;

Saída: Um AFD Mínimo $M' = (K', \Sigma, \delta', q_0', F') \mid M' \equiv M$;

Método:

1 - Elimine os estados inacessíveis.

2 - Elimine os estados mortos.

3 - Construa todas as possíveis classes de equivalência.

4 - Construa M' como segue:

a) $K' \rightarrow$ é o conjunto de CE obtidas.

b) $q_0' \rightarrow$ será a CE que contiver q_0.

c) $F' \rightarrow$ será o conjunto das CE que contenham pelo menos um elemento $\in$ F; isto é, $\{[q] \mid \exists\ p \in [q]$ e $p \in F$, onde [q] é um CE$\}$.

d) $\delta' \rightarrow \delta'([p], a) = [q] \leftrightarrow \delta(p, a) = q$ é uma transição de M e p e q são elementos de [p] e [q], respectivamente.

Exemplo: Minimize o AFD definido na seguinte tabela de transições:

	δ'	a	b
→*	A	G	B
	B	F	E
	C	C	G
*	D	A	H
	E	E	A
	F	B	C
*	G	G	F
	H	H	D

Temos que os estados acessíveis são: {A, G, B, F, E, C}. Portanto podemos eliminar os estados D e H. Assim o novo AFD é:

	δ'	a	b
→*	A	G	B
	B	F	E
	C	C	G
	E	E	A
	F	B	C
*	G	G	F

Nenhum dos estados do AFD acima é morto. Assim, podemos construir as classes de equivalência. No primeiro passo, duas classes de equivalência são construídas: F e K-F. Cada passo do algoritmo é representado numa linha da tabela abaixo:

	F	**K-F**
1	{A, G}	{B, C, E, F}
2	{A, G}	{B, F} {C, E}
3	{A, G}	{B, F} {C, E}

A primeira linha da tabela é a divisão dos estados em F e K-F. Para i > 1, a linha i é formada pela separação das CEs da linha i-1, até que uma linha seja repetida.

Algoritmicamente, a separação das CEs pode ser dada por:

```
SE houver mais de um estado na CE ENTÃO
        Crie uma nova CE aberta na linha i+1 para o primeiro estado da CE;
        PARA todos os outros estados da CE FAÇA
                Procure encaixar o estado numa CE aberta
                SE não for possível ENTÃO
                        crie uma nova CE aberta na linha i+1 para o estado;
                FIMSE
        FIMPARA
        Feche todas as CEs abertas
SENÃO
        Repita a CE na linha i+1
```

Na tabela acima, na linha 2, foi aberta uma classe para B, e a seguir os outros estados foram testados. C não pôde ficar na mesma classe de B (pois δ(B, b) = E e δ(C, b) = G, e E e G pertencem à CEs separadas na primeira linha. Assim, uma nova CE foi aberta para C. O estado E não pôde ficar na CE de B, mas pôde ficar na CE de C, assim não foi aberta uma nova CE para E. Já o estado

F pôde ficar na CE de B. Como não havia mais estados na CE da primeira linha, as duas CEs da segunda linha foram fechadas.

A repetição do algoritmo para a criação da linha 3 fez com que a linha 2 se repetisse. Assim, todas as CEs foram determinadas.

Denominando {A, G} = q_0, {B, F} = q_1 e {C, E} = q_2, temos o seguinte AFD Mínimo:

	δ'	a	b
→*	q_0	q_0	q_1
	q_1	q_1	q_2
	q_2	q_2	q_0

5. Construção do Analisador Léxico

No processo de compilação, o Analisador Léxico é responsável pela identificação dos *tokens*, ou seja, das menores unidades de informação repassadas para o Analisar Sintático.

Assim, pode-se dizer que o analisador léxico é responsável pela leitura dos caracteres da entrada, agrupando-os em palavras, que são classificadas em categorias. Estas categorias podem ser, basicamente, as seguintes:

Palavras reservadas: palavras que devem aparecer literalmente na linguagem, sem variações. Por exemplo, na linguagem PASCAL, algumas palavras reservadas são: BEGIN, END, IF, ELSE.

Identificadores: palavras que seguem algumas regras de escrita, porém podem assumir diversos valores. São definidos de forma genérica. Por exemplo, identificadores que devem iniciar com uma letra, seguida de letras ou dígitos. Pode-se representar estes identificadores pela seguinte expressão regular: (a + b + ... + c)(a + b + .. + z + 0 + 1 + ... + 9)*

Símbolos especiais: seqüências de um ou mais símbolos que não podem aparecer em identificadores nem palavras reservadas. São utilizados para composição de expressões aritméticas ou lógicas, por exemplo. Exemplos de símbolos especiais são: "**;**" (ponto-e-vírgula), "**:**" (dois pontos), "**:=**" (atribuição).

Constantes: também denominadas *literais*, são valores que são inseridos diretamente no programa, ao invés de serem uma referência para um endereço de memória. Por exemplo, números (904, 8695, -9, 12E-9), strings ("Hello world"), booleanos (TRUE, FALSE), etc.

Para a construção de um analisador léxico, são necessários 4 passos:

1 - definir as gramáticas e autômatos finitos determinísticos mínimos para cada token;

2 - juntar os autômatos num único autômato e determinizar (não minimizar);

3 - acrescentar um estado de erro, para tratar tokens inválidos;

4 - implementar o autômato numa linguagem de programação.

A construção de uma analisador léxico é demonstrada a seguir, com um exemplo:

Sejam os tokens desejados os seguintes:

PALAVRAS RESERVADAS: goto, do.

IDENTIFICADORES: começam com uma letra, seguidas de letras e "_". Expressão Regular: $(a + .. + z)(a +...+ z + \_)^*$

1º passo: definir as gramáticas e autômatos para cada token.

Token "GOTO": S ::= gA

A::= oB

B ::= tC

C ::= o

→ S	A	-	-
A	-	-	B
B	-	C	-
C	-	-	X
*** X**	-	-	-

g	**t**	**o**

Token “DO”:

S ::= dD

D ::= o

	d	**o**
→ S	D	-
D	-	Y
*** Y**	-	-

Token “Identificador”:

S ::= aE | bE | ... | zE

E ::= aE | bE | ... | zE | _E | ε

	a	**b**	**...**	**z**	**_**
→ S	E	E	E	E	-
***E**	E	E	E	E	E

Note que os autômatos acima já foram minimizados, sendo que o estado novo do autômato de identificadores foi eliminado por ser inalcançável.

2º passo: juntar os autômatos num só e determinizar:

Os autômatos acima serão agora colocados juntos num só autômato. Note que já foi tomado o cuidado de somente o nome do estado inicial coincidir. Todos os outros estados têm nomes diferentes. O autômato não-determinístico geral fica:

	g	**t**	**o**	**d**	**a**	**b**	**...**	**z**	**_**
→ S	**A, E**	**E**	**E**	**D, E**	**E**	**E**	**E**	**E**	**-**
A	**-**	**-**	**B**	**-**	**-**	**-**	**-**	**-**	**-**
B	**-**	**C**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
C	**-**	**-**	**X**	**-**	**-**	**-**	**-**	**-**	**-**
D	**-**	**-**	**Y**	**-**	**-**	**-**	**-**	**-**	**-**
*** X**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
*** Y**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**
*** E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**

O autômato acima, determinizado (e não minimizado) fica:

	g	t	o	d	a	b	...	z	_
→ S [S]	A	E	E	D	E	E	E	E	-
* A [A,E]	E	E	B	E	E	E	E	E	E
* B [B, E]	E	C	E	E	E	E	E	E	E
* C [C, E]	E	E	X	E	E	E	E	E	E
* D [D, E]	E	E	Y	E	E	E	E	E	E
* X [X, E]	E	E	E	E	E	E	E	E	E
* Y [Y,E]	E	E	E	E	E	E	E	E	E
* E [E]	E	E	E	E	E	E	E	E	E

O autômato acima poderia ser minimizado, mas isto faria com que todos os estados finais se juntassem num só. Isto faz com que informações importantes sejam perdidas. Note que o estado final do token **GOTO** está no estado que contém o estado final X, e o estado final do token **DO** está no estado final que contém o estado final Y. Os estados que contém o estado final E são os que reconhecem os identificadores. Onde há conflito entre identificadores e palavras reservadas a prioridade é das palavras reservadas.

Assim, se um token terminar no estado X do autômato acima ele será um GOTO, no estado Y será um DO, e nos demais estados finais será um Identificador.

3º passo: acrescentar o estado de erro:

Note que no autômato acima são definidas as transições somente para letras e o caracter "_". Para os demais caracteres (que são inválidos na nossa linguagem) é acrescentado um estado de erro. Este estado será alcançado quando houver um caracter inválido, bem como para as transições indefinidas para os caracteres válidos. Este estado de erro será um estado final, que reportará erro. O autômato acima ficará:

	g	t	o	d	a	b	...	z	_	etc.
→ S	A	E	E	D	E	E	E	E	F	F
* A	E	E	B	E	E	E	E	E	E	F
* B	E	C	E	E	E	E	E	E	E	F
* C	E	E	X	E	E	E	E	E	E	F
* D	E	E	Y	E	E	E	E	E	E	F
* X	E	E	E	E	E	E	E	E	E	F
* Y	E	E	E	E	E	E	E	E	E	F
* E	E	E	E	E	E	E	E	E	E	F
* F	F	F	F	F	F	F	F	F	F	F

(etc. denota os caracteres não válidos da linguagem).

4º passo: implementação:

O autômato acima é facilmente implementado numa estrutura de dados do tipo tabela. Ou seja, no reconhecimento de um novo token, parte-se do estado inicial, determinando-se o novo estado pela coluna correspondente ao caracter lido do arquivo. Repete-se este procedimento até achar um separador de token (espaço em branco, quebra de linha ou final de arquivo).

Quando um separador de token é encontrado, verifica-se qual o estado atual do autômato. Se for um estado não-final, houve um erro, que deve ser acusado. Se for um estado final, verifica-se o que o estado representa, e retorna-se o valor desejado.

No autômato acima, se um separador de tokens é encontrado, verifica-se o estado corrente. Cada estado representa um token ou erro, da seguinte forma:

S → erro;

X → token GOTO

Y → token DO

F → erro - caracter inválido

outro estado → Identificador.

EXERCÍCIOS:

1) Construa um A. F. D. para cada linguagem abaixo.

a) $\{w \mid w \in (0, 1)^+ \text{ e } |w| > 3\}$.

* b) $\{w \mid w \in (0, 1)^+$ e w comece com 0 e termine com 1$\}$

* c) $\{w \mid w \in (0, 1, 2)^+$ e a quantidade de 1's em w seja ímpar$\}$.

d) $\{w \mid w \in (a, b, c)^*$ e w não possua letras iguais consecutivas$\}$.

e) $\{w \mid$ w é um comentário PASCAL não aninhado, na forma (*x*), e $x \in (a, b, ..., z)^*$.

f) {begin, end}. Caso ocorra outra entrada o autômato deve ir para um estado de erro e continuar lá.

2) Construa as G. R. dos autômatos abaixo. A seguir determinize-os e minimize-os. Construa a G. R. dos autômatos resultantes.

* a)

	δ'	a	b
→	A	A, B	A
	B	C	-
	C	-	D
*	D	D	D

b)

	δ'	a	b	c	d
→	A	B, C	D	C	C
	B	-	D	C	C
*	C	-	-	-	D
	D	-	B, C	-	-

c)

	δ'	a	b	c
→*	S	A	B, F	S, F
	A	S, F	C	A
	B	A	-	B, S, F
	C	S, F	-	A, C
*	F	-	-	-

d)

	δ'	0	1
→	A	A, B	A
	B	C	A
	C	D	A
*	D	D	D

3) Construa os A. F. que reconhecem as linguagens geradas pelas seguintes G. R. Caso o A. F. seja não-determinístico, determinize-o. Minimize todos os A. F.

* a) S ::= 0S | 1S | 0A | 0C | 1B
A ::= 0A | 0C | 0
B ::= 1B | 1
C ::= 0C | 0A | 0

b) S ::= aA | aC | bB | bC
A ::= aF | a
B ::= bF | b
C ::= aA | aC | bB | bC
F ::= aF | bF | a | b

c) S ::= aA | bB
A ::= aS | aC | a
B ::= bS | bD | b
C ::= aB
D ::= bA

d) S ::= 0B | 1A | 1 | ε
A ::= 0B | ε
B ::= 0C | 0 | 1D
C ::= 0B | 1A | 1
D ::= 1C | 1

IV - Linguagens Livres de Contexto

1. Linguagens Auto-Embebidas

A importância das LLC's reside no fato de que praticamente todas as linguagens de programação podem ser descritas por este formalismo, e que um conjunto bastante significativo destas linguagens pode ser analisado por algoritmos muito eficientes. A maioria das linguagens de programação pertence a um subconjunto das LLC's que pode ser analisado por algoritmos que executam *n.log(n)* passos para cada símbolo da sentença de entrada. Outro subconjunto que corresponde às linguagens não ambíguas pode ser analisado em tempo n^2. No pior caso, uma LLC ambígua pode ser analisada em tempo n^3. O que ainda é bastante eficiente, se comparado às linguagens sensíveis ao contexto, que podem requerer complexidade exponencial (2^n), sendo, portanto, computacionalmente intratáveis, e às linguagens de tipo 0, que podem ser, inclusive, indecidíveis, isto é, o algoritmo de análise pode não parar nunca.

