Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e de Estatística

LINGUAGENS FORMAIS E COMPILADORES

(Prof. Olinto José Varela Furtado)

Capítulo I – Introdução

I.1 – Introdução a Compiladores

I.2 - Introdução a Teoria da Computação

O Que é **Teoria da Computação**?

A **Teoria da Computação** pode ser vista como um guia (um roteiro) que nos orienta no sentido de informar o que pode e o que não pode ser efetivamente computável, explicando porque, de que forma e com que complexidade. Neste sentido, a **Teoria da Computação** classifica os problemas computacionais em três classes:

a) **Problemas Indecidíveis** (ou impossíveis de serem solucionados);
b) **Problemas Intratáveis** (possíveis com recursos ilimitados, porém impossíveis com recursos limitados);
c) **Problemas Tratáveis** (possíveis de serem solucionadas com recursos limitados).

Esta classificação engloba problemas de toda a natureza, envolvendo desde problemas clássicos que fundamentam a teoria da computação até problemas (ou instâncias de problemas) práticos da ciência da computação, tais como:

1 – Existe programa para solucionar um determinado problema?
2 – Qual o poder de expressão de um determinado modelo de especificação?
3 – Dado um programa qualquer, ele sempre tem parada garantida?
4 – Dois programas P1 e P2 são equivalentes entre si?
5 – Uma determinada solução é a melhor solução para um dado problema?
6 – Qual o significado de um determinado programa?
7 – Dado um programa qualquer, este programa está correto?

Esta lista poderia ser expandida e detalhada, contudo, seu objetivo é enfatizar a abrangência da teoria da computação. Dos tópicos aqui listados, complexidade (5), semântica (6) e correção/construção (7), costumam ser tratados como disciplinas específicas e independentes, enquanto que os demais se classificam como problemas básicos da teoria da computação. Todos os problemas computacionais podem ser tratados (estudados) sob a ótica da Teoria das Linguagens Formais e Autômatos.

Segundo esta ótica, a teoria da computação pode ser vista como um conjunto de modelos formais (juntamente com suas propriedades) que fundamentam a ciência da computação. Tais modelos incluem Autômatos (Finitos, de Pilha e Máquinas de Turing) e Gramáticas, enquanto que as propriedades de interesse envolvem questões de decidibilidade, Inter-relacionamento entre modelos (abrangência, equivalência, etc...) e complexidade computacional.

Nesta apostila, abordaremos a **Teoria da Computação** segundo a ótica da **Teoria das Linguagens Formais e Autômatos.**

I.2.1 - Conceitos e Propósitos Fundamentais da Teoria da Computação

Com o objetivo de melhor fundamentar as questões cobertas pela teoria da computação e de identificar claramente a possibilidade de reduzir tais questões a problemas pertinentes a Teoria das Linguagens Formais, apresentaremos nesta seção alguns conceitos fundamentais e comentamos alguns dos principais propósitos que sustentam a teoria da computação.

Procedures e Algorítmos

O conceito de algoritmo é fundamental dentro da ciência da computação e pode ser definido formalmente segundo vários propósitos da teoria da computação (como será visto no final desta seção) ou informalmente em função da definição de procedure (como veremos a seguir).

Procedure: É m conjunto finito de passos (instruções), os quais podem ser executados mecanicamente em uma quantidade fixa de tempo e com uma quantidade fixa de esforço. Um bom exemplo de uma procedure é um programa de computador escrito em linguagem de máquina, pois tal programa possui um número finito de passos, todos executáveis mecanicamente com uma quantidade fixa de recursos.

Algoritmo: É uma procedure que sempre pára; ou seja, uma procedure cuja execução chegará ao final, independentemente de quais sejam suas entradas. Adicionalmente, dizemos que uma procedure termina para uma determinada entrada, se existe um número finito t, tal que após a execução de t instruções (não necessariamente distintas), ou não existem mais instruções a serem executadas, ou a última instrução executada foi uma instrução “halt”.

Exemplos:
1 – Dado um número inteiro positivo I, determinar se I é ou não um número primo.
2 – Dado um inteiro I, determinar se existe um número perfeito maior que I (obs: um número é perfeito se a soma de seus divisores (exceto ele mesmo) é igual ao próprio número).
3 – Dado um programa escrito em uma determinada linguagem de programação, determinar se esse programa está sintaticamente correto. Este problema é uma instância do seguinte problema genérico: dada uma cadeia de caracteres x determinar se essa cadeia é gerada por uma Gramática Livre de Contexto (ou reconhecida por um Autômato de Pilha).
4 – Dado um programa qualquer, determinar se existe alguma entrada para a qual o programa entrará em loop.

Os problemas enunciados nos exemplos 1 e 3 possuem representação algorítmica enquanto que os problemas dos exemplos 2 e 4 só são representáveis via procedures.

Conjuntos Recursivos e Conjuntos Recursivamente Enumeráveis

Um conjunto é dito Recursivamente Enumerável se ele pode ser representado (solucionado) por uma procedure, e Recursivo se ele pode ser representado (solucionado) por um algoritmo. Como procedures e algoritmos podem ser definidos formalmente através de vários modelos (gramáticas e autômatos, por exemplo), podemos também definir conjuntos recursivos e recursivamente enumeráveis em função de tais modelos.

Problemas Decidíveis e Indecidíveis X Algoritmos e Procedures

Um problema é decidível (tratável ou não) se e somente se ele é resolvível por um algoritmo, para qualquer entrada pertencente ao seu domínio; caso contrário ele é um problema indecidível.

A partir das definições acima, podemos notar claramente a relação entre problemas decidíveis e indecidíveis com conjuntos recursivos e recursivamente enumeráveis; ou seja, um problema é decidível se o conjunto de soluções das diversas instâncias deste problema é um conjunto recursivo, e indecidível caso tal conjunto seja recursivamente enumerável. Assim sendo, torna-se evidente que a questão da decidibilidade pode ser tratada formalmente através dos modelos que compõem a Teoria das Linguagens e Autômatos.

A classe dos problemas indecidíveis é significativamente representada pelo "HALTING PROBLEM" (problema da parada) que consiste em: "Dado uma procedure Z e uma entrada X, decidir (determinar) se Z termina quando aplicado a X. A indecidibilidade deste problema é extremamente útil para demonstrar a indecidibilidade de outros problemas através da redução destes para o "halting problem "".

Propósitos da Teoria da Computação

Até aqui definimos procedures e algoritmos de maneira intuitiva e informal. Contudo eles podem ser definidos rigorosamente (precisamente) através de vários formalismos conhecidos como propósitos (ou princípios) da Teoria da Computação. Tais formalismos tem sido explorados largamente na Ciência da Computação, onde servem como modelos na solução de diversos problemas práticos. Dentre os formalismos mais importantes, podemos citar:

a) Máquinas de Turing (Turing, 1936);
b) Gramáticas (Chomsky, 1959);
c) Algoritmos de Markov (Markov, 1951);
d) Lambda Calculus (Church, 1941);
e) Sistemas Post e Sistemas de Produção (Emil Post, 1936);
f) Funções Recursivas (Kleene, 1936).

Um ponto importante a ressaltar aqui, é que toda procedure (ou algoritmo) descrita por algum destes formalismos, pode também ser descrita através de qualquer um dos demais; fato este que sugere a equivalência entre os formalismos.

A aceitação destes formalismos dentro da teoria da computação é, em grande parte, decorrente da hipótese (conhecida como Tese de Church) de que todo processo computável – passível de ser descrito por uma procedure – pode ser realizado por uma Máquina de Turing. Esta tese, apesar de não ter sido provada formalmente, também não foi contradita e continua sendo universalmente aceita. Conseqüentemente podemos afirmar que Máquinas de Turing constituem o formalismo mais genérico para a representação de procedure e que qualquer outro formalismo será significativo se for considerado equivalente às máquinas de Turing. A demonstração formal da equivalência entre os diversos formalismos citados e máquinas de Turing, reforça a tese de Church.

I.3 – Introdução a Teoria das Linguagens Formais

O que é a **Teoria das Linguagens Formais**?

Para respondermos esta questão precisamos primeiro responder o que é Linguagem Formal, e para isto precisamos antes responder o que é **Linguagem**.

Inicialmente, de maneira bastante informal, podemos definir uma linguagem como sendo uma forma de comunicação. Elaborando um pouco mais esta definição, podemos definir uma linguagem como sendo "um conjunto de elementos (símbolos) e um conjunto de métodos (regras) para combinar estes elementos, usado e entendido por uma determinada comunidade".

Exemplos: 1 - Linguagens Naturais (ou idiomáticas)
2 - Linguagens de Programação, de Controle, de Consulta
3 – Protocolos de Comunicação

Contudo, apesar de intuitiva, esta definição não nos permite responder satisfatoriamente as duas primeiras questões; precisamos antes dar um sentido formal para a definição de linguagem. Faremos isto nas duas próximas seções.

I.3.1 – Conceitos Básicos

Alfabeto (ou vocabulário): É um conjunto finito, não vazio, de símbolos (elementos). Representaremos um alfabeto por.

Exemplos: V = {a, b, c,...,z }
V = {0, 1}
V = {a, e, i, o ,u}

Sentenças: Uma sentença sobre um alfabeto V, é uma seqüência (ou cadeia) finita de símbolos do alfabeto.

Exemplo de sentenças sobre V = { a , b }: a, b, aa, ab, bb, aaa, aab, aba, baa, ...

Tamanho de uma sentença: Seja w uma sentença $\forall$.

O tamanho da sentença w, denotado por |w|, é definido pelo número de símbolos (elementos do alfabeto) que compõem w.

Exemplos: Seja V = { a , b , c }
se x = aba, então |x| = 3
se x = c, então |x| = 1

Sentença vazia: É uma sentença constituída de nenhum símbolo; isto é, uma sentença de tamanho 0 (zero).

Observações: - Representaremos a sentença vazia por ε (épsolon).
- Por definição, |ε| = 0

Potência de uma sentença: Seja w uma sentença $\forall$. A n-ésima potência de w, representada por w^n, significa w repetido n vezes.

Exemplos: se x = ab, então x^3 = ababab
Para $\forall$ x, $x^0 = \varepsilon$

Fechamento de um Alfabeto

Seja V um alfabeto ∀.

- O fechamento reflexivo (ou simplesmente fechamento) de V, representado por V^*, é dado pelo conjunto de todas as possíveis seqüências que podem ser formadas a partir de V, inclusive a sentença vazia.
- O fechamento transitivo (ou fechamento positivo) de V, representado por V^+, é dado por $V^* - \{ \varepsilon \}$.

  Exemplos: Seja V = { 0, 1 }, temos que:

  $V^* = \{\varepsilon, 0, 1, 00, 01, 11, 000,...\}$

  $V^+ = \{0, 1, 00 ,01, 11, 000,...\}$

I.3.2 – Linguagens e suas Representações

Linguagem: Uma linguagem L sobre um alfabeto V, é um subconjunto de V^*; isto é,

$$\mathbf{L \subseteq V^*}$$

Representações de Linguagens: O estudo de linguagens está intimamente relacionado ao estudo das formas de representação dessas linguagens. O problema de representação de uma linguagem, por sua vez, está relacionado com o fato dela ser finita ou infinita:

- **Linguagem Finita**: É uma Linguagem que pode ser representada por enumeração.

Exemplo: A linguagem definida como sendo o conjunto dos inteiros positivos pares maiores que 0 e menores que 20, pode ser representado por: L = {2, 4, 6, 8, 10, 12, 14, 16, 18}.

- **Linguagem Infinita**: Neste caso, na impossibilidade de usarmos enumeração, precisamos encontrar uma representação finita para estas linguagens.

Exemplo: A linguagem definida como sendo o conjunto dos inteiros pares poderia ser representada por V ={2, 4, 6, 8, 10,...} que, que apesar de intuitiva, não é finita e nem precisa.

As representações finitas de linguagens classificam-se em Reconhecedores e Sistemas Geradores:

Reconhecedores – São dispositivos formais que nos permitem verificar se uma determinada sentença pertence ou não a uma determinada linguagem (é uma representação das sentenças de uma linguagem sob o ponto de vista do reconhecimento de tais sentenças). Esses dispositivos denominam-se *autômatos*; autômatos finitos, autômatos de pilha e máquinas de turing, por exemplo, podem ser destacados como importantes classes de autômatos.

Sistemas Geradores – São dispositivos formais dotados de mecanismos que permitem a geração sistemática das sentenças de uma linguagem (representação sob o ponto de vista da geração das sentenças de uma linguagem). Os principais sistemas geradores disponíveis são as *gramáticas*, dentre as quais, por exemplo, podemos destacar as gramáticas de CHOMSKY.

Observações: Todo *reconhecedor* e todo *sistema gerador* pode ser representado por *algorítmos* e/ou *procedures.*

Linguagens Formais: São linguagens que podem ser representadas de maneira finita e precisa através de sistemas com sustentação matemática (dispositivos formais ou modelos matemáticos).

Linguagem Recursiva: Uma linguagem é recursiva se existe um algoritmo capaz de reconhecer ou gerar as sentenças que compõem essa linguagem.

Linguagem Recursivamente Enumerável: É toda a linguagem cujas sentenças podem ser reconhecidas ou geradas por procedures.

