PIER MARCO RICCHETTI

GERAÇÃO DE ANALISADORES SINTÁTICOS BASEADOS EM TRANSDUTORES A PARTIR DE ESPECIFICAÇÕES GRAMATICAIS

Dissertação de Mestrado apresentada à
Escola Politécnica da Universidade de
São Paulo para a obtenção do título de
Mestre em Engenharia

São Paulo
2005

PIER MARCO RICCHETTI

GERAÇÃO DE ANALISADORES SINTÁTICOS BASEADOS EM TRANSDUTORES A PARTIR DE ESPECIFICAÇÕES GRAMATICAIS

Dissertação de Mestrado apresentada à
Escola Politécnica da Universidade de
São Paulo para a obtenção do título de
Mestre em Engenharia.

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr.
João José Neto

São Paulo
2005

FICHA CATALOGRÁFICA

Ricchetti, Pier Marco
Geração de analisadores sintáticos baseados em transdutores a partir de especificações gramaticais / P.M. Ricchetti. -- São Paulo, 2005.
146 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Análise sintática 2.Autômatos finitos 3.Gramática I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Aos meus pais,
Alfredo Ricchetti e
Sandra Maria Ricchetti

AGRADECIMENTOS

Ao grande amigo e orientador Prof. Dr. João José Neto, pela contribuição e incentivo inestimáveis na realização deste trabalho.

Aos colegas do LTA - Laboratório de Linguagens e Tecnologias Adaptativas e do Departamento de Engenharia de Computação e Sistemas Digitais.

Aos meus colegas professores, cujo apoio, incentivo e compreensão foram fundamentais para o desenvolvimento desta dissertação.

Aos meus alunos e a todos aos quais eu possa transmitir o meu conhecimento, principal razão de meu empenho.

A todos os que contribuíram, de certa forma, para que este trabalho se tornasse possível.

RESUMO

Neste trabalho, procura-se dar uma contribuição à área de linguagens formais a partir da proposta do desenvolvimento de técnicas associadas ao tratamento de linguagens, cujas gramáticas estão associadas a ações, e que permitirão automatizar a construção de analisadores sintáticos para tais linguagens. Adicionalmente, desenvolve-se uma ferramenta prática de auxílio à aplicação destas técnicas e de visualização dos passos que a compõem, e que produz como saída um analisador sintático baseado em autômatos de pilha estruturados, para a utilização na ferramenta *Adaptools*, desenvolvida no LTA – Laboratório de Linguagens e Tecnologias Adaptativas – do Departamento de Engenharia da Computação e Sistemas Digitais da Escola Politécnica da USP. O cunho didático da ferramenta e de seu desenvolvimento possibilita a sua utilização no ensino de técnicas adaptativas, de linguagens formais, de autômatos e de compiladores. Adicionalmente, uma breve discussão sobre a alteração das técnicas propostas, com o objetivo de adequá-las a gramáticas adaptativas, é feita ao final deste trabalho.

ABSTRACT

This work shows a contribution to the formal language field through the development of technical processes applied to grammars including semantic actions. Such processes and algorithms allow the construction of parsers for context-free languages with semantic actions. A practical tool implementing the proposed method is also presented. This tool shows the method by its steps, so it can be used as an educational tool. Moreover, due the application of proposed algorithms, a syntactic parser is automatically generated for use with *Adaptools*, a tool developed in LTA – Laboratório de Linguagens e Tecnologias Adaptativas – a division of the Computer Engineering and Digital Systems Department, at São Paulo University. Finally, as a further research proposal, considerations are made about modifying and improving the presented algorithms regarding their application to adaptive grammars.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1 - INTRODUÇÃO

A implementação de *parsers* sintáticos é uma etapa importante na construção de compiladores. Existem diversos métodos e algoritmos para a construção de *parsers* sintáticos, como pode ser observado por exemplo em (TREMBLAY; SORENSON, 1985), (APPEL, 1997) (PRICE; TOSCANI, 2004), (AHO; ULLMAN, 1972), (AHO; ULLMAN, 1973). Neste trabalho apresenta-se uma contribuição sobre o método apresentado em (NETO, 1993), e posteriormente modificado em (IWAI, 2000).

1.1 Apresentação

Os Analisadores Sintáticos ou *parsers*[1] permitem a análise de sentenças fornecidas quanto à sua adequação às regras gramaticais. Em (NETO, 1993) é proposto um método para a construção automática de reconhecedores sintáticos para gramáticas livres de contexto e também uma primeira extensão deste método de forma a permitir a geração de árvores sintáticas durante o reconhecimento de sentenças.

Esta dissertação analisa e transforma este método com o objetivo de adaptá-lo para a inclusão de ações semânticas, e para a construção de uma ferramenta de manipulação gramatical e geração de analisadores sintáticos. Esta ferramenta auxilia a aplicação e visualização dos passos de manipulação gramatical, e produz um *parser* ou analisador sintático baseado no formato do *Adaptools*, desenvolvido no LTA – Laboratório de Linguagens e Tecnologias Adaptativas do Departamento de Engenharia da Computação e Sistemas Digitais da Escola Politécnica da USP.

---

[1] A palavra *parser*, de origem inglesa é largamente utilizada em literatura técnica e se refere a *analisador sintático*.

1.2 Objetivos e motivação

Em (NETO, 1993) são propostos métodos de manipulação gramatical que, quando aplicados a uma gramática, devem gerar como resultado um transdutor de pilha com um número mínimo de ocorrências de não-determinismos, e que atua como analisador sintático desta gramática. A redução de não-determinismos facilita a construção do transdutor e contribui para a eficiência do reconhecimento de sentenças válidas para a gramática em questão.

O transdutor obtido tem duas funções: realizar a análise sintática sobre uma sentença fornecida, e gerar, como saída, uma árvore de derivação sintática que represente o reconhecimento desta sentença.

