UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

TRANSFORMAÇÃO DE GRAMÁTICAS LIVRES DO CONTEXTO PARA EXPRESSÕES REGULARES ESTENDIDAS

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

CLEISON VANDER AMBROSI

BLUMENAU, JUNHO/2000

2000/1-20

TRANSFORMAÇÃO DE GRAMÁTICAS LIVRES DO CONTEXTO PARA EXPRESSÕES REGULARES ESTENDIDAS

CLEISON VANDER AMBROSI

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. José Roque Voltolini da Silva — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. José Roque Voltolini da Silva

_____

Prof. Everaldo Artur Grahl

_____

Prof. Roberto Heinzle

DEDICATÓRIA

À meus pais Anísio e Agnes, por sempre indicar o melhor caminho à seguir nesta vida.

AGRADECIMENTOS

Agradeço à Deus por me dar saúde e condições de cursar esta faculdade.

À minha família pelo apoio e incentivo oferecidos no decorrer desta jornada, tornando possível a conclusão deste curso.

Especialmente ao professor e orientador deste trabalho, José Roque Voltolini da Silva, pela dedicação, paciência e atenção dispensada na elaboração deste trabalho.

À minha namorada Leila, pelo auxílio, incentivo e compreensão, os quais foram de fundamental importância para atingir este objetivo.

À todos os amigos da universidade que auxiliaram no desenvolvimento deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

RESUMO

Este trabalho de conclusão de curso visa o estudo e implementação de um protótipo para transformação de definições feitas em linguagens livres do contexto em expressões regulares estendidas, segundo proposto por José Roque Voltolini da Silva. A recursividade encontrada nas produções de uma gramática livre do contexto será eliminada durante o processo de transformação para uma expressão regular estendida. Uma vez feita a transformação, o teorema de Kleene será utilizado para retirar o indeterminismo. Ainda, também será utilizado o algoritmo de Silva para transformar uma expressão regular em um autômato finito determinístico.

ABSTRACT

This work of course conclusion aims at the study and implementation of a prototype for transformation of definitions made in context free languages in extended regular expressions second considered for José Roque Voltolini da Silva. The recursively found in the productions of a context free grammar will be eliminated during the transformation process an extended regular expression. When made the transformation, the theorem of Kleene will be used to remove the indeterminism. Still, also will be used the Silva algorithm to transform a regular expression into a deterministic finite automaton.

1 INTRODUÇÃO

Este trabalho visa o estudo da representação sintática das linguagens, em especial as linguagens livres do contexto, que são usadas principalmente para o desenvolvimento de analisadores sintáticos de linguagens de programação.

Uma linguagem de programação é definida por [JOS1987] como um conjunto de todos os textos que podem ser gerados a partir de uma gramática. Ela difere da linguagem natural por ser simples e direta, pois é através desta que uma máquina pode ser instruída. Por outro lado, linguagens de programação possuem propriedades em comum com as linguagens naturais. Toda linguagem possui uma sintaxe e semântica.

A sintaxe é definida como a formação de frases em uma linguagem de programação. Cada linguagem possui um vocabulário de símbolos e regras que são agrupados para formar expressões, declarações, comandos, e em última instância formar programas. Cada linguagem tem suas próprias regras para compor as frases, podendo ter comandos equivalentes e diferir sintaticamente ([WAT1991]). Segundo [LEW2000], a sintaxe manipula esses símbolos sem considerar os seus significados correspondentes, portanto não existe uma noção de programa “certo” ou “errado”.

Os símbolos de uma linguagem de programação são identificadores, literais, símbolos de operadores. Cada linguagem tem suas próprias regras por formar frases, portanto podem possuir comandos equivalentes que diferem sintaticamente ([JOS1987]).

A semântica é definida como o significado das frases de uma linguagem. Uma interpretação semântica deve ser dada aos símbolos para resolver o problema na realidade propriamente dita. Ao contrário da sintaxe, que é facilmente formalizável, a semântica exige notações muito mais complexas, de aprendizagem mais difícil e, em geral, apresentando simbologias bastante carregadas e pouco legíveis ([JOS1987] [WAT1991]).

As linguagens são classificadas em diversas classes, em uma ordem hierárquica, denominada hierarquia de Chomsky, resultando nas seguintes classes básicas de linguagens ([MEN1998]):

a) linguagens regulares ou tipo 3;
b) livres do contexto ou tipo 2;
c) sensíveis ao contexto ou tipo 1;
d) enumeráveis recursivamente ou tipo 0;

Na especificação das linguagens livres de contexto, uma série de verificações e transformações necessitam ser feitas, para posterior implementação (em computação). Estas verificações e transformações consistem na retirada da recursividade, na retirada do indeterminismo (tratado no jargão de “compiladores” como fatoração).

Métodos para tornar uma linguagem livre do contexto implementável podem ser vistos em [AHO1995].

Este trabalho aplica técnicas apresentadas em [WAT1991] e [SIL2000b] para transformação de definições em linguagens livres do contexto em expressões regulares estendidas. Para esta transformação, serão utilizadas algumas técnicas apresentadas em [WAT1991] para linguagens livres do contexto ‘Não Auto-Embutidas’ e [SIL2000b] para linguagens livres do contexto ‘Auto-Embutidas’. Será também aplicado o teorema de Kleene, definido para linguagens regulares nas expressões regulares estendidas.

Conforme [MEN1998], uma linguagem é dita linguagem livre do contexto se for gerada por uma gramática livre do contexto, onde esta é uma gramática onde o lado esquerdo das produções contém exatamente uma variável. Segundo [AHO1995], uma gramática livre do contexto possui quatro componentes:

a) um conjunto de *tokens*, conhecidos como símbolos terminais;
b) um conjunto de não terminais;
c) um conjunto de produções, onde uma produção consiste em um não-terminal, chamado de lado esquerdo da produção, uma seta e uma seqüência de *tokens* e/ou não-terminais, chamados de lado direito da produção;
d) uma designação a um dos não-terminais como o símbolo de partida.

Um exemplo de gramática pode ser vista no quadro 1.1.

QUADRO 1.1 – DEFINIÇÃO DE UMA GRAMÁTICA

G = (V, T, P, S) onde: V: conjunto finito de símbolos não-terminais T: conjunto finito de terminais P: conjunto finito de pares, denominados regras de produção S: elemento de V denominado variável inicial

Fonte: [MEN1998]

Ainda, este trabalho propõe-se a implementar um protótipo que automatize o processo de eliminação do indeterminismo em linguagens livres do contexto, usando o mesmo teorema de Kleene para eliminação do indeterminismo em linguagens regulares através da transformação de linguagens livres do contexto em expressões regulares estendidas. Esse teorema diz que toda linguagem regular com indeterminismo é possível transformar em uma linguagem determinística.

1.1 OBJETIVO

O presente trabalho tem como objetivo principal a transformação de linguagens livres do contexto em uma expressão regular estendida e, ainda utilizar as propriedades e teoremas das linguagens regulares (que podem ser representadas por expressões regulares) na nova expressão regular estendida proposta.

1.2 ORGANIZAÇÃO DO TEXTO

O primeiro capítulo apresenta uma introdução do trabalho contendo alguns conceitos fundamentais na área de linguagens de programação e computabilidade. Além disto, são apresentados os objetivos e a organização do texto.

O segundo capítulo dá uma visão geral sobre linguagens formais, abrangendo as quatro classes da hierarquia de Chomsky, com ênfase em Linguagens Livres do Contexto (Tipo 2) e Linguagens Regulares (Tipo 3). Este capítulo também aborda os conceitos básicos de um Compilador e seus principais módulos: o analisador léxico, sintático e semântico.

O terceiro capítulo mostra a relação entre das expressões regulares com as gramáticas livres do contexto, dando ênfase nos tratamentos de recursividade propostos por [SIL2000b].

O quarto capítulo apresenta o método para transformação de uma expressão regular em autômato finito determinístico através do teorema de Kleene conforme [MAN1974] e um novo método proposto por [SIL2000a] chamado de Algoritmo do Desmonte.

O quinto capítulo apresenta o desenvolvimento do protótipo, sua especificação, suas telas, características e comenta sobre o ambiente de desenvolvimento Borland Delphi 5.0.

O sexto capítulo relata as conclusões finais obtidas e possíveis extensões para o mesmo.

2 LINGUAGENS FORMAIS

A teoria das linguagens Formais foi originalmente desenvolvida na década de 1950 com o objetivo de desenvolver teorias relacionadas com as linguagens naturais. Entretanto, logo foi verificado que esta teoria era importante para o estudo de linguagens artificiais e, em especial, para as linguagens originárias da Ciência da Computação. O estudo das linguagens formais desenvolveu-se significativamente e com diversos enfoques, com destaque para aplicações em análise léxica e sintática de linguagens de programação, modelos de sistemas biológicos, desenho de hardware e relacionamentos com linguagens naturais ([MEN1998]).

Uma linguagem de programação (bem como qualquer modelo matemático) pode ser vista de duas formas ([MEN1998]):

a) como uma entidade livre, ou seja, sem qualquer significado associado;
b) como uma entidade juntamente com uma interpretação do seu significado.

De acordo com [DIV1999], uma linguagem formal ou simplesmente linguagem é um conjunto de palavras sobre um alfabeto, então supondo o alfabeto $\sum = \{ a, b \}$:

a) o conjunto vazio e o conjunto formado pela palavra vazia são linguagens sobre $\sum$;
b) o conjunto de palíndromos (palavras que tem a mesma leitura da esquerda para a direita e vice-versa) sobre $\sum$ é um exemplo de linguagem infinita, assim ε, a, b, aa, bb, aaa, aba, bab, bbb, aaaa, são palavras desta linguagem.

As linguagens formais preocupam-se com os problemas sintáticos das linguagens. A teoria da sintaxe possui construções matemáticas bem definidas, como por exemplo a Hierarquia de Chomsky ([MEN1998]).

2.1 HIERARQUIA DE CHOMSKY

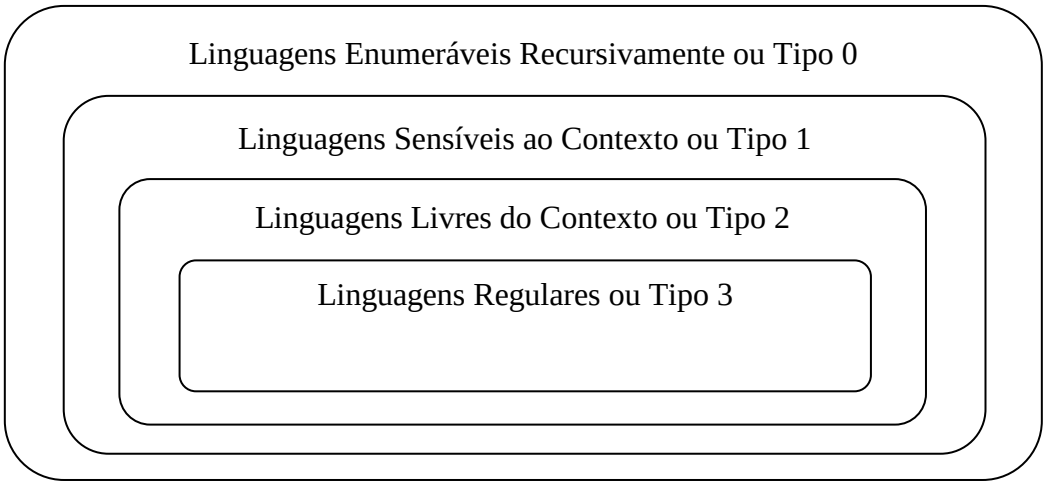
Em 1960 o lingüista Noam Chomsky propôs uma influente estrutura de pensamento para considerar a expressividade das linguagens de programação e o poder dos autômatos, sendo chamada de hierarquia de Chomsky ([LEW2000]).

Essa estrutura é composta por uma divisão do universo das linguagens em quatro classes, onde cada uma é contida pela classe imediatamente anterior ([MEN1998]), as quais são:

a) Linguagens Regulares ou Tipo 3;
b) Linguagens Livres do Contexto ou Tipo 2;
c) Linguagens Recursivas ou Tipo 1;
d) Linguagens Enumeráveis Recursivamente ou Tipo 0.

A figura 2.1 mostra a relação de correspondência entre as classes de linguagens.

FIGURA 2.1 – HIERARQUIA DE CHOMSKY



Fonte: [MEN1998]

2.1.1 LINGUAGENS ENUMERÁVEIS RECURSIVAMENTE OU TIPO 0

As Linguagens Enumeráveis Recursivamente ou Tipo 0 são aquelas que podem ser reconhecidas por uma Máquina de Turing. Considerando que, segundo a hipótese de Church, a Máquina de Turing é o mais geral dispositivo de computação, então a classe das linguagens Enumeráveis Recursivamente representa o conjunto de todas as linguagens que podem ser reconhecidas mecanicamente e em um tempo finito ([MEN1998]).

Analogamente às demais classes de linguagens, é possível representar as Linguagens Enumeráveis Recursivamente usando um formalismo axiomático ou gerador, na forma de

gramática, denominado Gramática Irrestrita. Como o próprio nome indica, uma Gramática Irrestrita não possui qualquer restrição sobre a forma das produções.

[DIV1999] cita como exemplos de linguagens enumeráveis recursivamente:

a) $\{ a^n b^n \mid n >= 0\}$

b) $\{ a^n b^n c^n \mid n >= 0\}$

c) $\{ a^i b^j c^k \mid i = j$ ou $j = k\}$

A classe das Linguagens Enumeráveis Recursivamente, inclui algumas palavras para as quais é impossível determinar mecanicamente se uma palavra *não* pertence à linguagem. Suponha-se que: L é uma linguagem, então para qualquer máquina M que aceita L, existe pelo menos uma palavra W que não pertencente a L que, ao ser processada por M, a máquina entra em *loop* infinito, processando indefinidamente ([DIV1999] [LEW2000]).

Portanto, em [DIV1999], define-se uma subclasse da Classe das Linguagens Enumeráveis, denominada Classe das Linguagens Recursivas, composta pelas linguagens para quais existe pelo menos uma Máquina de Turing que pára para qualquer entrada, aceitando ou rejeitando.

2.1.1.1 LINGUAGENS RECURSIVAS

Uma linguagem L é dita *Linguagem Recursiva* se existe uma Máquina de Turing M tal que :

a) ACEITA(M) = L

b) REJEITA(M) = Σ^* - L

c) LOOP(M) = ϕ

Portanto, a classe das linguagens recursivas define a classe de todas as linguagens que podem ser reconhecidas mecanicamente e para as quais sempre existe um reconhecedor que sempre pára, para qualquer entrada, reconhecendo ou rejeitando, destacando que a grande maioria da linguagens aplicadas são recursivas ([DIV1999]).

A seguir estão relacionadas algumas das principais propriedades das linguagens enumeráveis recursivamente e das linguagens recursivas, conforme [DIV1999]:

a) o complemento de uma linguagem recursiva, é uma linguagem recursiva. Consequentemente, existe um algoritmo que sempre para e que reconhece o complemento da linguagem;
b) uma linguagem L é recursiva se, e somente se, L e seu complemento são enumeráveis recursivamente;
c) a classe das linguagens recursivas está contida propriamente na classe das linguagens enumeráveis recursivamente.

2.1.2 LINGUAGENS SENSÍVEIS AO CONTEXTO OU TIPO 1

As Linguagens Sensíveis ao Contexto ou tipo 1, são definidas a partir das Gramáticas Sensíveis ao Contexto. O termo “sensível ao contexto” deriva do fato de que o lado esquerdo das produções da gramática pode ser uma palavra de variáveis ou terminais, definindo um “contexto” de derivação ([MEN1998]).

Uma gramática sensível ao contexto G é uma gramática G= (V,T,P,S) com a restrição de que qualquer regra de produção de P é de forma $\alpha \rightarrow \beta$, onde:

a) α é uma palavra de $(V \cup T)^{+}$;
b) β é uma palavra de $(V \cup T)^{*}$;
c) $|\alpha| \leq |\beta|$, executando-se, eventualmente, para $S \rightarrow \in$. Neste caso S não pode estar presente no lado direito de cada produção.