2. Gramáticas Livres de Contexto

Uma *gramática livre de contexto* (GLC) é denotada por G=(N,T,P,S), onde N e T são conjuntos disjuntos de variáveis e terminais, respectivamente. P é um conjunto finito de produções, cada uma da forma $A ::= \alpha$, onde A é uma variável do conjunto N e α é uma cadeia de símbolos $(N \cup T)^*$. Finalmente, S é uma variável especial denominada símbolo inicial.

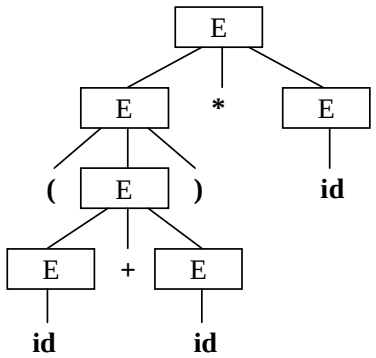
3. Árvore de Derivação

Muitas vezes é útil mostrar as derivações de uma GLC através de árvores. Estas figuras, chamadas árvores de derivação ou árvores sintáticas, impõe certas estruturas úteis às sentenças das linguagens geradas, principalmente as de programação.

Os vértices de uma árvore de derivação são rotulados com terminais ou variáveis, ou mesmo com a cadeia vazia. Se um vértice interior n é rotulado com A, e os filhos de n são rotulados com $X_1, X_2,...,X_k$, da esquerda para a direita, então $A::= X_1X_2...X_k$ deve ser uma produção da gramática. Considere, como exemplo, a gramática:

```
E  ::=   E + E
       | E * E
       | ( E )
       | id
```

Uma árvore de derivação para a sentença "(id+id)*id" gerada por esta gramática poderia ser:



Mais formalmente, seja G=(N,T,P,S) uma GLC. Então uma árvore é uma árvore de derivação para G, se:

a) cada vértice tem um rótulo, que é um símbolo de $N \cup T \cup \{\varepsilon\}$;

b) o rótulo da raiz é o símbolo S;

c) os vértices interiores são rotulados apenas com símbolos de N;

d) se o vértice n tem rótulo A, e os vértices n_1, n_2, ..., n_k são fílhos de n, da esquerda para a direita, com rótulos $X_1,X_2,...,X_k$, respectivamente, então A ::= $X_1X_2...X_k$ deve ser uma produção em P;

e) se o vértice n tem rótulo ε, então n é uma folha ε e é o único filho de seu pai.

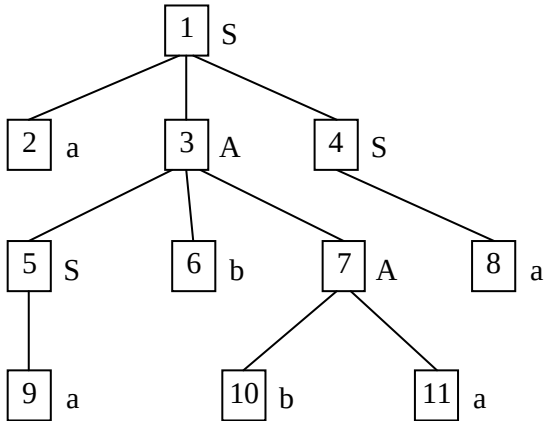
Exemplo:

Considere a gramática G=({S,A},{a,b},P,S), onde P consiste de:

S ::= aAS | a

A ::= SbA | SS | ba

A figura seguinte é uma árvore de derivação para uma sentença desta gramática, na qual os nodos foram numerados para facilitar a explicação:



Os vértices interiores são 1, 3, 4, 5 e 7. O vértice 1 tem o rótulo S, e seus filhos, da esquerda para a direita têm rótulos a, A e S. Note que S ::= aAS é uma produção. Da mesma forma, o vértice 3 tem rótulo A, e os rótulos de seus filhos são S, b e A, da esquerda para a direita, sendo que A := SbA é também uma produção. Os vértices 4 e 5 têm ambos o rótulo S. O único filho deles tem rótulo a, e S ::= a também é uma produção. Finalmente, o vértice 7 tem rótulo A, e seus filhos, da esquerda para a direita, são b e a, e A ::= ba também é produção. Assim, as condições para que esta árvore seja uma árvore de derivação para G foram cumpridas.

Pode-se estender a ordem "da esquerda para a direita" dos filhos para produzir uma ordem da esquerda para a direita de todas as folhas. No exemplo acima, o caminhamento da esquerda para a direita nas folhas da árvore produziria a seguinte seqüência: 2, 9, 6, 10, 11 e 8.

Pode-se ver que a árvore de derivação é uma descrição natural para a derivação de uma forma sentencial particular da gramática G.

Uma sub-árvore de uma árvore de derivação é composta por um vértice particular da árvore, juntamente com seus descendentes.

4. Derivação mais à Esquerda e mais à Direita

Se cada passo na produção de uma derivação é aplicado na variável mais à esquerda, então a derivação é chamada derivação mais à esquerda, Similarmente, uma derivação onde a cada passo a variável mais à direita é substituída, é chamada de derivação mais à direita.

Se w está em L(G), então w tem ao menos uma árvore de derivação. Além disso, em relação a uma árvore de derivação particular, w tem uma única derivação mais à esquerda e uma única derivação mais à direita.

Evidentemente, w pode ter várias derivações mais à esquerda e várias derivações mais à direita, já que pode haver mais de uma árvore de derivação para w. Entretanto, é fácil mostrar que, para cada árvore de derivação, apenas uma derivação mais à esquerda e uma derivação mais à direita podem ser obtidas.

Exemplo:

A derivação mais à esquerda correspondente à árvore do exemplo anterior é:

$S \rightarrow aAS \rightarrow aSbAS \rightarrow aabAS \rightarrow aabbaS \rightarrow aabbaa$

A derivação mais à direita correspondente é:

$S \rightarrow aAS \rightarrow aAa \rightarrow aSbAa \rightarrow aSbbaa \rightarrow aabbaa$

5. Ambigüidade

Se uma gramática G tem mais de uma árvore de derivação para uma mesma sentença, então G é chamada de *gramática ambígua*. Uma linguagem livre de contexto para a qual toda gramática livre de contexto é ambígua é denominada *linguagem livre de contexto inerentemente ambígua.*

6. Simplificações de Gramáticas Livres de Contexto

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

a) Cada variável e cada terminal de G aparecem na derivação de alguma palavra de L.

b) Não há produções da forma A ::= B, onde A e B são variáveis.

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

6.1. Símbolos Inúteis

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

Há dois tipos de símbolos inúteis: aqueles que não geram nenhuma cadeia de terminais e aqueles que jamais são gerados a partir de S. O primeiro caso corresponde aos *símbolos improdutivos* (ou *mortos*), e o o segundo caso corresponde aos *símbolos inalcançáveis.* Um terminal sempre é produtivo, mas pode ser inalcançável. Já o não-terminal pode ser tanto improdutivo quanto inalcançável, mas o símbolo inicial sempre é alcançável. A seguir serão vistos algoritmos para eliminar tanto o primeiro quanto o segundo tipo de símbolos inúteis.

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

Algoritmo: Eliminação de Símbolos Improdutivos

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

SP := $T \cup \{\varepsilon\}$
Repita
    Q := {X | X ∈ N e X ∉ SP e (existe pelo menos uma produção X ::= $X_1X_2,...X_n$ tal que $X_1 \in SP$, $X_2 \in SP$, ..., $X_n \in SP$)};
    SP := $SP \cup Q$;
Até Q = ∅;

```
N'= SP ∩ N;
Se S ∈ SP Então
        P':= {p | p∈P e todos os símbolos de p pertencem a SP}
Senão "L(G) = ∅";
        P':= ∅;
Fim Se
Fim.
```

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

Seja G a seguinte gramática:

S ::= ASB | BSA | SS | aS | ε
A ::= ABS | B
B ::= BSSA | A

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

S ::= aS
S ::= ε

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

S ::= ASB | BSA | SS | aS | ε
A ::= ABS | B
B ::= BSSA | A

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

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

S ::= SS | aS | ε

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

Algoritmo: Eliminação de Símbolos Inalcançáveis

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

SA := {S}
Repita
    M := {X | X ∈ N∪T e X ∉ SA e existe uma produção Y ::= αXβ e Y ∈ SA};
    SA := SA ∪ M;
Até M = ∅;
N'= SA ∩ N;
T'= SA ∩ T;
P':= {p | p∈P e todos os símbolos de p pertencem a SA};
Fim.

Para exemplificar, seja G:

S ::= aS | SB | SS | ε
A ::= ASB | c
B ::= b

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

S ::= <u>aS</u> | <u>SB</u> | <u>SS</u> | ε

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

B ::= <u>b</u>

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

S ::= aS | SB | SS | ε
B ::= b

6.2. Eliminação de ε-Produções

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

O método consiste em determinar, para cada variável A em N, se $A \rightarrow^* \varepsilon$. Se isto for verdade, se diz que a variável A é anulável. Pode-se assim substituir cada produção da forma $B ::= X_1X_2...X_n$ por todas as produções formadas pela retirada de uma ou mais variáveis X_i anuláveis.

Exemplo:

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

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

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

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

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

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

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

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

Algoritmo: Encontrar o Conjunto dos ε-não-terminais

Entrada: Uma GLC G=(N,T,P,S)
Saída: O conjunto E dos ε-não-terminais.

E := {ε}
Repita
    Q := {X | X ∈ N e X ∉ E e existe pelo menos uma produção $X ::= Y_1Y_2...Y_n$ tal que $Y_1 \in E$, $Y_2 \in E$, ... , $Y_n \in E$ };
    E := E ∪ Q;
Até Q = ∅;
Fim.

Seja G a seguinte gramática:

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

Inicialmente temos E = {ε}.

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

S ::= aS | SS | bA A ::= B<u>C</u> B ::= <u>CC</u> | ab| aAb<u>C</u> C ::= <u>ε</u>

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

S ::= aS | SS | bA A ::= <u>BC</u> B ::= <u>CC</u> | ab| aAb<u>C</u> C ::= <u>ε</u>

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

S ::= aS | SS | b<u>A</u> A ::= <u>BC</u> B ::= <u>CC</u> | ab | a<u>A</u>b<u>C</u> C ::= <u>ε</u>

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

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

Algoritmo: Eliminar Todos os ε-não-terminais

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

Construa o conjunto E
P' := {p | p ∈ P e p não é ε-produção }
Repita
    Se P' tem uma produção da forma A ::= αBβ, tal que B ∈ E, αβ $(N \cup T)^*$ e αβ ≠ ε, então
        inclua a produção A ::= αβ em P';
Até que nenhuma nova produção possa ser adicionada a P';
Se S ∈ E, então
    Adicione a P'as produções S' ::= S | ε;
    N' := N ∪ {S'};
Senão
    S' := S;
    N' := N;
Fim Se
Fim.

Exemplo:

Dada a seguinte GLC:

S ::= SaBC | ABb | BC
A ::= bC | aBC | a
B ::= BCA | bC | ε
C ::= ab | ε

O conjunto dos ε-não-terminais é E = {S, B, C}. A gramática ε-livre equivalente é:

S' ::= S | ε
S ::= SaBC | ABb | BC | aBC | SaC | SaB | Ab | C | B | aC | aB | Sa | a
A ::= bC | aBC | a | b | aB | aC
B ::= BCA | bC | CA | BA | b | A
C ::= ab

6.3. Produções Unitárias

Uma produção da forma A ::= α em uma GLC G=(N,T,P,S) é chamada de *produção unitária* se α é um não-terminal. Um caso especial de produção unitária é a produção A ::= A, também chamada *produção circular*. Tal produção pode ser removida imediatamente sem afetar a capacidade de geração da gramática. O algoritmo para eliminar produções unitárias assume que as

produções circulares já tenham sido eliminadas anteriormente. Essa eliminação é trivial, pois a identificação de produções circulares é bastante simples.