Na época do início deste trabalho, constatou-se a inexistência de ferramentas que implementassem os recursos apontados em (NETO, 1993) e que permitissem a visualização didática das fases de sua execução, gerando automaticamente os analisadores sintáticos desejados.

Neste trabalho estudam-se os métodos e técnicas propostos em (NETO,1993), e incluem-se ações semânticas na gramática fornecida como entrada. Estas ações fazem parte do processo de transformação, sendo indicadas como elementos que compõem a cadeia que representa a árvore de derivação final.

Adicionalmente, são propostos algoritmos que permitam a aplicação destes métodos, e uma ferramenta é desenvolvida para auxiliar o processo de construção de parsers sintáticos.

A utilização de uma ferramenta de apoio reduz o trabalho de simplificação da gramática e de geração do analisador sintático; a inclusão de ações semânticas proporcionará uma maior flexibilidade na definição da gramática.

Na ferramenta desenvolvida, todo o processo de obtenção do analisador é apresentado passo a passo, o que facilita o entendimento e a aplicação do processo para fins didáticos.

1.3 Histórico

Diversos estudos têm sido desenvolvidos no que diz respeito à exposição e detalhamento de métodos para a construção de *parsers* sintáticos.

Para o estudo de linguagens formais e autômatos e sua aplicação à construção de compiladores, sugere-se (HOPCROFT; ULLMAN, 1972), onde são apresentadas técnicas de *parsing* após uma breve exposição da teoria de autômatos e linguagens formais. São apresentados métodos de *backtracking*, *top-down* e *bottom-up*, CYK e *Earley*, bem como outros métodos como LL(k), LC(k), LR(k) para parsing sintático. Uma referência a compiladores pode ser obtida também em (MENEZES, P.B., 2000).

Para a construção de compiladores de linguagens computacionais, sugere-se (APPEL, 1997), onde são detalhados inicialmente os conceitos de tokens léxicos, expressões regulares e autômatos finitos. Mais adiante são apresentados a construção de *parsers*, árvores sintáticas e o processo de compilação. Na mesma linha sugere-se (WIRTH, 1996), onde são apresentados as estruturas, dados abstratos e técnicas de programação e (TREMBLAY; SORENSON, 1985), onde há a formalização de diferentes tipos de *parsers* (LR, SLR, LALR; LL(1)) e seus algoritmos. Em (NETO, 1987), expõe-se de forma detalhada aspectos relacionados à construção de compiladores e os formalismos envolvidos e as características de cada classe de linguagens. Um compilador completo é montado para uma linguagem simplificada, semelhante à linguagem BASIC.

A geração de reconhecedores sintáticos pode ser estudada em (NETO; MAGALHÃES, 1981), e soluções adaptativas podem ser estudadas em (NETO, 1988b); (NETO, 2001); (PEREIRA; NETO, 1997). A geração e reconhecimento de linguagens formais através de gramáticas modificáveis são propostos em (BURSHTEYN, 1990).

O método utilizado para a criação de um *parser* sintático baseado em transdutores (NETO, 1993) serviu como base para Iwai (2000), que a partir de algoritmos desenvolvidos anteriormente, propôs um tratamento para gramáticas dependentes de contexto. Ambos os trabalhos são importantes para a elaboração desta dissertação,

que aproveita os métodos propostos para a inclusão das ações semânticas e produção da ferramenta de software.

Em (PEREIRA, 1999) desenvolve-se uma ferramenta de auxílio ao desenvolvimento de reconhecedores sintáticos, denominada RSW e baseada em autômatos finitos, autômatos de pilha e autômatos adaptativos.

Em (BONFANTE; NUNES, 2001), explora-se o uso de estatística para a criação e tratamento de *parsers* sintáticos; (FREITAS, 2001) apresenta uma ferramenta de auxílio à implementação de aplicações que empregam mais de uma linguagem de programação (aplicações multilinguagem). Este método auxilia na utilização de diferentes linguagens adequadas a cada parte de um desenvolvimento, melhorando o aproveitamento de equipes de desenvolvimento com conhecimentos em diferentes linguagens.

Em (PISTORI; NETO, 2003); (PISTORI, 2003) é apresentada a ferramenta *Adaptools*, que permite a implementação e teste de autômatos adaptativos. São apresentados também as características de depuração, visualização gráfica e mecanismos de controle e execução de máquinas no *Adaptools*. Esta ferramenta será utilizada para a execução parcial do analisador sintático produzido pelo software desenvolvido neste trabalho.

Como informações adicionais sobre o estudo de linguagens e autômatos adaptativos, sugere-se (MENEZES, C. E. D.; NETO, 2002); (NETO; IWAI, 1998); (IWAI,2000); (PISTORI; NETO, 2002); (PISTORI; NETO; COSTA, 2003); (ROCHA, 2003).

1.4 Conteúdo desta dissertação

No capítulo 2, são apresentados os conceitos básicos da teoria.

Uma apresentação de formalismos adaptativos é feita no capítulo 3, onde são discutidos os autômatos adaptativos. Ao final, apresenta-se um exemplo de autômato adaptativo construído através de ações possíveis na ferramenta *Adaptools*.

Nos capítulos 4 e 5, são propostos métodos de preparação da gramática para a construção do autômato reconhecedor e a sua transformação em um transdutor que possibilite a construção do *parser* sintático. Este material serve como base para a elaboração de algoritmos que compõem a ferramenta.

O capítulo 4 apresenta o formato de entrada da gramática e as técnicas de manipulação utilizadas nos algoritmos desenvolvidos.

No capítulo 5, apresentam-se métodos para a construção do autômato de reconhecimento e sua transformação em transdutor sintático, sendo apresentado um exemplo de reconhecimento para ilustrar o processo.