A linguagem L={ ww | w é palavra de {a, b}*} pode ser gerada por uma gramática sensível do contexto como descrita no quadro 2.1.

QUADRO 2.1 – GRAMÁTICA SENSÍVEL AO CONTEXTO

```
G = ({S, X, Y, A, B, <aa>, <ab>, <ba>, <bb>}, {a, b}, P, S),
onde:

P = { S → XY | aa | bb | ∈,
   X → XaA | XbB | aa<aa> | ab<ab> | ba<ba> | bb <bb>,
   Aa → aA, Ab → bA, AY → Ya,
   Ba → aB, Bb → bB, BY → Yb,
   <aa>a → a<aa>, <aa>b → b<aa>, <aa>Y → aa,
   <ab>a → a<ab>, <ab>b → b<ab>, <ab>Y → ab,
   <ba>a → a<ba>, <ba>b → b<ba>, <ba>Y → ba,
   <bb>a → a<bb>, <bb>b → b<bb>, <bb>Y → bb }
```

FONTE: [MEN1998]

Executando-se para | w | $\leq$ 1, a gramática apresentada gera o primeiro W após X e o segundo W após Y, como segue:

a) a cada símbolo terminal gerado após X, é gerada uma variável correspondente;
b) esta variável "caminha" na palavra até passar por Y, quando deriva o correspondente terminal;
c) X deriva uma subpalavra de dois terminais e uma correspondente variável a qual "caminha" até encontrar Y, quando é derivada a mesma subpalavra de dois terminais.

2.1.3 LINGUAGENS LIVRES DO CONTEXTO OU TIPO 2

Uma linguagem é dita Linguagem Livre do Contexto ou tipo 2, se for gerada por uma Gramática Livre do Contexto (descrita na seção 2.3). O nome "*Livre do Contexto"* se deve ao fato de representar a mais geral classe de linguagens cuja produção é da forma $A \rightarrow \alpha$. Ou seja, em uma derivação a variável A deriva α sem depender ("livre") de qualquer análise dos símbolos que antecedem ou sucedem A ("contexto") na palavra que está sendo derivada ([MEN1998]).

As Linguagens Livres do Contexto e as correspondentes noções de Gramática Livre do Contexto são usadas principalmente para o desenvolvimento de Analisadores Sintáticos, uma importante parte de um compilador ([MEN1998]).

Uma aplicação de linguagens livres do contexto é na área de compiladores. Uma visão de um compilador pode ser vista na seção 2.2.

2.1.4 LINGUAGENS REGULARES OU TIPO 3

É o conjunto de todas as linguagens reconhecíveis através de autômatos finitos ([JOS1987]). Trata-se de uma classe de linguagens mais simples, sendo possível desenvolver algoritmos de reconhecimento ou de geração de pouca complexidade, grande eficiência e de fácil implementação ([MEN1998]).

O estudo das linguagens regulares pode ser abordado através dos formalismos:

a) operacional ou reconhecedor: define-se um autômato ou uma máquina abstrata, baseada em estados, em instruções primitivas e na especificação de como cada instrução modifica cada estado. Esta máquina dever ser suficientemente simples para não permitir dúvidas sobre a execução de seu código. O *Autômato Finito*, pode ser determinístico, não-determinístico ou com movimentos vazios;
b) axiomático ou gerador: associam-se regras às componentes da linguagem. As regras permitem afirmar o que será verdadeiro após a ocorrência de cada cláusula, considerando o que era verdadeiro antes da ocorrência. A abordagem axiomática utilizada será a *Gramática Regular*;
c) denotacional: define-se uma função que caracteriza o conjunto de palavras admissíveis na linguagem, sendo restrita às *Expressões Regulares*.

As linguagens regulares são caracterizadas por sua simplicidade e pela facilidade com que são obtidos seus reconhecedores, na forma de autômatos finitos. As construções básicas de que são formadas suas sentenças (identificadores, palavras reservadas, comentários, números decimais, etc.) formam uma linguagem regular, e em geral os compiladores optam por efetuar um processamento prévio do programa fonte, encarando-o como constituído de uma seqüência de tais elementos básicos regulares antes de iniciar a análise estrutural propriamente dita das sentenças livres de contexto que tais elementos formam ([JOS1987]).

As expressões regulares são compiladas num reconhecedor através da construção de um diagrama de transições generalizado chamado Autômato Finito ([JOS1987]).

Conforme [SIL1999], uma expressão regular representa um Conjunto Regular.

2.1.4.1 AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS

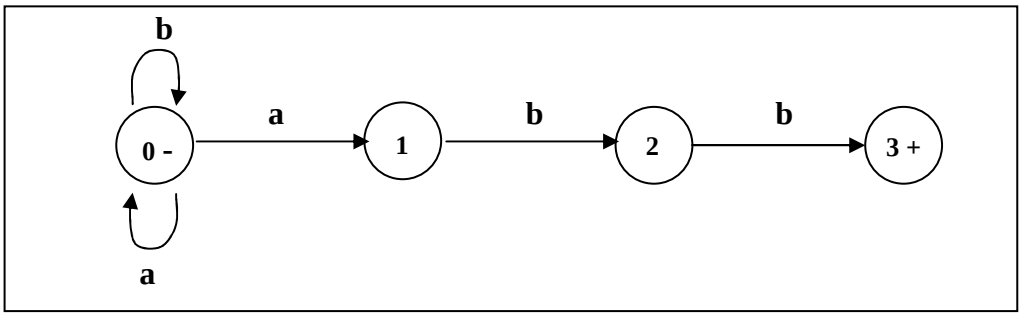
Um Autômato finito não-determinístico é um modelo matemático que consiste em:

a) um conjunto de estados S;
b) um conjunto de símbolos de entrada Σ (o alfabeto de símbolos de entrada);
c) uma função de transição, movimento, que mapeia pares estado-símbolo em conjuntos de estados;
d) um conjunto não-vazio de estados iniciais;

e) um conjunto de estados F distinguidos como estados de aceitação (ou finais).

O autômato não-determinístico assume um conjunto de estados alternativos, como se houvesse uma multiplicação da unidade de controle, uma para cada alternativa, processando independentemente, sem compartilhar recursos com as demais. Assim o processamento de um caminho não influi no estado, símbolo lido, e posição da cabeça dos demais caminhos alternativos ([MEN1998]). Na figura 2.2, a linguagem (a | b)*abb é mostrada na forma de um grafo de transições, do tipo autômato finito não-determinístico. O indeterminismo poderá ser verificado no nó 0 (inicial), onde saem duas arestas dirigidas (transições) com o mesmo elemento, (no caso, o símbolo ‘a’).

FIGURA 2.2 – AUTÔMATO FINITO NÃO-DETERMINÍSTICO



2.1.4.2 AUTÔMATOS FINITOS DETERMINÍSTICOS

Segundo [SIL1999], um autômato finito determinístico é um sistema de transições com número de estados finito e número de transições definidas finito.

Um autômato finito determinístico A sobre um alfabeto $\sum = \{ \alpha_1, \alpha_2, ..., \alpha_n\}$ é um grafo dirigido finito onde ([SIL1999]):

a) de cada nó partem no máximo *n* arestas, sendo que cada aresta corresponde a um α_i distinto $\in \Sigma$;
b) existe um nó denominado de nó inicial (-);
c) existe um conjunto de nós denominados de nós finais (+).

Considerando as definições citadas, é possível dizer que, seja a palavra $w \in \sum^*$:

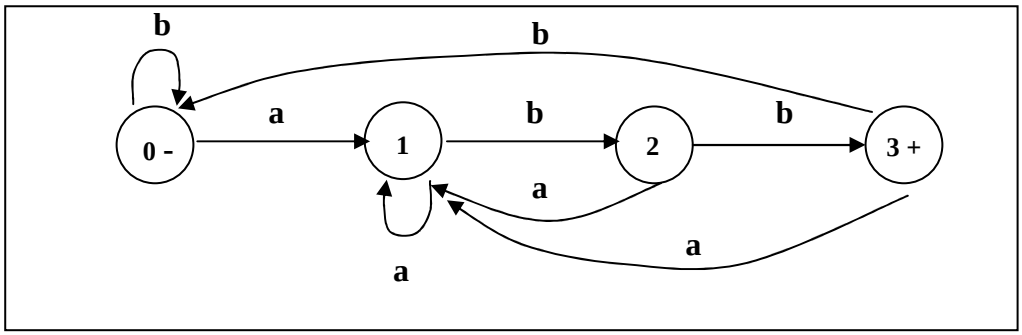
a) caminho-w de um nó *i* até um nó *j* representa a palavra *w* se a concatenação de todas

as letras das arestas do caminho forma w;

b) a palavra w ∈ ∑* será aceita (reconhecida) pelo autômato finito A se e somente se existe um caminho-w do nó inicial até um nó final.

A figura 2.3 representa a linguagem (a | b)*abb na forma de um grafo de transições do tipo autômato finito determinístico.

FIGURA 2.3 – AUTÔMATO FINITO DETERMINÍSTICO



2.1.4.3 EXPRESSÕES REGULARES

Expressão regular é definida a partir de conjuntos básicos (linguagens) e operações de concatenação e união. Trata-se de um formalismo denotacional, também considerado gerador, pois pode-se inferir como construir (“gerar”) as palavras de uma linguagem (conjunto regular). São expressas através do uso exclusivo dos terminais (*tokens* ou símbolos) da linguagem, sem o recurso da utilização de não-terminais ([JOS1987] [MEN1998]).

Uma expressão regular (ER) sobre um alfabeto Σ é indutivamente definida como segue ([MEN1998]):

a) ϕ (conjunto vazio) é uma ER e denota a linguagem vazia;

b) ∈ é uma ER e denota a linguagem contendo exclusivamente a palavra vazia, ou seja, {∈};

c) qualquer símbolo α pertencente ao alfabeto Σ é uma ER e denota a linguagem contendo a palavra unitária α, ou seja, { α }.

Se r e s são ER e denotam as linguagem R e S, respectivamente, então ([MEN1998]):

a) (r + s) é ER e denota a linguagem R ∪ S;

b) (rs) é ER e denota a linguagem RS = { uv | u ∈ R e v ∈ S };

c) (r*) é ER e denota a linguagem R* (veja definição de R* no quadro 2.2).

QUADRO 2.2 – DEFINIÇÃO DE R*

$R^* = R_0 \cup R_1 \cup R_2 \cup ... \cup R_i$

$R_0 = \{\wedge\}$;

$R_1 = R$;

$R_i = R_{i-1}. R$ para $i > 1$.

FONTE: [SIL1999]

Na tabela 2.1, são apresentadas algumas expressões regulares e as correspondentes linguagens, considerando ∑ = {a,b}.

TABELA 2.1 – EXPRESSÕES REGULARES (∑ = {a,b})

Expressão Regular	Linguagem Representada
cc	Somente a palavra cc
ba*	Todas as palavras que iniciam por b, seguido por zero ou mais a's
(a + b)*	Todas as palavras sobre {a,b}
(a + b)* cc (a + b)*	Todas as palavras contendo cc como subpalavra
a*ba*ba*	Todas as palavras contendo exatamente dois b
(a + b)*(cc + dd)	Todas as palavras que terminam com cc ou dd
(a + ∈) (b + ba)*	Todas as palavras que não possuem dois a consecutivos

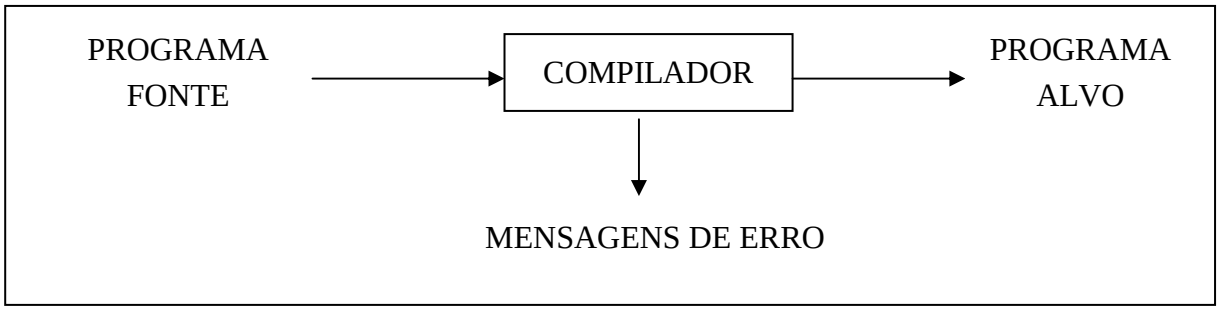
Fonte: [MEN1998]

2.2 COMPILADORES

Um compilador é um programa que efetua automaticamente a tradução de textos, redigidos em uma determinada linguagem de programação — a linguagem fonte — para uma outra forma que viabilize sua execução, ou seja, a linguagem alvo ([AHO1995] [JOS1987]), conforme mostrado na figura 2.4.

O compilador deve ser capaz de detectar se o programa fonte não é válido por qualquer razão (isto é, não corresponde à linguagem para a qual o compilador foi escrito), e, se assim for, imprimir uma mensagem apropriada. Este aspecto da compilação é designado por detecção de erros ([HUN1987]).

FIGURA 2.4 - UM COMPILADOR



Fonte: [AHO1995]

Um compilador, para realizar a parte de consistência, é formado por três grandes módulos: os módulos de análise léxica, de análise sintática e de análise semântica.

2.2.1 ANÁLISE LÉXICA

A análise léxica é a primeira fase do processo de compilação, onde a sua função é de ler os caracteres de um programa fonte de entrada e agrupar num fluxo de *tokens* que serão classificados segundo o tipo que pertencem (identificadores, constantes ou palavras da linguagem), pois o módulo de análise sintática deverá utilizá-los em seguida. A mais importante informação acerca desses caracteres é a classe à qual pertencem e não propriamente o seu valor ([AHO1995] [JOS1987]).

O analisador léxico também pode realizar tarefas secundárias ao nível da interface com o usuário. Uma delas é a de remover do programa fonte os comentários e espaços em branco, tabulações e caracteres de avanço de linha, e também relacionar as mensagens de erro do compilador com o programa fonte ([AHO1995]).

Tratando-se da análise léxica, utilizam-se os termos: *token*, *padrão*, *lexema e parser* com significados específicos, exemplificados na tabela 2.2. O termo *token* representa a menor unidade de informação de uma linguagem. O termo *padrão* é uma regra que descreve o conjunto de lexemas que podem representar um *token* particular nos programas fonte. O termo *lexema* é um conjunto de caracteres no programa fonte que é reconhecido pelo padrão de algum *token*. O termo *parser* significa analisador sintático.