Teoria das Linguagens Formais e dos Autômatos

Entende-se por Teoria das Linguagens Formais e dos Autômatos o estudo de modelos matemáticos que possibilitam a especificação e o reconhecimento de linguagens (no sentido amplo da palavra), suas classificações, estruturas, propriedades, características e inter-relacionamentos.

A importância desta Teoria na Ciência da Computação é dupla: Ela tanto apóia outros aspectos teóricos da Ciência da Computação (decidibilidade, computabilidade, complexidade computacional, por exemplo), como fundamenta diversas aplicações computacionais tais como processamento de linguagens, reconhecimento de padrões, modelagem de sistemas.

CAPÍTULO II – GRAMÁTICAS

II.1 – Motivação

Sabemos que uma linguagem é qualquer conjunto ou subconjunto de sentenças sobre um alfabeto, ou seja: dado um alfabeto V, uma linguagem L sobre esse alfabeto, é um subconjunto de V^* $\therefore L \subseteq V^*$.

Assim sendo, devemos nos preocupar em definir que subconjunto é esse.

A finalidade de uma gramática é definir o subconjunto de V^* que forma (define) uma determinada linguagem. Uma gramática define uma estrutura sobre um alfabeto de forma a permitir que apenas determinadas combinações sejam válidas, isto é, sejam consideradas sentenças (definindo assim a linguagem que ela representa).

O que é Gramática?

Uma gramática, de maneira informal, pode ser definida como sendo:

a) Um sistema gerador de linguagens;
b) Um sistema de reescrita;
c) Uma maneira finita de descrever (representar) uma linguagem;
d) Um dispositivo formal usado para especificar de maneira finita e precisa uma linguagem potencialmente infinita.

Exemplo intuitivo de uma Gramática:
(um subconjunto da gramática da língua portuguesa)

<sentença> :: = <sujeito> <predicado>
<sujeito> :: = <substantivo>
| <artigo> <substantivo>
| <artigo> <adjetivo> <substantivo>
<predicado> :: = <verbo> <objeto>
<substantivo> :: = joão | Maria | cachorro | livro | pão
<artigo> :: = o | a
<adjetivo> :: = pequeno | bom | bela
<verbo> :: = morde | le | olha
<objeto> :: = <substantivo>
| <artigo> <substantivo>
| <artigo> <adjetivo> <substantivo>

Notação utilizada:

< ..... > : categoria sintática ou gramatical;
::= : definido por
| : ou (alternativa)
α ::= β : regra de sintaxe (ou regra gramatical ou regra de produção)

II.2 – Definição Formal de Gramática

Formalmente definimos uma gramática G como sendo um quádrupla (sistema formal constituído de quatro elementos) **G = (Vn, Vt, P, S)** onde:

Vn – É um conjunto finito de símbolos denominados **não-terminais**. Estes símbolos também são denominados meta variáveis, ou seja, são os símbolos utilizados na descrição da linguagem.

Vt – É um conjunto finito de símbolos denominados **terminais**. São os símbolos da linguagem propriamente ditos, ou seja os símbolos que podem ser usados na formação das sentenças da linguagem.

Convenções: $\mathbf{Vn} \cap \mathbf{Vt} = \varphi$ e $\mathbf{Vn} \cup \mathbf{Vt} = \mathbf{V}$

P – É um conjunto finito de pares (α , β) denominado **produções** (ou regras gramaticais ou regras de sintaxe). Uma produção é representada por α **::=** β, onde $\alpha \in V^*VnV^* \wedge \beta \in V^*$, e significa que α **é definido por** β, ou ainda que α produz β ou equivalentemente que β é produzido (gerado) a partir de α.

S – É o **símbolo inicial** da gramática; deve pertencer a Vn. O símbolo inicial de uma gramática é o não-terminal a partir do qual as sentenças de uma linguagem serão geradas.

Exemplo: Formalizando o subconjunto da gramática da língua portuguesa, apresentado na seção anterior, teríamos:

Gportugues = (Vn, Vt, P, S), onde:

Vn = { <sentença> , <sujeito> , <predicado> , <substantivo> ,
<artigo> , <adjetivo> , <predicado> , <verbo> , <objeto> }

Vt = { joão, maria , cachorro, livro , pão, o , a , pequeno, bom , bela , morde , le , olha }

P = é o conjunto das regras gramaticais apresentado

S = <sentença>

Fazendo uma analogia entre o exemplo intuitivo e a noção formal de gramática, constatamos que:

Vn – são as categorias sintáticas ou gramaticais;

Vt – são as palavras utilizadas como símbolos da linguagem;

P – são as regras sintáticas (ou gramaticais);

S - é a categoria gramatical que sintetiza o que será produzido (gerado) pela gramática.

Notação a ser utilizada neste curso:

::= - $\rightarrow$

$\mathbf{V_N}$ – Letras de "A" a "T" e palavras escritas com letras maiúsculas

Vt – Letras de "a" a "t", palavras escritas com letras minúsculas, dígitos e caracteres especiais

Vt* - u, v, x, y, w, z

$\mathbf{\{V_N \cup Vt\}}$ – U, V, X, Y, W, Z

$\mathbf{\{V_N \cup Vt\}^*}$ - α, β, γ, δ, ... ,ω (exceto ε)

II.3 – Derivação e Redução

São operações de substituição que formalizam a utilização de gramáticas, sendo que:

Derivação: É a operação que consiste em substituir em string (ou parte dele) por outro, de acordo com as produções das gramáticas em questão, no sentido símbolo inicial $\rightarrow$ sentença;

Redução: É a operação que consiste na substituição de um string (ou parte dele) por outro, de acordo com as produções da gramática, no sentido sentença $\rightarrow$ símbolo inicial.

Observação: derivação é a operação adequada para geração de sentenças; enquanto que a operação de redução é adequada ao reconhecimento de sentenças.

Noção Formal de Derivação e Redução

Seja G = (Vn, Vt, P, S) uma gramática $\forall$.

Seja $\delta\alpha\gamma \in (Vn \cup Vt)^*$.

Derivação / redução em um passo (ou direta): dizemos que $\delta\alpha\gamma$ deriva em um passo (ou deriva diretamente) $\delta\beta\gamma$, se e somente se $\alpha \rightarrow \beta \in \mathbf{P}$; indicamos por $\delta\alpha\gamma \Rightarrow \delta\beta\gamma$.

Neste caso, dizemos ainda que $\delta\beta\gamma$ reduz-se a $\delta\alpha\gamma$ em um passo (ou diretamente); denotamos por $\delta\alpha\gamma \Leftarrow \delta\beta\gamma$.

Derivação / redução em zero ou mais passos: Por extensão, dizemos que α deriva em zero ou mais passos (ou simplesmente deriva) β, se existirem seqüências $\alpha 1, \alpha 2, ..., \alpha n$ tais que: $\alpha \Rightarrow \alpha 1 \Rightarrow \alpha 2 \Rightarrow ... \Rightarrow \alpha n \Rightarrow \beta$; esta forma de derivação é denotada por $\alpha \overset{*}{\Rightarrow} \beta$.

Analogamente β reduz-se a α em zero ou mais passos (ou simplesmente reduz-se); indicamos por $\alpha \overset{*}{\Leftarrow} \beta$.

Derivação / redução em um ou mais passos: Por outro lado, se tivermos certeza de que pelo menos um passo foi necessário para chegar em β a partir de α (ou vice-versa), então teremos uma nova forma de derivação (redução) denominada derivação (redução) em um ou mais passos; indicaremos por: $\alpha \overset{+}{\Rightarrow} \beta$ **(ou por** $\alpha \overset{+}{\Leftarrow} \beta$**).**

Observação: - Se $\alpha \overset{*}{\Rightarrow} \beta$ em 0 (zero) passos, então $\alpha = \beta$.

- Quando várias gramáticas estiverem sendo usadas simultaneamente, devemos usar o nome da gramática sob a seta de derivação (redução) para não causar dúvidas.

II.4 – Sentença, Forma Sentencial e Linguagem

Sentença – É uma seqüência só de terminais produzida (gerada) a partir do símbolo inicial de uma gramática; isto é, se $\mathbf{G = (Vn, Vt, P, S) \wedge S \overset{+}{\Rightarrow} x}$, então **x** é uma sentença pertencente à linguagem representada por G.

Forma Sentencial – É uma seqüência qualquer (composta por terminais e não-terminais) produzida (gerada) a partir do símbolo inicial de uma gramática; isto é,
se $\mathbf{G = (Vn, Vt, P, S) \wedge S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \ldots \Rightarrow \gamma \Rightarrow \ldots}$ então $\alpha, \beta, \ldots, \gamma, \ldots$são formas sentenciais de **G**.

Linguagem – Formalmente definimos a linguagem gerada por **G = (Vn, Vt, P, S),** denotada por **L(G),** como sendo: $\mathbf{L(G) = \{x \mid x \in Vt^* \wedge S \overset{+}{\Rightarrow} x\}}$; ou seja, uma linguagem é definida pelo conjunto de sentenças que podem ser derivadas a partir do símbolo inicial da gramática que a representa.

Gramáticas Equivalentes – Duas gramáticas **G1** e **G2** são equivalentes entre si, se e somente se **L(G1) = L(G2).**
Formalmente: $\mathbf{G1 \equiv G2 \Leftrightarrow L(G1) = L(G2).}$

Exemplos:

II.5 – Tipos de Gramáticas

(Classificação ou hierarquia de CHOMSKY)

Gramática Tipo 0:
(ou gramática sem restrições)

G = (Vn, Vt, P, S), onde:
$P = \{\alpha \rightarrow \beta \mid \alpha \in V^*VnV^* \wedge \beta \in V^*\}$

Gramática Tipo 1:
(ou Gramática Sensível ao Contexto – G.S.C.)

G = (Vn, Vt, P, S), onde:
$P = \{\alpha \rightarrow \beta \mid |\alpha| \leq |\beta|, \alpha \in V^*VnV^* \wedge \beta \in V^+\}$

Gramática Tipo 2:
(ou Gramática Livre de Contexto – G.L.C.)

G = (Vn, Vt, P, S), onde:
$P = \{A \rightarrow \beta \mid A \in Vn \wedge \beta \in V^+$

Gramática Tipo 3:
(ou Gramática Regular – G.R.)

G = (Vn, Vt, P, S), onde:
$P = \{A \rightarrow a\,X \mid A \in Vn, a \in Vt \wedge X \in \{Vn \cup \{\varepsilon\}\}\}$

Observação: As linguagens representadas (geradas) por G.S.C., G.L.C. e G.R. são denominadas, respectivamente, **Linguagens Sensíveis ao Contexto (L.S.C.) Linguagens Livres de Contexto (L.L.C.) e Linguagens Regulares (L.R.).**

II.6 –Sentença Vazia

Introdução: A motivação para o estudo de gramáticas foi à necessidade de se encontrar representações finitas para as linguagens.

Logicamente, se uma linguagem **L** possui uma descrição finita, então **L1 = L ∪ {ε}**, também deverá possuir uma descrição finita.

Pela definição dada, as G.S.C., G.L.C. e G.R. não aceitam produções da forma $S \rightarrow \varepsilon$; logo, segundo essa definição, ε (a sentença vazia) não pode pertencer as L.S.C., L.L.C. ou L.R.. Entretanto, a razão destas definições não é a sentença vazia em si, mas sim o fato de que a produção que possibilita a derivação da sentença vazia, pode inviabilizar a prova formal da existência de algoritmos associados a estas gramáticas.

Redefinição de G.S.C. G.L.C. e G.R.

Em função do exposto acima, vamos redefinir G.S.C., G.L.C. e G.R., permitindo que a produção $S \rightarrow \varepsilon \in P$, se e somente se:

1 – S for o símbolo inicial da gramática;
2 – S não aparecer no lado direito de nenhuma produção da gramática em questão.

Observação: Note que segundo esta redefinição, a produção $S \rightarrow \varepsilon$ só poderá ser usada uma vez: exatamente na derivação de ε (a sentença vazia).

Lema II.1: "Se G = (Vn, Vt, P, S) é uma G.S.C., então $\exists$ G_1 S.C. | $L(G_1) = L(G) \wedge$ o símbolo inicial de G_1 não apareça no lado direito de nenhuma produção de G_1".

Prova: Seja G = (Vn, Vt, P, S) uma G.S.C.;
Seja $S_1 = (Vn \cup Vt)$;
Seja $G_1 = (Vn \cup \{S\ G_1\}, Vt, P1, S1)$, onde:
$P\ G_1 = \{S\ G_1 \rightarrow \alpha \mid S\ G_1 \rightarrow \alpha\ \exists \leftrightarrow S \rightarrow \alpha \in P\} \cup P \wedge \underline{S}\ G_1$ será o símbolo inicial de G_1.