No capítulo 6 apresenta-se a ferramenta desenvolvida através de sua estrutura geral e de seus principais algoritmos, descritos em pseudocódigo simplificado.

No capítulo 7 são apresentados uma análise do trabalho realizado, sugestões para implementações futuras visando à dependência de contexto e conclusões.

2 - CONCEITOS

O objetivo deste capítulo é explorar os aspectos formais relevantes para a realização deste trabalho. A terminologia e conceitos apresentados foram obtidos em (NETO, 1987); (NETO,1993).

Uma *linguagem* é uma coleção de cadeias de símbolos, de comprimento finito, as quais são denominadas *sentenças*. Estas sentenças são formadas pela justaposição de elementos individuais, chamados de *símbolos*. Um conjunto finito não vazio de símbolos é denominado *alfabeto*.

As linguagens são representadas através de três mecanismos básicos:

(I) *Enumeração* das cadeias de símbolos que formam as suas sentenças;

(II) Conjunto de *leis de formação* das cadeias (denominado *gramática*);

(III) *Regras de aceitação* de cadeias (reconhecedores);

No mecanismo (I), todas as cadeias aparecem explicitamente na enumeração, e a validação de uma cadeia é feita através de uma busca no conjunto de sentenças da linguagem.

Para o mecanismo (II), apenas é possível se afirmar que uma cadeia pertence à linguagem, se a mesma puder ser sintetizada através da aplicação das leis de formação que compõem a gramática. Chama-se *derivação* o processo de obtenção de uma sentença a partir da gramática.

No mecanismo (III), dispõe-se de um conjunto de regras de aceitação que, quando aplicadas a uma cadeia, decidem a sua pertinência à linguagem.

As gramáticas são dispositivos *generativos*, enquanto que os reconhecedores são dispositivos de *aceitação*.

Para a construção de compiladores, o uso de gramáticas não é geralmente prático, sendo necessária a obtenção de reconhecedores que descrevam a mesma linguagem (NETO, 1987).

No final da década de 50, (CHOMSKY, 1957) realizou um estudo sobre linguagens, quanto à sua capacidade generativa. Seu sistema de desenvolvimento atraiu o interesse da comunidade científica e foi largamente utilizado em análise sintática. Chomsky também classificou as linguagens por ordem de complexidade, em *linguagens regulares*, *linguagens livres de contexto*, *linguagens sensíveis ao contexto*

e *linguagens recursivamente enumeráveis*, designadas como linguagens do *tipo 3*, *tipo 2*, *tipo 1* e *tipo 0*, respectivamente.

As linguagens regulares ou tipo 3 constituem o tipo mais simples de linguagem na hierarquia apresentada. Suas regras de formação de sentenças são básicas e baseadas apenas em operações de concatenação, união e repetição. Para estas linguagens, são utilizadas as gramáticas regulares ou lineares à direita, e seu reconhecimento pode ser realizado por reconhecedores simples como os autômatos finitos.

Uma classe um pouco mais complexa de linguagens é o das linguagens livres de contexto ou tipo 2, que são definidas por gramáticas cujas regras de produção são baseadas em substituições simples e incondicionais de não-terminais. Embora as leis de formação para linguagens de tipo 2 se assemelhem àquelas de tipo 3, as linguagens livres de contexto permitem adicionalmente a representação de construções aninhadas, o que não ocorre com as linguagens regulares. Neste caso, os mecanismos geradores são chamados gramáticas livres de contexto, e os correspondentes reconhecedores fazem uso de uma pilha, e são denominados Autômatos de Pilha.

Quando se impõe às leis de formação de uma gramática a única restrição de que nenhuma substituição possa reduzir o comprimento da forma sentencial, essas gramáticas são denominadas gramáticas sensíveis ao contexto, e as linguagens por elas geradas são linguagens sensíveis ao contexto ou do tipo 1. Os reconhecedores utilizados para esta classe de linguagens são as Máquinas de Turing com memória limitada.

As linguagens às quais nenhuma restrição é imposta às leis de formação, são também chamadas linguagens irrestritas, correspondendo às linguagens recursivamente enumeráveis ou do tipo 0, que utilizam as Gramáticas Irrestritas como regras de formação. Para este tipo de linguagem, utilizam-se as Máquinas de Turing com memória infinita como mecanismos reconhecedores.

Nas próximas seções, apresenta-se uma visão mais detalhada do uso de Autômatos e Gramáticas. As Máquinas de Turing não são escopo deste trabalho. Para referência, sugere-se (PAPADIMITRIOU, 1998).

2.1 Autômatos e transdutores

Um Autômato Finito é um dispositivo reconhecedor baseado em estados e transições, e que não utiliza memória auxiliar no processo de reconhecimento. É definido por uma quíntupla $M = (Q, \Sigma, P, q_0, F)$, onde Q define um conjunto finito de estados que constituem o autômato; Σ define um conjunto de símbolos de entrada (alfabeto); P é a *função de transição* de estados do autômato que indica as transições possíveis em cada configuração do autômato, e mapeia o produto cartesiano $Q \times (\Sigma \cup \{\varepsilon\})$ em Q; q_0 representa o estado inicial do autômato ($q_0 \in Q$), e F é o conjunto de *estados finais* ou *estados de aceitação* do autômato. Caso o autômato tenha, para cada elemento do conjunto $Q \times \Sigma$ um único estado possível $q' \in Q$, este é dito determinístico.

No caso determinístico, P assume a forma de uma função de transição $P : Q \times \Sigma \rightarrow Q$, responsável pelo controle de estados do autômato, e define a partir do seu estado corrente e de cada símbolo de Σ, o novo estado do autômato.

O reconhecimento de uma sentença w da linguagem definida por este autômato é feito aplicando-se a função de transição para cada um dos símbolos da sentença a partir do estado inicial q_0. Ao término dessa operação, o estado corrente deverá pertencer ao conjunto F para que a sentença seja aceita pelo autômato.