TABELA 2.2 – EXEMPLOS DE *TOKENS*

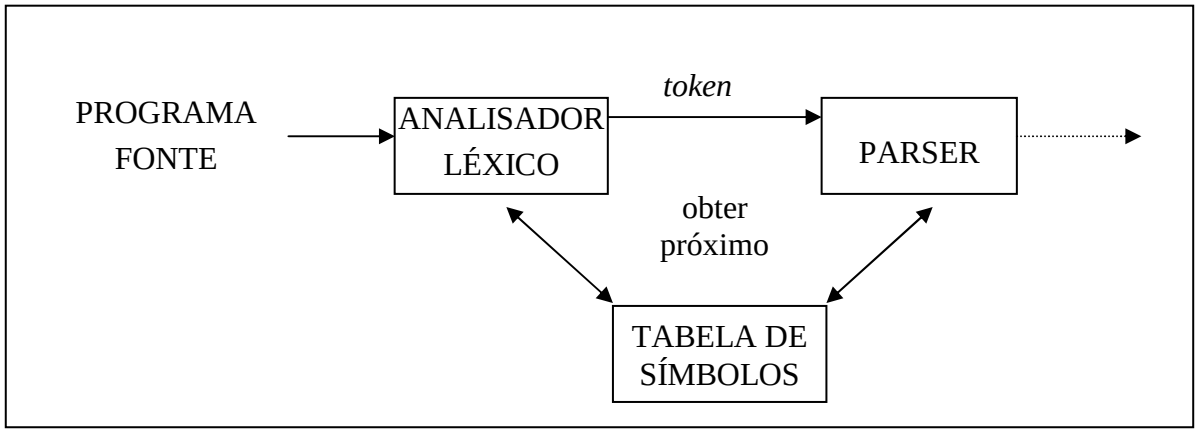
Token	Lexemas Exemplo	Descrição Informal
Const	Const	Const
If	If	If
Relação	<, <=, =, <>, >, >=	< ou <= ou = ou <> ou >= ou >
Id	pi, contador, D2	letra seguida por letras e/ou digitos
Num	3.1416, 0 , 6.02E23	qualquer constante numérica
Literal	“conteúdo da memória”	quaisquer caracteres entre aspas, exceto aspas

Fonte: [AHO1995]

Os analisadores léxicos são módulos funcionais do compilador, cujas funções são ativadas inúmeras vezes durante o processo de compilação de um programa fonte. Assim sendo, basta notar que cada símbolo, número, palavra reservada, identificador, correspondem ao resultado da execução de uma chamada deste módulo. Portanto convém que os analisadores léxicos sejam construídos com extremo cuidado, utilizando técnicas na qual se obtenha um programa de alta eficiência, pois caso contrário isto pode prejudicar seriamente o desempenho do compilador ([JOS1987]).

Quando o analisador léxico é inserido entre o *parser* e o fluxo de entrada, o mesmo interage com os dois, conforme mostrado na figura 2.5.

FIGURA 2.5 - INTERAÇÃO DO ANALISADOR LÉXICO COM O PARSER



Fonte: [AHO1995]

2.2.2 ANÁLISE SINTÁTICA

O analisador sintático é o segundo grande módulo componente dos compiladores e que se pode caracterizar como o mais importante, na maioria dos compiladores, por sua característica de controlador das atividades do compilador. Tem como principal função verificar se uma cadeia de *tokens* proveniente do analisador léxico, pode ser gerada pela gramática da linguagem-fonte. A partir desta cadeia, o analisador sintático verifica a ordem de apresentação dos *tokens* na seqüência, identificando, em cada situação, o tipo da construção sintática por eles formada, de acordo com a gramática na qual se baseia o reconhecedor, relatando quaisquer erros de sintaxe de uma forma entendível ([AHO1995] [JOS1987]).

Além das funções mencionadas acima, o analisador sintático executa outras de grande importância, conforme [JOS1987]:

a) identificação de sentenças;
b) detecção de erros de sintaxe;
c) recuperação de erros;
d) correção de erros;
e) montagem da árvore abstrata da sentença;
f) comando da ativação do analisador léxico;
g) comando do modo de operação do analisador léxico;
h) ativação de rotinas da análise referente às dependências de contexto da linguagem;
i) ativação de rotinas de análise semântica;
j) ativação de rotinas de síntese do código objeto.

Os métodos de análise sintática mais comumente utilizados na construção de compiladores são o descendente (*top-down*) e o ascendente (*bottom-up*). Esses termos se referem à ordem na qual os nós da árvore gramatical são construídos. No primeiro, a construção se inicia na raiz e prossegue em direção às folhas, enquanto que no segundo, a construção se inicia nas folhas e procede em direção à raiz ([AHO1995]).

2.2.3 ANÁLISE SEMÂNTICA

A terceira fase do processo de compilação, a análise semântica, refere-se à verificação de erros semânticos no programa fonte, e a captura de informações de tipo para a fase subseqüente de geração de código. Nesta verificação de tipos, o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, muitas definições nas linguagens de programação requerem que o compilador relate um erro a cada vez que um número *real* seja usado para indexar um *array* ([AHO1995]).

A ação da análise semântica no processo de compilação consiste em incluir, na árvore sintática, informações adicionais que venham a facilitar a obtenção do programa objeto correspondente. A árvore sintática, enriquecida com as informações semânticas, está pronta para ser processada, com a finalidade de ser traduzida para a forma de programa objeto ([JOS1987]). Informações sobre árvore sintática podem ser vistas na seção 2.3.2.1.

Segundo [JOS1987], ações semânticas são funções ativadas pelo analisador sintático digiridos pela sintaxe sempre que forem atingidos certos estados do reconhecimento, ou sempre que determinadas transições ocorrerem durante a análise do programa fonte.

Algumas das principais funções das ações semânticas são ([JOS1987]):

a) criação e manutenção de tabelas de símbolos;
b) associar aos elementos da tabela de símbolos seus respectivos atributos;
c) manter informações sobre o escopo dos identificadores;
d) representar tipos de dados;
e) analisar restrições quanto a utilização dos identificadores;
f) verificar o escopo dos identificadores;
g) verificar a compatibilidade de tipos;
h) efetuar a tradução do programa.

2.3 GRAMÁTICAS LIVRES DO CONTEXTO

Uma Gramática Livre do Contexto é uma gramática onde o lado esquerdo das produções contém exatamente uma variável ([MEN1998]).

São Gramáticas Livres do Contexto, aquelas em que é levantado o condicionamento das substituições impostas pelas regras definidas pelas produções ([JOS1987]).

[AHO1995] cita que a gramática livre de contexto ou BNF (Forma de Backus-Naur) é uma notação amplamente aceita para a especificação da sintaxe de uma linguagem de programação.

Uma gramática livre do contexto é formada pelos seguintes componentes ([AHO1995] [WAT1991]):

a) *Terminais*, também conhecidos como *tokens*: estes símbolos são usados de fato quando se escreve a linguagem;
b) *Não-Terminais:* são variáveis sintáticas que denotam novas produções, criando assim uma nova hierarquia na gramática;
c) *Símbolo de Partida*: é a designação atribuída a um dos não-terminais do lado esquerdo da primeira produção da gramática;
d) *Conjunto de Produções*: as produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste em um não-terminal (aqui chamado de *lado esquerdo da produção*), seguido por uma seta (→) (veja definição na tabela 2.3), seguido por uma cadeia de terminais e não-terminais, chamados de *lado direito* da produção.

Através do uso da notação BNF, é possível representar, qualquer gramática livre do contexto, pois é uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentenças. Para tanto, cada produção corresponde a uma regra de substituição, em que para cada símbolo da linguagem são associadas uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição ([JOS1987]). Exemplos de produções em uma gramática livre do contexto podem ser vistas no quadro 2.3.

QUADRO 2.3 – PRODUÇÕES DE UMA GRAMÁTICA LIVRE DO CONTEXTO

```
A → B ‘a’ | C ‘b’ | A ‘c’
B → C ‘c’ | B ‘a’ |
C → C ‘b’ | ∈
```

Fonte: [JOS1987]

Uma gramática livre do contexto é dita ambígua, se existe uma palavra que possua duas ou mais árvores de derivação ([MEN1998]). Maiores detalhes sobre árvores de derivação e ambigüidade podem ser vistos nas seções 2.3.2 e 2.3.3 respectivamente.

2.3.1 BNF

A BNF (*Backus-Naur form*) é uma metalinguagem que tem sido utilizada com sucesso para especificação da sintaxe de linguagens de programação, desde que foi publicada pela primeira vez no relatório de especificação da linguagem Algol 60 por John Backus e Peter Naur ([JOS1987] [WAT1991]).

Trata-se de uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentenças, portanto cada produção corresponde a uma regra de substituição, em que para cada símbolo da metalinguagem são associadas uma ou mais cadeias de símbolos, indicando as diversas possibilidades de substituição. Os símbolos correspondem a não-terminais da gramática que esta sendo especificada. As cadeias podem ser formadas de terminais e ou não-terminais e do símbolo $\in$, que representa a cadeia vazia ([JOS1987]). A simbologia adotada pela BNF é mostrada na tabela 2.3.

Devido a equivalência da BNF com as gramáticas livres de contexto, esta é amplamente aceita como metalinguagem, através da qual são construídas gramáticas e os diagramas de estados que representam autômatos finitos, através dos quais são construídos reconhecedores para uma importante classe de linguagens de programação ([JOS1987]).

TABELA 2.3 – SIMBOLOGIA ADOTADA PELA BNF

$\rightarrow$	É o símbolo da metalinguagem que associa a um não-terminal um conjunto de cadeias de terminais e/ou não-terminais, incluindo o símbolo da cadeia vazia. O não-terminal em questão é escrito à esquerda deste símbolo, e as diversas cadeias, à sua direita. Lê-se como “pode ser”
|	É o símbolo que separa as diversas cadeias (opções) que constam à direita do símbolo $\rightarrow$. Lê-se como “OU”
$\in$	Representa a cadeia vazia na notação BNF
‘X’	Representa um *terminal* (literal) da linguagem que está sendo definida, e pertence ao conjunto de todos os *tokens* que compõem as sentenças da linguagem. Deve ser denotado tal como figura nas sentenças da linguagem
E	Representa um não-terminal, cujo nome é dado por uma cadeia de caracteres quaisquer.

Fonte: Baseado em [JOS1987]

2.3.2 ÁRVORE DE DERIVAÇÃO

Para uma gramática livre de contexto, a representação visa a derivação de palavras, que pode ser mostrada na forma de árvore, denominada de árvore de derivação ou árvore gramatical. Em aplicações como compiladores e processadores de textos, freqüentemente é conveniente representar a derivação de palavras na forma de árvore, partindo do símbolo inicial como raiz e terminando em folhas de terminais, onde suas folhas são lidas da esquerda para a direita e formam o produto da árvore, que é a cadeia gerada ou derivada a partir do não terminal à raiz da árvore ([AHO1995][MEN1998]). Formalmente a árvore gramatical possui as seguintes propriedades:

a) a raiz é rotulada pelo símbolo de partida;
b) cada folha é rotulada por um *token* ou o símbolo vazio;
c) cada nó interior é rotulado por um não-terminal.

Uma árvore de derivação pode ser vista como uma representação gráfica para uma derivação que filtre a escolha relacionada à ordem de substituição, onde cada nó interior de uma árvore gramatical é rotulado por algum não-terminal A, e que os filhos de um nó são rotulados, da esquerda para direita, pelos símbolos do lado direito da produção pelos quais A foi substituído na derivação. As folhas da árvore são rotuladas por não-terminais ou terminais

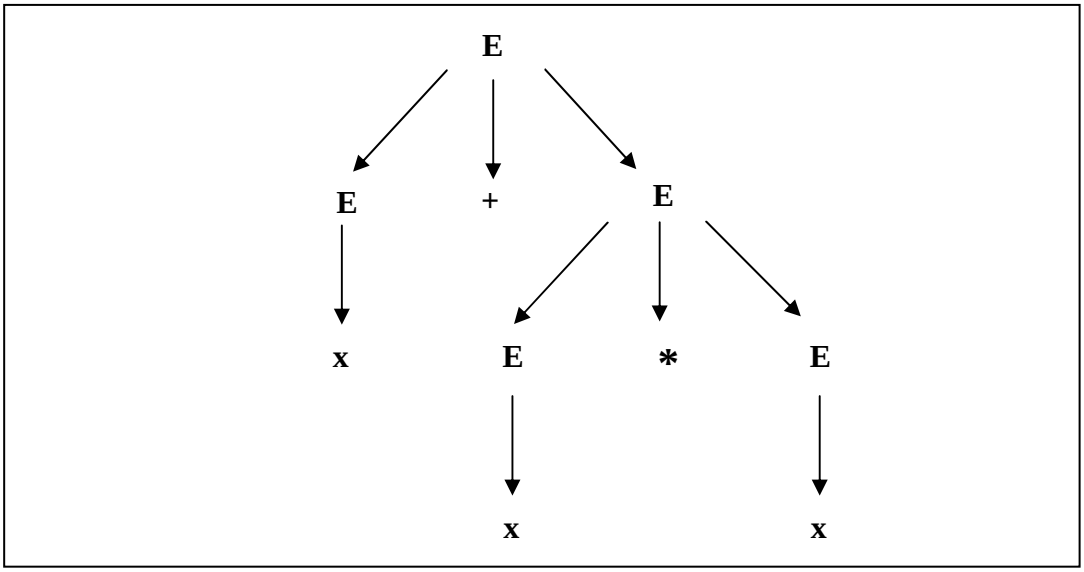
e lidos da esquerda para direita ([AHO1995]). No quadro 2.4 (a) é mostrada uma produção para expressões aritméticas contendo dois operadores e um operando. Uma derivação para a expressão x + x * x também é mostrada no quadro 2.4 (b) e na figura 2.6 a respectiva árvore de derivação.

QUADRO 2.4 – PRODUÇÃO E DERIVAÇÃO PARA A EXPRESSÃO x + x * x

(a) P = {E → E + E | E * E | E | x }
(b) E → E + E → x + E → x + E * E → x + x * x

FONTE: [MEN1998]

FIGURA 2.6 –ÁRVORE DE DERIVAÇÃO BASEADA NO QUADRO 2.4

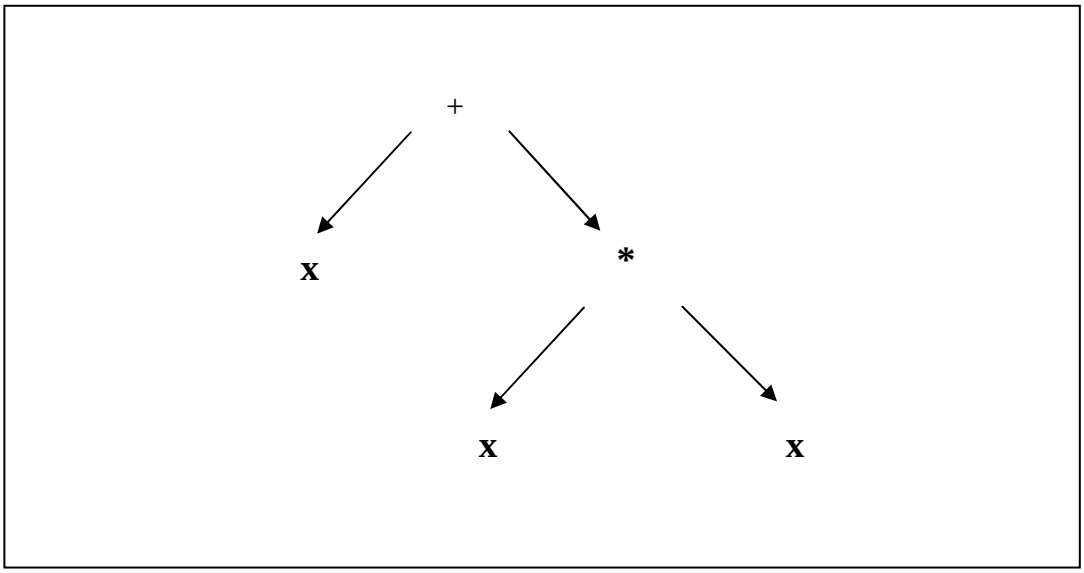


FONTE: [MEN1998]

2.3.2.1 ÁRVORE DE DERIVAÇÃO X ÁRVORE SINTÁTICA

As árvores de derivação diferem das árvores sintáticas porque as distinções superficiais de forma, desimportantes para a tradução, não figuram nas árvores sintáticas. Os operadores e palavras chave não figuram folhas, mas sim são associados ao nó interior que seria o pai daquelas folhas na árvore gramatical ([AHO1995]). Um exemplo de árvore de derivação pode ser visto na figura 2.6 e um exemplo de árvore sintática pode ser visto na figura 2.7.