Para completar a prova, é necessário mostrar que $L(G_1) = L(G)$:

$L(G_1) = L(G) \Leftrightarrow$ 1 - $L(G) \subseteq L(G_1)$
2 - $L(G_1) \subseteq L(G)$

1 – Seja $w \in L(G)$ $\therefore S \overset{+}{\Rightarrow} w$
se $S \overset{+}{\Rightarrow} w$
então $S \Rightarrow \alpha \overset{*}{\Rightarrow} w$
se $S \Rightarrow \alpha$
então $S \rightarrow \alpha \in P$ e, por definição, $S_1 \rightarrow \alpha \in P_1$

Logo, $S_1 \Rightarrow \alpha \Rightarrow w \therefore S_1 \overset{+}{\Rightarrow} w \therefore L(G) \subseteq L(G_1)$

2 – Seja $w \in L(G_1)$ $\therefore S_1 \overset{+}{\Rightarrow} w$
se $S_1 \overset{+}{\Rightarrow} w$
então $S_1 \Rightarrow \alpha \overset{*}{\Rightarrow} w$
se $S_1 \Rightarrow \alpha$
então $S_1 \rightarrow \alpha \in P_1$ e, para que esta produção exista, é necessário que P contenha a produção $S \rightarrow \alpha$, portanto:
$S \Rightarrow \alpha \overset{*}{\Rightarrow} w \therefore S \overset{+}{\Rightarrow} w \therefore L(G_1) \subseteq L(G)$

Conclusão: De 1 e 2, tem-se que **$L(G_1) = L(G)$.**

OBS.: 1 – O lema II.1 vale também para G.L.C. e G.R..
2 – Se G é S.C., L.C. ou Regular, então G_1 será respectivamente S.C., L.C. ou Regular.

Teorema II.1: "Se L é S.C., L.C. ou REGULAR, então $L_1 = L \cup \{\varepsilon\}$ e $L_2 = L - \{\varepsilon\}$ serão respectivamente S.C., L.C. ou Regular".
Prova: Imediata a partir da redefinição de gramáticas e do LEMA II.1.

II.7 – Recursividade das G.S.C.

Definição: Uma gramática G é RECURSIVA se existe um algoritmo que determine para qualquer seqüência w, se w é ou não gerada por G.

Teorema II.2: "Se G = (Vn, Vt, P, S) é uma G.S.C., então G é RECURSIVA".

Prova: (* através de algoritmo *)

Seja G = (Vn, Vt, P, S) uma G.S.C. | $S \rightarrow \varepsilon \notin P$;

Seja w uma seqüência $\forall$ |w| = n;

Seja T_M o conjunto de formas sentenciais α | $|\alpha| \leq N \wedge S \overset{*}{\Rightarrow} \alpha$ em M passos.

Algoritmo II.1:

1 – To = { S }
2 – M $\leftarrow$ 1
3 – $T_M \leftarrow T_{M-1} \cup \{\alpha \mid \beta \Rightarrow \alpha$, onde $\beta \in T_{M-1} \wedge |\alpha| \leq n\}$
4 – Se $T_M = T_M$-1
    então fim
    senão M $\leftarrow$ M+1;
        calcule novo T_M (volte ao passo 3).

Conclusão: Como em uma G.S.C. as formas sentenciais não são decrescentes de tamanho ($|\alpha| \leq |\beta|$), e como o número de sentenças de tamanho n é finito (no máximo K^n onde K é o número de símbolos de Vt), sempre existirá um M | $T_M = T_M$-1 $\therefore$ o algoritmo II.1 sempre determinará para qualquer w, se w é ou não gerado por G.

Capítulo III – Autômatos Finitos e Conjuntos Regulares

Geradores X Reconhecedores

Gramáticas Tipo 0	→	Máquinas de Turing
Gramáticas S.C.	→	Autômatos Limitados Lineares
Gramáticas L.C.	→	Autômatos de Pilha
Gramáticas Regulares	→	Autômatos Finitos

Autômatos Finitos são reconhecedores de linguagens regulares;

Entende-se por reconhecedor de uma linguagem "L", um dispositivo que tomando uma seqüência w como entrada, respondem "SIM" se w $\in$ L e "NÃO" em caso contrario.

Tipos de Autômatos Finitos:

Autômato Finito Determinístico (A.F.D.)

Autômato Finito Não Determinístico(A.F.N.D.)

III.1 A.F.D.

Formalmente definimos um A.F.D. como sendo um sistema formal M = (K, Σ, δ, qo, F), onde:

K → É um conjunto finito não vazio de **Estados**;

Σ → É um **Alfabeto**, finito, de entrada;

δ → **Função de Mapeamento** (ou função de transição)
definida em: **K x Σ → K**

qo → $\in$ K, é o **Estado Inicial**

F → $\subseteq$ K, é o conjunto de **Estados Finais**

Interpretação de δ

A interpretação de uma transição $\delta(q, a) = p$, onde $q \wedge p \in K \wedge a \in \Sigma$, é a seguinte: se o "Controle de M" esta no estado "q" e o próximo símbolo de entrada é "a", então "a" deve ser reconhecido e o controle passar para o estado "p".

Significado Lógico de um Estado

Logicamente um estado é uma situação particular no processo de reconhecimento de uma sentença.

Sentenças Aceitas por M

Uma seqüência x é aceita (reconhecida) por um A.F. M = (K, Σ δ, qo, F),

$$\delta(qo, x) = p \mid p \in F.$$

Linguagem Aceita por M

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

$$T(M) = \{x \mid \delta(qo, x) = p \wedge p \in F\}$$

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

Diagrama de Transição

Um diagrama de transição para um A.F. M é um grafo direcionado e rotulado, onde os vértices representam os estados e fisicamente são representados por círculos (sendo que o estado inicial é possui uma seta com rótulo "**Inicio**" e os estados finais são representados por círculos duplos), e 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.

Tabela de Transições

É uma representação tabular de um A.F..

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 δ(q, a) = p, senão será indefinido.

III.2 A.F.N.D.

Um A.F.N.D. é um sistema formal **M = (K, Σ, δ, qo, F),** onde:

K, Σ qo, F → possuem a mesma definição dos A.F.D.

δ → É uma função de mapeamento, definido em **K x Σ = ρ(K)**; sendo que **ρ(K)** é um subconjunto de K; isto eqüivale a dizer que $\delta(q, a) = p_1, p_2, ..., p_n$. 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.

	Vantagem	**Desvantagem**
A.F.D.	Implementação Trivial	Não natural na representação de algumas L.R.
A.F.N.D.	Representação mais natural de algumas LR	Implementação complexa

III.3 Transformação de A.F.N.D. para A.F.D.

Teorema 3.1: "Se L é um conjunto aceito por um **A.F.N.D**., então ∃ um **A.F.D**. que aceita L"

Prova: Seja M = (K, Σ, δ, qo, F) um A.F.N.D..

Construa um A.F.D. M' = (K', Σ, δ', qo', F') como segue:

1 – K' = {ρ(k)} - isto é, cada estado de M' será um subconjunto de estados de M.
2 – qo' = [qo] - ou seja, qo' será o ρ(k) composto apenas por q0.
obs.: representaremos um estado q ∈ K' por [q].
3 – F' = {ρ(K) | ρ(K) ∩ F ≠ φ}
4 – Para cada ρ(K) ⊂ K'
definimos δ'(ρ(K),a) = ρ'(K),
onde ρ'(K) = {p | para algum q ∈ ρ(K), δ(q, a) = p};
ou seja, se ρ(K) = [q1, q2, ..., qr] ∈ K' e se
δ(q1,a) = p1, p2, ..., pj
δ(q2,a) = pj+1, pj+2, ..., pk
: : : :
δ(qr, a) = pi, pi+1, ..., pn são as transições de M,
então ρ(K) = [p1, ..., pj, pj+1, ..., pr, pi, ..., pn] será um estado de M',
e M' conterá a transição: δ'(ρ(K),a) = ρ'(K).

Para concluir a prova do teorema, basta, mostrar que T(M') = T(M).

Exemplo:

Seja M um A.F.N.D. definido por:

δ	a	b
→qo	qo,q1	qo
q1	---	q2
q2	---	q3
*q3	---	---

onde T(M) = { }

Defina **M' = (K', Σ', δ', qo', F')**, onde:

K' = {
Σ' = {a, b)
qo =
F' =

δ'	a	b
→[qo]		

T(M') = { }

III.4 Relação entre G.R. e A.F.

Teorema 3.2: "Se G = (Vn, Vt, P, S) é uma G.R.,
então $\exists$ um A.F. M = (K, Σ, δ, qo, F) | T(M) = L(G)".

Prova: a – Mostrar que M existe
b – Mostrar que T(M) = L(G)

a) Defina M, como segue:

1 – K = Vn $\cup$ {A}, onde A é um símbolo novo
2 - Σ = Vt
3 – qo = S
4 – F = {A, S} se S $\rightarrow \varepsilon \in$ P
{A} se S $\rightarrow \varepsilon \in$ P
5 – Construa δ de acordo com as regras a, b e c.
a) Para cada produção da forma B $\rightarrow$ a $\in$ P, crie a transição δ (B, a) = A
b) Para cada produção da forma B $\rightarrow$ a C $\in$ P crie a transição δ (B, a) = C
c) Para todo a $\in$ Vt, δ (A, a) = - (indefinido)

b) Para mostrar que T(M) = L(G), devemos mostrar

que : **1 – L(G) $\subseteq$ T(M)**
2 – T(M) $\subseteq$ L(G)

1 – L(G) $\subseteq$ T(M)

Mostrar que L(G) está contido em T(M), significa mostrar que se x $\in$ L(G) então T(M) contém x, ou seja, M aceita x.

Seja x = $a_1a_2...a_n \in$ L(G).

Se x $\in$ L(G), então S $\stackrel{+}{\Rightarrow}$ x

Como G é uma G.R., a derivação S $\stackrel{+}{\Rightarrow}$x é da forma

$$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow ... \Rightarrow a_1a_2...na_{-1}A_{n-1} \Rightarrow a_1a_2...a_n$$

Logo,

$S \rightarrow a_1A_1$, $A_1 \rightarrow a_2A_2$, ... $An_{-1} \rightarrow a_n$ são produções de G.

Assim sendo, por definição, M consistirá das seguitnes transições:

$\delta (S,a_1) = A_1$, $\delta (A_1,a_2) = A_2$, ... $\delta (A_{n-1}, a_n) = A$

Portanto, como $x = a_1a_2...a_n$, δ (S, x) = A $\wedge$ A $\in$ F,

conclui-se que T(M) contém x.

Mas, e se **ε** $\in$ L(G) ???

Se $\varepsilon \in$ L(G) é porque S $\rightarrow \varepsilon \in$ P

Neste caso, por definição, S $\in$ F e, portanto, T(M) contém ε.

Logo**, L(G) $\subseteq$ T(M).**

2 – T(M) $\subseteq$ L(G)

Mostrar que T(M) está contido em L(G), significa mostrar que, se T(M) contém x, então $x \in L(G)$.
Seja $x = a_1a_2...a_n$ uma seqüência aceita por M.

Se M aceita x, então $\exists$ uma seqüência de estados S, A_1, A_2, ..., A_{n-1}, A |
$\delta(S, a_1) = A_1$,
$\delta(A_1, a_2) = A_2$
:
$\delta(A_{n-1}, a_n) = A$, onde S é o estado inicial e A é um estado final.

por definição, para que essas transições existam, G deverá possuir as seguintes produções:

$$\mathbf{S \rightarrow a_1A_1,\ A_1 \rightarrow a_2A_2,\ ...\ An\text{-}_1 \rightarrow a_n}$$

e, se essas produções existem, então
$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow ... \Rightarrow a_1a_2...na_{-1}A_{n-1} \Rightarrow a_1a_2... a_n$ é uma derivação em G;

logo, como $x = a_1a_2...a_n \wedge S \overset{+}{\Rightarrow} x$, então $\underline{x \in L(G)}$

Mas, e se $\varepsilon \in T(M)$?!?
Neste caso, por definição, $S \in F$;
e se $S \in F$ é porque $S \rightarrow \varepsilon \in P$,
logo $\varepsilon \in L(G)$.

Assim, **$\underline{T(M) \subseteq L(G)}$.**

$\underline{Conclusão}$: De $\underline{1}$ e $\underline{2}$, temos que **$\underline{T(M) = L(G)}$**

$\underline{Teorema\ 3.3}$: "Se M = (K, Σ, δ, qo, F) é um A F., então $\exists$ uma G.R. G = (Vn, Vt, P, S) | L(G) = T(M)".
$\underline{Prova}$: a – Mostrar que G existe
b – Mostrar que L(G) = T(M)

a) Seja M = (K, Σ, δ, qo, F) um A.F.D..
Construa uma G.R. G = (Vn, Vt, P, S), como segue:
1 – Vn = K
2 – Vt = Σ
3 – S = qo
4 – Defina P, como segue:
a) $\underline{Se}$ δ(B, a) = C $\underline{então}$ adicione $\underline{B \rightarrow aC}$ em P
b) $\underline{Se}$ δ(B, a) = C $\wedge$ C ∈ F $\underline{então}$ adicione $\underline{B \rightarrow a}$ em P
c) $\underline{Se}$ qo ∈ F,
$\underline{então}$ ε ∈ T(M).
Neste caso, L(G) = T(M) – {ε},

contudo, usando o teorema 2.1 podemos encontrar uma G.R. $G_1 \mid L(G_1) = L(G) \cup \{\varepsilon\}$ e portanto, $L(G_1)$ será igual a T(M)

Senão, $\varepsilon \notin T(M)$ e $L(G) = T(M)$.

b) Para mostrar que L(G) = T(M), devemos mostrar

que: 1 – $T(M) \subseteq L(G)$

2 – $L(G) \subseteq T(M)$

Isto pode ser feito de maneira análoga a demonstração do teorema 3.2.