Caso isto não ocorra ou em algum momento não existam transições compatíveis com um determinado símbolo, então o autômato rejeitará a sentença.

Caso o autômato tenha mais de um estado possível a transitar a partir de um dado estado e de um símbolo da cadeia w, isto caracterizará a presença de não-determinismo, e P assumirá a forma $P : Q \times \Sigma \rightarrow 2^Q$, ou seja, P será uma relação.

Um autômato tem como característica transitar entre seus estados em resposta aos símbolos da cadeia de entrada. No entanto, muitas vezes faz-se necessária a geração de novas cadeias ou a alteração da máquina em análise, durante o processo de aceitação. Autômatos que possuem esta característica adicional são denominados *transdutores*. Sua definição é similar àquela utilizada pelo autômato convencional correspondente, porém acrescida de regras de modificação associadas às transições. Os transdutores podem ser adequados para as situações que incluam a geração de cadeia de saída ou a execução de ações durante o processo de reconhecimento. Este

trabalho tem como objetivo a construção de um transdutor que será responsável por validar uma determinada cadeia de entrada, e ao mesmo tempo produzir como saída uma cadeia de símbolos que representa a correspondente árvore sintática de reconhecimento.

2.2 Autômatos de pilha

Quando se agrega uma memória auxiliar organizada em pilha a um autômato, utiliza-se outra classe de reconhecedores, os Autômatos de Pilha. Estes autômatos são empregados no reconhecimento de linguagens livres de contexto, na implementação de analisadores sintáticos para linguagens de programação, em que há, entre outros, os aninhamentos do tipo *if... then ... else*, os aninhamentos de parênteses em fórmulas matemáticas, os blocos *Begin...End* existentes em Pascal, ou *{* e *}* em linguagem C.

A seguir, apresenta-se um formalismo utilizado na literatura para o autômato de pilha, que se denomina *autômato de pilha estruturado* (NETO, 1987), (NETO, 1993), inspirado em (CONWAY, 1963 apud NETO, 1993), (BARNES, 1972 apud NETO, 1993) e (LOMET, 1973 apud NETO, 1993).

Um autômato de pilha estruturado é representado pela óctupla $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$, onde Q é um conjunto finito não-vazio de estados $q_{i,j}$; A é um conjunto de submáquinas a_i; Σ é um conjunto de símbolos de entrada da submáquina a_i, Γ é um conjunto de símbolos da pilha; P é um conjunto de produções do autômato ; $Z_0 \in \Gamma$ é o símbolo inicial da pilha do autômato; $q_0 \in Q$ é o estado inicial do autômato, e $F \subseteq Q$ é um conjunto de estados finais do autômato.

A é um conjunto de submáquinas $a_i = (Q_i , \Sigma_\iota , P_i , E_i , S_i)$ do autômato, onde:

$Q_i \subseteq Q$ ... é o conjunto dos estados $q_{i,j}$ da submáquina a_i

$\Sigma_\iota \subseteq \Sigma$ ... é o conjunto dos símbolos de entrada da submáquina a_i

$E_i \subseteq Q_i$ ... é o estado de entrada da submáquina a_i

$S_i \subseteq Q_i$ ... é o conjunto de estados finais ou de saída de a_i

$P_i \subseteq P$ ... é o conjunto de produções da submáquina a_i

O autômato M pode ser representado por uma *situação* t, em que $t = (\gamma, q, \alpha\tau)$, onde:

$\gamma \in \Gamma^*$ ... representa o conteúdo da pilha;

$q \in Q$ ... representa o estado de M;

$\alpha \in \Sigma^*$ ... representa a parte da cadeia ainda não analisada;

$\tau \notin \Sigma$ ... representa uma marca de fim de texto

Diz-se que o autômato M reconhece uma cadeia de entrada se, e somente se for possível, a partir de uma situação inicial t_0 de M, através da aplicação sucessiva de produções $p \in P$, seja atingida uma situação final t_n:

$t_0 = (Z_0, q_0, \alpha_0\tau) \Rightarrow^* t_n = (Z_0, q_F, \tau)$

onde Z_0 indica a ausência de dados na pilha, q_F indica um estado final de M, e τ indica que a cadeia de entrada foi esgotada.

2.3 Gramáticas

Gramáticas são dispositivos que definem as leis de formação de uma linguagem. Através de um processo de substituições sucessivas de elementos formadores destas regras, produzem-se as cadeias das linguagens definidas pelas gramáticas.

Uma gramática G é definida por uma como quádrupla ordenada G = (V, Σ, P, S).

Chama-se de V ao vocabulário da gramática, isto é, ao conjunto de todos os símbolos utilizados para a definição das leis de formação das sentenças da linguagem. Pode-se dividir estes símbolos em dois conjuntos: o conjunto Σ, dos terminais, que serão os símbolos formadores de sentenças da linguagem, e o conjunto N dos símbolos não-terminais, que são utilizados para identificar grupos de leis de formação de sentenças. Estes conjuntos devem obedecer à relação $V = \Sigma \cup N$, com $\Sigma \cap N = \square$.

O processo de geração de sentenças de uma dada linguagem inicia-se a partir de um não-terminal $S \in N$, que será o primeiro símbolo a ser analisado na gramática, e que é denominado raiz da gramática. As regras que definem a lei de formação de sentenças fazem parte do conjunto P, e são denominadas regras de produção, escritas na forma $\alpha \rightarrow \beta$, onde α é uma cadeia contendo, no mínimo, um não-terminal, e β é uma cadeia eventualmente vazia, composta de terminais e/ou não-terminais, ou seja, $\alpha \in V^* N V^*$ e $\beta \in V^*$

A Linguagem Gerada a partir de uma gramática G é dada por $L(G) = \{ F \in \Sigma^* \mid S \Rightarrow_G {}^* F \}$.