FIGURA 2.7 –ÁRVORE SINTÁTICA BASEADA NO QUADRO 2.4



2.3.3 AMBIGÜIDADE

Uma gramática é ambígua, quando a mesma gera duas ou mais árvores de derivação. No desenvolvimento e otimização de alguns tipos de algoritmos de reconhecimento, é conveniente que a gramática usada não seja ambígua, entretanto nem sempre é possível eliminar essa ambigüidade, sendo mais fácil definir linguagens para as quais qualquer gramática livre do contexto seja ambígua ([MEN1998][WAT1991]).

Um exemplo de gramática ambígua e de como reescrevê-la afim de eliminar a ambigüidade será mostrado nos quadros 2.5 e 2.6 e na figura 2.8. Será utilizado, como exemplo, a gramática ambígua do “ELSE-VAZIO”, lembrando que o “**outro**” significa qualquer outro enunciado.

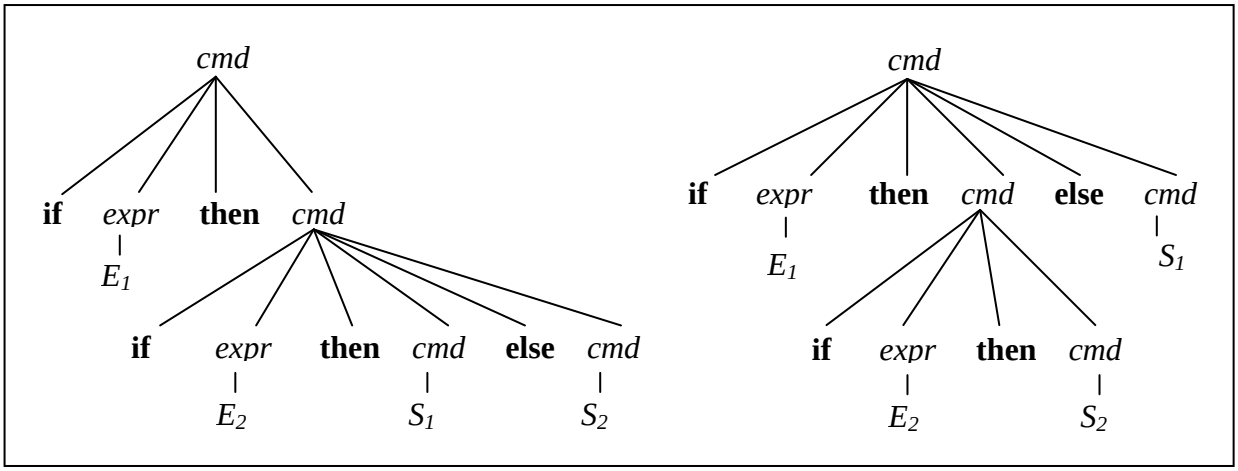
QUADRO 2.5 – GRAMÁTICA AMBÍGUA DO “ELSE-VAZIO”

cmd → **if** *expr* **then** *cmd*
| **if** *expr* **then** *cmd* **else** *cmd*
| **outro**

FONTE: [AHO1995]

A gramática mostrada no quadro 2.5 é ambígua uma vez que: **If** E_1 **then if** E_2 **then** S_1 **else** S_2, possui duas árvores, conforme mostra a figura 2.8.

FIGURA 2.8 - DUAS ÁRVORES GRAMATICAIS PARA A MESMA SENTENÇA



FONTE: [AHO1995]

Nas linguagens de programação com sentenças condicionais desta forma, prefere-se a primeira árvore gramatical, utilizando-se a regra geral de "associar cada **else** ao **then** anterior mais próximo ainda não associado" ([AHO1995]). Essa regra de inambigüidade pode ser incorporada diretamente a gramática. Sendo assim é possível rescrever a gramática sob a forma inambígua apresentada no quadro 2.6.

QUADRO 2.6 – GRAMÁTICA DO "ELSE-VAZIO" SEM AMBIGÜIDADE

cmd →	*cmd_associado*
	| *cmd_não associado*
cmd_associado →	**if** *expr* **then** *cmd_associado* **else** *cmd_associado*
	| **outro**
cmd_não associado →	**if** *expr* **then** *cmd*
	| **if** *expr* **then** *cmd_associado* **else** *cmd_não associado*

FONTE: [AHO1995]

2.3.4 FATORAÇÃO À ESQUERDA

A fatoração à esquerda resume-se em, quando não estiver claro qual das duas produções alternativas usar (*indeterminismo*) para expandir um não-terminal *A*, rescrever as produções *A* e postergar a decisão até que tenha-se verificado a entrada o suficiente para realizar uma escolha ([AHO1995] [JOS1987]).

Tomando como exemplo as duas produções mostradas no quadro 2.7, ao verificar o *token* de entrada **if**, não se pode decidir qual produção escolher para expandir *cmd*. Considerando que, se $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, são duas produções A, e a entrada não começa por uma cadeia vazia derivada a partir de α, portanto não se sabe se a expansão de A será feita em $\alpha\beta_1$ ou em $\alpha\beta_2$, gerando assim um *indeterminismo*. Para resolver esse problema será feita a fatoração à esquerda, postergando-se a decisão e expandindo A para $\alpha A'$, e após enxergar a entrada derivada a partir de α, A' será expandido em β_1 ou em β_2 ([AHO1995] [LEW2000]). As produções originais (quadro 2.8 (a)), fatoradas à esquerda são demonstradas no quadro 2.8 (b).

QUADRO 2.7 – PRODUÇÃO COM INDETERMINISMO

cmd → **if** *expr* **then** *cmd* **else** *cmd* | **if** *expr* **then** *cmd*

Fonte: [AHO1995]

QUADRO 2.8 – FATORAÇÃO DE UMA PRODUÇÃO

(a) com indeterminismo $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$	(b) sem indeterminismo $A \rightarrow \alpha A'$ $A' \rightarrow \beta_1 \mid \beta_2$

Fonte: [AHO1995]

2.3.5 RECURSIVIDADE À ESQUERDA

A recursividade à esquerda existe quando o não-terminal no lado esquerdo da produção é igual ao primeiro *token* do lado direito ou igual ao primeiro *token* encontrado após

um operador condicional OU. Neste caso exemplificado no quadro 2.9, o *token E* do lado direito é chamado recursivamente e o analisador roda para sempre, causando um laço infinito ([AHO1995]).

QUADRO 2.9 – PRODUÇÃO COM RECURSIVIDADE

E → E + T | T

Fonte: Baseado em [AHO1995]

O processo para eliminação da recursividade à esquerda consiste em ([AHO1995]):

a) criar uma nova produção derivada da produção recursiva (no quadro 2.10 será chamada de *E*') e nesta produção copiar todos os *tokens* da opção recursiva, exceto o *token* recursivo, acrescentando ao final da nova produção, o seu não-terminal do lado esquerdo, seguido do operador OU e do *token* vazio (∈);
b) na produção original, acrescentar o não-terminal do lado esquerdo da produção nova em cada opção, exceto na opção recursiva, que será eliminada da produção.

QUADRO 2.10 – PRODUÇÃO SEM RECURSIVIDADE

E → T E'
E' → + T E' | ∈

Fonte: Baseado em [AHO1995]

2.3.6 RECURSIVIDADE INDIRETA

Conforme [AHO1995], a recursividade indireta ou não imediata, ocorre quando um não-terminal E deriva uma produção de tal modo que somente possa ser recursiva após algumas substituições do não-terminal na produção principal como é mostrado no quadro

QUADRO 2.11 – PRODUÇÕES COM RECURSIVIDADE INDIRETA

S → A a | b
A → A c | S d | ∈

FONTE: [AHO1995]

2.11.

O não-terminal *S* é recursivo à esquerda porque S → A a → S d a, mas não é diretamente recursivo. Para solucionar este problema [AHO1995] substitui as produções da forma inversa àquela que foram criadas, portanto substitui ***S*** na produção ***A*** gerando a produção **A→ A c | A a d | b d | ∈ ,** que agora possui recursividade direta, esta será retirada gerando as produção sem recursividade indireta conforme mostrado no quadro 2.12.

QUADRO 2.12 – PRODUÇÕES DO QUADRO 2.11 SEM RECURSIVIDADE INDIRETA

S → A a | b A → b d A' | A' A' → c A' | a d A' | ∈

FONTE: [AHO1995]

2.4. NOTAÇÃO PÓS-FIXA

A notação pós-fixa é uma notação na qual os operadores figuram após seus operandos ([AHO1995]).

Uma expressão *E*, pode ser definida na forma pós-fixa seguindo as regras ([AHO19995]):

a) se *E* for uma variável ou constante, então a notação pós-fixa para *E* será o próprio *E;*
b) se *E* for uma expressão da forma E_1 *op* E_2, onde op é qualquer operador binário, então a forma pós-fixa para E será E_1' E_2' *op*, onde E_1' e E_2' são as notações pós-fixas para E_1 e E_2, respectivamente;
c) se *E* for uma expressão da forma (E_1), então a notação pós-fixa para E_1 será também a notação pós-fixa para *E*.

Os parênteses não são necessários na notação pós-fixa porque a posição e o número de argumentos dos operadores permitem somente uma única decodificarão de uma expressão

pós-fixa. Um exemplo de uma expressão regular (a) e sua forma equivalente pós-fixada (b) é mostrada no quadro 2.13.

QUADRO 2.13 – EXPRESSÃO REGULAR INFIXA (a) E PÓS-FIXA (b)

(a)	**(b)**
a (b + c)* + ca* + ^	**a b c + * . c a * . + ^ +**

Fonte: [SIL2000a]

2.5 TEOREMA DE KLEENE

Em 1956, o matemático Stephen Kleene demonstrou pela primeira vez que os conjuntos regulares são exatamente os conjuntos que as máquinas de estado finito podem reconhecer ([GER1995]). Seu teorema, conhecido como teorema de Kleene, está descrito no quadro 2.14.

QUADRO 2.14 – TEOREMA DE KLEENE

"Qualquer conjunto reconhecido por uma máquina de estado finito é regular, e qualquer conjunto regular pode ser reconhecido por uma máquina de estado finito".

FONTE: [GER1995]

O teorema de Kleene estabelece as limitações, bem como as capacidades, de máquinas de estado finito, pois existem certamente diversos conjuntos que não são regulares e que pelo Teorema de Kleene, não há máquina de estado finito capaz de reconhecer ([GER1995]).

Segundo [MAN1974], o teorema de Kleene afirma:

a) para todo grafo de transições T sobre um alfabeto ($\sum$) existe uma expressão regular R sobre o $\sum$ tal que o conjunto de todas as palavras reconhecidas por R é igual ao conjunto de todas as palavras reconhecidas por T;
b) para toda expressão regular R sobre o $\sum$ existe um autômato finito determinístico A sobre o $\sum$ tal que o conjunto de todas as palavras reconhecidas por A é igual ao conjunto de todas as palavras reconhecidas por R.

3 GRAMÁTICAS LIVRES DO CONTEXTO x EXPRESSÕES REGULARES

Cada construção que possa ser descrita por uma expressão regular também pode ser descrita por uma gramática livre do contexto. Um exemplo desta equivalência é apresentado na tabela 3.1.

TABELA 3.1 – EQUIVALÊNCIA ENTRE UMA EXPRESSÃO REGULAR E UM GRAMÁTICA LIVRE DO CONTEXTO

(a | b)* a bb	$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$ $A_1 \rightarrow bA_2$ $A_2 \rightarrow bA_3$ $A_3 \rightarrow \in$

FONTE: [AHO1995]

Para efetuar a transformação de uma gramática livre do contexto em uma expressão regular, [SIL2000b] estende a expressão regular através da inclusão de dois operadores (‘~’ e ‘ ” ’, respectivamente nomeados como contador de iterações e repetidor de iterações previamente definidas). Ainda, [SIL2000b] trata as possíveis formas de recursividade (auto-embutidas, não-auto-embutidas à esquerda e à direita) que possam surgir em uma gramática livre do contexto e simultaneamente mostra como convertê-las para uma expressão regular. A seguir são mostradas estas regras.

3.1 PRODUÇÕES RECURSIVAS AUTO-EMBUTIDAS

Uma gramática livre do contexto G é auto-embutida se e somente se, para algum símbolo não-terminal N de G, existe um α N β tal que N é também uma produção, e que α e β são não-terminais não-vazios, implicando que as produções sejam recursivas ([WAT1991]). Os exemplos do quadros 3.1 e 3.2 mostram que a gramática define N em termos de si mesmo.

Porém, mesmo que uma gramática seja recursiva, não implica necessariamente que ela seja auto-embutida, como será discutido na seção 3.2.

QUADRO 3.1 – PRODUÇÃO RECURSIVA AUTO-EMBUTIDA

$N \rightarrow \alpha\ N\ \beta$

FONTE: [WAT1991]

Segundo [SIL2000b], para converter a produção auto-embutida descrita no quadro 3.2 em uma expressão regular e eliminar a recursividade, deve-se para cada opção auto-embutida:

a) substituir o *token*-recursivo por todas as opções classificadas como normais ou não-auto-embutidas (já sem recursividade);
b) acrescentar o *token* '~' após os *tokens* α;
c) acrescentar o *token* ' " ' após os *tokens* β.

QUADRO 3.2 – PRODUÇÃO RECURSIVA AUTO-EMBUTIDA COM VÁRIAS OPÇÕES

$N \rightarrow \alpha_1\ N\ \beta_1 \mid x \mid y$

FONTE: [SIL2000b]

Um exemplo desta transformação é mostrado no quadro 3.3.

QUADRO 3.3 – TRANSFORMAÇÃO DE PRODUÇÃO RECURSIVA AUTO-EMBUTIDA

$N \rightarrow (\alpha_1) \sim (x \mid y) (\beta_1)$ "

FONTE: [SIL2000b]

3.2 PRODUÇÕES RECURSIVAS NÃO-AUTO-EMBUTIDAS

Para uma gramática ser Não-Auto-Embutida, basta que a produção seja recursiva à esquerda (α deve ser vazio em relação ao quadro 3.1) ou recursiva à direita (β deve ser vazio em relação ao quadro 3.1), como demonstrado no quadro 3.4.

QUADRO 3.4 – PRODUÇÕES RECURSIVAS NÃO-AUTO-EMBUTIDAS À ESQUERDA (a) E À DIREITA (b)

a) $N \rightarrow N \beta$	b) $N \rightarrow \alpha N$

FONTE: [WAT1991]

Para eliminação da recursividade das produções não-auto-embutida à esquerda e à direita, segue-se o método apresentado por [WAT1991]. No quadro 3.5 é apresentado uma produção com recursividade não-auto-embutida à esquerda e à direita.

QUADRO 3.5 –PRODUÇÃO RECURSIVA NÃO-AUTO-EMBUTIDA COM VÁRIAS OPÇÕES

$$N \rightarrow N \alpha_1 \mid N \alpha_2 \mid \alpha_3 N \mid \alpha_4 N \mid \beta_1 \mid \beta_2$$

Para a transformação da produção do quadro 3.5, o qual possui recursividade tanto à direita quanto à esquerda, procede-se da seguinte maneira:

a) caso exista recursividade não-auto-embutida à esquerda, coloca-se as opções sem recursividade não-auto-embutida à esquerda por primeiro entre parênteses;
b) e após, coloca-se as opções recursivas não-auto-embutidas à esquerda entre parênteses, sem o *token*-recursivo (somente os α's) sob o operador estrela (*). A produção resultante é mostrada no quadro 3.6;

QUADRO 3.6 – RETIRADA DA RECURSIVIDADE À ESQUERDA DA PRODUÇÃO DO QUADRO 3.5

$$N \rightarrow (\alpha_3 N \mid \alpha_4 N \mid \beta_1 \mid \beta_2) \ (\alpha_1 \mid \alpha_2)^*$$

c) verifica-se se ainda existe recursividade entre os parênteses, existindo, (no caso da recursividade não-auto-embutida à direita), coloca-se as opções recursivas não-auto-embutidas à direita entre parênteses sem o *token* recursivo (somente os α's) sob o operador estrela;
d) após coloca-se o restante das opções que não são recursivas não-auto-embutidas à direita entre parênteses. A produção resultante é mostrada no quadro 3.7.