III.5 Minimização de Autômatos Finitos

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

1 – Não possui estados inacessíveis;

2 – Não possui estados mortos;

3 – Não possui estados equivalentes.

Estados Inacessíveis:

Um estado $q \in K$ é inacessível (ou inalcançável) quando não existe w_1 tal que a partir de qo, q seja alcançado; ou seja, não existe $w_1 \mid \delta(qo, w_1) = q$, onde w_1 é uma sentença ou parte dela.

Estados mortos:

Um estado $q \in K$ é morto se ele $\notin F \wedge \exists\ w_1 \mid \delta(q, w1) = p$, onde $p \in F \wedge w_1$ 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 q1, q2, ..., qj são equivalentes entre sí, se eles pertencem a uma mesma classe de equivalência.

Classes de Equivalência (CE):

Um conjunto de estados q1, q2, ..., qj está em uma mesma CE se δ (q1, a), δ(q2, a), ..., δ(qj, a), para cada $a \in \sum$, resultam respectivamente nos estados qi, qi+1, ..., qn, e estes pertencem a uma mesma CE.

Algoritmo para Construção das Classes de Equivalência

1 – Crie, se necessário, um estado ϕ para representar as indefinições;

2 – Divida 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.

Algoritmo para construção do A.F. Mínimo

Entrada: Um A.F.D. M = (K, Σ, δ, qo, F);
Saída: Um A.F.D. Mínimo M' = (K', Σ, δ', qo', F') | M' ≡ M;
Método:

1 – Elimine os estados Inacessíveis;
2 – Elimine os estados Mortos;
3 – Construa todas as possíveis Classes de equivalência de M.
4 – Construa M', como segue:

a) K' - é o conjunto de CE obtidas;
b) qo' - é a CE que contem qo;
c) F' - é o conjunto das CE que contenham pelo menos um elemento ∈ F; ou seja : {[q] | ∃ p ∈ F em [q], onde [q] é uma CE};
d) δ' - δ'([p], a) = [q] ⇔ δ(p_1, a) = q_1 é uma transição de M ∧ p_1 e q_1 são elementos de [p] e [q] respectivamente.

Exemplo: Minimize o A.F.D. definido pela seguinte tabela de transição:

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

III.6 – Conjuntos Regulares e Expressões Regulares

Conjuntos Regulares (C.R.)

Um C.R. pode ser definido de quatro maneiras equivalentes:

1 – (* definição matemática (primitiva) *)

Seja Σ um alfabeto qualquer.
Definimos um C.R. sobre Σ, como segue:
a – **ϕ** é um C.R. sobre Σ;
b – **{ε}** é um C.R. sobre Σ;
c – **{a}**, para todo a ∈ Σ, é um C.R. sobre Σ;
d – Se **P** e **Q** são **C.R**. sobre Σ, então:
1 – **P ∪ Q** (união),
2 – **P.Q** (ou PQ) (concatenação),
3 – **P** * (fechamento).
Também são C.R. sobre ∑;
e – Nada mais é C.R.

Portanto, um subconjunto de ∑* é uma C.R. se e somente se ele é: ϕ, {ε}, {a}, ou pode ser obtido a partir destes através de um número finito de aplicação das operações de **União, Concatenação e Fechamento.**

2 – C.R. são as linguagens geradas por Gramáticas Regulares.

3 – C.R. são as linguagens reconhecidas por Autômatos Finitos.

4 – C.R. são os conjuntos denotados (representados) por Expressões Regulares.

Expressões Regulares (E.R.)

As E.R. sobre um alfabeto Σ e os C.R. que elas denotam, são definidos como segue:

1 – **ϕ** é uma E.R. e denota o C.R. ϕ;
2 – **ε** é uma E.R. e denota o C.R. {ξ};
3 – **a** para todo a ∈ Σ, é uma E.R. e denota o C.R. {a};
4 – Se p e q são E.R. denotando os C.R. P e Q respectivamente, então:
    a – **(p | q)** é uma E.R. denotando o C.R. P ∪ Q;
    b – **(p.q)** ou **(pq)** é uma E.R. denotando o C.R. PQ;
    c – **(p)*** é uma E.R. denotando o C.R. P*;
5 – Nada mais é E.R..

Observações:
1 – Os parênteses podem ser eliminados; neste caso, para evitar ambigüidades, considera-se a seguinte ordem de precedência: 1) * 2) . 3) |
2 – Para simplificar E.R., podemos utilizar as seguintes abreviaturas:

$p^+ = pp^*$
$p^? = p \mid \varepsilon$
$p^{£}q = p(qp)^*$

Relação entre E.R. e C.R.

1 – Para todo C.R. ∃ pelo menos uma E.R. que o denota;
2 – Para toda E.R. é possível construir o C.R. que ela denota;
3 – Duas E.R. são equivalentes se elas denotam o mesmo C.R..

III.7 - Implementação de Autômatos Finitos

Existem duas formas básicas para implementação de A.F.:

- **Implementação Específica**

- **Implementação Geral (ou genérica);**

Implementação Específica

Consiste em representar cada estado do A.F. através de um conjunto de instruções, ou seja, consiste em, literalmente, programar cada estado do A.F..

Exemplo:

TAB	a	b	c
q1	1	2	4
q2	4	2	3
q3	1	4	3
q4	4	4	4

(* um exemplo de uma implementação específica para o AF dado *)

```
inicio
q0: leia CAR
    se CAR = 'a'
    então va-para q0
    senão  se  CAR ≠ 'b'
            então va-para qerro
            fim se
    fim se

q1: leia CAR
    se CAR = 'b'
    então va-para q1
    senão se CAR ≠ 'c'
            então va-para qerro
            fim se
    fim se

q2: leia CAR
    se CAR = '$'
    então escreva 'Sequência Reconhecida'  pare
    senão se CAR = 'c'
            então va-para q2
            senão se CAR = 'a'
                    então va-para q0
                    fim se
            fim se
    fim se

q erro: enquanto CAR ≠ '$' faça
              leia CAR
          fim enquanto
          escreva 'Sequência não Reconhecida'
fim
```

Implementação GERAL

Esta forma de implementação de A.F. requer uma tabela de transições e um vetor de estados finais e, logicamente, consiste na elaboração de um procedimento genérico para interpretar a tabela de transições em função da seqüência de entrada a ser analisada.

Exemplo: Para o A.F. dado anteriormente, teríamos:

Tabela de Transições **Vetor de Estados Finais**

	TAB	a	b	c	<> a, b, c
(q_0)	1	1	2	4	4
(q_1)	2	4	2	3	4
(q_2)	3	1	4	3	4
(q_{erro})	4	4	4	4	4

VE	F
1	0
2	0
3	1
4	0

Observação: VEF [EST] = 0 → EST $\notin$ F
VEF [EST] = 1 → EST $\in$ F

Procedimento

```
Inicio
   (* inicialização de TAB e VEF *)
   leia EST, CAR  (* estado inicial, próximo caracter *)
   enquanto CAR ≠ '$' faça
        EST:= TAB[EST, CAR]
        leia CAR
   fim enquanto
   se VEF [EST] = 1
      então escreva 'sequência reconhecida'
      senão escreva 'sequência não reconhecida'
    fim se
fim
```

Observações:

- A escolha entre uma e outra forma de implementação depende da aplicação;
- Os exemplos de implementação apresentados são ilustrativos, na prática qualquer implementação deve ser adaptada segundo seus objetivos. Vale, portanto a filosofia de implementação.

III.8 – Principais Propriedades e Problemas de Decisão sobre Conjuntos (Linguagens) Regulares

Propriedades Básicas de C.R.

A classe dos C.R. é fechada sobre as operações de UNIÃO, CONCATENAÇÃO, COMPLEMENTO, e INTERSECÇÃO, isto é:

1 – UNIÃO:
Se L_1 e L_2 são C.R., então
$L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$ também é um C.R.

2 – CONCATENAÇÃO:
Se L_1 e L2 são C.R., então
$L_1L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$ também é um C.R.

3 – COMPLEMENTO:
Se $L_1 \subseteq \Sigma^*$ é um C.R., então
$\Sigma^* - L_1$ também é um C.R.

4 – INTERSECÇÃO:
Se L_1 e L_2 são C.R., então
$L_1 \cap L_2 = \{x \mid x \in L_1 \wedge x \in L_2\}$ também é um C.R.

Problemas de Decisão sobre C.R.

Os problemas de decisão sobre C.R. podem ser formulados e demonstrados através de qualquer um dos modelos usados para expressar tais conjuntos. Usaremos aqui o modelo de Autômatos finitos em função de sua grande força de expressão e também por sua aplicabilidade:

1 – MEMBERSHIP
Dado um AF M = (K, Σ, δ, q0, F) e um string $x \in \Sigma^*$.
X é aceito por M? ou seja, $x \in T(M)$?

2 – EMPTINESS
Dado um AF M = (K, Σ, δ, q0, F), $T(M) = \varphi$?

3 – FINITENESS
Dado um AF M = (K, Σ, δ, q0, F), T(M) é finita?

4 – CONTAINMENT
Dados dois AF's M1 e M2, $T(M1) \subseteq T(M2)$?

5 – EQUIVALENCE
Dados dois AF's M1 e M2, T(M1) = T(M2)?

6 – INTERSECTION
Dados dois AF´s M1 e M2, $T(M1) \cap T(M2) = \varphi$?

Todos os problemas acima apresentados são problemas decidíveis.

III.9 – Aplicações de A.F. e E.R.

Apesar da simplicidade destas ferramentas, existe uma grande variedade de software cuja especificação e/ou implementação pode ser bastante simplificada se realizada em termos de A.F. e E.R., resultando em software´s mais eficientes. Dentre as diversas aplicações que fazem uso de A.F. e E.R. podemos citar:

a) Analisador Léxico: Os tokens (símbolos básicos) de uma linguagem de programação geralmente podem ser especificados eficientemente através de E.R., as quais podem ser automaticamente convertidas para A.F.D. equivalentes, cuja implementação (o analisador léxico propriamente dito) é trivial.
Isto permite inclusive, a geração automática de um analisador léxico a partir da especificação formal dos "tokens" de uma linguagem de programação.

b) Editores de Texto: As operações de busca e substituição de cadeias de caracteres em um texto, podem ser eficientemente realizadas, se a cadeia for expressa em termos de E.R. e a operação realizada em termos de um A.F. usufruindo da possibilidade da conversão automática de E.R. para A.F., de forma transparente ao usuário.

c) Além das aplicações acima (tidas como as principais) podemos também destacar o uso destas ferramentas nas seguintes áreas:

c.1) Protocolos de comunicação
c.2) Projeto (modelagem) de sistemas operacionais
c.3) Path Expression
c.4) Problemas específico tais como: segurança de arquivos, desorientação em sistemas de hipertexto, modelagem de redes neurais, compressão de dados, etc...

Capítulo IV – Gramáticas Livres de Contexto e Autômatos de Pilha

Introdução: Dentre os quatro tipos de gramáticas estudadas, as G.L.C. são as mais importantes na área de compiladores e linguagens de programação, pelo fato de especificarem eficientemente as construções sintáticas usuais.

As Várias Definições de G.L.C.

Inicialmente definimos uma G.L.C. como sendo G = (Vn, Vt, P, S), onde $P = \{A \rightarrow \alpha \mid A \in Vn \wedge \alpha \in (Vn \cup Vt)^+\}$.

Posteriormente estendemos esta definição permitindo a existência da produção $S \rightarrow \varepsilon$, desde que S fosse o símbolo inicial de G e não aparecesse no lado direito de nenhuma das produções de P. As gramáticas assim definidas denominam-se G.L.C. ε - LIVRES.

Agora, redefiniremos (em definitivo) as G.L.C. permitindo a existência de produções da forma $A \rightarrow \varepsilon$ para $\forall A \in Vn$.

Formalmente, passamos a definir uma G.L.C. G como sendo G = (Vn, Vt, P, S), onde

$P = \{A \rightarrow \alpha \mid A \in Vn \wedge \alpha \in (Vn \cup Vt)^*\}$, ou seja, agora α pode ser ε.

IV.1 – Árvore de Derivação (A.D.) ou Árvore Sintática (A.S.)

É um método estruturado para representar as derivações de uma G.L.C..

Definição:

Seja G = (Vn, Vt, P, S) uma G.L.C..

Uma **Árvore** é uma **Árvore de Derivação** de G, se:

a) Todos os nodos da árvore forem rotulados com símbolos pertencentes à Vn ∪ Vt;

b) O nodo raiz da árvore for rotulado com S, o símbolo inicial de G;
c) Todo nodo com descendentes (além dele próprio) possuir como rótulo um símbolo $\in$ Vn;
d) Para todo nodo n (rotulado por A) com descendentes diretos n1, n2, ..., nk (rotulados, respectivamente, por A1, A2, ..., Ak) existir uma produção
A → A1 A2 ... Ak em P

Exemplos:

IV.2 – Limite de uma Árvore de Derivação

O limite de uma A.D. é o string obtido pela concatenação (da esquerda para a direita) das folhas da árvore; ou seja, é a forma sentencial cuja derivação esta sendo representada pela A.D.

Lema 4.1: "Se D é uma A.D. com limite α em G então S → α."