De acordo com a classe de linguagens que uma gramática produz, classifica-se as gramáticas em Irrestritas, Sensíveis ao Contexto, Livres de Contexto e Gramáticas Lineares à Direita (ou à Esquerda).

As Gramáticas Irrestritas geram linguagens irrestritas ou do tipo 0, e por isso são chamadas de gramáticas de tipo 0. Suas produções são da forma:

$\alpha \rightarrow \beta$, com $\alpha \in V^* N V^*$ e $\beta \in V^*$

As Gramáticas Sensíveis ao Contexto geram linguagens do tipo 1, e por isso são chamadas de gramáticas de tipo 1. Suas produções são da forma:

$\alpha \rightarrow \beta$, com $|\alpha| \leq |\beta|$, onde $\alpha \in V^* N V^*$ e $\beta \in V^*$

As Gramáticas Livres de Contexto geram linguagens do tipo 2, e por isso são chamadas de gramáticas de tipo 2. Suas produções são da forma:

$A \rightarrow \alpha$, onde $A \in N$, e $\alpha \in V^*$

A forma mais simples de gramática é aquela geradora das linguagens regulares, nas quais as produções atendem à forma:

$A \rightarrow \alpha B$

$A \rightarrow \alpha$

onde $\alpha \in \Sigma^*$; $A, B \in N$.

As linguagens geradas por este tipo de gramática também são conhecidas como linguagens do tipo 3, e portanto esta gramática é chamada de gramática de tipo 3 na hierarquia de Chomsky.

As linguagens, gramáticas e reconhecedores apontados relacionam-se conforme a tabela I abaixo:

	Linguagem	Gramática	Reconhecedor
tipo 0	Recursivamente Enumerável	Irrestrita	Máquinas de Turing
tipo 1	Dependente de Contexto	Dependente de Contexto	Máquinas de Turing com Memória Limitada
tipo 2	Livre de Contexto	Livre de Contexto	Autômatos de Pilha
tipo 3	Regular	Linear	Autômatos Finitos

Tabela I - Relação linguagem □ gramática □ reconhecedor

Obs: Máquinas de Turing não são utilizadas neste trabalho. Para consulta, sugere-se (PAPADIMITRIOU, 1998).

2.3.1 Notações

O objetivo desta seção é apresentar três formalismos para a representação de linguagens: a *BNF*, a *notação de Wirth* e a *notação de Wirth Modificada*. A notação BNF não foi utilizada neste trabalho, estando aqui indicada como exemplo adicional.

2.3.1.1 Backus-Naur Form (BNF)

É uma metalinguagem bastante conhecida para a especificação de linguagens de programação, criada inicialmente para o Algol 60. Esta metalinguagem permite a representação de linguagens livres de contexto de modo recursivo. Para esta metalinguagem utilizam-se os seguintes símbolos:

<x> - que representa um não-terminal x;

::= - associa um não-terminal a um conjunto de cadeias terminais e/ou símbolos não-terminais;

|	-	indica separação. Lê-se como *ou*;
X	-	representa um terminal;
∈	-	representa a cadeia vazia;
yz	-	representa uma cadeia formada por dois símbolos.

2.3.1.2 Notação de Wirth

Esta notação é uma variante da BNF, sendo mais legível e mais adequada para a substituição de recursões por iterações. Os seguintes símbolos representativos são utilizados pela notação de Wirth:

X	-	representa o nome de um não-terminal;
“y”	-	representa uma cadeia de caracteres que simboliza um terminal;
[z]	-	z representa uma cadeia ou um conjunto de cadeias terminais e/ou não-terminais. Os colchetes indicam que z é opcional, e equivale à indicação z |ε ;
(z)	-	representa z quando utilizado em agrupamentos;
{z}	-	representa opcionalidade no conjunto de terminais e/ou não-terminais representados por z;
zt	-	é a concatenação de duas cadeias, z e t, em qualquer uma das representações indicadas acima;
z | t	-	representa a alternância entre z e t, sendo ambos representados em qualquer uma das formas acima;
=	-	é o símbolo que separa os elementos do lado esquerdo e direito da produção, equivale ao símbolo ::= do BNF
.	-	é o símbolo que representa o final de uma regra de produção.

Formaliza-se esta notação, através da figura abaixo (NETO,1993):

Notação-de-Wirth	=	regra { regra } .
regra	=	não-terminal “=” expressão “.” .
expressão	=	termo { “|” termo } .
termo	=	fator { fator } .
fator	=	não-terminal
		terminal
		“ε”
		“(“ expressão “)”
		“[“ expressão “]”
		“{“ expressão “}” .

Fig. 1 – Notação de Wirth

2.3.1.3 Notação de Wirth Modificada

É a notação utilizada para o tratamento gramatical realizado nesta dissertação.

É uma modificação da notação de Wirth tradicional, com o objetivo de gerar descrições sintáticas mais claras e compactas. Em fatores ou elementos básicos que compõem regras, inclui-se a indicação de repetição através do símbolo \, seguindo-o de uma expressão sintática.

Analogamente ao exposto em 2.3.1.2, formaliza-se esta notação, através da figura abaixo (NETO,1993):

Notação-de-Wirth-Modificada	=	(regra \ ε) .
regra	=	não-terminal “=” expressão “.” .
expressão	=	(termo \ “|”) .
termo	=	(fator \ ε) .
fator	=	não-terminal
		terminal
		“ε”
		“(“ expressão (“\” expressão | ε) “)” .

Fig. 2 – Notação de Wirth Modificada

2.3.2 Inclusão de ações semânticas

A inclusão de ações semânticas à gramática original possibilita a inclusão de mecanismos adicionais ao reconhecimento de sentenças, como por exemplo a geração de código e o tratamento de algumas dependências de contexto.