Algoritmo: Eliminar Produções Unitárias

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

Para toda $A \in N$ faça:
    $N_A := \{B \mid A \rightarrow^* B, \text{ com } B \in N\}$;
Fim Para
$P' := \varnothing$;
Para toda produção $B ::= \alpha \in P$ faça:
    Se $B ::= \alpha$ não é uma produção unitária, então:
        $P' := P' \cup \{A ::= \alpha \mid B \in N_A\}$;
    Fim Se
Fim Para
Fim.

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

S ::= aB | aA | A B ::= aA | B | A
A ::= C | aaa C ::= ε | cBc

Primeiramente é necessário determinar o conjunto de N_K para cada não terminal K:

$N_S = \{S, A, B, C\}$ $N_B = \{B, A, C\}$ $N_A = \{A, C\}$ $N_C = \{C\}$

As produções de cada símbolo K são as produções $R ::= \alpha$, onde R é um elemento de N_K e α é uma produção não-unitária. No exemplo, as produções de S são as produções não unitárias de S, A, B e C. Assim, a nova GLC sem produções unitárias é

S ::= aB | aA | aaa | cBc | ε B ::= aA | aaa | cBc | ε
A ::= aaa | ε C ::= ε | cBc

Nesta nova GLC, o não-terminal C é inalcançável.

7. Fatoração

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

S ::= aSB | aSA
A ::= a
B ::= b

fonte do não-determinismo

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

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

$A ::= \alpha A'$

A' ::= β | γ

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

S ::= aSS'
S'::= B | A
A ::= a
B ::= b

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

S ::= AC | BC
A ::= aD | cC
B ::= aB | dD
C ::= eC | eA
D ::= fD | AB

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

S ::= aCC | cBC | aBC | dDC
A ::= aD | cB
B ::= aB | dD
C ::= eC | eA
D ::= fD | AB

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

S ::= aS' | cBC | dDC
S'::= CC | BC
A ::= aD | cB
B ::= aB | dD
C ::= eC'
C' ::= C | A
D ::= fD | AB

8. Eliminação de Recursão à Esquerda

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

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

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

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

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

Para eliminar as recursões diretas à esquerda nas produções:

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

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

$A ::= \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_m A'$
$A' ::= \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_n A' \mid \varepsilon$

onde A' é um novo não-terminal.

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

Algoritmo: Eliminar Recursões à Esquerda

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

N' := N;
P' := P;
Ordene os não-terminais de N' em uma ordem qualquer (por exemplo, A_1, A_2, A_3, ..., A_n);
Para i de 1 até n, faça:
    Para j de 1 até i-1 faça
        Substitua as produções de P' da forma $A_i ::= A_j\gamma$ por produções da forma $A_i ::= \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$, onde δ_1, δ_2, ... δ_k são os lados direitos das produções com lado esquerdo A_j, ou seja, $A_j ::= \delta_1 \mid \delta_2 \mid ... \mid \delta_k$.
    Fim Para;
    Elimine as recursões diretas das produções de P'com lado esquerdo A_i;
Fim Para;
Fim.

Note que este algoritmo já elimina as recursões à esquerda diretas e indiretas. Para exemplificar, considere a seguinte gramática:

S ::= Sb | Cc | Ab
A ::= Sc | ab
C ::= Scd | Cba | b

Primeiramente, (i = 1), a recursão a esquerda de S é eliminada:

S ::= CcS' | AbS'
A ::= Sc | ab
C ::= Scd | Cba | b
S' ::= bS' | ε

No segundo passo (i = 2), temos que substituir S nas produções de A que começam por S:

S ::= CcS' | AbS'
A ::= CcS'c | AbS'c | ab
C ::= Scd | Cba | b
S' ::= bS' | ε

Em seguida, eliminar a recursão a esquerda direta:

S ::= CcS' | AbS'
A ::= CcS'cA' | abA'
C ::= Scd | Cba | b
S' ::= bS' | ε
A' ::= bS'cA' | ε

No terceiro passo (i = 3), temos que substituir S (j = 1) nas produções de C que começam por S:

S ::= CcS' | AbS'
A ::= CcS'cA' | abA'
C ::= CcS'cd | AbS'cd | Cba | b
S' ::= bS' | ε
A' ::= bS'cA' | ε

Em seguida substituir A (j = 2) nas novas produções de C que começam por A:

S ::= CcS' | AbS'
A ::= CcS'cA' | abA'
C ::= CcS'cd | CcS'cA'bS'cd | abA'bS'cd | Cba | b
S' ::= bS' | ε
A' ::= bS'cA' | ε

Por fim, eliminar as recursões a esquerda diretas:

S ::= CcS' | AbS'
A ::= CcS'cA' | abA'
C ::= abA'bS'cdC' | bC'
S' ::= bS' | ε
A' ::= bS'cA' | ε
C' ::= cS'cdC' | cS'cA'bS'cdC' | baC' | ε

9. Tipos Especiais de GLC

A seguir serão identificados alguns tipos especiais de GLC

a) Gramática Própria: Uma GLC é própria se:

a.1) Não possui produções cíclicas (ver definição seguinte);

a.2) É ε-livre;

a.3) Não possui símbolos inúteis.

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

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

c.1) L(G) não é vazia;

c.2) Se $A ::= \alpha \in P$ então $A \neq \alpha$

c.3) G não possui símbolos inúteis

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

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

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

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

g.1) A ::= BC, com A, B e C ∈ N, ou

g.2) A ::= a; com A ∈ N e a ∈ T.

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

10 Principais Notações de GLC

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

<S> ::= a<S> | ε <E> ::= <E>+id | id

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

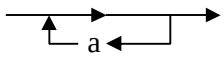
<S> ::= {a} <E> ::= id{+id}

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

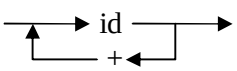
$<S> ::= a^*$ $<E> ::= id(+id)^*$

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

<S> :



<E>:



11. Conjuntos First e Follow

11.1 Conjunto First

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

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

a) X = ε. Neste caso, FIRST(X) = {ε}.

b) $X \in T$. Neste caso, FIRST(X) = {X}.

c) $X \in N$. Neste caso, se $X ::= \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n \mid$ são todas as produções com lado esquerdo X, então $FIRST(X) = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup ... \cup FIRST(\alpha_n)$.

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

Algoritmo: Cálculo do Conjunto First de uma Cadeia
Entrada: Uma cadeia de símbolos $X \in (N \cup T)^*$ de uma gramática G=(N,T,P,S). Uma tabela TABFIRST com o conjunto FIRST de cada não-terminal de G.

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

Se X = ε então:
        F := {ε}.
Senão se X ∈ T então:
        F = {X}
Senão se X ∈ N então:
        Sejam X ::= α1 | α2 | ... | αn as produções com lado esquerdo X, então
                F := FIRST(α1) ∪ FIRST(α2) ∪...∪ FIRST(αn);
Senão
        i := 0;
        F := ∅;
        Repita:
                i := i+1;
                Xi := o i-ésimo símbolo da cadeia X;
                F := F ∪ TABFIRST(Xi) - {ε};
                Se X = ε então:
                        F := F ∪ {ε};
                Fim Se
        Até X ∈ T ou X = ε ou (X ∈ N e ε ∉ TABFIRST(X))
Fim Se
```

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

1. Crie uma tabela TABFIRST cujas linhas são rotuladas com os não-terminais da gramática e com uma coluna rotulada como “FIRST”, que irá conter os símbolos pertencentes ao FIRST de cada não-terminal. Inicialize cada uma das posições desta tabela com ∅ (conjunto vazio).
2. Para cada um dos não-terminais, calcule o seu conjunto FIRST utilizando o algoritmo acima.
3. Repita o passo 2 até que não ocorra mais nenhuma modificação na tabela.

Para exemplificar, considere a gramática:

S ::= ABS | aA
A ::= ε | a
B ::= Bb | cd

Inicialmente constrói-se a tabela:

N	FIRST
S	∅
A	∅
B	∅

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

N	FIRST
S	{a}
A	∅

B	∅

Calculando o FIRST(A) temos {ε, a}. Então a tabela fica:

N	FIRST
S	{a}
A	{ε, a}
B	∅

Calculando o FIRST(B) temos FIRST(B) = FIRST(Bb) ∪ FIRST(cd) = ∅ ∪ {c}. A tabela fica:

N	FIRST
S	{a}
A	{ε, a}
B	{c}

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

N	FIRST
S	{a, c}
A	{ε, a}
B	{c}

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

11.2. Conjunto Follow

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

Algoritmo: Cálculo do Conjunto Follow de um Não-Terminal

Entrada: Uma gramática G=(N,T,P,S).
Saída: Uma tabela TABFOLLOW com o conjunto FOLLOW de cada não-terminal de G.

Para todo X∈N, exceto S:
    TABFOLLOW(X) := ∅;
Fim Para
TABFOLLOW(S) := {$}; (marca de final de sentença)
Para toda ocorrência de um não-terminal X em uma produção Y ::= αXβ ∈ P com β ≠ ε:
    TABFOLLOW(X) := TABFOLLOW(X) ∪ FIRST(β) - {ε}
Fim Para;
Repita:
    Para cada produção da forma X ::= $Y_1Y_2...Y_n$ faça:
        i ::= n+1;
        Repita:
            i := i-1;
            Se Y_i ∈ N então:
                TABFOLLOW(Y_i) := TABFOLLOW(X) ∪ TABFOLLOW(Y_i)
            Fim Se