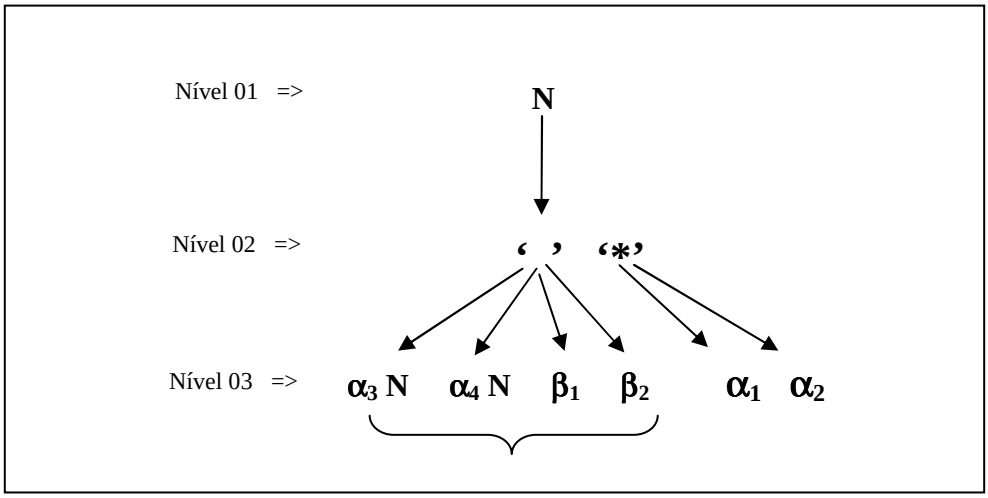
QUADRO 3.7 – RETIRADA DA RECURSIVIDADE
À DIREITA DA PRODUÇÃO DO QUADRO 3.6

$$N \rightarrow ((\alpha_3 \mid \alpha_4)^* (\beta_1 \mid \beta_2)) \ (\alpha_1 \mid \alpha_2)^*$$

3.3 ÁRVORE SINTÁTICA PARA TRANSFORMAÇÃO DE LINGUAGENS LIVRES DO CONTEXTO EM EXPRESSÕES REGULARES

Conforme visto no quadro 3.6, essa produção ainda possui recursividade, portanto a árvore sintática será utilizada para distribuir a produção de forma a manter em níveis inferiores as opções que estiverem envolvidas por parênteses. Sendo assim cada *token* poderá ter "irmãos" e "filhos" e quando algum parênteses for encontrado será criado um *token* "pai" de acordo com o token que tiver após os parênteses, este pode ser '*', '~' , ' " ' ou ' ', e os tokens que estiverem dentro dos parênteses serão guardados como filhos do *token* anteriormente alocado. Desta maneira a retirada da recursividade em produções que possuem parênteses torna-se mais fácil pois, deve-se somente retirar a recursividade dos tokens com o mesmo pai, que conforme é mostrado na figura 3.1 são os *tokens* filhos de ' ' e pertencentes ao nível 03.

FIGURA 3.1 – REPRESENTAÇÃO NA FORMA DE ÁRVORE
SINTÁTICA PARA A EXPRESSÃO DO QUADRO 3.6



Caso exista recursividade indireta na árvore o mesmo processo descrito na seção 2.3.6 poderá ser aplicado.

3.4 EXEMPLO DA ELIMINAÇÃO DA RECURSIVIDADE E CONVERSÃO DE UMA GRAMÁTICA LIVRE DO CONTEXTO EM UMA EXPRESSÃO REGULAR ESTENDIDA

Antes de apresentar um exemplo mais completo, é necessário definir alguns termos utilizados, tais como:

a) opção: cada grupo de *tokens* contidos após o *token* "pode ser" (→) ou entre operadores "OU", são chamados de **opção**, portanto cada produção pode ter uma ou mais opções delimitadas por operadores 'OU' e caracterizadas como:
   - sem recursividade;
   - auto-embutidas;
   - não-auto-embutidas à esquerda;
   - não-auto-embutidas à direita.

b) *token*-recursivo: é quando algum *token* de uma opção é o mesmo *token* do lado esquerdo da produção.

Conforme estas definições [SIL2000b] propõe um método que, baseado na gramática mostrada no quadro 3.8, transformará a gramática livre do contexto em uma expressão regular, retirando automaticamente a recursividade. O algoritmo é apresentado no quadro 3.9.

QUADRO 3.8 – EXEMPLO DE TRANSFORMAÇÃO DE UMA GRAMÁTICA LIVRE DO CONTEXTO EM EXPRESSÃO REGULAR E ELIMINAÇÃO DA RECURSIVIDADE

a) N → N α_1 | α_2 N | β_1 N β_2 | X | Y | β_3 N β_4

b) N → β_1 N β_2 | β_3 N β_4 | (α_1* (X | Y)) | ((X | Y) α_2*)

c) N → β_1~ ((α_1* (X | Y)) | ((X | Y) α_2*)) β_2" |
β_3~ ((α_1* (X | Y)) | ((X | Y) α_2*)) β_4"

FONTE: [SIL2000b]

QUADRO 3.9 – ALGORITMO PARA TRANSFORMAÇÃO DE UMA GRAMÁTICA LIVRE DO CONTEXTO EM EXPRESSÃO REGULAR E ELIMINAÇÃO DA RECURSIVIDADE

Baseado na produção (a) do quadro 3.8:

a) gerar a produção intermediária (b) do quadro 3.8:
- criar uma nova produção mantendo as opções recursivas auto-embutidas para posterior substituição;
- transformar as opções não-auto-embutidas em expressão regular como visto na seção 3.2.

b) gerar a produção (c) do quadro 3.8, e para cada opção recursiva auto-embutida:
- acrescentar o *token* '~' antes do *token* recursivo;
- acrescentar o *token* ' " ' ao final da opção;
- substituir o *token* recursivo, pelas opções geradas e delimitadas por '( )' no passo a;

FONTE: [SIL2000b]

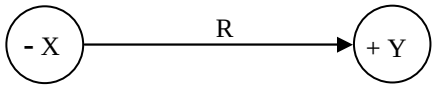
4 TRANSFORMAÇÃO DE UMA EXPRESSÃO REGULAR EM GRAFO DE TRANSIÇÕES

Manna [MAN1974] apresenta um algoritmo para transformar uma expressão regular em um autômato finito determinístico, baseado no Teorema de Kleene. Tal algoritmo é descrito em dois passos apresentados na tabela 4.1 e no quadro 4.1 respectivamente.

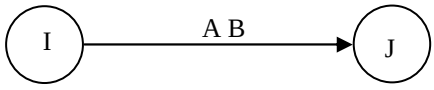
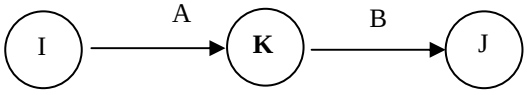
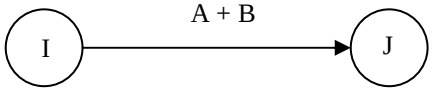
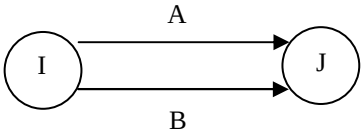
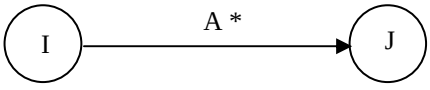
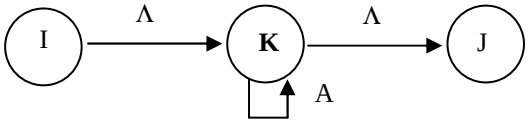
Um exemplo da aplicação do algoritmo proposto por [MAN1974] será apresentado considerando a expressão regular descrita no quadro 4.2 e o alfabeto como sendo $\sum = \{a,b\}$.

TABELA 4.1 – PASSO 1: TRANSFORMAÇÃO DE UMA EXPRESSÃO REGULAR EM GRAFO DE TRANSIÇÕES

Primeiro constrói-se o grafo de transição T tal que todo conjunto de palavras reconhecidas por T é igual ao conjunto de palavras reconhecidas pela expressão Regular R. Inicia-se com um grafo de transição generalizado da forma:



Após, quebra-se R sucessivamente adicionando novos vértices e arestas, até que todas arestas estejam rotuladas somente por letras ou por vazio (Λ), usando as seguintes regras:

(a)	**(b)**
	
	
	

Fonte: [SIL2000a]

QUADRO 4.1 – ALGORITMO PARA TRANSFORMAÇÃO DE UMA EXPRESSÃO REGULAR EM UM AUTÔMATO FINITO DETERMINÍSTICO

A partir do grafo de transições gerado no passo 1 (tabela 4.1), são realizadas as ações:

a) expandir o grafo até que cada aresta esteja marcada com um símbolo $\sigma_i \in \Sigma$;

b) construir uma tabela de transições:
   - coluna: símbolo $\sigma_i \in \Sigma$;
   - linha : subconjunto M dos nós do grafo;
   - a primeira linha representa M_0 - conjunto dos nós iniciais e todos os nós atingíveis a partir dos nós iniciais por palavras vazias;
   - para a coluna $\sigma_i \in \Sigma$ e a linha M_0, formar um conjunto M_j dos nós atingíveis a partir dos nós de M_i por caminhos σ_i. Se a partir de um nó com um elemento σ_i atingir um nó onde este atinge outros com a palavra vazia, os outros atingidos diretamente pela palavra vazia também são considerados no conjunto M_j;
   - se M_j ainda não possuir uma linha na tabela, acrescenta-se uma linha M_j;
   - repetir o procedimento até completar a tabela.;

c) formação do Autômato Finito (A.F.):
   - a cada novo conjunto M_i corresponde um nó m_i no A.F.;
   - o nó inicial do A.F. é m_0;
   - o nó m_i é final se e somente se o conjunto M_i possuir pelo menos um nó final do grafo de transições.

FONTE: [SIL2000a]

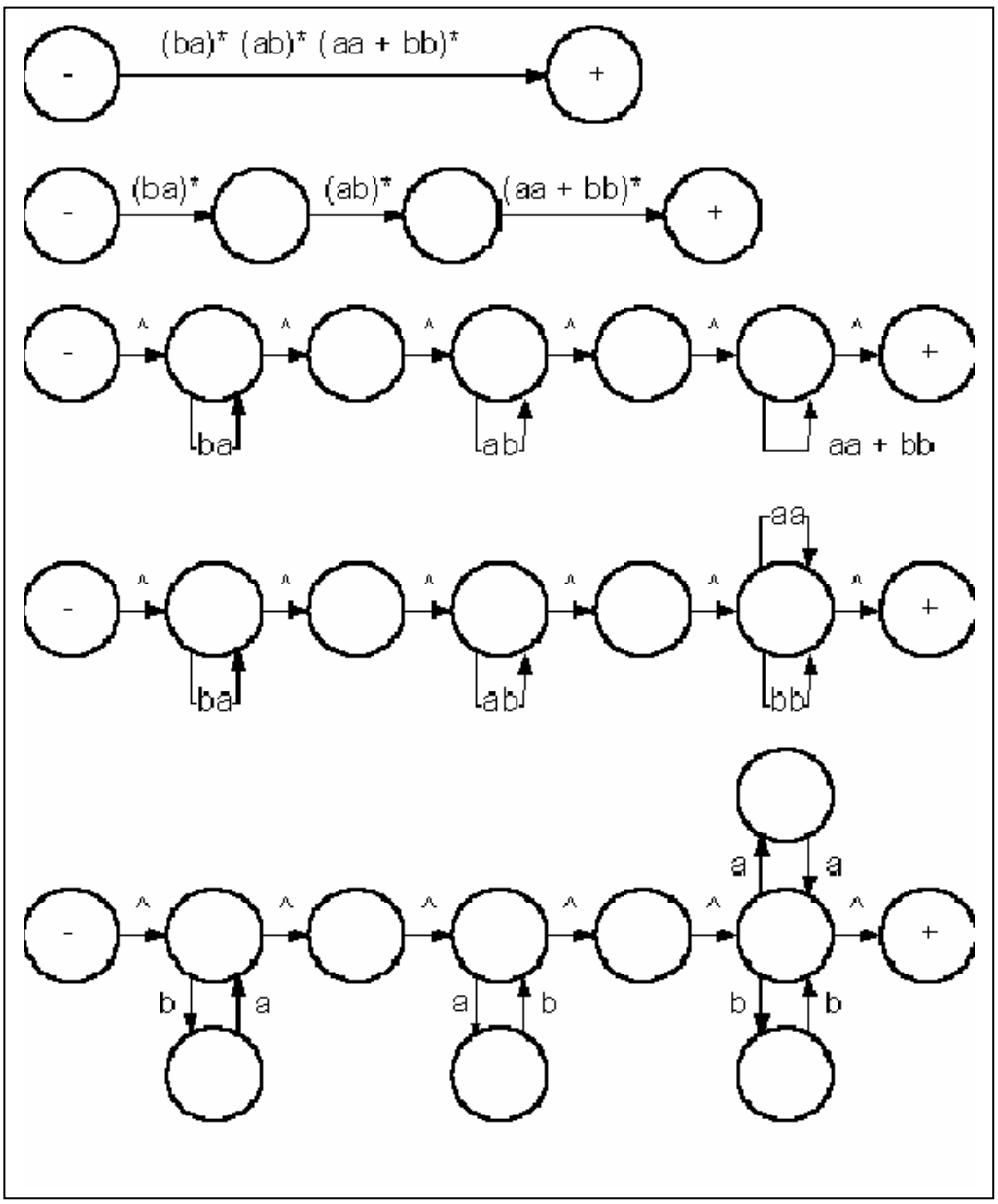
QUADRO 4.2 – EXPRESSÃO REGULAR

(ba)* (ab)* (aa + bb)*

FONTE: [SIL2000a]

A seguir, aplica-se o passo 1 descrito na tabela 4.1 para transformar a expressão regular em um grafo de transições. Tal procedimento é demonstrado na figura 4.1.

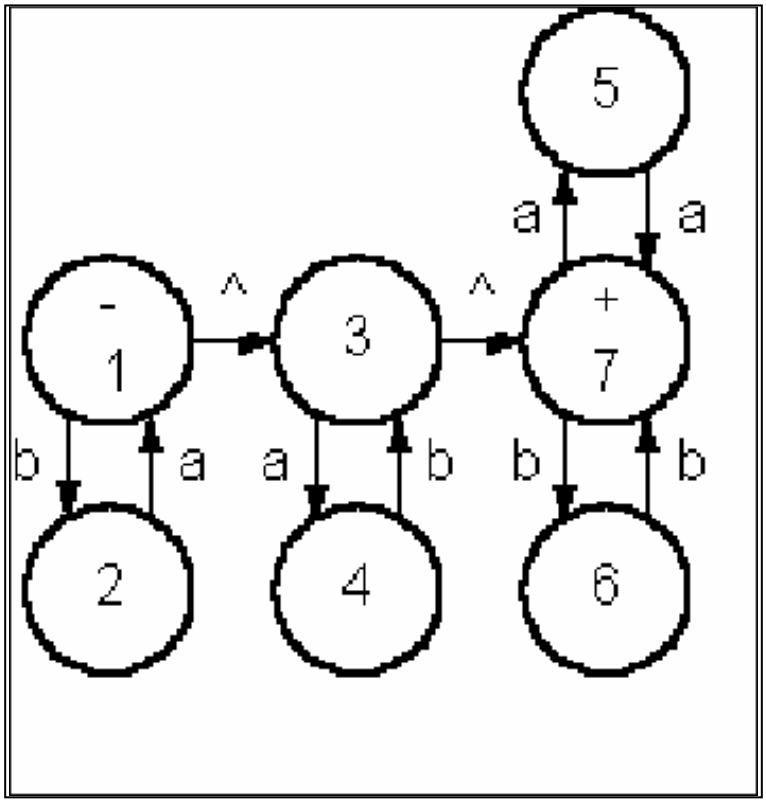
FIGURA 4.1 - APLICAÇÃO DO PASSO 1 DA TABELA 4.1 PARA A EXPRESSÃO DO QUADRO 4.2



FONTE: [SIL2000a]

A figura 4.1 apresenta o grafo de transições, o qual pode ser simplificado, como mostra a figura 4.2.