Para a definição de linguagens livres de contexto com a inclusão de ações semânticas, as meta-linguagens tradicionais como BNF e Wirth não dispõem de recursos suficientes, e por este motivo são utilizados formalismos como as gramáticas de atributos (TREMBLAY; SORENSON,1985) e as gramáticas de dois níveis (PAGAN, apud NETO, 1993). As gramáticas de atributos são capazes de efetuar associações entre objetos da linguagem e atributos declarados para estes objetos. A utilização destas gramáticas permite a inclusão de mecanismos que possibilitem não somente o reconhecimento, mas também o tratamento de dependências de contexto e geração de código (NETO, 1993).

Para este trabalho, propõe-se a inclusão de ações semânticas de forma semelhante à gramática de atributos, definindo-se a quádrupla:

$G = (V, \Sigma, R, S)$

Onde R corresponde a um conjunto de regras de produção estendidas pela inclusão de ações semânticas, ou seja, R é formado por $P \cup A$, onde A representa um conjunto de ações semânticas a serem aplicadas às regras, respeitando-se o formato descrito em 4.1.

Como exemplo, a gramática G = ({S, X}, {0, 1}, R, S) é uma gramática que inclui ações semânticas em seu conjunto de regras de produção. Estas ações permitem a conversão de números binários em números decimais.

O conjunto de regras de produção estendidas que definem a gramática G é dada por:

R = {
S → { } X{A}
X → { } 0 {B}
X → { } 1 {C}
X → X{ } 0 {D}
X → X{ } 1 {E}
}

Notar que o lado direito de cada regra possui um par de chamadas a ações semânticas, denotadas por chaves. Considerando-se *valor* como uma variável, o significado de cada uma das ações semânticas é definido pelo conjunto *A* :

A = {

{A} : imprimir valor

{B} : valor ← 0

{C} : valor ← 1

{D} : valor ← 2*valor

{E} : valor ← 2*valor+1

{ } : sem ação

}

A cadeia 110 gera, portanto, a derivação abaixo:

S □ { } X{A} □ { } X{ } 0 {D}{A} □ { } X{ } 1 {E}{ } 0 {D}{A} □

□ { } { } 1 {C}{ } 1 {E}{ } 0 {D}{A}

A aplicação das ações na seqüência obtida acima gera:

{C}	...	valor ← 1	(valor = 1)
{E}	...	valor ← 2*valor+1	(valor = 3)
{D}	...	valor ← 2*valor	(valor = 6)
{A}	...	imprimir valor	(imprimir 6)

O reconhecimento de 110 produz, portanto, o valor decimal 6.

2.4 Árvores de derivação

Muitas vezes a análise direta sobre reconhecedores não proporciona a possibilidade visual da execução passo-a-passo na realização de análise sintática de determinada sentença de uma linguagem. A utilização de árvores sintáticas ou de derivação poderá auxiliar na visualização do processo de derivação de sentenças.

As árvores sintáticas constituem-se em uma forma de representação que utiliza como elementos nós e ligações (ou ramificações) que os conectam. O nó de topo da árvore é denominado *raiz*, e aos nós terminais atribui-se o nome de *folhas*. Neste tipo de

construção, um nó sempre está conectado de forma descendente a apenas um *ancestral* (nó ascendente).

Exemplo

Seja a gramática G = (V, Σ, P, E), utilizada para a análise de expressões matemáticas, e considerando-se que, sendo n = número, E = expressão, T = termo e F = fator:

- V = {+, -, *, /, (,), n, E, T, F}
- Σ = {+, -, *, /, (,), n}
- P = {

  $E \rightarrow E + T \mid E - T \mid E * T \mid E / T \mid T$

  $T \rightarrow T * F \mid T / F \mid T + F \mid T - F \mid F$

  $F \rightarrow (E)$

  $F \rightarrow n$

  }

A sentença (n - n) * (n) + n gera a derivação abaixo:

E □ T □ T + F □ T * F + F □ F * F + F □ (E) * F + F □

(E – T) * F + F □ (T – T) * F + F □ (F – T) * F + F □

(F – F) * F + F □ (n – F) * F + F □ (n – n) * F + F □

(n – n) * (E) + F □ (n – n) * (T) + F □

(n – n) * (F) + F □ (n – n) * (n) + F □

(n – n) * (n) + n

A árvore sintática que representa a geração desta expressão está ilustrada na figura abaixo:

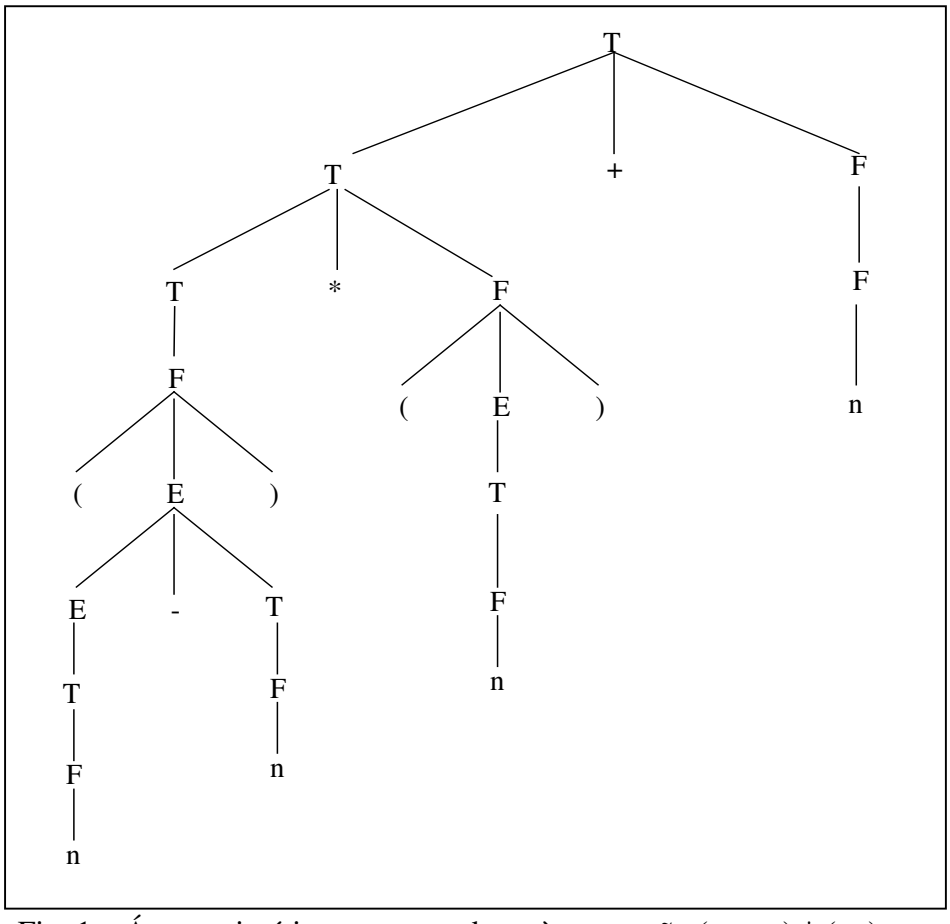


Fig. 1 – Árvore sintática correspondente à expressão (n - n) * (n) + n

O processo de construção do analisador sintático proposto neste trabalho envolve o desenvolvimento de uma *pilha de transdução* que será utilizada para a emissão de uma cadeia que represente a árvore de derivação de uma sentença da linguagem, durante o processo de reconhecimento. Esta pilha é construída através de autômato adaptativo. Por este motivo, no capítulo a seguir, exploram-se os conceitos de autômato adaptativo e define-se a notação utilizada para representá-los.

3 - AUTÔMATOS ADAPTATIVOS

Neste capítulo, estudam-se formalismos em que a adaptatividade é associada às representações convencionais dos autômatos e das gramáticas, permitindo-se a alteração dinâmica de sua configuração. Os formalismos assim idealizados são denominados adaptativos.

Inicialmente, apresenta-se uma introdução ao conceito geral de *formalismo adaptativo*, sendo em seguida detalhado o caso particular dos Autômatos Adaptativos (NETO, 1993);(NETO, 2001).

Um dispositivo adaptativo muda seu comportamento dinamicamente como resposta direta aos estímulos de entrada, sem a interferência de agentes externos. Para que isto seja possível, dispositivos adaptativos devem ser auto-modificáveis. Em outras palavras, quaisquer possíveis mudanças no comportamento do dispositivo devem ser totalmente conhecidas. Por este motivo, dispositivos adaptativos devem ser capazes de detectar todas as situações que possam causar possíveis modificações e de reagir adequadamente, impondo mudanças ao comportamento do dispositivo.

3.1 Autômato adaptativo

A inclusão de adaptatividade em autômatos é implementada através da inclusão de ações adaptativas a serem executadas *antes* e/ou *depois* de suas transições. Estes autômatos, denominados *Autômatos Adaptativos,* podem ter sua topologia alterada ao longo do processamento de uma cadeia de entrada. Para cada transição especificada, poderão ser associadas ações que permitam a inclusão ou eliminação de estados e transições, como resultado da aplicação de suas regras. Denomina-se aqui *configuração do autômato* à topologia e ao conjunto de parâmetros que definem um autômato adaptativo em um dado momento.

A cada vez que uma *transição adaptativa* é executada, a máquina de estados que implementa o autômato adaptativo sofre alguma mudança em sua configuração, obtendo-se uma nova máquina de estados. Para cada cadeia de entrada pode-se definir uma *máquina de estados inicial* (antes do início do processamento da cadeia de entrada), *máquinas de estados intermediárias* (geradas ao longo do processamento

da cadeia) e uma *máquina de estados final* (correspondente ao término do processamento da cadeia de entrada).

O reconhecimento de uma cadeia por um autômato adaptativo é realizado pelos seguintes passos:

- início de reconhecimento da cadeia de entrada, com o autômato posicionado no estado inicial da *máquina de estados inicial*;
- execução de uma seqüência de transições, formada por transições próprias da *máquina de estados corrente* e por transições adaptativas, provocando a evolução da máquina de estados corrente para uma nova máquina de estados, na qual se dará seqüência ao reconhecimento;
- término do reconhecimento: normal (final de reconhecimento em estado final com a cadeia de entrada esgotada), ou por erro de sintaxe (quando a cadeia de entrada se esgota sem que se tenha atingido um estado final, ou quando não há uma transição válida a partir do estado corrente com a cadeia de entrada ainda não esgotada).

Um autômato adaptativo é então caracterizado pelos seguintes elementos: uma cadeia de entrada w a ser reconhecida; uma máquina de estados inicial E^0, que representa a máquina utilizada no início da operação do autômato; a máquina de estados E^n, que implementa o autômato adaptativo ao final do reconhecimento da cadeia de entrada w ($n \geq 0$); e as máquinas E^i, cada qual representando o autômato adaptativo imediatamente após a i-ésima execução de suas transições adaptativas ($0 \leq i \leq n$).

Assumindo-se que uma cadeia w a ser reconhecida é composta por sub-cadeias α^i, ($0 \leq i \leq n$), o reconhecimento de uma cadeia w por um autômato adaptativo é caracterizado macroscopicamente pela seqüência de reconhecimentos de cada uma das sub-cadeias α^i da cadeia w pelas correspondentes máquinas de estados E^i ($0 \leq i \leq n$). A *trajetória de reconhecimento* da cadeia w corresponde a uma seqüência (E^0, α^0), (E^1, α^1), ..., (E^n, α^n), com $w = \alpha^0\alpha^{1\cdots}\alpha^n$, onde (E^i, α^i) representa o consumo da sub-cadeia α^i pela máquina de estados E^i.