```
            Até  Yi ∉ N ou ε ∉ TABFIRST(Yi)
        Fim Para
Até que não haja mais modificações na tabela TABFOLLOW
```

Seguem duas observações importantes:

a) A função FIRST é definida para cadeias de símbolos, FOLLOW só é definida para não-terminais.

b) Os elementos de um conjunto FIRST pertencem ao conjunto $T \cup \{\varepsilon\}$; os elementos de um conjunto FOLLOW pertencem ao conjunto $T \cup \{\$\}$.

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

S ::= ABS | aA A ::= ε | a B ::= Bb | cd

cuja tabela FIRST é:

N	FIRST
S	{a, c}
A	{ε, a}
B	{c}

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

N	FOLLOW
S	{$}
A	∅
B	∅

Para o primeiro passo, temos as ocorrências de Y ::= αXβ em S ::= ABS e B ::= Bb, assim:

FOLLOW(A) = FOLLOW(A) ∪ FIRST(BS) = ∅ ∪ {c} = {c}

FOLLOW(B) = FOLLOW(B) ∪ FIRST(S) = ∅ ∪ {a, c} = {a, c}

FOLLOW(B) = FOLLOW(B) ∪ FIRST(b) = {a, c} ∪ {b} = {a, b, c}

Temos então a tabela FOLLOW intermediária:

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

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

FOLLOW(S) = FOLLOW(S) ∪ FOLLOW(S) = {$}

Como ε ∉ FIRST(S), a segunda regra termina para esta produção. Para a produção S ::= aA, temos:

FOLLOW(A) ::= FOLLOW(A) ∪ FOLLOW(S) = {c, $}

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

N	FOLLOW

S	{$}
A	{c, $}
B	{a, b, c}

EXERCÍCIOS

1. Seja G a seguinte GLC:

S ::= aScSa | ε

a) Mostre três palavras (sentenças) em L(G) e alguma árvore de derivação para cada uma delas.

b) Mostre todas as árvores sintáticas para a palavra “aacacaacacaa”.

2. Seja G a seguinte GLC:

S ::= SS | aSf | af | bSe | be | cSd | cd

mostre todas as árvores de derivação para a palavra “abafcdefbcbdde”.

3. Seja G uma GLC:

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

encontre uma GLC reduzida G’ equivalente a G retirando os símbolos improdutivos e inalcançáveis.

4. Dada uma GLC:

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

a) Quais são os ε-não-terminais de G?

b) $\varepsilon \in L(G)$

c) Construa uma GLC ε-livre equivalente a G (aplicando o algoritmo).

5. Considere a seguinte GLC:

S ::= B | aB
A ::= B | aB
B ::= C | aC
C ::= b

a) Quais são as produções unitárias de G?

b) Construa uma GLC equivalente livre de produções unitárias.

6. A linguagem L(G) = { $a^i b^j c^m d^n$ | (i=j e n=m) ou (i=n e j=m)} é inerentemente ambígua? Caso positivo, construa G ambígua que a representa. Caso negativo, construa G não-ambígua que a representa. Em qualquer caso, construa todas as árvores de derivação e todas as derivações das seguintes sentenças: abcd e bbcc.

7. Elimine os símbolos inúteis da seguinte GLC:

S ::= aSa | FbD
D ::= Dd | fF | c
A ::= aA | CA | ε

E ::= BC | eE | EB
B ::= bB | FE
F ::= fF | Dd
C ::= cCb | AcA

8. Transforme em ε-livre e elimine as produções unitárias das seguintes GLC's:

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

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

9. Elimine as recursões à esquerda das seguintes GLC's:

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

* b) S ::= BaS | ε
B ::= SAa | Bb
A ::= Sa | ε

10. Fatore as seguintes GLC's:

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

b) P ::= DL | L
D ::= dD | ε
L ::= L;C | C

c) C::= V=exp | id(E)
V ::= id[E] | id
E ::= exp,E | exp

11. Procure responder as seguintes questões justificando a resposta:

a) Toda linguagem livre de contexto pode ser gerada por uma GLC própria?

b) Toda linguagem livre de contexto pode ser representada por uma GLC na forma normal de Chomsky?

c) Toda GLC pode ser fatorada?

d) Ambigüidade e recursão à esquerda implicam em não-fatoração?

12. A união, concatenação e intercessão das linguagens de duas GLC quaisquer são também linguagens livres de contexto?

13. Calcule o FIRST e FOLLOW das gramáticas resultantes dos exercícios anteriores.

V - Autômatos de Pilha

1. Introdução

Assim como as linguagens regulares têm nos autômatos finitos seus sistemas reconhecedores, também as linguagens livres de contexto têm seus sistemas reconhecedores: os *autômatos de pilha* (AP).

Porém, neste caso, a relação entre GLCs e APs não é tão satisfatória como a relação entre GRs e AFs. O autômato de pilha é um dispositivo não-determinístico, e a versão determinística aceita apenas um subconjunto de todas as linguagens livres de contexto. Felizmente, este subconjunto contém a sintaxe da maioria das linguagens de programação.

2. Autômatos de Pilha

O autômato de pilha é, essencialmente, um autômato finito com controle tanto da entrada quanto de uma pilha. Esta pilha, acoplada a um controle finito, pode ser usada para reconhecer conjuntos não-regulares. Considere a linguagem livre de contexto gerada pela gramática S ::= 0S0 | 1S1 | c. Não é difícil mostrar que L não pode ser reconhecida por um autômato finito. Para aceitar L, usaremos um controle finito com dois estados e_1 e e_2 e uma pilha na qual colocamos pratos azuis, verdes e vermelhos. O dispositivo vai operar de acordo com as seguintes regras:

1. A máquina inicia com um prato vermelho no topo e com o controle finito no estado e_1.
2. Se a entrada é 0 e o controle está no estado e_1, um prato azul é colocado na pilha. Se a entrada é 1 e o dispositivo está no estado e_1, um prato verde é colocado na pilha. Em ambos os casos, o dispositivo permanece no estado e_1.
3. Se a entrada é c e o dispositivo está no estado e1, ele muda para o estado e2 e não modifica a pilha.
4. Se a entrada é 0 e o dispositivo está no estado e_2 com um prato azul, que representa 0, no topo da pilha, o prato é removido. Se a entrada é 1 e o dispositivo está no estado e_2 com um prato verde, que representa 1, no topo da pilha, o prato é removido. Em ambos os casos, o controle finito permanece no estado e_2.
5. Se o dispositivo está no estado e_2 e um prato vermelho está no topo da pilha, o prato é removido sem esperar pela próxima entrada.
6. Para todos os casos não descritos acima, o dispositivo não pode fazer nenhuma transição ou movimento na pilha.

Dizemos que o dispositivo descrito acima aceita uma cadeia de entrada se, quando processar o último símbolo da cadeia, a pilha da pratos fica completamente vazia. Note que quando a pilha está vazia, mais nenhum movimento é possível.

As regras do dispositivo são resumidas na seguinte tabela:

Prato no topo	Estado	Entrada		
		0	1	c
Azul	e_1	Empilhe azul Fique em e_1	Empilhe verde Fique em e_1	Vá para e_2
	e_2	Desempilhe Fique em e_2	-	-
Verde	e_1	Empilhe azul Fique em e_1	Empilhe verde Fique em e_1	Vá para e_2
	e_2	-	Desempilhe Fique em e_2	-
Vermelho	e_1	Empilhe azul Fique em e_1	Empilhe verde Fique em e_1	Vá para e_2
	e_2	Sem consumir a entrada, desempilhe		

Essencialmente, o dispositivo opera da seguinte forma: no estado e_1 o dispositivo faz uma imagem de sua entrada, colocando um prato azul no topo da pilha cada vez que um 0 aparece na entrada e um prato verde cada vez que um 1 aparece na entrada. Quando a entrada é "c", o dispositivo vai para o estado e_2. A seguir, o restante da entrada é comparado com a pilha, removendo um prato azul cada vez que o símbolo da entrada é um 0 e um prato verde cada vez que o símbolo é um 1. Se o prato no topo da pilha for da cor errada, o dispositivo pára e nenhum processamento posterior na entrada é possível. Se todos os pratos correspondem às entradas, eventualmente o prato vermelho do fundo da pilha será exposto. O prato vermelho é imediatamente removido e o dispositivo aceitou a entrada. Todos os pratos podem ser removidos apenas quando a cadeia após o "c" é a reversa da que precedia "c".

2.1. Definição Formal

O autômato de pilha (AP) é composto por uma entrada, um autômato finito e uma pilha. A pilha é uma cadeia de símbolos de algum alfabeto. O símbolo mais à esquerda da pilha é considerado como seu topo. A máquina será não-determinística, tendo um número finito de escolhas de movimentos a efetuar em cada situação. Os movimentos podem ser de dois tipos.

No primeiro tipo de movimento o símbolo da entrada é usado. Dependendo do símbolo da entrada, do topo da pilha e do estado do autômato finito, algumas escolhas são possíveis. Cada escolha consiste de um estado seguinte para o autômato finito e uma cadeia de símbolos (possivelmente vazia) para ser trocada pelo topo da pilha. Após selecionar uma escolha, um símbolo é consumido da entrada.

O segundo tipo de movimento (chamado ε-movimento) é similar ao primeiro, exceto que ele não consome símbolos da entrada. Este tipo de movimento permite que o AP manipule a pilha sem ler símbolos da entrada.

Finalmente, devemos definir a linguagem aceita por um autômato de pilha. Há duas formas naturais para fazer isto. A primeira, que já foi vista, é definir a linguagem aceita como sendo o conjunto de todas as entradas para as quais alguma seqüência de movimentos faz com que a pilha do autômato fique vazia. Esta linguagem é conhecida como a linguagem aceita por pilha vazia.

A segunda forma de definir a linguagem reconhecida é similar à forma utilizadas para autômatos finitos. Isto é, designamos alguns estados como estados finais e definimos a linguagem aceita como o conjunto de todas as entradas para as quais alguma escolha de movimentos leve a um estado final.

Como podemos ver, as duas definições são equivalentes no sentido de que um conjunto que pode ser aceito por pilha vazia em um AP pode ser aceito por um estado final em outro AP, e vice-versa.

Reconhecimento por estado final é a noção mais comum, mas é fácil provar o teorema básico dos autômatos de pilha usando reconhecimento por pilha vazia. Este teorema estabelece que uma linguagem é aceita por um AP se e somente se ela é uma linguagem livre de contexto.

Um *autômato de pilha* M é uma tupla (E, A, P, t, e_0, p_0, F), onde:

- E é um conjunto finito de estados;
- A é um alfabeto de entrada;
- P é um alfabeto da pilha;
- $e_0 \in E$ é o estado inicial;
- $p_0 \in P$ é o símbolo inicial da pilha;
- $F \subseteq E$ é o conjunto de estados finais;
- t é um mapeamento de $E \times \{A \cup \{\varepsilon\}\} \times P$ em subconjuntos finitos de $E \times P^*$.

A menos que seja explicitamente dito, usaremos letras minúsculas do início do alfabeto para denotar símbolos da entrada e letras do fim do alfabeto para denotar cadeias de símbolos de entrada. Letras maiúsculas denotarão símbolos da pilha e letras gregas indicarão cadeias de símbolos da pilha.

2.1.1. Movimentos

A interpretação de:

$t(e, a, p) = \{(d_1, \gamma_1), (d_2, \gamma_2), ..., (d_m, \gamma_m)\}$

onde "e" e d_i, $1 \le i \le m$, são estados, a está em A, p é um símbolo da pilha, e γ_i está em P^* para $1 \le i \le m$, é que o AP no estado e, com o símbolo de entrada a e p no topo da pilha, pode, para algum i, entrar no estado d_i, trocar p pela cadeia γ_i e consumir um símbolo da entrada. É adotada a convenção de que o símbolo mais à esquerda de γ_i será colocado no topo da pilha.

A interpretação de:

$t(e, \varepsilon, p) = \{(d_1, \gamma_1), (d_2, \gamma_2), ..., (d_m, \gamma_m)\}$

é que o AP no estado e, independente do símbolo da entrada, e com o valor p no topo da pilha, pode entrar no estado d_i e trocar p por γ_i, para algum i, $1 \le i \le m$. Neste caso, nenhum símbolo da entrada é consumido.

Considere o seguinte exemplo:

$M = (\{e_1, e_2\}, \{0, 1, c\}, \{R, B, G\}, t, e_1, R, \varnothing)$

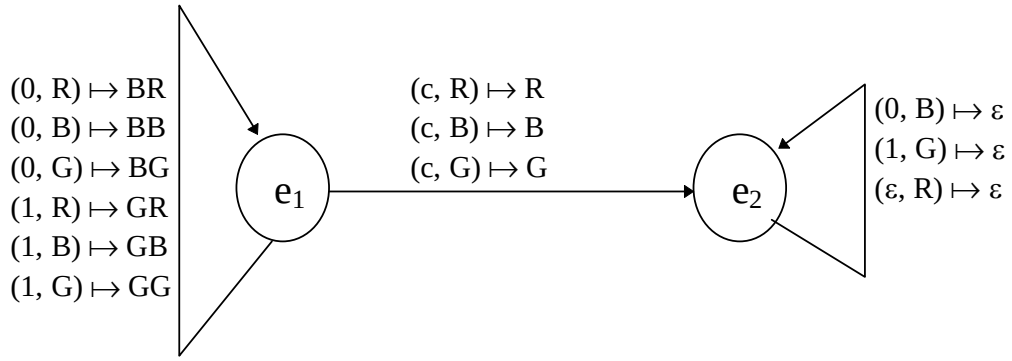
$t(e_1, 0, R) = \{(e_1, BR)\}$	$t(e_1, 0, B) = \{(e_1, BB)\}$	$t(e_1, 0, G) = \{(e_1, BG)\}$
$t(e_1, 1, R) = \{(e_1, GR)\}$	$t(e_1, 1, B) = \{(e_1, GB)\}$	$t(e_1, 1, G) = \{(e_1, GG)\}$
$t(e_1, c, R) = \{(e_2, R)\}$	$t(e_1, c, B) = \{(e_2, B)\}$	$t(e_1, c, G) = \{(e_2, G)\}$
$t(e_2, \varepsilon, R) = \{(e_2, \varepsilon)\}$	$t(e_2, 0, B) = \{(e_2, \varepsilon)\}$	$t(e_2, 1, G) = \{(e_2, \varepsilon)\}$

O AP acima reconhece a linguagem $\{ wcw^{reversa} \mid w \in (0 + 1)^* \}$ por pilha vazia. Note que para um movimento no qual o AP escreve um símbolo no topo da pilha, t tem o valor (e, γ), onde $|\gamma| = 2$. Por exemplo, $t(e_1, 0, R) = \{(e_1, BR)\}$. Se γ fosse de comprimento 1, o AP simplesmente trocaria o símbolo no topo da pilha. Isto nos permite usar γ igual a ε quando queremos desempilhar.

2.1.2. Notação Gráfica

A notação gráfica dos AP é parecida com a notação dos autômatos finitos. O que muda são os rótulos das transições. Ao invés de apenas indicar o símbolo da entrada, é necessário indicar também o símbolo que deve estar no topo da pilha e a transformação a se fazer na pilha. Por exemplo, uma transição do estado e_1 para e_2, rotulada com $(c, R) \mapsto R$ representa a transição $t(e_1, c, R) = \{(e_2, R)\}$