FIGURA 4.2 – GRAFO SIMPLIFICADO DA FIGURA 4.1



FONTE: [SIL2000a]

A seguir, é aplicado o item b do quadro 4.1, resultando a tabela de transições descrita na tabela 4.2.

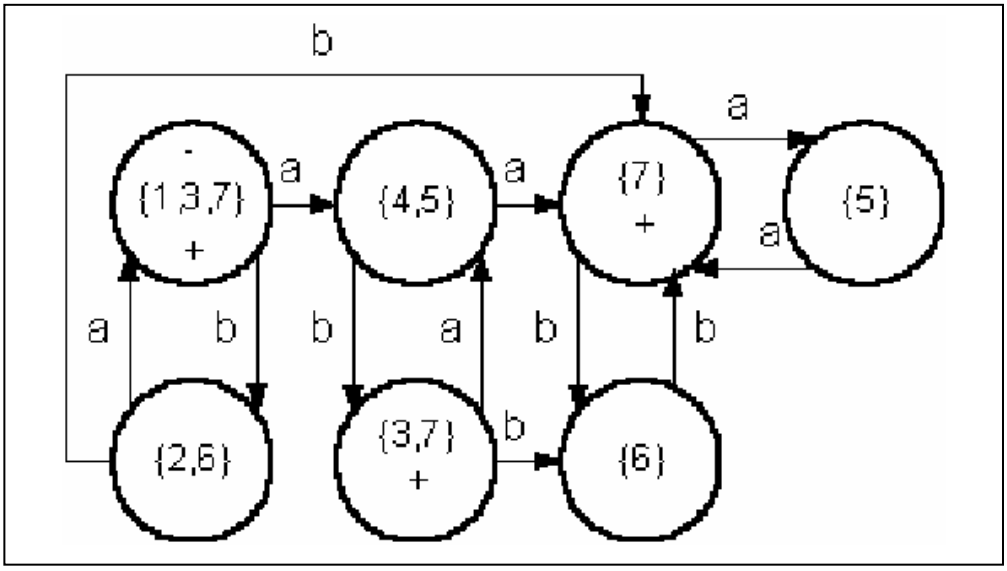
TABELA 4.2 – TABELA DE TRANSIÇÕES

Nó Autômato Finito	Mi	Mj	
		a	b
S1 (– +)	{1,3,7}	{4,5}	{2,6}
S2	{4,5}	{7}	{3,7}
S3	{2,6}	{1,3,7}	{7}
S4 (+)	{7}	{5}	{6}
S5 (+)	{3,7}	{4,5}	{6}
S6	{5}	{7}	{ }
S7	{6}	{ }	{7}

FONTE: [SIL2000a]

Após, é aplicado o item c do quadro 4.1, conseguindo-se assim o autômato finito determinístico, como mostrado na figura 4.3.

FIGURA 4.3 – AUTÔMATO FINITO GERADO A PARTIR DA TABELA 4.2



FONTE: [SIL2000a]

Para transformar a expressão regular num grafo determinístico, anteriormente transformou-se a expressão regular em uma notação pós-fixada. A especificação de uma expressão regular e ações para transformá-la em uma pós-fixada é mostrado na seção 2.4.

Uma vez tendo a expressão regular pós-fixada, a mesma é transformada em um grafo de transições para o qual foi utilizado o algoritmo do desmonte [SIL2000a] o qual é apresentado na seção 4.2.

4.1 DEFINIÇÃO DA BNF PARA EXPRESSÕES REGULARES

[SIL2000a] apresenta a definição formal de expressões regulares utilizando a linguagem livre do contexto. A definição do quadro 4.3 já está normalizada (sem recursividade à esquerda e já foi fatorada).

Através da definição, é criado uma forma pós-fixada da expressão regular, através da ação semântica **EMITIR**. Um exemplo de uma expressão regular (a) e sua forma equivalente pós-fixada (b) são mostrados no quadro 4.4.

QUADRO 4.3 – DESCRIÇÃO ATRAVÉS DE UMA LINGUAGEM LIVRE DO CONTEXTO DE EXPRESSÕES REGULARES

```
^  →   símbolo vazio
R  →   expressão regular
SR →   simples expressão regular
RR →   resto da expressão regular

R  →   SR RR;

RR →   '+' SR
   |   '.' SR
   |   ^;

SR →   T SR1

SR1→   '+' T SR1     [EMITIR '+'];
   |   ^;

T  →   F T1;

T1 →   '.' F T1      [EMITIR '.']
   |   ^;

F  →   '(' R ')' F1
   |   #S F1         [EMITIR 'Simbolo'];

F1 →   '*' F1        [EMITIR '*']
   |   ^;
```

FONTE: [SIL2000a]

4.2 APLICAÇÃO DO ALGORITMO DO DESMONTE E TRANSFORMAÇÃO DO GRAFO EM AUTÔMATOS FINITO DETERMINÍSTICO

Uma vez transformada a expressão regular infixada em pós-fixada, o que é feito baseado na definição através de gramática livre de contexto e a ação semântica **EMITIR**, monta-se uma tabela para construção do grafo. Baseado no algoritmo do desmonte proposto por [SIL2000a]. Ainda nesta seção, é também mostrado a transformação do grafo de transições em um grafo determinístico (autômato finito determinístico).

A seguir, será detalhado o funcionamento do algoritmo do desmonte exemplificando o seu uso através da expressão regular mostrada no quadro 4.4.

QUADRO 4.4 – EXPRESSÃO REGULAR INFIXA (a) E PÓS-FIXA (b)

(a)	**(b)**
a (b + c)* + ca* + ^	**a b c + * . c a * . + ^ +**

Fonte: [SIL2000a]

O grafo para o exemplo da quadro 4.4 (b) possui 6 nós (2 nós, sendo um para o nó inicial e outro para o nó final e mais 1 nó para cada operador "*" e "."). Este número é determinado conforme o Teorema de Kleene, quando mostra o algoritmo de transformação de uma expressão regular em um grafo de transições, conforme pode ser visto no passo 1 descrito na tabela 4.1.

A tabela para construção do grafo possui o formato descrito na tabela 4.3. Esta tabela é iniciada marcando-se na primeira coluna no campo *nó de saída* o nó inicial e na coluna *próximo nó* o nó final. Para a expressão da quadro 4.4 (b) o nó inicial é 1 e o nó final é 6.

Após, monta-se a tabela, começando a leitura da expressão regular na forma pós-fixada de trás para frente. Na quadro 4.4 (b), o primeiro elemento a ser lido é o símbolo "+", após os símbolos "^", "+", ".", "*", "a", "c", ".", "*", "+", "c", "b" e por último o símbolo "a". Estes símbolos são acrescentados na tabela, sempre na posição livre da segunda coluna (símbolo), pesquisando-se do final da tabela para o início.

TABELA 4.3 – FORMATO DA TABELA PARA CONSTRUÇÃO DO GRAFO A PARTIR DA EXPRESSÃO REGULAR PÓS-FIXADA

NÓ DE SAÍDA	SÍMBOLO	PRÓXIMO NÓ
1		6

FONTE: [SIL2000a]

Para os símbolos que são operadores ("+", "." e "*") acrescentam-se novas arestas no final da tabela.

Para o símbolo “+” acrescenta-se duas novas arestas, sendo que os *nós de saídas* é o nó de saída do último símbolo acrescentado e os *próximos nós* é o próximo nó do último símbolo acrescentado.

Para o símbolo “.” acrescenta-se também duas novas arestas (linhas na tabela), sendo que a primeira linha o *nó de saída* é o nó de saída do último símbolo acrescentado e o *próximo nó* o próximo nó disponível no grafo (no exemplo, o próximo nó disponível no início é o 2). A segunda linha, o *nó de saída* é o próximo nó da aresta anterior incluída e o *próximo nó* é o próximo nó do último símbolo acrescentado. O campo *símbolo* da aresta é deixado livre.

Para o símbolo “*” acrescenta-se três novas arestas (linhas na tabela). A primeira linha o *nó de saída* é o nó de saída do último símbolo acrescentado e o *próximo nó* o próximo nó disponível no grafo. A segunda linha, o *nó de saída* é o próximo nó da aresta anterior incluída e o *próximo nó* é o próximo nó do último símbolo acrescentado. O campo *símbolo* destas duas arestas são preenchidos com o símbolo vazio (^). A terceira aresta, o *nó de saída* e o *próximo nó* são o próximo nó da primeira aresta incluída. O campo *símbolo* da aresta, é deixado livre.

Um exemplo com a tabela completa é mostrado na tabela 4.4, criada a partir da expressão pós-fixada mostrada na quadro 4.4 (b).

Após construída a tabela do grafo, ignoram-se as linhas nas quais possuem no campo *símbolo* operadores. As linhas restantes são arestas do grafo. O nó inicial do grafo é sempre o *nó de saída* da primeira linha (no exemplo da tabela 4.4 é o nó 1) e o nó final também é sempre o *próximo nó* da primeira linha (no exemplo da tabela 4.4 é o nó 6).

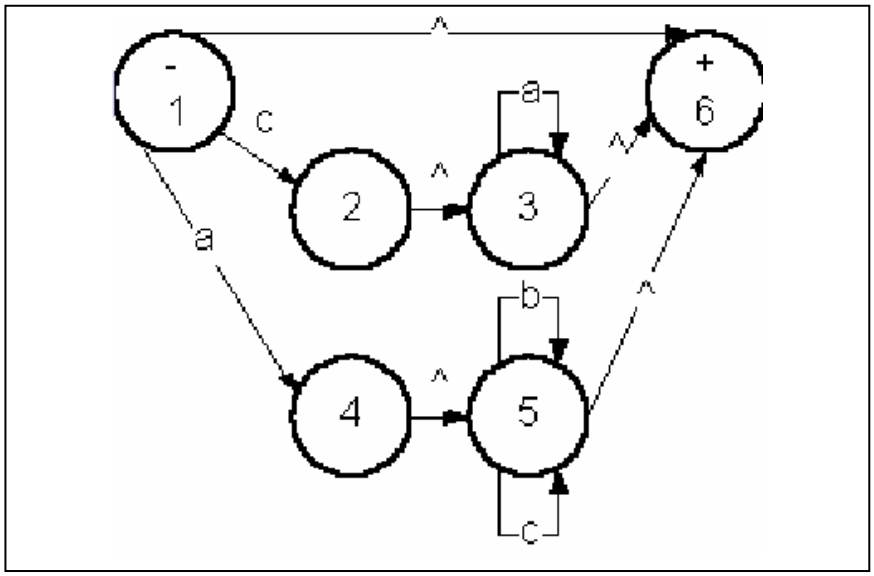
TABELA 4.4 – FORMATO DA TABELA COMPLETA PARA CONSTRUÇÃO DO GRAFO A PARTIR DA EXPRESSÃO REGULAR PÓS-FIXADA

NÓ DE SAÍDA	SÍMBOLO	PRÓXIMO NÓ
1	**+**	**6**
1	**+**	**6**
1	**^**	**6**
1	**.**	**6**
1	**.**	**6**
1	**C**	**2**
2	*****	**6**
2	**^**	**3**
3	**^**	**6**
3	**A**	**3**
1	**A**	**6**
4	*****	**3**
4	**^**	**4**
5	**^**	**6**
5	**+**	**5**
5	**B**	**5**
5	**C**	**5**

FONTE: [SIL2000a]

A figura 4.4 apresenta o grafo de transições, construído a partir da tabela mostrada na tabela 4.4.

FIGURA 4.4 – GRAFO DE TRANSIÇÕES CONSTRUÍDO A PARTIR DA TABELA APRESENTADA NA TABELA 4.4



FONTE: [SIL2000a]

A seguir, é aplicado o item b do passo 2, do algoritmo de Kleene apresentado no quadro 4.1, resultando a tabela de transições descrita na tabela 4.5.

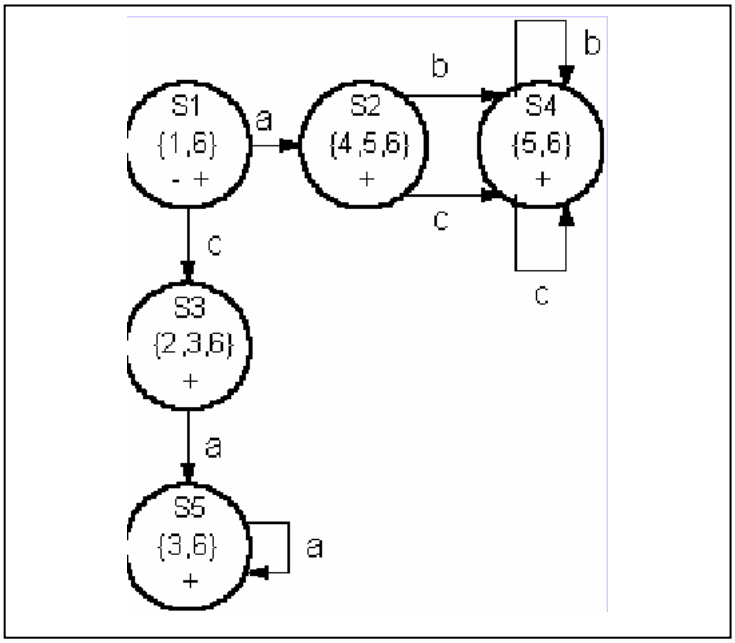
TABELA 4.5 – TABELA DE TRANSIÇÕES

Nó Autômato Finito	Mi	Mj		
		A	b	c
S1 (– +)	{1,6}	{4,5,6}	{}	{2,3,6}
S2 (+)	{4,5,6}	{}	{5,6}	{5,6}
S3 (+)	{2,3,6}	{3,6}	{}	{}
S4 (+)	{5,6}	{}	{5,6}	{5,6}
S5 (+)	{3,6}	{3,6}	{}	{}

FONTE: [SIL2000a]

Após é aplicado o item c do algoritmo de Kleene apresentado no quadro 4.1, conseguindo-se assim o autômato finito determinístico, como mostrado na figura 4.5.

FIGURA 4.5 – AUTÔMATO FINITO MONTADO A PARTIR DA TABELA 4.5



FONTE: [SIL2000a]

5 DESENVOLVIMENTO DO PROTÓTIPO

A seguir será apresentado o protótipo, mostrando a sua especificação, implementação, e o seu uso.

5.1 ESPECIFICAÇÃO DO PROTÓTIPO

Na figura 5.1 é apresentado o fluxograma geral do processo de transformação de uma linguagem livre do contexto em uma expressão regular estendida e o desmonte da mesma.