Lema 4.2: "Para toda derivação de α em G, $\exists$ uma A.D. com limite α."

IV.3 – Derivação mais a Esquerda e mais a Direita

Seja a derivação $S \Rightarrow \alpha \Rightarrow \beta$.

Considerando que β é obtido de α pela substituição de um Não-Terminal de α, temos que:

a) Se o símbolo substituído foi o mais da esquerda, então esta é uma derivação mais à esquerda;
b) Se o símbolo substituído foi o mais da direita, então esta é uma derivação mais à direita;

Observação: Se α é obtido por uma derivação mais a esquerda ou mais a direita dizemos que α é uma forma sentencial **Esquerda** ou **Direita**, respectivamente.

IV.4 – Gramática Ambígua

Definição: Uma G.L.C. G = (Vn, Vt, P, S) é ambígua se para algum x $\in$ L(G) existe mais de uma A.D..

De outra forma, G é ambígua se existir algum x $\in$ L(G), tal que x possua mais de uma **Derivação mais a Esquerda** ou mais de uma **Derivação mais a Direita**.

Exemplos:

IV.5 – Transformações (Simplificações) em G.L.C.

IV.5.1 – Eliminação de Símbolos Inúteis

Definição: Um símbolo $X \in Vn \cup Vt$ é inútil em uma G.L.C. G = (Vn, Vt, P, S), se não existe em G uma derivação da forma:

$$S \overset{*}{\Rightarrow} w\,X\,y \overset{*}{\Rightarrow} w\,x\,y$$

onde w, x, y, $\in Vt^*$, $S \in Vn \wedge X \in Vn \cup Vt$; ou seja, um símbolo é inútil se ele é infértil (não gera string só de terminais) ou inalcançável (não aparece em nenhuma forma sentencial de G).

Algoritmo IV.1

Objetivo – Encontrar o conjunto de **Não Terminais Férteis**.
Entrada – Uma G.L.C. G = (Vn, Vt, P, S).
Saída – NF – Conjunto de Não-Terminais **Férteis**.
Método:

Construa conjuntos N0, N1, ..., como segue:
$i \leftarrow 0$
$Ni \leftarrow \phi$
repita
    $i \leftarrow i + 1$
    $Ni \leftarrow Ni\text{-}1 \cup \{A \mid A \rightarrow \alpha \in P \wedge \alpha \in (Ni\text{-}1 \cup Vt)^*\}$
até $Ni = Ni\text{-}1$
$NF \leftarrow Ni$

Fim

Observações: Uma variante do algoritmo IV.1 pode ser usada para verificar se L(G) é vazia: basta verificar se S (o símbolo inicial de G) $\in$ NF.
Exemplo:

Algoritmo IV.2:

Objetivo: Eliminar símbolos **Inalcançáveis**.
Entrada: Uma G.L.C. G = (Vn, Vt, P, S).
Saída: Uma G.L.C. G' = (Vn', Vt', P', S') na qual todos os símbolos $\in$ (Vn' $\cup$ Vt') sejam alcançáveis.
Método:

Construa conjuntos V0, V1, ..., como segue:
$i \leftarrow 0$; $Vi \leftarrow \{S\}$
repita
    $i \leftarrow i + 1$
    $Vi \leftarrow Vi\text{-}1 \cup \{X \mid A \rightarrow \alpha\,X\,\beta \in P, A \in Vi\text{-}1 \wedge \alpha\,e \in (Vn \cup Vt)^*\}$
até $Vi = Vi\text{-}1$

Construa G' = (Vn', Vt', P', S'), como segue:

a) Vn' ← Vi ∩ Vn
b) Vt' ← Vi ∩ Vt
c) P' ← conjunto de produções de P, que envolvam apenas símbolos de Vi
d) S' ← S

Fim

Exemplo:

Algoritmo IV.3

Objetivo: Eliminar **símbolos inúteis**.
Entrada: Uma G.L.C. G = (Vn, Vt, P, S).
Saída: Uma G.L.C. G' = (Vn', Vt', P', S') | L(G') = L(G) e nenhum símbolo de G' seja inútil.
Método:

1 – Aplique o algoritmo IV.1 para obter NF;

2 – Construa G1 = (Vn ∩ NF, Vt, P1, S), onde P1 contém apenas produções envolvendo símbolos pertencentes à NF ∪ Vt;

3 – Aplique o ALG IV.2 em G1, para obter G1' = (Vn', Vt', P', S');

Fim

Exemplo:

IV.5.2 – Transformações de G.L.C. em G.L.C. ε - Livre

Toda G.L.C. pode ser transformada em uma G.L.C. ε - **Livre**, através do seguinte algoritmo:

Algoritmo IV.4:

Objetivo: Transformar uma G.L.C. em uma G.L.C. ε - LIVRE equivalente
Entrada: Uma G.L.C. G = (Vn, Vt, P, S).
Saída: Uma G.L.C. ε - **Livre** G' = (Vn', Vt', P', S') | L(G') = L(G).
Método:

1 – Construa Ne = $\{A \mid A \in Vn \wedge A \overset{*}{\Rightarrow} \varepsilon\}$.

2 – Construa P' como segue:

a) Inclua em P' todas as produções de P, com exceção daquelas da forma $A \rightarrow \varepsilon$.
b) Para cada produção de P da forma:

$$\mathbf{A \rightarrow \alpha\ B\ \beta \mid B \in Ne \wedge \alpha, \beta \in V^*}$$

inclua em P' a produção $A \rightarrow \alpha\beta$

OBS: caso exista mais de um não terminal pertencente a Ne no lado direito de uma produção, faça todas as combinações entre esses não terminais.

c) Se $S \in Ne$, adicione a P' as seguintes produções:

$S' \rightarrow S$ e $S' \rightarrow \varepsilon$ incluindo S' em N' ($N' = N \cup \{S'\}$)

caso contrário, faça $N' = N$ e $S' = S$.

3 – A gramática transformada será: G' = (Vn', Vt', P', S').

Fim.

Exemplos:

1) $S \rightarrow a\ S\ b \mid c \mid \varepsilon$

2) $S \rightarrow AB$
   $A \rightarrow aA \mid \varepsilon$
   $B \rightarrow bB \mid \varepsilon$

IV.5.3 – Eliminação (Remoção) de Produções Simples

Definição: Produções Simples são produções da forma $A \rightarrow B$, onde A e $B \in Vn$.
As produções simples de uma G.L.C. podem se removidas através do seguinte algoritmo:

Algoritmo IV.5:

Entrada: Uma G.L.C. ε - **Livre** G = (Vn, Vt, P, S).
Saída: Uma G.L.C. ε - **Livre** G' = (Vn', Vt', P', S') sem produções simples | L(G') = L(G).
Método:

1 – Para todo $A \in Vn$, construa $NA = \{B \mid A \overset{*}{\Rightarrow} B\}$
2 – Construa P' como segue:
se $B \rightarrow \alpha \in P$ e não é uma produção simples,
então adicione a P' as produções da forma:

$$\mathbf{A \rightarrow \alpha, \text{ para todo } A \mid B \in NA}$$

3 – Faça G' = (Vn, Vt, P', S).

Fim.

Exemplos:

1) $S \rightarrow FGH$
   $F \rightarrow G \mid a$
   $G \rightarrow dG \mid H \mid b$
   $H \rightarrow c$

2) S → a B c D e
   B → b B | E | F
   D → d D | F | d
   E → e E | e
   F → f F | f

IV.5.4 – Fatoração de G.L.C.

Uma G.L.C. está **fatorada** se ela é determinística; isto é, não possui A – produções cujo lado direito inicie com o mesmo conjunto de símbolos ou com símbolos que geram (derivam) seqüências que iniciem com o mesmo conjunto de símbolos.
Exemplos:

Processo de Fatoração

Para fatorar uma G.L.C. alteramos as produções envolvidas no não-determinismo, da seguinte maneira:

a) As produções com **Não-Determinismo Direto** da forma:
   $A \rightarrow \alpha\ \beta \mid \alpha\ \gamma$
   Devem ser substituídas por:
   $A \rightarrow \alpha\ A'$
   $A' \rightarrow \beta \mid \gamma$
b) **O Não-Determinismo Indireto** é retirado através de sua transformação em um não-determinismo **Direto** (via derivações sucessivas), o qual é eliminado como previsto no item a.

Exemplos:

IV.5.5 – Eliminação de Recursão a Esquerda

Definições:

a) Um **Não-Terminal** A em uma G.L.C. G = (Vn, Vt, P, S) é recursivo se $A \overset{+}{\Rightarrow} \alpha\ A\ \beta$, para $\alpha \wedge \beta \in V^*$.
b) Uma **Gramática** com pelo menos um NT recursivo a **Esquerda** ou a **Direita**, é uma **Gramática Recursiva** a **Esquerda** ou a **Direita**, respectivamente.

Recursão a Esquerda

a) **Direta** (ou **Imediata**) : Uma G.L.C. G = (Vn, Vt, P, S) possui recursão a esquerda direta, se p contém pelo menos uma produção da forma $A \rightarrow A\alpha$.
b) **Indireta :** Uma G.L.C. G = (Vn, Vt, P, S) possui recursão a esquerda indireta, se existe em G uma derivação da forma:
$\mathbf{A \Rightarrow ... \overset{*}{\Rightarrow} A\beta.}$

Eliminação de Recursões a Esquerda Imediata

Sejam as seguintes A - produções de G:

$\mathbf{A \rightarrow A\alpha 1 \mid A\alpha 2 \mid ... \mid A\alpha n \mid \beta 1 \mid \beta 2 \mid ... \mid \beta m}$

Obs.: onde nenhum βi começa com "A".

Substitua estas produções pelas seguintes:

$\mathbf{A \rightarrow \beta 1A' \mid \beta 2A' \mid ... \mid \beta mA'}$
$\mathbf{A' \rightarrow \alpha 1A' \mid \alpha 2A' \mid ... \mid \alpha nA' \mid \varepsilon}$

Observações: onde A' é um novo não-terminal.

Processo para Eliminar Recursões a Esquerda (diretas e indiretas)

Entrada: Uma G.L.C. Própria G = (Vn, Vt, P, S).

Saída: Uma GLC G' = (Vn', Vt, P', S) | L(G') = L(G) ∧ G' não possui Recursão a Esquerda.

Método:

1 – Ordene os não-terminais de G em uma ordem qualquer (digamos: A1, A2, ..., An);
2 – Para i = 1, n faça
  Para j = 1, i – 1 faça
    Substitua as produções da forma

    $\mathbf{Ai \rightarrow Aj\ \gamma}$

    por produções da forma

    $\mathbf{Ai \rightarrow \delta 1\ \gamma \mid \delta 2\ \gamma \mid ... \mid \delta k\ \gamma}$

    onde δ1, δ2, ..., δk são os lados direitos das
      Aj – produções (Aj → δ1 | δ2 | ... | δk)
  fim para
  Elimine as rec. esq. **Diretas** das Ai – produções
 fim para
3 – **Fim**.

Exemplos:

IV.6 – TIPOS ESPECIAIS DE G.L.C

Gramática Própria:
Uma G.L.C. é **Própria**, se:

- Não possui **Ciclos**;
- É ε - **Livre**;
- Não possui **Símbolos Inúteis**.

Gramática Sem Ciclos:
G = (Vn, Vt, P, S) é uma G.L.C. Sem Ciclos (ou Livre de Ciclos) se não existe em G nenhuma derivação da forma $A \rightarrow A$, para $\forall A \in Vn$.

Gramática Reduzida:
Uma G.L.C. G = (Vn, Vt, P, S) é uma G.L.C. Reduzida se:

- L(G) não é vazia;
- Se $A \rightarrow \alpha \in P$, $A \neq \alpha$;
- G não possui Símbolos Inúteis.

Gramática de Operadores:
Uma G.L.C. G = (Vn, Vt, P, S) é de Operadores, se ela não possui produções cujo lado direito contenha NT consecutivos.

Gramática Unicamente Inversível:
Uma G.L.C. Reduzida é Unicamente Inversível se ela não possui produções com lados direitos iguais.

Gramática Linear:
Uma G.L.C. G = (Vn, Vt, P, S) é Linear se todas as suas produções forem da forma
$A \rightarrow x\ B\ w \mid x$, onde $A, B \in Vn \wedge x, w \in Vt^*$.

Forma Normal de Chomsky (F.N.C.):
Uma G.L.C. está na F.N.G. se ela é ε - LIVRE e todas as suas produções (exceto, possivelmente, $S \rightarrow \varepsilon$) são da forma:

- $A \rightarrow BC$, com A, B e C $\in$ Vn ou
- $A \rightarrow a$, com $A \in Vn \wedge a \in Vt$.