Para o AP acima, a representação gráfica é:



3. Autômatos de Pilha Determinísticos

O autômato de pilha mostrado na seção anterior é determinístico no sentido de que no máximo um movimento é possível para cada situação. Formalmente, dizemos que um AP $M = (E, A, P, t, e_0, p_0, F)$ é *determinístico* se:

1. Para cada $e \in E$ e $p \in P$, sempre que $t(e, \varepsilon, p)$ não for vazio, então $t(e, a, p)$ é vazio para todo $a \in A$.
2. Para nenhum $e \in E$, $p \in P$ e $a \in A \cup \{\varepsilon\}$, acontece que $t(e, a, p)$ contenha mais de um elemento.

A primeira regra descreve o não determinismo entre o consumo ou não do símbolo da entrada. A segunda regra é análoga à regra dos AF determinísticos. Ou seja, se o controle estiver num determinado estado, para as combinações de símbolo de entrada e símbolo no topo da pilha, somente um novo estado e uma substituição na pilha devem ser possíveis.

Sejam as seguintes transições num AP:

a) $t(e_1, \varepsilon, F) = (e_2, GF)$
b) $t(e_1, a, F) = (e_2, GF)$
c) $t(e_2, a, F) = (e_3, GF)$
d) $t(e_2, a, F) = (e_3, HF)$
e) $t(e_3, b, G) = (e_4, \varepsilon)$
f) $t(e_3, b, G) = (e_5, \varepsilon)$

Os seguintes não-determinismos estão representados nas transições acima:

1. O AP, no estado e_1, pode chegar a e_2 consumindo ou não o símbolo da entrada a (transições (a) e (b));
2. O AP, no estado e_2, consumindo o símbolo de entrada a, pode chegar a e_3 com símbolos diferentes no topo da pilha (transições (c) e (d));
3. No estado e_3, o AP tem que consumir o mesmo símbolo de entrada, e fazer as mesmas alterações na pilha, porém pode ir para estados diferentes, e_4 ou e_5 (transições (e) e (f)).

Para autômatos finitos, os modelos determinísticos e não determinísticos são equivalentes com respeito às linguagens reconhecidas. Isto não é verdade, porém, para os autômatos de pilha. De fato, $ww^{reversa}$ pode ser reconhecida por um autômato de pilha não-determinístico, mas não por um AP determinístico.

Para ilustrar, seja o AP para reconhecer a linguagem $\{ ww^{reversa} \mid w \in (a + b)^*\}$:

$M = (\{e_1, e_2\}, \{a, b\}, \{R, B, G\}, t, e_1, R, \varnothing)$

$t(e_1, a, R) = \{(e_1, BR)\}$ $\quad$ $t(e_1, a, B) = \{(e_1, BB), (e_2, \varepsilon)\}$

$t(e_1, a, G) = \{(e_1, BG)\}$ $\quad$ $t(e_1, b, R) = \{(e_1, GR)\}$

$t(e_1, b, B) = \{(e_1, GB)\}$ $t(e_1, b, G) = \{(e_1, GG), (e_2, \varepsilon)\}$

$t(e_1, \varepsilon, R) = \{(e_2, \varepsilon)\}$ $t(e_2, a, B) = \{(e_2, \varepsilon)\}$

$t(e_2, b, G) = \{(e_2, \varepsilon)\}$ $t(e_2, \varepsilon, R) = \{(e_2, \varepsilon)\}$

Note que o AP acima é não-determinístico, porque no estado e_1, toda vez que o símbolo de entrada corresponde ao símbolo no topo da pilha, há duas possibilidades: continuar a empilhar ou começar a desempilhar. Também há a possibilidade de desempilhar o símbolo inicial da pilha, sem consumir nenhuma entrada. Isto é necessário para reconhecer a sentença vazia, que faz parte da linguagem.

Isto porque não há um símbolo que marca a metade da cadeia, como na linguagem do primeiro exemplo deste capítulo. Desta forma, não há possibilidade de se construir um AP determinístico para a linguagem aqui proposta.

Exercícios

1) Construa Autômatos de Pilha para reconhecer as seguintes linguagens:

a) $\{a^{i+3}b^{2i+1} \mid i \geq 0\} \cup \{a^{2i+1}b^{3i} \mid i \geq 0\}$

* b) $\{a^ib^jc^jd^je^3 \mid i, j \geq 0\}$

* c) $\{a^ib^j \mid i, j \geq 0$, com $i = 2j$ ou $2i = j\}$

d) $\{a^ib^jc^kd^l \mid i, j, k, l \geq 0, i < l$ e $j \neq k\}$

e) $\{a^ib^jb^ia^j \mid i, j \geq 0\}$

VI - Análise Sintática

1. Introdução

Uma analisador sintático, ou *parser*, é um sistema capaz de construir uma derivação para qualquer sentença em alguma linguagem L(G) baseado em uma gramática G.

Um parser também pode ser visto como um mecanismo para a construção de árvores de derivação.

Seja G = (N, T, P, S) uma GLC com as produções de P numeradas de 1 a p e uma derivação $S \rightarrow^{+} x$. A seqüência formada pelo número das produções utilizadas na derivação $S \rightarrow^{+} x$ constitui o *parse* de x em G.

O *parse ascendente* é constituído pela seqüência invertida dos números das produções utilizadas em $S \rightarrow^{+}_{dir} x$, onde $\rightarrow^{+}_{dir}$ denota a derivação mais a direita.

O *parse descendente* é constituído pela seqüência dos números das produções utilizadas em $S \rightarrow^{+}_{esq} x$, onde $\rightarrow^{+}_{esq}$ denota a derivação mais a esquerda.

Seja a seguinte GLC:

S::= aAB | aB (1, 2) A ::= bc | bAc (3, 4) B ::= eBf | d (5, 6)

onde os números entre parênteses representam os números das produções.

Para a sentença *abbcceedff* o parse ascendente é {3, 4, 6, 5, 5, 1} (que é seqüência de derivações mais a direita {1, 5, 5, 6, 4, 3} invertida). Já o parse descendente é {1, 4, 3, 5, 5, 6} (seqüência de derivações mais a esquerda).

2. Classes de Analisadores Sintáticos

Existem duas classes fundamentais de analisadores sintáticos, definidas em função da estratégia utilizada na análise.

Os analisadores *ascendentes* procuram chegar ao símbolo inicial da gramática a partir da sentença a ser analisada, olhando a sentença, ou parte dela, para decidir qual produção será utilizada na *redução*.

Os analisadores *descendentes* procuram chegar à sentença a partir do símbolo inicial de G, olhando a sentença ou parte dela para decidir que produção deverá ser usada na *derivação*.

3. Analisadores Ascendentes (Família LR)

Sob o ponto de vista lógico, um analisador LR consiste de duas partes: um algoritmo de análise sintática (padrão para todas as técnicas da família) e uma tabela de parsing (específica para cada técnica e para cada gramática).

As principais razões da grande importância desta família de analisadores na teoria de parsing e, conseqüentemente, no desenvolvimento de compiladores, são:

a) analisam praticamente todas as construções sintáticas de linguagens de programação que podem ser representadas por GLC;

b) são mais gerais que os outros analisadores ascendentes e que a maioria dos descendentes e que a maioria dos descendentes sem back-tracking;

c) possuem a propriedade de detecção imediata de erros sintáticos;

d) o tempo de análise é proporcional ao tamanho da sentença a ser analisada.

Em contrapartida, pode-se destacar como ponto negativo dos analisadores LR a complexidade da construção da tabela de parsing, bem como o espaço requerido para seu armazenamento (especialmente as técnicas mais abrangentes).

3.1. Estrutura dos Analisadores LR

Um analisador LR se compõe de:

- um algoritmo de análise sintática;
- uma tabela de análise sintática;
- uma pilha de estados (ou pilha sintática) que conterá um histórico da análise efetuada. Ela é inicializada com o estado inicial da análise sintática;
- uma entrada que conterá a sentença a ser analisada, seguida por uma marca de final de sentença (por exemplo, o símbolo "$").
- uma GLC com as produções numeradas de 1 a p.

3.2. Algoritmo de Análise Sintática LR

O algoritmo de análise sintática LR consiste em comparar o estado do topo da pilha e o próximo símbolo da entrada e, com base nessas informações, consultar a tabela de parsing para decidir a próxima ação a ser efetuada e efetuá-la. Esta ação pode ser:

a) PARAR

b) ERRO

c) TRANSFERIR(S)

d) REDUZIR(R)

Os procedimentos associados às ações têm os seguintes significados:

a) PARAR - Fim de análise. O procedimento consiste em encerrar a análise sintática.

b) ERRO - Indica a ocorrência de um erro sintático. O procedimento é o seguinte: as rotinas de recuperação, se existirem, deverão ser ativadas para que a situação de erro seja contornada e a análise possa continuar; senão a análise deve ser encerrada.

c) TRANSFERIR(S) - Significa o reconhecimento sintático do símbolo da entrada. O procedimento consiste em retirar o símbolo da entrada, e o estado S indicado na tabela de parsing deverá ser empilhado.

d) REDUZIR(R) - Significa que uma redução pela produção n^o R deve ser efetuada. O procedimento consiste em retirar da pilha sintática tantos estados quantos forem os símbolos do lado direito da produção R, e o símbolo do lado esquerdo dessa produção deverá ser colocado na entrada antes da próxima ação do analisador.

Observação: o processo de determinar e efetuar as ações sintáticas deverá ser repetido até que a ação PARAR seja encontrada, ou quando um erro for encontrado (nas implementações sem recuperação de erro).

3.3. A Tabela de Parsing LR

Para implementar uma tabela de parsing pode-se usar uma matriz ou uma lista. No caso de matriz, como normalmente as tabelas de parsing são bastante esparsas, utilizam-se técnicas de compactação de matrizes.

Por exemplo, considere a seguinte gramática LR(0):

0 : E' ::= E$

1, 2 : E ::= E+T | T

3, 4 : T ::= T*F | F

5, 6 : F ::= (E) | id

A tabela SLR(1) correspondente seria:

	id	(	)	+	*	$	E	T	F
0	S_5	S_4					S_1	S_2	S_3
1				S_6		PARE			
2			R_2	R_2	S_7	R_2			
3			R_4	R_4	R_4	R_4			
4	S_5	S_4					S_8	S_2	S_3
5			R_6	R_6	R_6	R_6			
6	S_5	S_4						S_9	S_3
7	S_5	S_4							S_{10}
8			S_{11}	S_6					
9			R_1	R_1	S_7	R_1			
10			R_3	R_3	R_3	R_3			
11			R_5	R_5	R_5	R_5			

3.4. A Configuração de um Analisador LR

A configuração de um analisador LR é um par cujo primeiro elemento é o conteúdo da pilha sintática e o segundo é a entrada a ser analisada. Por exemplo, a configuração:

$(S_0\ S_1\ S_2\ ...\ S_m,\ a_i\ a_{i+1}\ a_{i+2}\ ...\ a_n\ \$)$

tem o seguinte significado: os primeiros i-1 símbolos da entrada já foram analisados e a próxima ação a ser efetuada será determinada pelo estado S_m (topo da pilha) e por a_i (próximo símbolo da entrada).

3.5. Gramáticas LR(0)

Uma das motivações para o estudo de linguagens livres de contexto determinísticas é a sua habilidade em descrever a sintaxe de linguagens de programação. Vários sistemas de construção de compiladores requerem especificação sintática na forma de GLC's restritas que permitem apenas a representação de GLC's determinísticas. Além disso, o parser produzido por tais geradores de compiladores é essencialmente um autômato de pilha determinístico.

Nesta seção será mostrado um tipo restrito de GLC chamado gramática LR(0). Esta classe de gramáticas é a primeira de uma família chamada de *família de gramáticas LR*. O nome LR(0) significa "*left-to-right scan on the input producing a rightmost derivation and using 0 symbols of lookahead on the input*" (busca de símbolos da esquerda para a direita na entrada produzindo uma derivação mais à direita e usando 0 símbolos de contexto de entrada).

As gramáticas LR(0) definem exatamente as linguagens livres de contexto determinísticas que têm a *propriedade do prefixo*. Esta propriedade estabelece que se w está em L, nenhum prefixo próprio de w está em L. Note que a propriedade do prefixo não é muito severa porque a introdução de uma marca de final de sentença ($) converte toda linguagem livre de contexto determinística em uma linguagem livre de contexto determinística com a propriedade do prefixo. Assim, $L\$ = \{w\$ \mid w \in L\}$ é uma linguagem livre de contexto determinística com a propriedade do prefixo sempre que L for uma linguagem livre de contexto determinística.

3.6. Itens LR

Para introduzir as gramáticas LR(0) são necessárias algumas definições preliminares. Em primeiro lugar, um *item*, para uma dada GLC, é uma produção com um ponto em alguma posição do lado direito, incluindo o início e o fim. No caso da produção vazia, B ::= ε, o único item possível é B ::= •.

A seguinte gramática será usada em uma série de exemplos subseqüentes:

S'::= Sc S ::= SA | A A ::= aSb | ab

Esta gramática, cujo símbolo inicial é S', gera as cadeias de parênteses corretamente aninhados, tratando a e b como abre e fecha parênteses, respectivamente, e c como uma marca de final de sentença.

Os itens para a gramática são

S' ::= •Sc	S ::= •SA	A ::= •aSb
S' ::= S•c	S ::= S•A	A ::= a•Sb
S' ::= Sc•	S ::= SA•	A ::= aS•b
	S ::= •A	A ::= aSb•
	S ::= A•	A ::= •ab
		A ::= a•b
		A ::= ab•

A seguir, serão usados os símbolos $\rightarrow^*_D$ e $\rightarrow_D$ para denotar as derivações mais a direita e passos de derivações mais a direita, respectivamente. Uma *forma-sentencial-direita* é uma forma sentencial que pode ser derivada por uma derivação mais a direita. Um *controlador* de uma forma sentencial direita $x\beta y$ para uma GLC G é uma sub-cadeia β, tal que:

$$S \rightarrow^*_D xAy \rightarrow_D x\beta y,$$

com $A \in N$. Em outras palavras, um controlador de $x\beta y$ é uma sub-cadeia que pode ser introduzida no último passo de uma derivação mais a direita de $x\beta y$. Note que neste contexto, a posição de b em $x\beta y$ é importante.