FIGURA 5.1 – FLUXOGRAMA GERAL

Conforme fluxograma apresentado na figura 5.1, os seguintes itens são abordados:

a) edição das produções: será utilizado o componente TRichEdit do Delphi para edição e gravação das produções. Serão utilizados as caixas de diálogo padrões do windows para abertura e gravação dos arquivos de produção, não necessitando assim, apresentar as mesmas;
b) verificação sintática: este processo consiste em analisar as produções digitadas (vide figura 5.6), alocando-as em uma lista encadeada para utilização posterior. Caso seja encontrado algum erro de sintaxe nas produções informadas será aberto uma nova janela apresentando e indicando a posição do erro e o seu motivo como mostrado na figura 5.7;
c) conferência de declaração de produções: a partir de uma busca na lista encadeada das produções informadas, todos os tokens *não-terminais*, deverão possuir uma única produção correspondente, caso isso não ocorra, o processo será abortado e uma janela (conforme figura 5.8) contendo a relação de inconsistências encontradas será apresentada;
d) montagem da expressão regular: consiste em substituir todos os não-terminais por suas correspondentes declarações, resultando em uma única expressão regular, devido ao fato de que produções podem possuir recursividade, e ao retirá-las as produções resultantes poderão conter parênteses, optou-se por guardar as produções simultaneamente em uma estrutura de *árvore n-ária* (com vários filhos) para facilitar a retirada da recursividade. A estrutura de representação da árvore pode ser vista na figura 3.1. O processo de retirada da recursividade é apresentado no capítulo 3;
e) desmonte da expressão regular: após a expressão regular for montada ela será submetida ao algoritmo do desmonte proposto por [SIL2000b] (mostrado na seção 4.2), onde primeiramente a expressão será transformada em uma forma pós-fixada (esta é baseada em uma definição de uma BNF para expressões regulares, mostrada na seção 4.1), gerando após, uma representação em forma de tabela de um autômato finito determinístico para a expressão regular passada, conforme mostrado na figura 5.10.

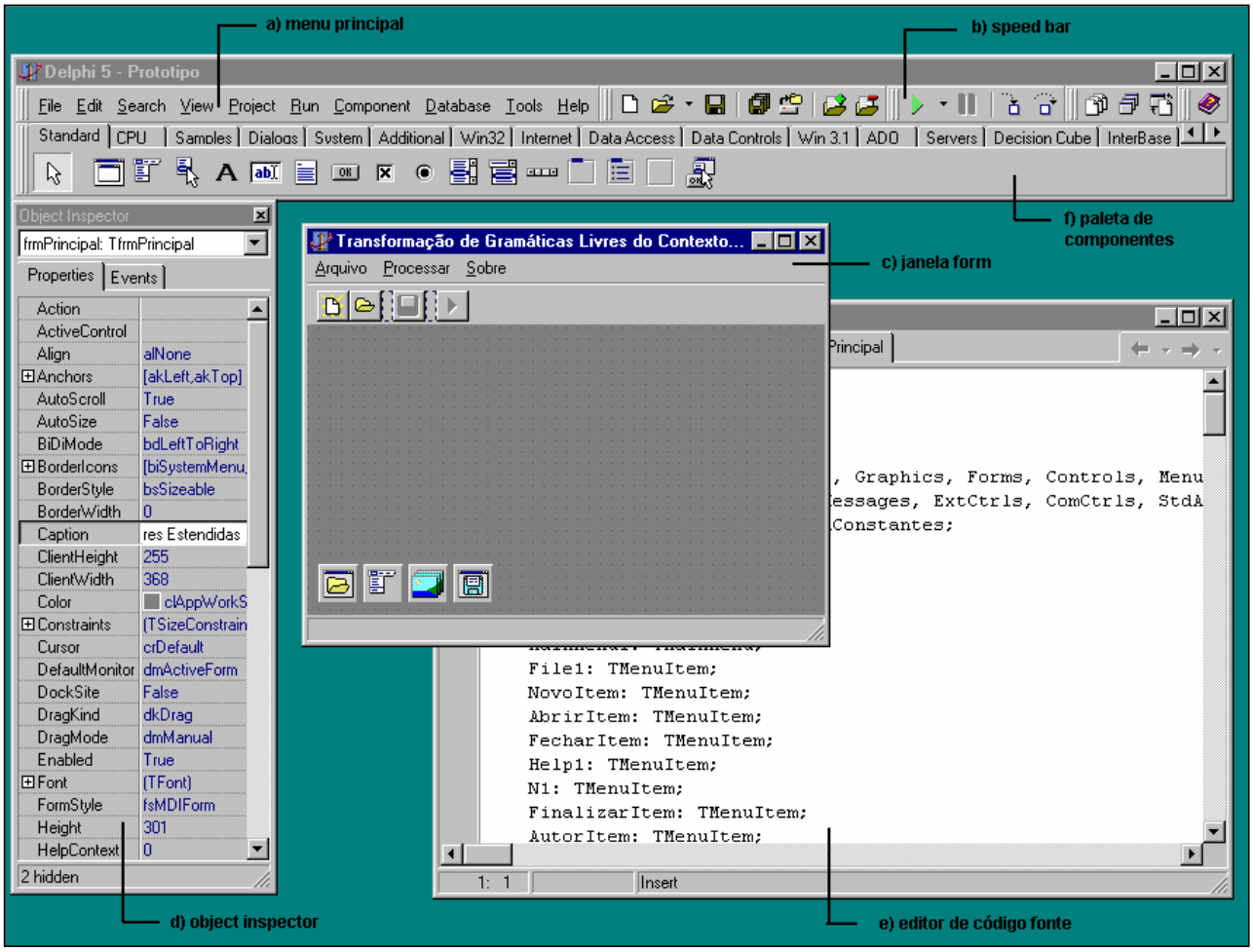
5.2 AMBIENTE DE DESENVOLVIMENTO

Para o desenvolvimento deste protótipo foi utilizado a ferramenta Borland Delphi 5.0. Seu uso deve-se ao fato de permitir a criação rápida de um aplicativo no padrão Windows e ser ferramenta que utiliza, como linguagem de programação, o *Object Pascal* descendente da linguagem de programação *Pascal*, fortemente utilizada em meio acadêmico.

As principais janelas do Delphi são mostradas na figura 5.2 e explanadas individualmente ([JOH1997]):

a) menu principal: dá acesso a todo o conjunto de ferramentas que auxiliam no desenvolvimento, depuração, gerenciamento de componentes;
b) *speed bar*: ou botões de velocidade são atalhos para as funções utilizadas com maior freqüência;
c) janela *form*: é a janela em que são colocados os componentes visuais do Delphi e modelados para construir a interface da maneira que o desenvolvedor desejar;
d) *object inspector*: exibe as propriedades e eventos de componentes ou do formulário selecionado que podem ser definidas em tempo de projeto;
e) editor de código fonte: permite ao desenvolvedor escrever seu código na linguagem *Object Pascal*;
f) paleta de componentes VCL (*Visual Component Library*): contém os componentes utilizados para desenvolvimento de uma aplicação padronizada com o ambiente Windows, também são encontrados componentes usados para trabalhar com imagens, banco de dados, geração de relatórios, geração confecção de gráficos etc.

FIGURA 5.2 – AMBIENTE DE DESENVOLVIMENTO DELPHI 5.0



5.3 APRESENTAÇÃO DO PROTÓTIPO

A execução do protótipo tem início com a abertura da janela principal e da janela de apresentação, mostrada na figura 5.3.

A janela principal possui um menu com todas as funções necessárias para operar o protótipo, que também podem ser acionadas através da barra de botões da janela, mensagens informativas sobre cada botão ou opção do menu são visualizadas na barra de *Status*, conforme figura 5.4.

As funções para manipulação dos arquivos de produção e outras opções do menu principal são mostradas na figura 5.5.

FIGURA 5.3 – TELA DE APRESENTAÇÃO

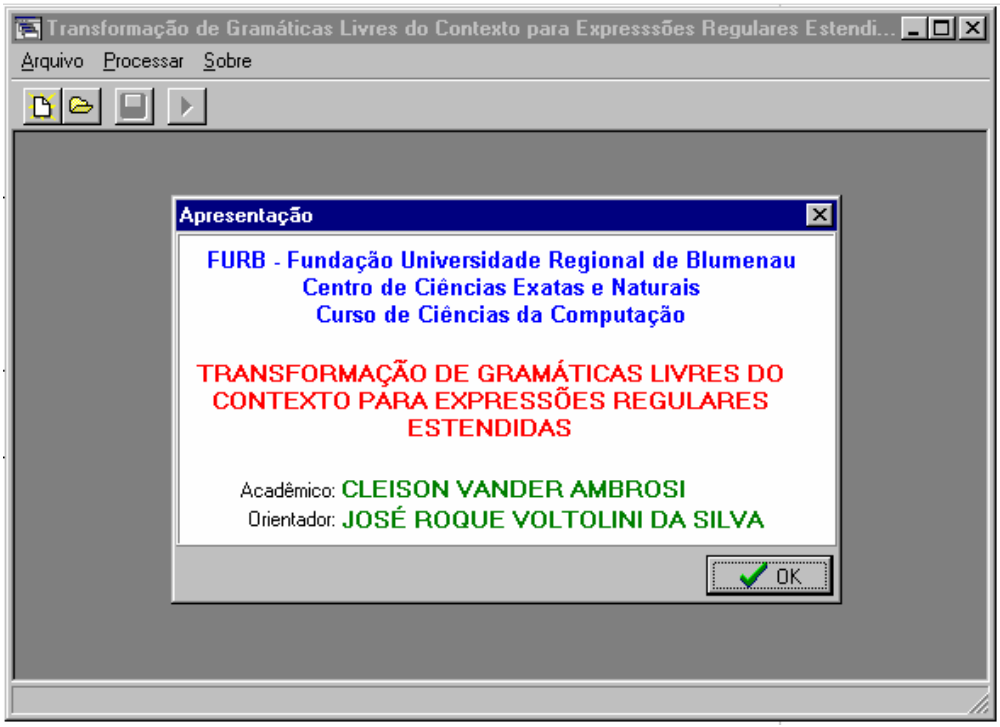


FIGURA 5.4 – TELA PRINCIPAL

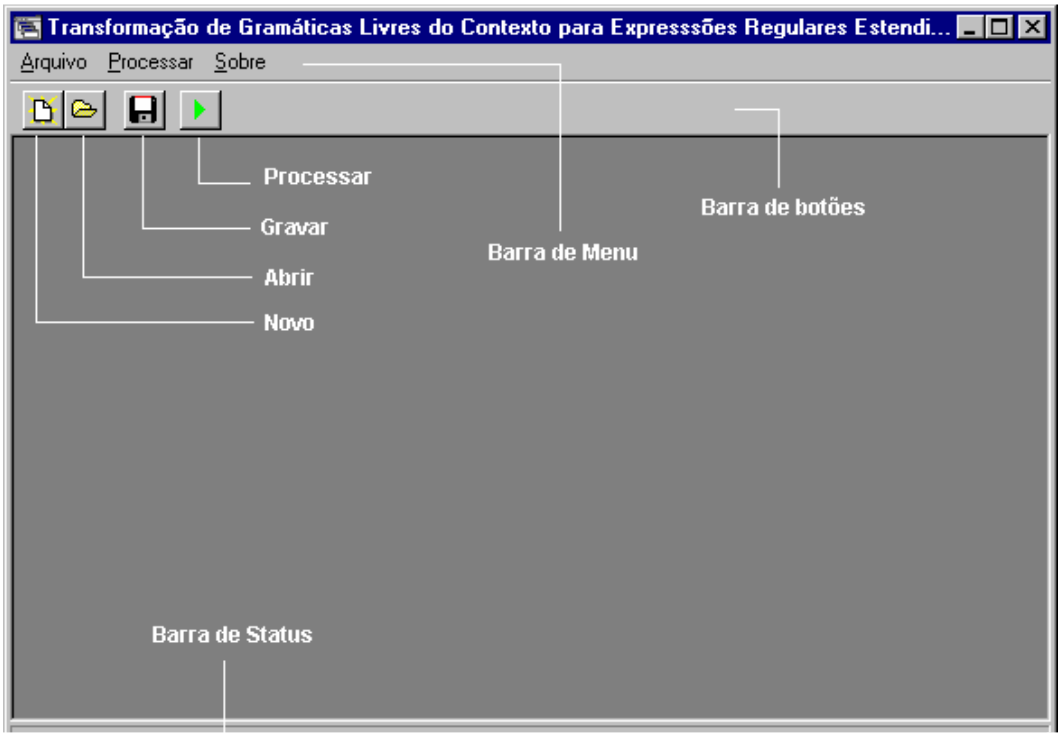
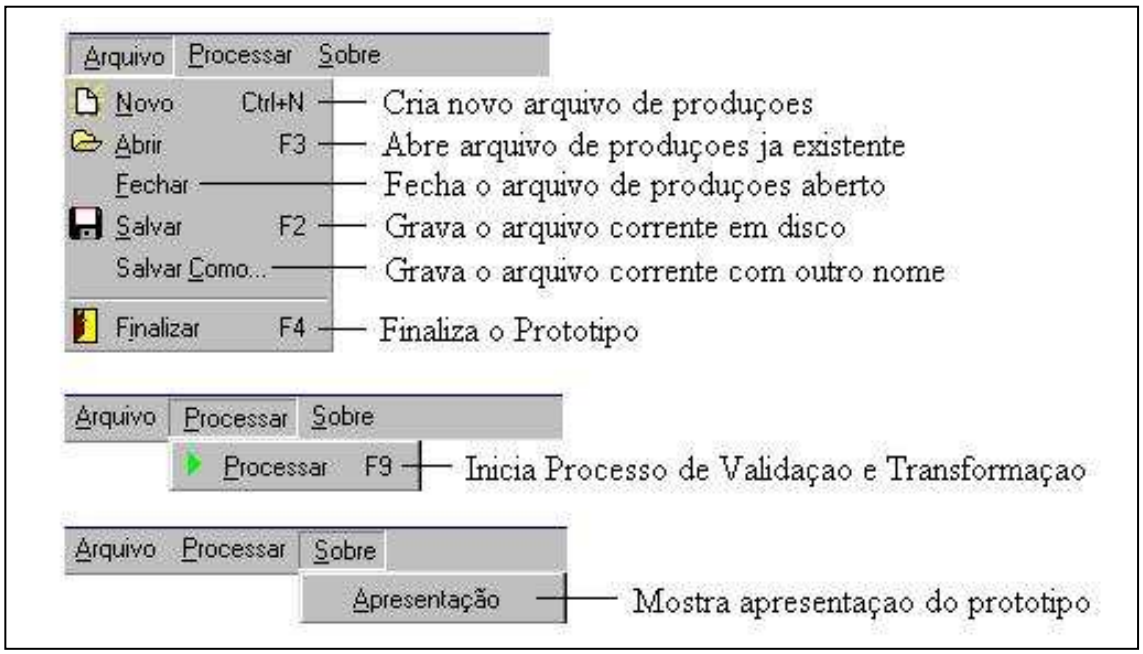
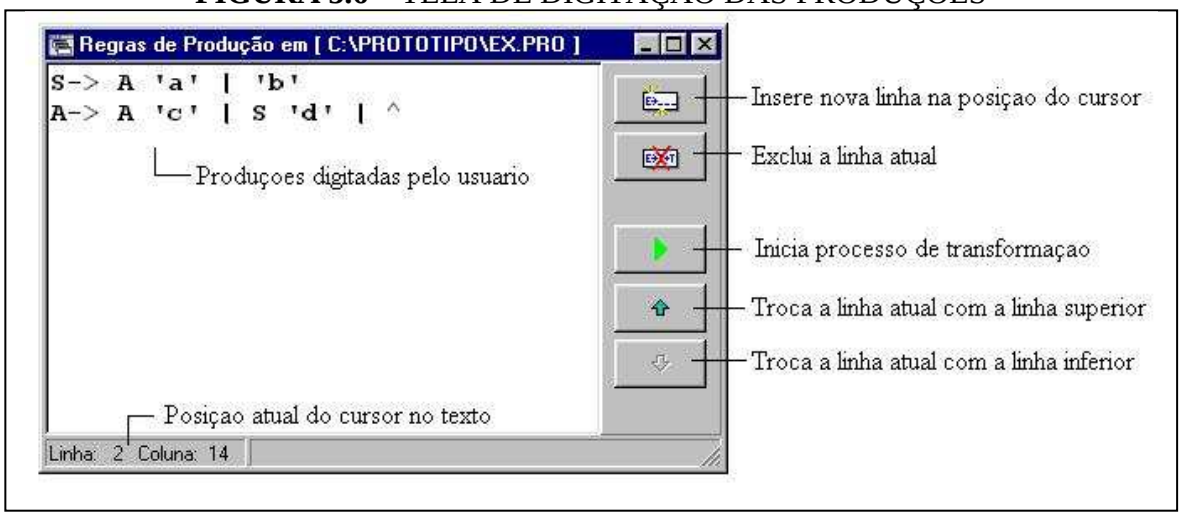


FIGURA 5.5 – DESCRIÇÃO DOS ITENS DO MENU



O protótipo permite que as produções a serem transformadas sejam digitadas na janela de digitação das produções, conforme a figura 5.6.