Forma Normal de Greibach (F.N.G.):
Uma G.L.C. está na F.N.G. se ela é ε - LIVRE e todas as suas produções (exceto, possivelmente, $S \rightarrow \varepsilon$) são da forma:

$$\mathbf{A \rightarrow a\alpha \mid a \in Vt, \alpha \in Vn^* \wedge A \in Vn.}$$

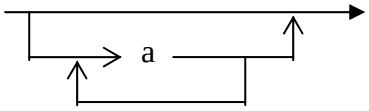
Principais Notacoes De G.L.C.:

BNF (**Backus Naur Form**) : Notação utilizada na especificação formal da sintaxe de Linguagens de Programação.
Exemplos: 1) <S> :: = a <S> | ε
2) <E> :: = E + id | id

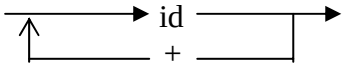
BNFE (**BNF Extended**) : Equivalente a **BNF,** permite uma especificação mais compacta da sintaxe de uma Linguagem de Programação.
Exemplos: 1) <S> :: = {a}
2) <E> :: = id {+ id}

RRP : É uma notação de G.L.C. onde os lados direito das produções são especificados através de E.R. envolvendo Vt e Vn de uma gramática.
Exemplos:
1) <S> :: = a*
2) <E> :: = id $^{£}$+

Diagrama Sintático : É uma notação gráfica de G.L.C..
Exemplos: 1) <S> :

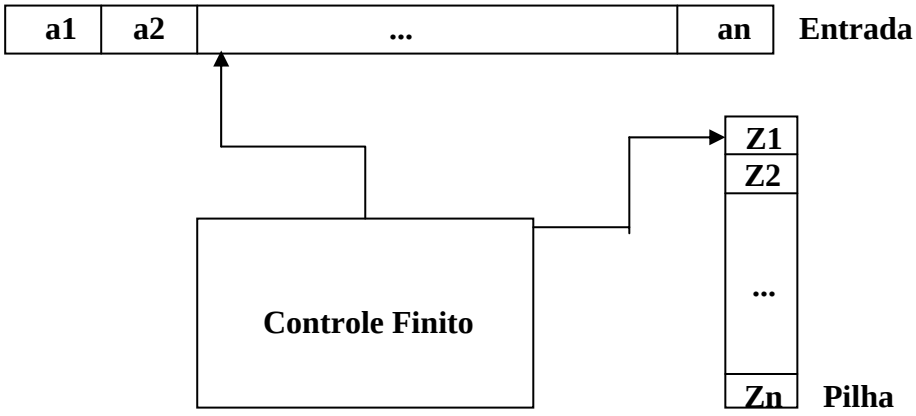


2) <E> :



IV.7 – Autômatos de Pilha (PDA)

Um autômato de pilha, também denominado "**Push Down Automata**", é um dispositivo não-determinístico reconhecedor de Linguagens Livres de Contexto (L.L.C.), que possui a seguinte estrutura geral:



Formalmente, um autômato de pilha (PDA) é um sistema definido por:

P = (K, ∑, Γ, δ, qo, Zo, F), onde

- **K** é um conjunto finito de **Estados**
- ∑ é o alfabeto finito de **Entrada**
- **Γ** é o alfabeto finito de **Pilha**
- **δ** é uma **Função De Mapeamento**, definida por:

$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \rightarrow \{K \times \Gamma^*\}$$

- **q0** $\in$ K, é o **Estado Inicial** de P
- **Z0** $\in \Gamma$, é o **Símbolo Inicial da Pilha**
- **F** $\subseteq$ K, é o conjunto de **Estados Finais**.

Movimentos de um PDA

Os movimento de um PDA são determinados com base nas seguintes informações:

- próximo símbolo da entrada,
- símbolo do topo da pilha
- estado corrente

Existem dois tipos de movimentos:

1 – Movimento dependente da entrada (a-move)

$\delta(q, a, Z) = \{(p1, \gamma1), (p2, \gamma2), ..., (pm, \gamma m)\}$ onde:

$a \in \Sigma$

$q, p1, p2, ..., pm \in K$

$Z \in \Gamma$

$\gamma i \in \Gamma^*$, para $i \leq i \leq m$

Significado – O PDA P no estado q, tendo a na entrada e Z no topo da pilha, deve entrar no estado pi (para qualquer i de 1 a m), substituindo Z por γi na pilha e avançando um símbolo de entrada (ou seja, reconhecendo o símbolo a).

1 – Movimento independente da entrada (ε -move)

$\delta(q, \varepsilon, Z) = \{(p1, \gamma1), (p2, \gamma2), ..., (pm, \gamma m)\}$ onde:
ε é a sentença vazia
$q, p1, p2, ..., pm \in k$
$Z \in \Gamma$
$\gamma i \in \Gamma^*$, para $i \leq i \leq m$

Significado – O PDA P no estado q, independente do símbolo de entrada e com Z no topo da pilha, pode entrar no estado pi ($i \leq i \leq m$) e substituir Z por γi na pilha, sem avançar o cabeçote de leitura da entrada (ou seja, sem reconhecer nenhum símbolo da entrada).

Descrição Instantânea (ou Configuração) de um PDA

A descrição instantânea de um PDA é dada por (q, w, γ), onde:

$q \in K$, representa o estado atual (corrente);
$w \in \sum^*$, é a porção da entrada ainda não analisada;
$\gamma \in \Gamma^*$, é o conteúdo (corrente) da pilha.

Os movimentos efetuados por um PDA são denotados por uma seqüência de configurações, de forma que:
Se $(q, aw, Z\gamma)$ é uma configuração e
se P contém a transição $\delta (q, a, Z) \rightarrow (q', a)$, onde:
$q, q' \in K$
$a \in \sum \cup \{\varepsilon\}$
$w \in \sum^*$
$\Gamma \supset Z$ (ou $Z \subset \Gamma$)
$\alpha \in \Gamma^*$

então a aplicação (o uso) desta transição deixará P na seguinte configuração:
$(q', w, \alpha\gamma)$
Tal movimento é denotado por:

$$\mathbf{(q, aw, Z\gamma) \vdash (q', w, \alpha\gamma)}$$

Linguagem Aceita por um PDA

A **Linguagem** aceita por um PDA pode ser definida de duas maneiras, conforme o modo como é efetuado o reconhecimento das sentenças:

1 – **Linguagem Aceita por Estado Final** (**T(P)**)

$$T(P) = \{w \mid (q0, w, Z0) \vdash^* (p, \varepsilon, \gamma), \text{ onde } p \in F \wedge \gamma \in \Gamma^*\}$$

2 – **Linguagem Aceita Por Pilha Vazia** (N(P))

$$N(P) = \{w \mid (q0, w, Z0) \vdash^{*} (p, \varepsilon, \varepsilon), \text{ para } \forall\ p \in K\}$$

Observação: Linguagens aceitas por estados finais e por pilha vazia constituem a mesma classe de linguagens, pois para toda linguagem aceita por um PDA P1 por pilha vazia, existe um PDA P2 que reconhece essa linguagem por estado final e vice-versa.

Exemplos:

Representação Gráfica de um PDA

Um PDA também pode ser representado por um grafo de transições, similar aquele utilizado na representação de A.F.. A diferença básica está no rótulo das arestas, os quais deverão indicar além do símbolo de entrada, o símbolo do topo da pilha e a seqüência de símbolos que deverá substituir o símbolo do topo da pilha, isto é, o rótulo deve ser da forma:

$$(a, Z) \rightarrow \gamma, \text{ onde } a \in \Sigma$$
$$Z \in \Gamma$$
$$\gamma \in \Gamma^{*}.$$

Exemplo:

Autômato de Pilha Determinístico

Um PDA $P = (K, \sum, \Gamma, \delta, q0, Z0, F)$ é determinístico se:

1 – Para cada $q \in K$ e $Z \in \Gamma$, sempre que $\delta(q, \varepsilon, Z) \neq \phi$ então
$(q, a, Z) = \phi$ para todo $a \in \sum$.

2 – Para $q \in K$, $Z \in \Gamma$ e $a \in \sum$, existe no máximo uma transição envolvendo $\delta(q, a, Z)$

A condição (1) evita a possibilidade de uma escolha entre um movimento independente da entrada (ε - move) e um movimento envolvendo um símbolo da entrada, enquanto que a condição (2) evita uma escolha múltipla em função do símbolo corrente da entrada.
Exemplos:

IV.8 – Equivalência Entre PDA e G.L.C.

Teorema: "Se L é uma L.L.C., então existe um PDA P | P aceita L".

A equivalência entre PDA e G.L.C. é provada pela demonstração de que:

1 – É sempre possível construir a partir de uma G.L.C. G um PDA P que aceite L(G) por pilha vazia (e portanto, em função da equivalência, aceita também por estado final);

2 – É sempre possível construir uma G.L.C. G a partir de um PDA P de forma que ambos aceitam a mesma linguagem. A prova deste teorema pode ser encontrada na bibliografia indicada ([HOP 79], por exemplo).

Exemplo:

IV.9 – Propriedades e Problemas de Decisão das LLC

Nesta seção apresentaremos as principais propriedades e os problemas de decisão básicos relativos a LLC, com o objetivo de despertar o interesse por tais aspectos e ilustrar algumas questões práticas. O assunto pode ser encontrado de forma mais ampla e profunda na bibliografia indicada.

Propriedades

A classe das LLC é fechada sobre as seguintes operações:

a) União

Se L1 e L2 são LLC, então
L1 $\cup$ L2 também é uma LLC.

b) Concatenação

Se L1 e L2 são LLC, então
L1L2 também é uma LLC.

c) Fechamento

Se L é uma LLC, então L^* e L^+ também são LLC.

d) Intersecção Com Conjunto Regular

Se L1 é LLC e L2 é um CR, então
L1 $\cap$ L2 é uma LLC

Observações: 1 – As LLC não são fechadas sobre as operações de **Intersecção** e **Complemento**.
2 – A principal aplicação das propriedades de LLC é a demonstração de que determinadas linguagens não são LLC.

Problemas de Decisão

1 – **Membership**

Dada uma G.L.C. G = (Vn, Vt, P, S) e um string x, **x $\in$ L(G)?**

Observação: Isto pode ser demonstrado pelo algoritmo do Teorema II.2 (CAP.II) através de uma derivação de no máximo 2*|x|-1 passos.

2 – **Emptiness**

Dada uma G.L.C. G = (Vn, Vt, P, S), **L(G) = φ?**

Observação: Este problema pode ser resolvido pelo algoritmo que determina o conjunto de NT férteis de G, bastando testar se o símbolo inicial de G é fértil.

3 – **Finiteness**

Dada uma G.L.C. G = (Vn, Vt, P, S), **L(G) é Finita?**

Observação: Este problema pode ser resolvido com auxílio dos conceitos de **Símbolos Inúteis** e **Não-Terminais Recursivos**.

4 – **Containment**

Dadas duas G.L.C. G1 e G2, **L(G1) ⊆ L(G2)?**

5 – **Equivalence**

Dadas duas G.L.C. G1 e G2, **L(G1) = L(G2)?**

6 – **Interssecção**

Dadas duas G.L.C. G1 e G2, **L(G1) ∩ L(G2) = φ?**

7 – **Ambigüidade**

Dada uma G.L.C. G = (Vn, Vt, P, S), **G é ambígua?**

Observações: Os problemas 1, 2 e 3 são **decidíveis**, os demais são **indecidíveis.**

IV.10 – Aplicações

As LLC são de grande importância prática, notadamente nas seguintes aplicações:

1 – Definição (especificação) de linguagens de programação;

2 – Na formalização da noção de parsing e conseqüentemente na implementação de parser's;

3 – Esquemas de tradução dirigidos pela sintaxe (Transducer's);

4 – Processamento de string's de modo geral.

Dentre essas aplicações, as duas primeiras são fundamentais para a área de compiladores, pois graças à possibilidade de formalização das LLC (via G.L.C. e PDA) foi possível construir uma teoria geral de parsing que levou a proposição de diversas e eficientes técnicas (algoritmos) de análise sintática. Técnicas estas que, por serem baseadas em aspectos formais, possibilitam a geração automática de parser's (analisadores sintáticos).

Deve-se salientar que G.L.C. constituem um modelo apropriado para especificação de linguagens de programação e quaisquer outras linguagens que não possuam dependência de contexto, enquanto que um PDA é um modelo natural de um parser.

Ainda na construção de compiladores (e no processo de tradução, em geral) G.L.C. e PDA têm sido bastante utilizados por outros dispositivos (formais ou semiformais) empregados na implementação de analisadores semânticos (gramáticas de atributo, por exemplo) e como base para esquemas de tradução dirigidos pela sintaxe.

Capítulo V – Análise Sintática

V.1 – Definições preliminares

Parser (Analisador Sintático) → É um algoritmo que recebendo como entrada uma sentença x, emite como saída:

1 – O **Parse** de x, se x pertence à linguagem; ou
2 – **Erro Sintático**, caso x não pertença à linguagem.

Parsing (Analise Sintática) → É o efeito da execução do **Parser**.

Parse → Seja G = (Vn, Vt, P, S) uma G.L.C. com as produções de P numeradas de 1 a p e uma derivação $S \stackrel{+}{\Rightarrow} x$.

O **Parse** de x em G, é a seqüência formada pelo número das produções utilizadasna derivação $S \stackrel{+}{\Rightarrow} x$. Mais especificamente:

1 – **Parse Ascendente** (Botton-up) → É a seqüência invertida dos números das produções utilizadas na derivação mais a direita de x em G ($S \stackrel{+}{\Rightarrow}_{dir} x$.

2 – **Parse Descedente** (Top-down) → É a seqüência dos números das produções utilizadas na derivação mais a esquerda de x em G ($S \stackrel{+}{\Rightarrow}_{esq} x$.