Um *prefixo viável* de uma forma sentencial direita γ é qualquer prefixo de γ que termine antes do (ou no) último símbolo do controlador de γ. Se o controlador ocupa as posições i até j, na cadeia, então os prefixos viáveis são todos os k-prefixos com $k \leq j$.

Por exemplo, na gramática acima há uma derivação mais a direita:

$$S' \rightarrow Sc \rightarrow SAc \rightarrow SaSbc$$

Assim, SaSbc é uma forma sentencial direita, e seu controlador é aSb. Em qualquer gramática não ambígua sem símbolos inúteis, que a derivação mais a direita de uma dada forma sentencial é única, assim, seu controlador é único. Assim, pode-se falar de "o controlador", ao invés de "um controlador". Os prefixos viáveis de SaSbc são {ε, S, Sa, SaS, SaSb} (ou seja, do início da forma sentencial até o último símbolo do controlador).

Dizemos que um item $A ::= \alpha\bullet\beta$ é *válido* para um prefixo viável $x\alpha$ se existe uma derivação mais à direita:

$$S \rightarrow^*_D xAy \rightarrow_D x\alpha\beta y$$

Saber quais itens são válidos para um dado prefixo viável é importante para encontrar uma derivação mais a direita reversa, como segue. Um item é denominado *completo* se o ponto está na posição mais a direita do item. Se $A ::= \alpha\bullet$ é um item completo válido para γ, então pode ser que $A ::= \alpha$ tenha sido usada no último passo e que a forma sentencial direita precedente na derivação de γy era xAy.

Porém isto é apenas uma suspeita, já que $A ::= \alpha\bullet$ pode ser válido para γ devido a uma derivação mais a direita $S \rightarrow^* xAy' \rightarrow \gamma y'$. Certamente pode haver dois ou mais itens válidos para γ, ou pode haver um controlador para γy que inclua símbolos de y. Intuitivamente, uma gramática é definida como LR(0) se em cada situação deste tipo xAy é realmente a forma sentencial direita prévia par γy. Neste caso é possível começar com uma cadeia de terminais w que está em L(G), e assim é uma forma sentencial direita de G, e trabalhar para trás, no sentido das formas sentenciais direitas anteriores, até chegarmos a S. Tem-se assim uma derivação mais à direita de w.

Por exemplo, considerando a gramática anterior e a forma sentencial direita abc. Já que $S \rightarrow^*_D Ac \rightarrow abc$ vemos que A ::= ab• é válido para o prefixo viável ab. Também vemos que A ::= a•b é válido para o prefixo viável a, e A ::= •ab é válido para o prefixo viável ε. Como A ::= ab• é um item completo, podemos ser capazes de deduzir que Ac foi a forma sentencial direita prévia de abc.

3.7. Cálculo dos Conjuntos de Itens Válidos

A definição de gramáticas LR(0) e o método para aceitar L(G) para uma gramática LR(0) por um autômato de pilha determinístico dependem de conhecer o conjunto de itens válidos para cada prefixo viável γ. Segue que para toda GLC G, qualquer que seja, o conjunto de prefixos é um conjunto regular, e este conjunto regular é aceito por um autômato finito não-determinístico cujos estados são os itens de G. Aplicando a determinização a este autômato, obtemos um autômato finito determinístico cujo estado, em resposta ao prefixo viável γ é o conjunto de itens válidos para γ.

O AFND M que reconhece os prefixos viáveis para a GLC G = (N, T, P, S) é definido como segue. Seja A = (E, N∪T, t, e_0, E), onde E é o conjunto de itens para G mais o estado e_0, o qual não é um item. Considerando α, β, γ ∈ (N∪T)* e A, B ∈ N, vamos definir:

1. t(e_0, ε) = {S ::= •α | S ::= α é uma produção}
2. t(A, α•Bβ, ε) = {B ::= •γ | B ::= γ é uma produção}
3. t(A ::= α•Xβ, X) = {A ::= αX•β}

A regra 2 permite a expansão de uma variável B que aparece imediatamente a direita do ponto. A regra 3 permite mover o ponto sobre qualquer símbolo X da gramática, se X é o próximo símbolo da entrada.

O AFND para a gramática do exemplo anterior é mostrado a seguir

Regra 1	t(q_0, ε) = S' ::= •Sc
Regra 2	t(S' ::= •Sc, ε) = S ::= •SA t(S' ::= •Sc, ε) = S ::= •A t(S ::= •SA, ε) = S ::= •SA t(S ::= •SA, ε) = S ::= •A t(S ::= •A, ε) = A ::= •aSb t(S ::= •A, ε) = A ::= •ab t(S ::= S•A, ε) = A ::= •aSb t(S ::= S•A, ε) = A ::= •ab t(S ::= a•Sb, ε) = S ::= •SA t(S ::= a•Sb, ε) = S ::= •A

Regra 3	t(S' ::= •Sc, S) = S' ::= S•c t(S' ::= S•c, c) = S' ::= Sc• t(S ::= •A, A) = S ::= A• t(S ::= •SA, S) = S ::= S•A t(S ::= S•A, A) = S ::= SA• t(A ::= •aSb, a) = A ::= a•Sb t(A ::= a•Sb, S) = A ::= aS•b t(A ::= aS•b, b) = A ::= aSb• t(A ::= •ab, a) = A ::= a•b t(A ::= a•b, b) = A ::= ab•

3.8. Definição de uma Gramática LR(0)

Dizemos que uma gramática G é uma gramática LR(0) se

1) Seu símbolo inicial não aparece no lado direito de nenhuma produção, e
2) Para todo prefixo viável γ de G, sempre que A ::= α• é um item completo válido para γ, então nenhum outro item completo e nenhum item com um terminal no lado direito do ponto é válido para γ.

Não há proibição a que vários itens incompletos sejam válidos para γ.

Para calcular o conjunto de itens válidos para cada prefixo viável, converta o AFND cujos estados são itens em um AFD. No AFD, o caminho do estado inicial rotulado com a seqüência γ leva ao estado que é o conjunto de itens válidos para γ. Assim, deve-se construir o AFD e inspecionar cada estado par ver se uma violação das condições LR(0) acontece.

Por exemplo, o AFD mostrado a seguir foi construído a partir do AFND anterior. Todos os estados, exceto I_0, I_1, I_3 e I_6 consistem de um único item completo. Os estados com mais de um item não têm itens completos, e o símbolo inicial S' não aparece no lado direito de nenhuma produção. Assim, a gramática que viemos usando como exemplo é, de fato, LR(0).

Sejam os estados do AFD:

I_0 = {e_0, S' ::= •Sc, S ::= •SA, S ::= •A, A::= •aSb, A ::= •ab}
I_1 = {S' ::= S•c, S ::= S•A, A ::= •aSb, A ::= •ab}
I_2 = {S ::= A•}
I_3 = {A ::= a•Sb, A ::= a•b, S ::= •SA, S ::= •A, A ::= •aSb, A ::= •ab}
I_4 = {S' ::= Sc•}
I_5 = {S ::= SA•}
I_6 = {A ::= aS•b, S ::= S•A, A::= •aSb, A ::= •ab}
I_7 = {A ::= ab•}
I_8 = {A ::= aSb•}

As transições são:

$t(I_0, S) = I_1$	$t(I_0, A) = I_2$	$t(I_0, a) = I_3$	$t(I_1, c) = I_4$
$t(I_1, a) = I_5$	$t(I_3, a) = I_3$	$t(I_3, S) = I_6$	$t(I_3, A) = I_2$
$t(I_3, b) = I_7$	$t(I_6, a) = I_3$	$t(I_6, b) = I_8$	

3.9. Construção do Conjunto LR

Há um método opcional para calcular o conjunto de itens para uma gramática LR. Este método não envolve determinização de autômatos pois gera o autômato determinizado diretamente a partir da gramática. Porém a aplicação do método é mais complexa.

Serão definidos, inicialmente, dois procedimentos auxiliares. O algoritmo 6.3 define o método propriamente dito.

Algoritmo 6.1: Fechamento de um estado e_i
Entrada: Um estado e_i e uma GLC G=(N, T, P, S)
Saída: O estado e_i fechado

Repita
    Se X ::= α•Yβ pertence ao estado e_i e Y ∈ N então
        Para todas as produções da forma Y ::= γ faça
            Adicione o item Y ::= •γ ao estado e_i
        FimPara
    FimSe
Até não haver mais modificações a fazer em e_i.

Algoritmo 6.2: Cálculo dos estados sucessores de e_i
Entrada: Um estado e_i e uma GLC G=(N, T, P, S)
Saída: Um conjunto de núcleos U e um conjunto de transições R

Faça U = R = {}
Seja Δ o conjunto dos símbolos terminais e não terminais que ocorrem imediatamente à direita do ponto nos itens pertencentes a e_i.
Para todo símbolo δ ∈ Δ faça
    Seja I = {X ::= α•δβ} o conjunto dos itens de e_i onde δ ocorre imediatamente à direita do ponto.
    Coloque em U um estado aberto e_δ com os itens de I no núcleo
    Coloque em R uma transição de e_i rotulada com δ para e_δ.
Fim Para
Retorne U e R.

Algoritmo 6.3: Construção do autômato de itens válidos
Entrada: Uma GLC G=(N, T, P, S)
Saída: Um autômato de Itens LR A = (E, A, t, e_0, F)

Crie um estado inicial e_0 aberto com o núcleo S'::= •S$
Coloque e_0 em E
Coloque N∪T em A
Repita
    Para todo estado aberto ei ∈ E faça:
        Faça fechamento(ei)
        Adicione a E os estados de sucessores(ei)
        Adicione a t as transições de sucessores(ei)
    Fim Para
Até que E não contenha mais estados abertos.

3.10. Construção da Tabela de Parsing SLR(1)

O algoritmo para a construção da tabela de parsing SLR(1), usando o autômato finito determinístico da seção anterior, é o seguinte:

Algoritmo 6.4: Construção da Tabela de Parsing SLR(1)
Entrada: Um AFD com os itens de uma GLC LR(0)
Saída: Uma tabela de parsing para a GLC da entrada.

Crie uma linha para cada estado do AFD e uma coluna para cada símbolo de T∪N∪{$}.

Coloque TRANSFERE(e_i) nos estados e_j que tem transições para estados e_i nas colunas referentes aos símbolos que rotulam a transição.
Para cada estado e_i com itens completos faça
    Para cada item completo faça
        Coloque REDUZIR(N) na interseção da linha correspondente ao estado e_i com as colunas correspondentes a símbolos que pertencem a FOLLOW(A), onde N é o número da produção no item completo em questão e A é o símbolo do lado esquerdo desta produção.
    Fim Para
Fim Para
Coloque PARE na interseção da linha referente ao estado que contenha o item S' ::= S•$, onde S' é o novo símbolo inicial, na coluna do símbolo $.
As posições da tabela que ficarem vazias representarão as situações de erro.

A *condição SLR(1)*, que é a condição para a tabela construída pelo algoritmo acima seja SLR(1), é que não existam estados inadequados (estados com conflitos TRANSFERE-REDUZ e/ou conflitos REDUZ-REDUZ). Isto equivale a dizer que cada posição da tabela deverá conter no máximo uma ação.

Para que um analisador LR aceite qualquer sentença correta e detecte pelo menos um erro em cada sentença incorreta, a tabela de parsing deve possuir as seguintes propriedades:

a) A condição de erro nunca será encontrada para as sentenças sintaticamente corretas.

b) Cada TRANSFERE deverá especificar um único estado, o qual dependerá do símbolo da entrada.

c) Cada REDUZ só é realizado quando os estados do topo da pilha (os símbolos que eles representam) forem exatamente os símbolos do lado direito de alguma produção de G.

d) O PARE só será encontrado quando a análise estiver completa.

4. Analisadores Descendentes

A idéia geral desta classe de analisadores resume-se a uma tentativa de construir a derivação mais à esquerda da sentença de entrada, ou, de maneira equivalente, a uma tentativa de construir a árvore de derivação da sentença a partir do símbolo inicial da gramática em questão. Estes analisadores podem ser implementados com ou sem "back-tracking".

A utilização de back-tracking permite que um conjunto maior de GLCs possa ser analisado. Entretanto, apresenta várias desvantagens, dentre as quais pode-se citar:

a) Maior tempo necessário para a análise;

b) Dificuldade na recuperação de erros;

c) Problemas na análise semântica e geração de código.

Todos estes problemas são decorrentes do não-determinismo.

As implementações sem back-tracking, apesar de limitarem a classe de gramáticas que podem ser analisadas (como será visto adiante), tornam-se mais vantajosas pelo fato de serem técnicas determinísticas e superarem as deficiências práticas das implementações sem back-tracking.

4.1. Analisadores Descendentes sem Back-Tracking

Para uma GLC poder ser analisada por estes analisadores, ela deve satisfazer as seguintes condições:

a) não possuir recursão à esquerda;

b) Estar fatorada, isto é, se A ::= $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ são as produções para o não-terminal A de uma determinada GLC, então FIRST(α_i) $\cap$ FIRST(α_j) = $\varnothing$, $1 \leq i, j \leq n$, $i \neq j$.

c) Para todo A $\in$ N, tal que A $\rightarrow^*$ ε, FIRST(A) $\cap$ FOLLOW(A) = $\varnothing$.

A seguir serão apresentadas duas técnicas de análise sintática descendente sem back-tracking.

4.1.1. Técnica de Implementação Descendente Recursivo

A técnica "descendente recursivo" consiste na construção de um conjunto de procedimentos (normalmente recursivos), um para cada símbolo não terminal da gramática em questão.

A principal desvantagem desta técnica é que ela não é geral, ou seja, os procedimentos são específicos para cada gramática. Além disso, o tempo para análise é maior se comparado com a técnica que será vista a seguir. Também existe a necessidade de uma linguagem que permita recursividade para sua implementação.

Por outro lado, a principal vantagem desta técnica é a simplicidade. Além disso, ela aceita uma classe maior de linguagens pelo fato de que em alguns casos a condição (c) pode ser relaxada.

Exemplo:

O analisador sintático descendente recursivo da gramática:

E ::= TE' E' ::= +TE' | ε T ::= (E) | id

seria composto pelos seguintes procedimentos:

```
(* programa principal *)
início
        analisa(símbolo);
        E(símbolo);
fim

procedimento E(simb);
início
        T(simb);
        Elinha(simb);
fim

procedimento Elinha(simb);
inicio
        se simb = "+" então
                analisa(simb);
                T(simb);
                Elinha(simb);
        fim se;
fim;

procedimento T(simb);
início
        se simb = "(" então
                analisa(simb);
                E(simb);
                analisa(simb);
                se simb = ")" então
                        analisa(simb);
                senão
                        erro;
        senão
                se simb = "id" então
                        analisa(simb);
                senão
                        erro;
        fim se;
fim;
```

4.1.2. Parser Preditivo (LL)

O parser preditivo é uma maneira eficiente de implementar um parser descendente recursivo. O parser preditivo consiste de:

a) Entrada: contendo a sentença (programa) a ser analisada.

b) Pilha: usada para simular a recursividade, ela prevê a parte da entrada que está para ser analisada. Ela é inicializada com $ e o símbolo inicial da gramática em questão.

c) Tabela de parsing: ou tabela de análise sintática. Ela contém as ações a serem efetuadas. É uma matriz M(A, a), onde A $\in$ N e a $\in$ T.

O algoritmo do parser preditivo tem como função determinar, a partir de X (o elemento do topo da pilha) e de "a" (o próximo elemento da entrada), a ação a ser executada, a qual poderá ser:

a) Se X = a = $, o algoritmo anuncia o final da análise.

b) Se X = a ≠ $, o analisador retira S do topo da pilha e a da entrada.

c) Se X ≠ a e X ∈ T, então a situação é de erro.

d) Se X ∈ N, o analisador consulta a tabela de parsing M(X, a), a qual poderá conter o número de uma produção ou um indicativo de erro.

Se M(X, a) contém o número de uma produção, e esta produção é, por exemplo, X ::= uVw, então X, que está no topo da pilha, deve ser substituído por wVu (com u no topo).

Se M(X, a) contém um indicativo de erro, então o analisador deve ativar os procedimentos de recuperação de erros. Em implementações sem recuperação de erros a análise é encerrada.

O termo "preditivo" deve-se ao fato de que a pilha sempre contém a descrição do restante da sentença (se ela estiver correta); isto é, a pilha prevê a parte da sentença que deve estar na entrada para que a sentença esteja correta.

Algoritmo 6.5: Análise Sintática Descendente

```
Repita:
        Faça X ser o topo da pilha.
        Faça "a" ser o próximo símbolo da entrada
        Se X é terminal ou $ então
                Se X = a então
                        retire X do topo da pilha
                        retire a da entrada
                Senão
                        erro
                Fim Se
        Senão (* X é não-terminal *)
                Se M(X, a) = X ::= Y1Y2…Yk então
                        retire X da pilha
                        coloque YkYk-1…Y2Y1 na pilha (* com Y1 no topo *)
                Senão
                        erro
                Fim Se
        Fim Se
Até X = $ (* pilha vazia, análise concluída *)
```

4.1.2.1. Construção da Tabela de Parsing para o Parser Preditivo

A idéia geral é a seguinte: Se A ::= α ∈ P e a ∈ FIRST(α), então se A está no topo da pilha e a é o próximo símbolo da entrada, deve-se expandir (derivar) A, usando a produção A ::= α.

Se α = ε ou $\alpha \rightarrow^{*} \varepsilon$ deve-se observar que ε nunca aparecerá na entrada. Neste caso, se a ∈ FOLLOW(A) deve-se expandir (derivar) A através da produção A ::= α.

Algoritmo 6.6: Construção da Tabela de Parsing