FIGURA 5.6 – TELA DE DIGITAÇÃO DAS PRODUÇÕES



Caso for encontrado algum erro ou inconsistência nas produções digitadas, serão apresentadas as telas de erro ou de inconsistência conforme figuras 5.7 e 5.8, respectivamente.

FIGURA 5.7 – RELAÇÃO DE ERROS ENCONTRADOS

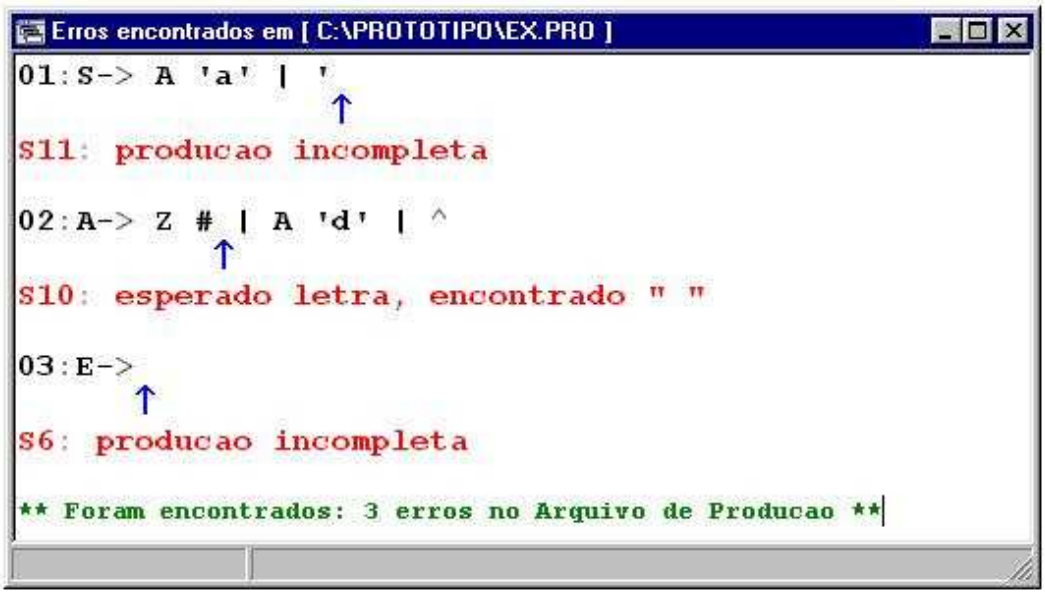
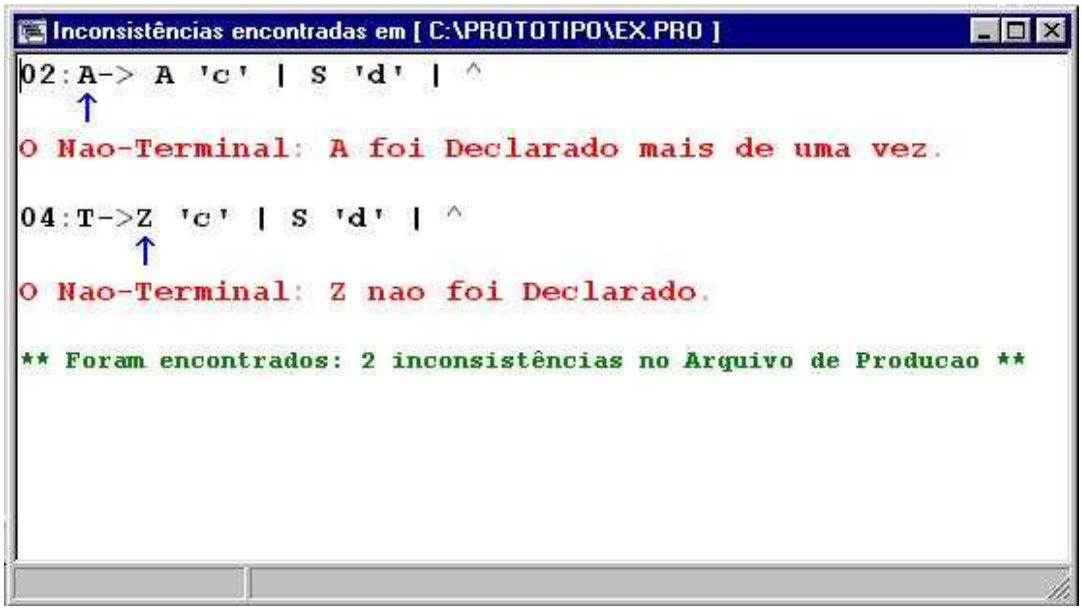
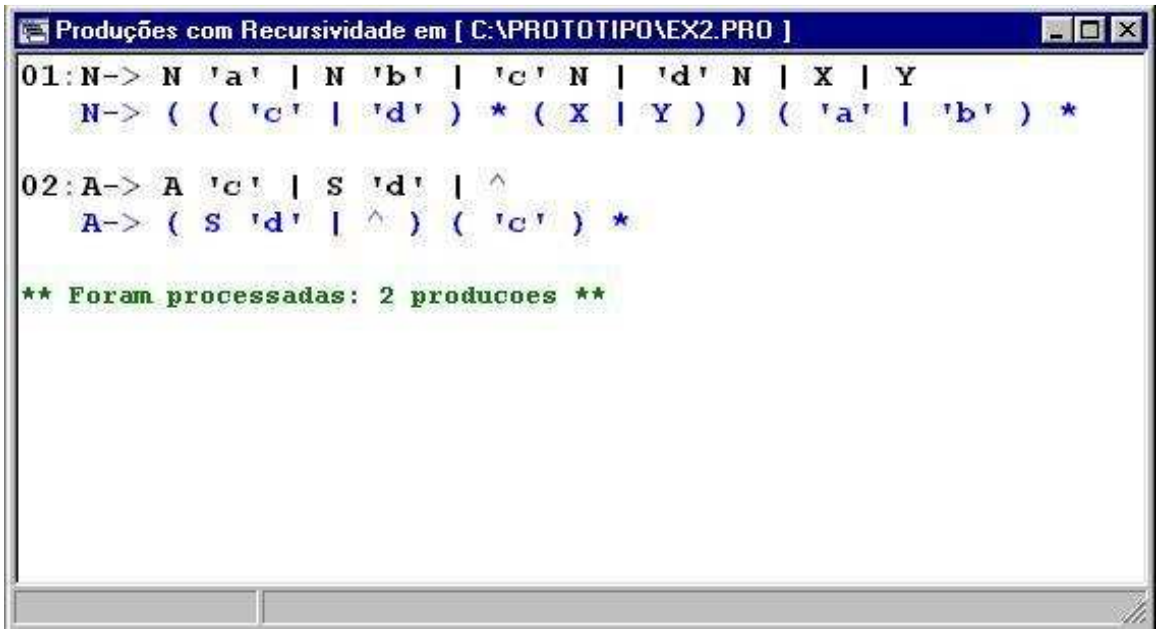


FIGURA 5.8 – RELAÇÃO DE INCONSISTÊNCIAS



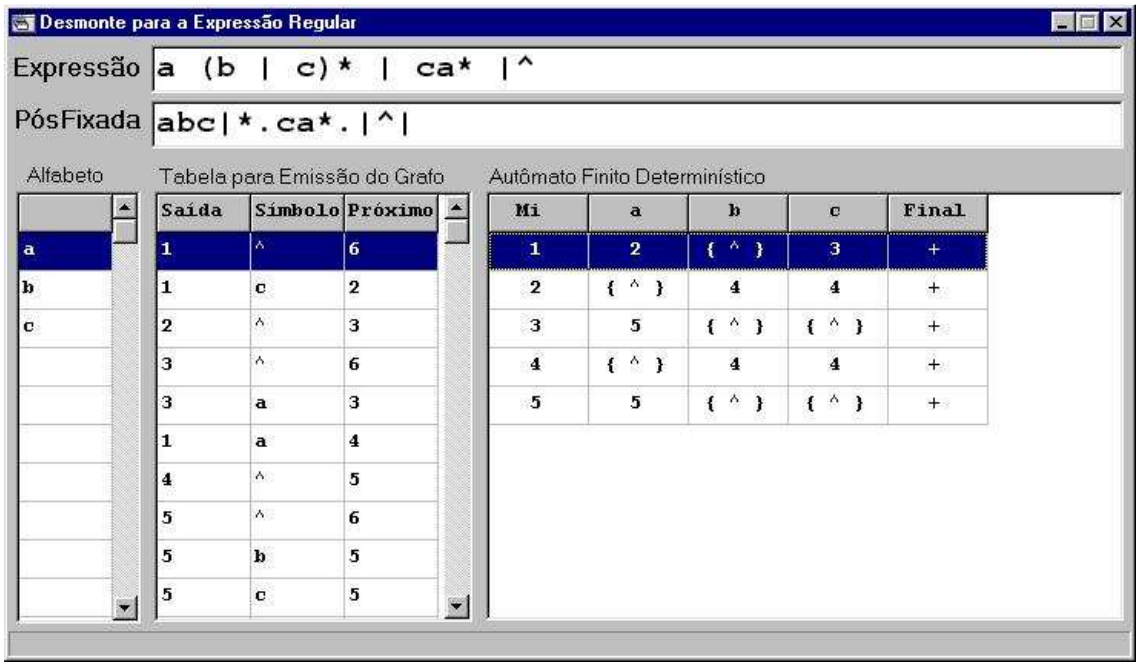
Depois das produções serem validadas, o processo de transformação em expressão regular estendida e da retirada da recursividade (se houver) são iniciados, e uma tela de informação com as produções na sua forma original e na forma sem recursividade é mostrada, conforme a figura 5.9.

FIGURA 5.9 – TELA DE PRODUÇÕES COM RECURSIVIDADE



A tela final do processo de transformação do protótipo (figura 5.10), mostra a expressão regular estendida, sua forma pós-fixada, o alfabeto, a tabela gerada pelo algoritmo de desmonte (tabela para emissão do grafo) e o autômato finito determinístico para a expressão passada. Tais processos foram abordados no capítulo 4.

FIGURA 5.10 – TELA DE APRESENTAÇÃO DO ALGORITMO DO DESMONTE



6 CONCLUSÃO

Este trabalho atingiu o objetivo proposto, o qual era a transformação de uma linguagem livre do contexto em uma expressão regular estendida. Verificou-se durante o desenvolvimento, que para tal transformação, era necessário retirar todos os tipos de recursividade (auto-embutidas e não-auto-embutidas), o que inicialmente não foi previsto, mas teve que ser realizado para cumprir com os objetivos.

Ainda, informações para retirada da recursividade auto-embutida, foi somente encontrada em [SIL2000b], sendo que este documento ainda está em fase de elaboração e alguns problemas foram encontrados, os quais foram solucionados juntamente com o autor.

Para atingir o objetivo de transformar uma linguagem livre do contexto para uma expressão regular foram introduzidos dois novos operadores de iteração (‘~’ e ‘ ” ’), modificando suas características originais e tornando-a estendida.

Foi utilizado o algoritmo do desmonte para transformar a expressão regular estendida em um grafo, a partir do qual foi feito a transformação para um autômato finito determinístico. Frisa-se que esta transformação para autômato finito determinístico não leva em consideração ainda os novos dois operadores de iteração introduzidos na expressão regular (estendida). A falta de tal informação impossibilita a transformação do autômato finito determinístico em um programa de computador de forma automática.

Para as duas classificações de linguagem (regulares e livres do contexto) obtém-se duas notações diferentes para representá-las. Com este trabalho é proposto uma única notação para representar os dois tipos de linguagem.

O protótipo pode servir como ferramenta para fazer as normalizações das estruturas sintáticas (recursividade, fatoração). Essas estruturas sintáticas são usadas principalmente para definições de compiladores e essas definições são normalizadas para torná-las implementáveis.

6.1 LIMITAÇÕES

A prova formal das transformações de linguagens livres do contexto com produções auto-embutidas, não foi feita. A prova das não-auto-embutidas pode ser vista em [WAT1991].

Ainda, como já mencionado acima (conclusão), a transformação do autômato finito determinístico em programa quando da existência de produções com recursividade auto-embutida não foi realizada (limitação de tempo). Em [SIL2000b] é feito um ensaio.

O protótipo suporta grafos determinísticos de no máximo 255 nós, devido a limitação do tipo de dados (*set)* do Pascal.

6.2 EXTENSÕES

Como proposta para trabalhos futuros, sugere-se a ampliação do número de nós possíveis para um grafo determinístico no protótipo, o que pode ser resolvido com a utilização de listas encadeadas alocadas dinamicamente.

Ainda, a validação de linguagens livres do contexto que possuem recursividade auto-embutida quando da transformação para expressões regulares estendidas (que usam os operadores ‘~’ e ‘ ” ’) não foi feita de maneira formal, sendo que um método para tal verificação é sugerido como extensão deste trabalho para dar uma maior segurança nos métodos aplicados.

Também, quando da transformação da expressão regular estendida para autômato finito determinístico, não se considerou os operadores ‘~’ e ‘ “ ’, o que precisa ser feito para automatizar a transformação do autômato finito determinístico em um programa que possua função equivalente.

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO1995] AHO, Alfred V, SETHI, Ravi, ULMAN, Jeffrey D**. Compiladores, princípios, técnicas e ferramentas**. Trad. Daniel de Ariosto Pinto. Rio de Janeiro : Livros Técnicos e Científicos, 1995.

[DIV1999] DIVÉRIO, Tiajarú Asmuz. **Teoria da computação: máquinas universais e computabilidade**. Porto Alegre : Sagra Luzzatto, 1999.

[GER1995] GERSTING, Judith L. **Fundamentos matemáticos para ciência da computação.** 3.Ed. Rio de Janeiro : Livros Técnicos e Científicos. Editora, 1995.

[HUN1987] HUNTER, Robin. **Compiladores - sua concepção e programação em pascal.** Lisboa : Editorial Presença, 1987.

[JOH1997] JOHN, Paul Mueller. **Guia para o Delphi 2**. Trad. João Eduardo Nóbrega Tortello. São Paulo : Makron Books, 1997.

[JOS1987] JOSÉ NETO, João. **Introdução à compilação**. Rio de Janeiro : Livros Técnicos e Científicos, 1987.

[LEW2000] LEWIS, Harry R., PAPADIMITRIOU, Christos H. **Elementos de teoria da computação.** Porto Alegre : Bookman, 2000.

[MAN1974] MANNA, Zohar. **Mathematical theory of computation.** New York : McGraaw-Hill, 1974.

[MEN1998] MENESES, Paulo Fernando Blauth. **Linguagens formais e autômatos**. Porto Alegre : Sagra Luzzatto, 1998.

[SIL1999] SILVA, José Roque Voltolini da. **Computabilidade: Roteiro de aula da disciplina de Teoria da Computação.** Blumenau, 1999. Curso de Ciências da Computação, Universidade Regional de Blumenau .

[SIL2000a] SILVA, José Roque Voltolini da**. Proposta de um novo algoritmo para transformação de uma Expressão Regular em um Autômato Finito Determinístico**. Artigo não publicado, 2000. Universidade Regional de Blumenau.

[SIL2000b] SILVA, José Roque Voltolini da**. Uma extensão proposta para o teorema de Kleene para aplicação em linguagens livres do contexto objetivando a eliminação de indeterminismos**. Artigo não publicado, 2000. Universidade Regional de Blumenau.

[WAT1991] WATT, David A. **Programming language syntax and semantics.** New York : Prentice Hall, 1991.