Exemplos:

Conjunto First

Definição: Seja α uma seqüência qualquer gerada por G. Definimos como sendo **first**(α) o conjunto de símbolos terminais que iniciam α ou seqüências derivadas (direta ou indiretamente) de α.
Observações: Se $\alpha = \varepsilon$ ou $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, então $\varepsilon \in$ **first**(α).

Algoritmo V.1:

Para calcular **first**(X) para todo $X \in Vn \cup Vt$, aplicamos as seguintes regras:

a) Se $X \in Vt$, então **first**(X) = {X};

b) Se $X \in Vn \wedge X \rightarrow a\alpha \in P$, então coloque a em **first**(X); da mesma forma, se $X \rightarrow \varepsilon \in P$, coloque ε em **first**(X);

c) Se $X \rightarrow Y1\ Y2\ ...\ Yk \in P$, então, para todo i | $Y1\ Y2\ ...\ Yi\text{-}1 \in Vn \wedge$ **first**(Yj), para j = i, i-1, contenha ε, adicione **first**(Yi) – {ε} em **first**(X).

Em outras palavras:
1 – Coloque **first**(Y1), exceto ε, em first(X);
2 – Se ε ∈ **first**(Y1) então coloque **first**(Y2), exceto ε em **first**(X);
3 – Se ε ∈ **first**(Y2) então ....
.....
até Yk.
4 – Finalmente, se para todo i (de 1 a k) **first**(Yi) contém ε, então adicione ε em **first**(X).

Conjunto Follow

Definição: Definimos **Follow**(A), para todo A ∈ Vn, como sendo o conjunto de símbolos terminais que podem aparecer imediatamente após A em alguma forma sentencial de G.

Algoritmo V.2:

Para todo A ∈ Vn, aplique as regras abaixo, até que **Follow**(A) esteja completo (isto é, não sofra nenhuma alteração):

1 – Coloque $ (a marca de final de sentença) em **Follow**(S), onde S é o Símbolo Inicial da gramática em questão;

2 – Se $a \rightarrow \alpha B\beta \in P \wedge \beta \neq \varepsilon$, então adicione **First**(β), exceto ε, em **Follow**(B);

3 – Se $A \rightarrow \alpha B$ (ou $A \rightarrow \alpha B\beta$, onde $\varepsilon \in$ **First**(β)) ∈ P, então adicione **Follow**(A) em **Follow**(B).

Exemplos e Exercícios:

V.2 – Classes de Analisadores

Existem duas classes fundamentais de analisadores sintáticos, definidos em função da estratégia utilizada na análise:

Analisadores Descendentes (Top-down)

Consistem em uma tentativa de chegar-se a sentença a partir do símbolo inicial de G, olhando a sentença ou parte dela para decidir que produção deverá ser usada na **derivação**.

Analisadores Ascendentes (Bottom-up)

Consistem em uma tentativa de chegar-se ao símbolo inicial de G a partir da sentença a ser analisada, olhando a sentença ou parte dela para decidir que produção deverá ser usada na **redução**.

V.2.1 – Analisadores Descendentes (Top-Down)

A idéia geral desta classe de analisadores resume-se a uma tentativa de construir a derivação mais à esquerda da sentença de entrada, ou equivalentemente, uma tentativa de construir a A.D. da sentença a partir do símbolo inicial da gramática em questão. Estes analisadores podem ser implementados com (analisadores não-deterministicos) ou sem back-track (analisadores deterrminísticos):

a) A **utilização de back-track** permite que um conjunto maior de G.L.C.(incluindo gramáticas ambíguas) possa ser analisado, entretanto apresenta várias desvantagens (todas decorrentes do **Não-Determinismo**), dentre as quais podemos destacar:

1 – Maior tempo necessário para a análise;

2 – Dificuldade na recuperação de erros;

3 – Problemas na análise semântica e geração de código.

Os analisadores descendentes não-determinísticos (dentre os quais destaca-se a técnica da **Força bruta**), nada mais são do que implementações diretas da idéia geral dessa classe de analisadores, ou seja: consistem em uma tentativa de construção (explicita) da árvore de derivação da sentença em análise, a partir do símbolo inicial da gramática. O uso do mecanismo de back-track faz-se necessário em função da possível presença de não-determinismo neste processo.

b) As **implementações sem back-track** apesar de limitarem a classe de gramáticas que podem ser analisadas – como veremos adiante – tornam-se mais vantajosas pelo fato de serem técnicas determinísticas e superarem as deficiências praticas das implementações com back-track. Assim sendo, enfatizaremos neste curso as técnicas determinísticas (ou seja, as técnicas que não fazem uso do mecanismo de back-track).

Analisadores Descendentes sem Back-Track

Para que um G.L.C. possa ser analisada deterministicamente por estes analisadores, ela deve satisfazer as seguintes condições:

a) Não possuir **Recursão à Esquerda**;

b) Estar **Fatorada**, isto é, se $A \rightarrow \alpha1 \mid \alpha2 \mid ... \mid \alpha n$ são as A-produções de uma determinada G.L.C. então
$$\textbf{First}(\alpha1) \cap ... \cap \textbf{First}(\alpha n) = \varphi;$$

c) Para todo $A \in Vn \mid A \overset{*}{\Rightarrow} \varepsilon$, $\textbf{First}(A) \cap \textbf{Follow}(A) = \varphi$.

Técnicas de Implementação

Descendente Recursivo: Esta técnica consiste basicamente 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 de análise é maior (se comparado com a técnica que veremos a seguir) e existe a necessidade de uma linguagem que permita recursividade para sua implementação.

Por outro lado, a principal vantagem desta técnica, é a simplicidade de implementação e a facilidade para inserir as diferentes funções do processo de compilação, diretamente nos procedimentos que realizam a análise sintática de cada não-terminal da gramática; contudo esta vantagem verifica-se apenas para gramáticas/linguagens pequenas e simples.

Outro inconveniente desta técnica é a dificuldade para validação do analisador construído, uma vez que os procedimentos não estão livres de erros de programação e ou erros decorrentes da não adequação da gramática para esta classe de analisadores (por exemplo, a presença de ambigüidade ou recursões à esquerda indiretas, não percebidas pelo projetista).

Exemplo:

O Parser **Descendente Recursivo** da Gramática

G: E → TE'
E' → + TE' | ε
T → (E) | id

seria composto pelos seguintes procedimentos:

```
(* Programa Principal *)
begin
      scanner (simb);
      E (simb);
end;
```

```
procedure E (simb);
begin
        T (simb);
        Elinha (simb);
end;
```

```
procedure Elinha (simb);
begin
        if simb = "+"
        then begin
                scanner (simb);
                T (simb);
                Elinha (simb);
             end;
end;
```

```
procedure T (simb);
begin

   ...

end;
```

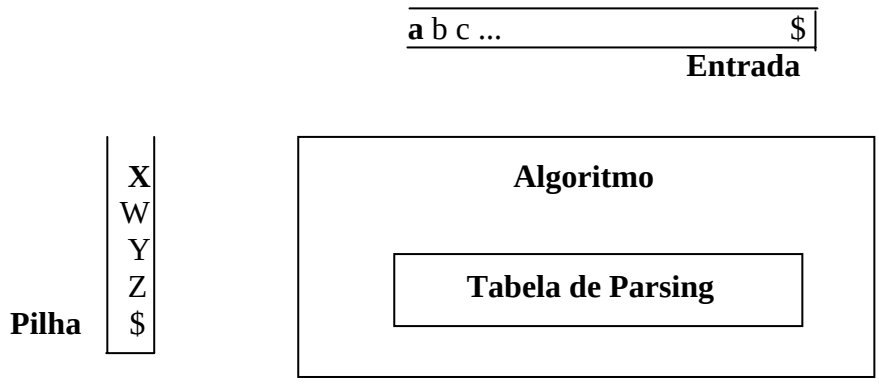
PARSER PREDITIVO (LL)

É uma maneira eficiente de implementar um parser descendente recursivo (por isso, é também denominado parser descendente recursivo tabular).

Um **parser preditivo** consiste de:

1 – **Entrada** → contendo a sentença (programa) a ser analisada.
2 – **Pilha** → usada para simular a recursividade, ela prevê a parte da entrada que está para ser analisada (é inicializada com $ e o símbolo inicial da gramática em questão).
3 – **Tabela de Parsing** (ou tabela de análise sintática) → Contém as ações a serem efetuadas. É uma matriz M(A, a), onde $A \in Vn \wedge a \in Vt$.
4 – **Algoritmo de análise sintática**.

Estrutura de um Parser Preditivo



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

a) Se X = a = $ → o analisador anuncia o final da análise.

b) Se X = a ≠ $ → o analisador retira **X** do topo da pilha e **a** da entrada. Esta ação corresponde ao reconhecimento sintático de **a.**

c) Se X ∈ Vt e X ≠ a → situação de erro (* 'X' era o símbolo esperado *). Neste caso, devem ser ativados os procedimentos de recuperação de erro (caso existam) ou a análise sintática deve ser encerrada, diagnosticando o erro encontrado.

d) Se X ∈ Vn → 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, digamos. 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 diagnosticar o erro encontrado e ativar os procedimentos de recuperação de erros (em implementações sem recuperação de erros a análise é encerrada).

Observações: 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, prevê a parte da sentença que deve estar na entrada para que a sentença esteja correta.

Algoritmo de Análise Sintática – Parser Preditivo

```
repita
(* X – topo da pilha *)
(* a – próximo símbolo da entrada *)

se x é terminal ou $
então se X = a
      então retira X do topo da pilha
            retira a da entrada
      senão erro( )
senão (* X é não terminal *)
      se M(X, a) = X → Y1 Y2 ... Yk
      então retira X da pilha
            coloca Yk Yk-1 ... Y2 Y1 na pilha
            (* deixando Y1 sempre no topo *)
      senão erro( )
até X = $ (* pilha vazia, análise concluída *)
```

Construção da Tabela de Parsing do Parser Preditivo

Idéia Geral – se A → α ∈ P ∧ a ∈ **First**(α), então, se A está no topo da pilha e a é o próximo símbolo da entrada, devemos expandir (derivar) A, usando a produção A → α. Mas, e se α = ε ou $\alpha \stackrel{*}{\Rightarrow} \varepsilon$!? – note que ε nunca aparecerá na entrada – neste caso, se a ∈ **Follow**(A) expandimos (derivamos) A através da produção A → α.

Algoritmo para Construção da Tabela de Parsing

1.Para cada produção $A \rightarrow \alpha \in P$, execute os passos 2 e 3.

2.Para todo $a \in$ **First**(α), exceto ε,
coloque o número da produção $A \rightarrow \alpha$ em M(A, a).

3.Se $\varepsilon \in$ **First**(α), coloque o número da produção $A \rightarrow \alpha$ em M(A, b), para todo $b \in$ **Follow**(A).

4.As posições de M que ficarem indefinidas, representarão as situações de erro.

Considerações Gerais

A tabela de parsing dos parser´s preditivos deve possuir a propriedade de que, em 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:

1 – Não possuir recursão à esquerda;

2 – Estar fatorada;

3 – Para todo $A \in Vn \mid A \overset{*}{\Rightarrow} \varepsilon$, **First**(A) $\cap$ **Follow**(A) = φ

As G.L.C. que satisfazem estas condições são denominadas G.L.C. **LL(K)** – isto é G.L.C. que podem ser analisadas deterministicamente da esquerda para a direita (**L**eft-to-right) e o analisador construirá uma derivação mais à esquerda (**L**eftmost derivation), sendo necessário a cada passo o conhecimento de **K** símbolos de lookahead (símbolos de entrada que podem ser vistos para que uma ação seja determinada).

Somente G.L.C. 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 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 **G.L.C. LL(1).**

V.2.2 – Analisadores Ascendentes

A formulação dos algoritmos de análise sintática ascendente baseia-se em um algoritmo primitivo denominado **Algoritmo Geral Shift-Reduce** (**Avança-Reduz**). Este algoritmo utiliza:

a) Uma **Pilha Sintática** (inicialmente vazia);

b) Um **Buffer** de **Entrada** (contendo a sentença a ser analisada);

c) Uma **G.L.C.** (com as produções numeradas de 1 a p);

d) Um procedimento de análise sintática, o qual consiste em **transferir** ("Shiftar") os símbolos da entrada (um a um) para a pilha até que o conteúdo da pilha (ou parte dele) coincida com o lado direito de uma produção. Quando isto ocorrer, o lado direito da produção

deve ser **substituído** pelo (reduzido ao) lado esquerdo da produção em questão. Este processo deve ser repetido ate que toda a sentença de entrada tenha sido analisada.

Observações: 1 – As reduções são prioritárias em relação ao shifts.
2 – Se ao final do processo a **Entrada** estiver vazia e a pilha sintática contiver apenas o símbolo inicial de G, então a sentença analisada estará sintaticamente correta.

Deficiências do Algoritmo Shift-Reduce

Embora esta técnica de análise sintática possa ser aplicada a qualquer GLC (esta é sua grande vantagem), ela apresenta várias deficiências que inviabilizam seu uso na prática. São elas:

1 – Requer muito tempo para analise;

2 – Só detecta erro sintático após consumir toda a sentença a ser analisada e, além disso, não identifica o ponto onde ocorreu o erro.