```
Para cada produção A ::= α ∈ P, faça:
    Para todo a ∈ FIRST(α), exceto ε, faça:
        Coloque o número da produção A ::= α em M(A, a);
    Fim Para;
    Se ε ∈ FIRST(α) então
        Coloque o número da produção A ::= α em M(A, b) para todo b ∈ FOLLOW(A);
    Fim Se;
```

Fim Para;
As posições de M que ficarem indefinidas representarão as situações de erro.

A tabela de parsing dos analisadores preditivos deve possuir a propriedade de que, para cada entrada da tabela M exista no máximo uma produção (seu número). Isto viabiliza a análise determinística da sentença de entrada. Para que esta propriedade se verifique, a gramática considerada deverá satisfazer as seguintes condições:

a) Não possuir recursão à esquerda;

b) Estar fatorada;

c) Para todo a $\in$ N tal que A $\rightarrow^{*}$ ε, FIRST(A) $\cap$ FOLLOW(A) = $\varnothing$.

As GLCs que satisfazem estas condições são denominadas GLCs LL(k), isto é, GLCs que podem ser analisadas deterministicamente da esquerda para a direita, e o analisador construirá uma derivação mais a esquerda, sendo necessário a cada passo o conhecimento de k símbolos de "look-ahead" (símbolos da entrada que devem ser vistos para que uma ação seja determinada).

Somente GLCs LL(k) podem ser analisadas pelos analisadores preditivos (as demais causam conflitos na construção da tabela de parsing ou fazem com que o analisador entre em "loop"). Por isso, os analisadores preditivos são também denominados *analisadores LL(k)*. Na prática, usa-se k = 1, obtendo-se desta forma *analisadores LL(1)* para GLC LL(1).

Exercícios

1) Efetue a análise sintática das sentenças seguintes usando a técnica LR(1), com base na tabela do texto:

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

2) Construa a tabela de parsing SLR(1) para as seguintes GLCs:

a) E ::= E+T | T
T ::= T * F | F
F ::= (E) | id

b) C ::= if E then C else C | if E then C | com
E ::= exp

* c) B ::= AB'
B' ::= or AB' | ε
A ::= CA'
A' ::= and CA' | ε
C ::= exp | (B) | not(C)

3) Construa o parse descendente recursivo para as gramáticas do exercício 2.

4) Construa o parse preditivo para as gramáticas do exercício 2.

Respostas aos Exercícios Propostos

Capítulo II

Exercício 2:

Uma sentença possível é: “aababa”, e sua derivação é:

$S \rightarrow aSba \rightarrow aaSbaba \rightarrow aababa$

A linguagem da gramática é $L(G) = \{a^n(ba)^n, n \geq 0\}$

Exercício 3:

a) S ::= aB | a
B ::= aB | bB | cB | a

ou, alternativamente:

S ::= a | aBa
B ::= aB | bB | cB | ε

ou, ainda:

S ::= aB
B ::= ε | bC | cC | aC | a
C ::= bC | cC | aC | a

b) S ::= aA | bA | cA | ε
A ::= aS | bS | cS

d) S ::= 1SBC | 1BC
CB ::= BC
1B ::= 10
0B ::= 00
0C ::= 02
2C ::= 22

f) S ::= aSc | aBc
B ::= bB | ε

Exercício 5

a) S ::= A | B
A ::= aaAb | abbb
B ::= aBbbb | aaaa

b) S ::= aSb | aAb (estas produções garantem que i, k >0, i = k)
A ::= aA | a (garantia de que i > k, já que um a deve ser gerado para parar.)

Exercício 7

a) $((0 + 1)(0 + 1))^*$

b) $(0 + 1)^*101(0 + 1)^*101(0 + 1)^*$

Capítulo III

Exercício 1:

a)

	δ	0	1
→	q_0	q_1	-
	q_1	q_1	q_2
*	q_2	q_1	q_2

b)

	δ	0	1	2
→	q_0	q_0	q_1	q_0
*	q_1	q_1	q_0	q_1

Exercício 2:

a) Gramática regular:

A ::= aA | aB | bA
B ::= aC
C ::= bD | b
D ::= aD | bD | a | b

Determinização:

	δ'	a	b
→	[A]	[A, B]	[A]
	[A, B]	[A, B, C]	[A]
	[A, B, C]	[A, B, C]	[A, D]
*	[A, D]	[A, B, D]	[A, D]
*	[A, B, D]	[A, B, C, D]	[A, D]
*	[A, B, C, D]	[A, B, C, D]	[A, D]

Renomeando os estados:

	δ'	a	b
→	A	B	A
	B	C	A
	C	C	D
*	D	E	D
*	E	F	D
*	F	F	D

Classes de Equivalência:

K-F	**F**
{A, B, C}	{D, E, F}
{A, B} {C}	{D, E, F}
{A} {B} {C}	{D, E, F}
{A} {B} {C}	{D, E, F}

Chamando a CE {D, E, F} de D, temos o seguinte AFD mínimo:

	δ'	a	b
→	A	B	A

	B	C	A
	C	C	D
*	D	D	D

A GR do AFD mínimo é a seguinte:

A ::= aB | bA
B ::= aC | bA
C ::= aC | bD | b
D ::= aD | bD | a | b

Exercício 3:

a) Autômato Finito:

	δ’	0	1
→	S	S, A, C	S, B
	A	A, C, D	-
	B	-	B, D
	C	A, C, D	-
*	D	-	-

Determinização:

	δ’	0	1
→	[S]	[S, A, C]	[S, B]
	[S, A C]	[S, A, C, D]	[S, B]
	[S, B]	[S, A, C]	[S , B, D]
*	[S, A, C, D]	[S, A, C, D]	[S, B]
*	[S, B, D]	[S, A, C]	[S, B, D]

Renomeando:

	δ’	0	1
→	S	E	F
	E	G	F
	F	E	H
*	G	G	F
*	H	E	H

Classes de Equivalência:

K-F	**F**
{S, E, F}	{G, H}
{S} {E} {F}	{G} {H}

Como as CEs contém apenas um estado, não há mais o que fazer, e o AFD já é mínimo.

A GR equivalente é:

S ::= 0E | 1F
E ::= 0G | 1F | 0
F ::= 0E | 1H | 1

G ::= 0G | 1F | 0
H ::= 0E | 1H | 1

Capítulo IV

Exercício 8:

a) Conjunto dos ε-não-terminais: E = {B, D, C, A, S}

Eliminação das ε-produções:
S ::= AB | aS | A | B | a
A ::= bA | BCD | b | CD | BD | BC | C | D | B
B ::= dB | C | d
C ::= cCc | BD | cc | B | D
D ::= CD | d | D | C

Eliminando as produções unitárias:

N_S = {S, A, B, C, D} N_A = {A, C, D, B} N_B = {B, C, D}
N_C = {C, B, D} N_D = {D, C, B}

Gramática sem produções unitárias:
S ::= AB | aS | a | bA | BCD | b | CD | BD | BC | dB | d | cCc | cc
A ::= bA | BCD | b | CD | BD | BC | cCc | cc | d | dB
B ::= dB | d | cCc | BD | cc | CD
C ::= cCc | BD | cc | dB | d | CD
D ::= CD | d | cCc | BD | cc | dB

Exercício 9:

b) S ::= BaS | ε
B ::= AaB'
A ::= aA' | A'
B' ::= aSAab' | bB' | ε
A' ::= aB'aSaA' | ε

Obs: S não gerou S' porque não tinha recursão à esquerda direta. Em B houve a substituição do S mais à esquerda. Em A houve a substituição de S e depois de B.

Exercício 10:

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

Substituindo B nas produções de S, temos:

S ::= bcD | bBcd | bcd
B ::= bB | b
D ::= dD | d

Eliminando o não-determinismo direto, temos:

S ::= bS’
S’ ::= cD | Bcd | cd
B ::= bB’
B’ ::= B | ε
D ::= dD’
D’ ::= D | ε

Substituindo D em S’, temos:

S ::= bS’
S’ ::= cdD’ | Bcd | cd
B ::= bB’
B’ ::= B | ε
D ::= dD’
D’ ::= D | ε

E, por último, tirar o não-determinismo direto:

S ::= bS’
S’ ::= cdS’’ | Bcd
S’’ ::= D’ | ε
B ::= bB’
B’ ::= B | ε
D ::= dD’
D’ ::= D | ε

Exercício 13:

Cálculo do FIRST e FOLLOW do resultado do exercício 9:

S ::= BaS | ε
A ::= aA’ | A’
A’ ::= aB’aSaA’ | ε
B ::= AaB’
B’ ::= aSAab’ | bB’ | ε

N	FIRST(N)	FOLLOW(N)
S	{a, ε}	{$, a}
B	{a, ε}	{a}
A	{a, ε}	{a}
B’	{a, b ε}	{a}
A’	{a, ε}	{a}

FIRST e FOLLOW da GLC:

S ::= ABC | aB | bA | ε
B ::= bA | b | ε
A ::= ab | Cb | cB | ε
C ::= cA | ABc | ε

N	FIRST(N)	FOLLOW(N)
S	{a, b, ε, c}	{$}
A	{a, c, ε, b}	{b, c, a, $}
B	{b, ε}	{c, a, b, $}
C	{c, a, b, ε}	{b, $}

CAPÍTULO V

Exercício 1:

Os AP's aqui apresentados reconhecem as sentenças por pilha vazia:

b) Alfabeto da pilha: A, B, C.

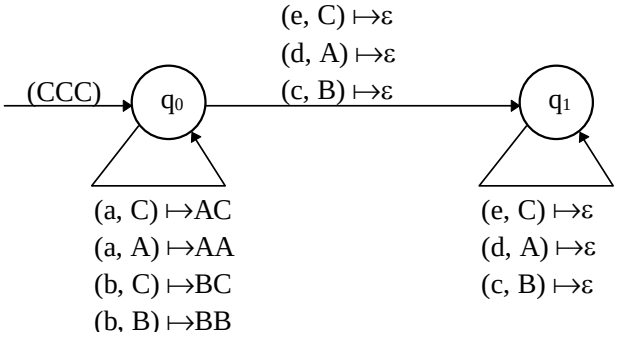
A = Empilhado quando reconhece a, para controlar o número de d's

B = Empilhado quando reconhece b, para controlar o número de c's

C = Controla o número de e's

Estado inicial da pilha: CCC (três C's empilhados).

Forma gráfica:



c) Alfabeto da pilha: A, C, D.

A = Empilhado quando reconhece a, para controlar o número de b's.

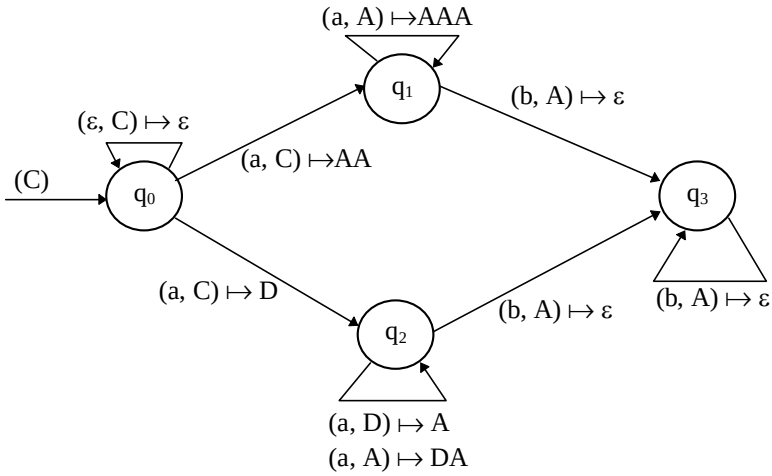
C = Utilizado para reconhecer a sentença vazia.

D = usado para o caso de 2i = j, em que um A deve ser empilhado para cada 2 a's reconhecido. Para o primeiro empilha-se um D, e para o segundo o D é trocado por A.

Estado inicial da pilha: C.

O AP é não-determinístico, já que não se sabe em qual dos dois casos a sentença irá se encaixar (i = 2j ou 2i = j).

Forma gráfica:



Capítulo VI

Exercício 2:

c) Primeiramente, introduz-se a propriedade do prefixo na gramática, e numera-se as produções:

0 S ::= B$
1 B ::= AB'
2, 3 B' ::= or AB' | ε
4 A ::= CA'
5, 6 A' ::= and CA' | ε
7, 8, 9 C ::= exp | (B) | not C

O algoritmo utilizado é o fechamento de estados, que já cria o AF determinístico. Assim, e_0 é iniciado com (o núcleo de cada estado é sublinhado):

e_0 = {S ::= •B$}

e o fechamento de e_0 resulta em:

e_0 = { S ::= •B$ B ::= •AB' A ::= •CA' C ::= •exp C::= •(B) C ::= •not C }

As transições de e_0 são:

$t(e_0, B) = e_1$ $t(e_0, A) = e_2$ $t(e_0, C) = e_3$ $t(e_0, exp) = e_5$

$t(e_0, () = e_4$ $t(e_0, not) = e_6$

Os núcleos dos novos estados são:

e_1 = {S ::= B•$}
e_2 = {B ::= A•B'}
e_3 = {A ::= C•A'}
e_4 = {C ::= (•B)}
e_5 = {C ::= exp •}
e_6 = {C ::= not • C}

Os estados do autômato são:

e_0 = { S ::= •B$ B ::= •AB' A ::= •CA' C ::= •exp C::= •(B) C ::= •not C }
e_1 = {S ::= B•$ }
e_2 = {B ::= A•B' B' ::= •or AB' B' ::= •}
e_3 = {A ::= C•A' A' ::= •and CA' A' ::= •}
e_4 = {C ::= (•B) B ::= •AB' A ::= •CA' C ::= •exp C::= •(B) C ::= •not C }
e_5 = {C ::= exp • }
e_6 = {C ::= not •C C ::= •exp C::= •(B) C ::= •not C }
e_7 = {B ::= AB'• }
e_8 = {B' ::= or •AB' A ::= •CA' C ::= •exp C::= •(B) C ::= •not C }
e_9 = {A ::= CA'•}
e_{10} = {A' ::= and •CA' C ::= •exp C::= •(B) C ::= •not C }
e_{11} = {C ::= (B•)}
e_{12} = {C ::= not C•}
e_{13} = {B' ::= orA•B' B' ::= •or AB' B' ::= •}
e_{14} = {A' ::= and C•A' A' ::= •and CA' A' ::= •}
e_{15} = {C ::= (B)•}
e_{16} = {B' ::= or AB'•}
e_{17} = {A' ::= and CA'•}

As transições do autômato são:

$t(e_0, b) = e_1$ $t(e_0, A) = e_2$ $t(e_0, C) = e_3$ $t(e_0, () = e_4$
$t(e_0, exp) = e_5$ $t(e_0, not) = e_6$ $t(e_2, B') = e_7$ $t(e_2, or) = e_8$

$t(e_3, A') = e_9$	$t(e_3, and) = e_{10}$	$t(e_4, B) = e_{11}$	$t(e_4, A) = e_2$
$t(e_4, C) = e_3$	$t(e_4, exp) = e_5$	$t(e_4, () = e_4$	$t(e_4, not) = e_6$
$t(e_6, C) = e_{12}$	$t(e_6, exp) = e_5$	$t(e_6, () = e_4$	$t(e_6, not) = e_6$
$t(e_8, A) = e_{13}$	$t(e_8, C) = e_3$	$t(e_8, exp) = e_5$	$t(e_8, () = e_4$
$t(e_8, not) = e_6$	$t(e_{10}, C) = e_{14}$	$t(e_{10}, exp) = e_5$	$t(e_{10}, () = e_4$
$t(e_{10}, not) = e_6$	$t(e_{11},)) = e_{15}$	$t(e_{13}, B') = e_{16}$	$t(e_{13}, or) = e_8$
$t(e_{14}, A') = e_{17}$	$t(e_{14}, and) = e_{10}$		

Para a montagem da tabela SLR(1), é necessário o FOLLOW de cada não-terminal:

N	FIRST(N)	FOLLOW(N)
B	{exp, (, not }	{ $,) }
B'	{ or, ε }	{ $,) }
A	{ exp, (, not }	{ or , $,) }
A'	{and, ε }	{ or, $,) }
C	{ exp, (, not }	{ and, or, $ }

A tabela SLR(1) é a seguinte:

	or	**and**	**exp**	**(**	**)**	**not**	**$**	**B**	**B'**	**A**	**A'**	**C**
0			S5	S4		S6		S1		S2		S3
1							PARE					
2	S8				R3		R3		S7			
3	R6	S10			R6		R6				S9	
4			S5	S4		S6		S11		S2		S3
5	R7	R7			R7		R7					
6			S5	S4		S6						S12
7					R1		R1					
8			S5	S4		S6				S13		S3
9	R4				R4		R4					
10			S5	S4		S6						S14
11					S15							
12	R9	R9			R9		R9					
13	S8				R3		R3		S16			
14	R6	S10			R6		R6				S17	
15	R8	R8			R8		R8					
16					R2		R2					
17	R5				R5		R5					

Para exemplificar, analise a sentença
not(exp) and (exp or exp)

Obs: quando a tabela SLR(1) mandar reduzir por uma produção do tipo X ::= ε, nenhum símbolo é retirado da pilha, somente X é escrito na entrada. Por exemplo, a configuração $\{e_0\ e_6\ e_4\ e_3,\ )\ and\ (exp\ or\ exp)\$\}$, consultada na tabela, leva a R6, ou reduzir pela produção 6, que é a produção A' ::= ε. A nova configuração será $\{e_0\ e_6\ e_4\ e_3,\ A')\ and\ (exp\ or\ exp)\$\}$

Exercício 4:

A mesma gramática é considerada, com suas produções numeradas:

1 B ::= AB'
2, 3 B' ::= or AB' | ε
4 A ::= CA'

5, 6 A' ::= and CA' | ε
7, 8, 9 C ::= exp | (B) | not C

Esta GLC é LL(1), pois

a. não possui recursão à esquerda;

b. está fatorada;

c. para todo $X \in N$, se $X \rightarrow^* \varepsilon$, então $FIRST(X) \cap FOLLOW(X) = \varnothing$.

Os conjuntos FIRST e FOLLOW são:

N	FIRST(N)	FOLLOW(N)
B	{exp, (, not }	{ $,) }
B'	{ or, ε }	{ $,) }
A	{ exp, (, not }	{ or , $,) }
A'	{and, ε }	{ or, $,) }
C	{ exp, (, not }	{ and, or, $ }

A tabela de parsing é:

	or	and	exp	(	)	not	$
B			1	1		1	
B'	2				3		3
A			4	4		4	
A'	6	5			6		6
C			7	8		9	

Novamente, para exemplificar, analise a sentença:

not (exp and exp) and (exp or exp).

Referências Bibliográficas

HOPCROFT, J. E. and ULLMAN, J. D.: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.