3 – Pode rejeitar sentenças corretas – pelo fato de que nem sempre que o lado direito de uma produção aparece na pilha a ação correta é uma redução, fato este que caracteriza o **Não-Determinismo** do método.

O problema do não-determinismo pode ser contornado através do uso da técnica de **Back-Tracking**; contudo, o uso desta técnica inviabiliza o método na pratica (em função da complexidade de tempo e espaço).

Na prática, as técnicas de análise sintática ascendente usuais superam as deficiências da técnica **Shift-Reduce com Back-Tracking** (apesar de fundamentarem-se nela), apresentando as seguintes características:

1 – Tempo de análise diretamente proporcional ao tamanho da sentença;

2 – Detecção de erros sintáticos no momento da ocorrência;

3 – São técnicas **Determinísticas**, isto é, em qualquer situação, haverá sempre uma única ação a ser efetuada.

Contudo, estas características tem um custo associado: a imposição de restrições às GLC, para que as mesmas possam ser analisadas deterministicamente. Uma restrição comum à todas as técnicas deterministicas, é a exigência de que a GLC não seja ambígua.

Principais Técnicas de Análise Sintática Ascendente

A classe de analisadores ascendentes determinísticos é composta por uma série de técnicas, dentres as quais destacam-se grandemente duas famílias de técnicas:

1 – **Analisadores de Precedência** (Simples, estendida e de operadores) - Estes analisadores baseiam-se no algoritmo **Shift-Reduce,** acrescido de relações de precedência entre os símbolos da gramática; relações estas que definem, de forma determinística a ação a ser efetuada em uma dada situação.

2 - **Analisadores LR** - Esta família de analisadores também baseia-se nas operações shift e reduce (é, na verdade, uma evolução natural do algoritmo geral shift-reduce). A diferença fundamental é que estas operações (shift e reduce) são realizados sempre deterministicamente, com base no estado corrente da análise e nas propriedades estruturais da gramática em questão.

Além de teoricamente importante, a família LR (especialmente as técnicas SLR(1) e LALR(1)) é a mais utilizada na implementação de analisadores sintáticos, e por isso será estudada de forma mais detalhada neste curso.

Analisadores LR

Introdução – A família dos analisadores LR é formada por uma série de técnicas onde as principais são: LR(0), SLR(1), LALR(1) e LR(1), em ordem crescente no sentido de força (abrangência de G.L.C.) e complexidade de implementação.

Os analisadores **LR** são assim denominados pelo fato de analisarem a sentença de entrada da esquerda para a direita (**L**eft-to-right) e construírem uma derivação mais à direita (**R**ightmost derivation) na ordem inversa.

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 independente da gramática) e uma Tabela de Parsing (construída de forma especifica 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:

1 – Analisam praticamente todas as construções sintáticas de linguagem de programação que podem ser representadas de forma não ambígua por G.L.C..

2 – São mais gerais que os outros analisadores ascendentes e que a maioria dos descendentes sem back-track.

3 – Possuem a propriedade de detecção imediata de erros sintáticos.

4 – O tempo de análise é proporcional ao tamanho da sentença a ser analisada.

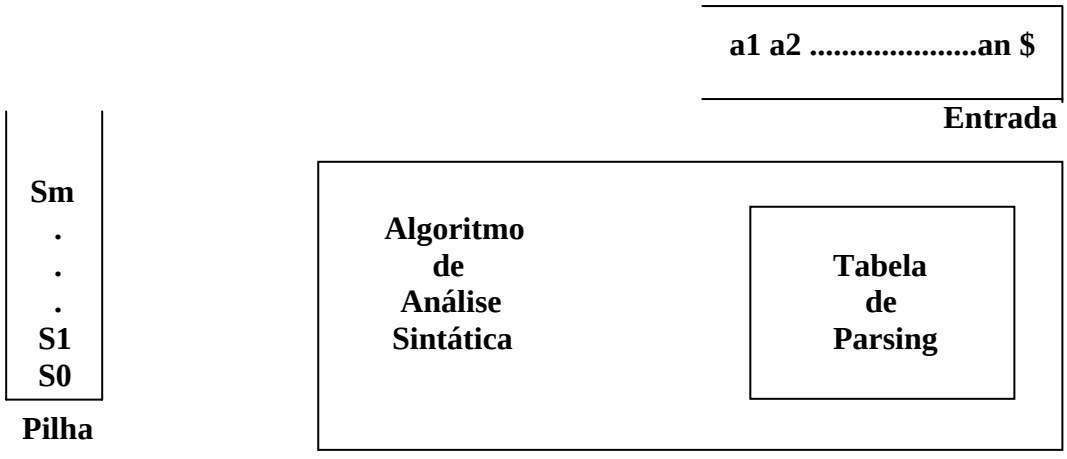
Em contrapartida, podemos destacar como ponto negativo dos analisadores LR, a complexidade de construção da tabela de parsing bem como o espaço requerido para seu armazenamento (especialmente as técnicas mais abrangentes).

Composição dos Analisadores LR

Um analisador LR compõe-se se:

a) **Algoritmo** d**e Análise Sintática** → padrão para todas as técnicas da família.
b) **Tabela** d**e Análise Sintática** (ou tabela de parsing) → específica para cada técnica e para cada gramática.
d) **Pilha de Estados** (ou pilha sintática) → conterá um histórico da análise efetuada; é inicializada com o estado inicial da análise sintática.
e) **Entrada** → conterá a sentença a ser analisada, seguida por $ (a marca de final de sentença).
f) **Uma G.L.C.** → com as produções numeradas de 1 a p.

Estrutura Geral de um Analisador LR



Algoritmo da Análise Sintática LR

Consiste em determinar S_m (o estado do topo da pilha) e a_i (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 poderá ser:

1 – **Shift S**
2 – **Reduce R**
3 – **Erro**
4 – **Halt** (ou **Accept**)

Significado e Procedimentos Associados às Ações

a) **Halt**: → Fim de análise;
→ **Procedimento** – A análise sintática deve ser encerrada.
b) **Erro**: → Indica a ocorrência de um erro sintático.
→ **Procedimento** – 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) **Shift S** → Significa o reconhecimento sintático do símbolo da entrada.
→ **Procedimento** – O símbolo da entrada deve ser retirado e o estado S, indicado na tabela de Parsing, deverá ser empilhado.
d) **Reduce R** → Significa que uma redução pela produção número R deve ser efetuada.
→ **Procedimento** – Deverão ser retirados pa pilha sintática tantos estados quantos forem os símbolos do lado direito da produção R, e o símbolo do lado esquerdo dessa produção deverá ser tratado como um símbolo de entrada na próxima ação do analisador.

Observações: O processo de determinar e efetuar as ações sintáticas deverá ser repetido até que a ação **Halt** seja encontrada ou, para as implementações sem recuperação de erros, um erro seja detectado.

Tabela de Parsing LR

A tabela de parsing dos analisadores LR é dividida em duas partes:
a) Tabela **Action** → contém as transições (**Shift**, **Reduce**, **Erro** e **Halt**) com os símbolos terminais.
b) Tabela **Goto** → contém as transições (**Goto** e **Erro**) com os símbolos não-terminais (Um **Goto** é um **Shift** especial sobre um símbolo não-terminal).

Implementação: Normalmente utiliza-se **Matriz** ou **Lista**. No caso de matriz – como normalmente as tabelas de parsing são bastante esparsas – utilizam-se técnicas de compactação de tabelas.

Configuração de um Analisador LR

É um par, cujo 1º elemento é o conteúdo da pilha sintática e o segundo é a entrada a ser analisada. Por exemplo, a configuração:

$$(S0_{X1}\ S1_{X2}\ S2\ \ldots\ S_m,\ a_i\ a_{i+1}\ a_{i+2}\ \ldots\ a_n\$)$$

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).

Observações: X1, X2, ..., XM só existem logicamente.

Gramáticas LR: Uma G.L.C. é LR se cada sentença que ela gera pode ser analisada deterministicamente da esquerda para a direita por um analisador LR.

Pertencem a classe das gramáticas LR, todas as G.L.C. para as quais é possível construir a tabela de parsing LR sem que hajam conflitos (isto é, em cada posição da tabela haverá no máximo uma ação registrada).

Gramáticas LR(K): Idem a definição anterior, considerando-se K símbolos de **Lookahead**. Na pratica usa-se K = 1.

Observações: G.L.C. analisáveis por analisadores SLR(1), LALR(1) e LR(1) são também denominadas, respectivamente, gramáticas SLR(1), LALR(1) e LR(1).

Exemplo: Para a G.L.C.:

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	S5	S4					1	2	3
1				S6		HALT			
2			R2	R2	S7	R2			
3			R4	R4	R4	R4			
4	S5	S4					8	2	3
5			R6	R6	R6	R6			
6	S5	S4						9	3
7	S5	S4							10
8			S11	S6					
9			R1	R1	S7	R1			
10			R3	R3	R3	R3			
11			R5	R5	R5	R5			
	ACTION						GOTO		

Exercício: Efetuar a análise sintática das sentenças:

x = id * id

y = (id +) id * id

Construção da Tabela de Parsing SLR(1)

Definicões Gerais:

a) **Item LR(0):** Um item LR(0) (ou simplesmente item) em uma **Gramática** G é uma produção de G com uma marca (fisicamente representada por um ponto) em uma posição de seu lado direito. Denota-se um item por um par de inteiros [i, j], onde i é o no da produção e j a posição em que se encontra o ponto.

Exemplo:

Observação: Um item [i, j] tem o seguinte significado: As informações disponíveis indicam que a produção i está sendo utilizada e que os primeiros j símbolos de seu lado direito já foram analisados.

b) Item Completo: Item em que a "marca" está após o último símbolo do lado direito da produção.

c) Estado: É uma coleção de informações sobre o progresso (situação) da análise sintática em um determinado momento. É composto por um conjunto de itens.

Observações: A coleção de estados criada para uma G.L.C. (denominada **Coleção** LR(0)) é a base para a construção da tabela de parsing SLR(1).

d) Núcleo de um Estado: Conjunto de itens que deu origem ao estado.

Algoritmo para Construção da Coleção Lr(0):

a) Inicialização

1 – Numere as produções de G de 1 a p;

2 – Aumente G com a produção S' → S$; S' passará a ser o símbolo inicial de G e $ será usado como marca de final de sentença)

3 – Defina o núcleo do estado inicial (estado 0), como sendo o item [0, 0], onde a produção no 0 é a produção incluída.

b) **Construção dos Estados**

1 – Faça o **Fechamento** do núcleo;

2 – Faça o **Fechamento** dos itens criados em a;

3 – Determine o núcleo dos **Estados Sucessores**;

4 – Repita a, b e c para todos os núcleos criados, até que nenhum **Estado Sucessor** novo seja criado e que todos os estados estejam completos (isto é, todos os itens fechados).

Fechamento → Se um determinado item contém a "marca" na frente de um não-terminal A, o fechamento deste item será o próprio item + o conjunto de itens [i, o] existentes, onde i é o número de uma produção cujo lado esquerdo é o não-terminal A. Se a "marca" precede um símbolo terminal ou se o item é completo, o fechamento será o próprio item.

Estados Sucessores → São os estados alcançados através da movimentação da "marca" sobre um determinado símbolo. O núcleo do estado sucessor será formado pelo conjunto de itens resultantes da movimentação acima referida. Observação: Não existe estado sucessor relativo ao item [0, 1] (≡ S' → S, $), uma vez que este item indica o final da análise.

Algoritmo para Construção da Tabela de Parsing SLR(1)

a) Crie uma linha para cada estado e uma coluna para cada símbolo $\in \{Vn \cup Vt \cup \{\$\}\}$.

b) Coloque SHIFT (ou GOTO) nos estados que deram origem a estados sucessores (nas colunas referentes aos símbolos sobre os quais houve a movimentação da marca).

c) Para cada estado i com itens completos, faça:

Para cada item completo, faça:

Coloque REDUCE N na intersecção da linha correspondente ao estado i com as colunas correspondentes a símbolos $\in$ follow(A); onde N é o número da produção envolvida no item completo em questão e A é o símbolo do lado esquerdo desta produção.

d) Coloque HALT na intersecção da linha referente ao estado que contenha o item [0, 1], na coluna do símbolo $.

e) As posições que ficarem vazias após a execução dos passos b, c e d, representarão as situações de erro.

Condição SLR(1): A condição para que a tabela construída pelo algoritmo acima seja SLR(1), é que não existam estados **Inadequados** (estados com conflitos **Shift**-**Reduce** e/ou **Reduce**-**Reduce**); isto equivale a dizer que cada posição da tabela deverá conter no máximo uma ação.

Propriedade das Tabelas LR

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) **Condição De Erro**: Nunca será encontrada para sentenças sintaticamente corretas.
b) Cada **Shift** deverá especificar um único estado, o qual dependerá do símbolo da entrada.
c) Cada **Reduce** 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, isto é, um *handle* da gramática.
d) **Halt** só será encontrado quando a análise estiver completa.

Exercícios: Construa a tabela de **Parsing SLR**(1) para as seguintes G.L.C.:

1 – E → E + T | T
T → T * F | F
F → (E) | id

2 – C → if E then C else C
| if E then C
| com
E → exp

3 – B → AB'
B' → or AB' | ξ
A → CA'
A' → and CA' | ξ
C → exp | (B) | not C