Em um autômato adaptativo, as *transições adaptativas* aplicadas ao reconhecimento de uma cadeia de entrada w são representadas pelo agrupamento adequado de ações elementares de inserção e de remoção de elementos, ocasionadas respectivamente por operações de inserção de novas transições ou de eliminação de transições existentes. Agrupamentos dessas ações elementares definem uma *ação adaptativa*.

Uma *transição adaptativa* é então caracterizada por uma quádrupla $p_i = (t_i, A_i, t_i', B_i)$, onde t_i é a *configuração* específica do autômato para que a transição adaptativa p_i seja aplicada; A_i é a ação adaptativa a ser aplicada ao autômato adaptativo antes de sua alteração para a nova configuração t_i' ; t_i' é a configuração para a qual o autômato deve ser conduzido pela aplicação da transição adaptativa p_i; e B_i a ação adaptativa a ser aplicada ao autômato adaptativo após a alteração de sua situação para t_i'.

As ações adaptativas que caracterizam as transições adaptativas são definidas a partir de *funções adaptativas*. Uma *função adaptativa* é definida pela ênupla $(F, P, V, G, T, E, q, A, B)$, onde:

- F representa o nome da função adaptativa;
- P é uma lista ordenada de parâmetros formais (p_1, p_2, ...) da função adaptativa F. Estes parâmetros são preenchidos automaticamente antes do início da execução da função adaptativa;
- V é o conjunto de nomes de variáveis preenchidos uma única vez na execução da função adaptativa, como resultado de ações adaptativas de consulta;
- G é o conjunto de geradores ou seja, variáveis especiais utilizados nas funções adaptativas;
- T é uma seqüência de padrões a serem consultados nas produções da máquina corrente para o preenchimento de variáveis referenciadas;
- E é uma seqüência de padrões a serem consultados nas produções da máquina corrente para o preenchimento de variáveis referenciadas, as quais serão eliminadas posteriormente;
- q é uma seqüência de transições a serem inseridas na máquina de estados corrente;

- A representa um conjunto de ações adaptativas A_i , a serem executadas antes da execução da função adaptativa declarada;
- B representa um conjunto de ações adaptativas B_i , a serem executadas após a execução da função adaptativa declarada;

Uma *ação adaptativa* é definida então por um par ordenado (F, Π), sendo F o nome da função adaptativa correspondente à ação adaptativa, e Π uma seqüência de argumentos que correspondem aos parâmetros formais (p_1, p_2, ...) da função adaptativa F.

3.2 Notação utilizada

Neste trabalho, as funções adaptativas serão descritas através de ações adaptativas elementares, da seguinte forma:

<nome>: <tipo>[<origem>, <input>, <destino>, <push>, <output>, <adap>]

Onde:

<nome> ... indica o nome da função adaptativa (parâmetro F em 3.1)

<tipo>... indica o tipo de ação elementar a ser realizada:

- ? caso se trate de uma ação de consulta;
- + caso se trate de uma ação de inclusão;
- - caso se trate de uma ação de exclusão.

Estas ações elementares atuam sobre os parâmetros V, G, T, E, definidos em 3.1.

<origem>... é o estado de origem de uma transição adaptativa;

<input>... representa o símbolo a ser consumido de uma cadeia w;

<destino>... é o estado de destino após a execução da transição;

<push> ... representa um símbolo a ser empilhado;

<output>... é um símbolo a ser produzido como saída;

<adap>... é uma nova função adaptativa a ser executada antes ou após a execução de uma transição;

A construção das ações adaptativas desta forma, implementa a definição dos demais parâmetros definidos em 3.1 (*P, q, A, B).*

Esta é uma forma de representação prática, baseada no formato adotado para a ferramenta *Adaptools* (PISTORI, 2003).

Ao conjunto de ações elementares assim descritas atribui-se um nome que corresponderá, no *Adaptools*, ao nome da função adaptativa que está sendo definida.

A chamada a uma função adaptativa é feita através de sua indicação no campo <adap>. Caso a chamada da função adaptativa deva ser ativada antes do consumo do símbolo da cadeia de entrada, denota-se o fato incluindo-se um ponto depois do nome da função, caso contrário inclui-se um ponto antes do nome da função.

Adicionalmente, definem-se os seguintes símbolos a serem utilizados nos campos indicados acima:

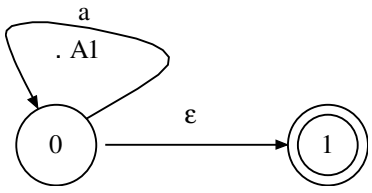
eps ... denota a cadeia vazia;

nop ... denota "nenhuma operação" (indica o não preenchimento do campo);

*n ... denota um gerador, de nome n.

3.3 Exemplo de utilização da notação

Representa-se a seguir o reconhecimento de sentenças que pertençam à linguagem $L = \{a^n b^n c^n$, para $n \geq 0\}$, por meio de um autômato adaptativo, cujo diagrama de estados e transições está apresentado a seguir:



Neste diagrama, a é o símbolo a ser consumido, e *.A1* é uma função adaptativa *A1* a ser executada *após* o consumo do símbolo *a*, devido à posição do ponto.

A função adaptativa *.A1* é composta por um conjunto de ações adaptativas elementares, representadas abaixo. Para facilitar o entendimento, as ações foram nomeadas como *L1*, *L2*, *L3*, *L4* e *L5*.

L1: ?[?x , eps, ?y , ?z , nop, nop]
(procura-se uma transição em vazio (eps), partindo de um estado qualquer *x* e finalizando em um estado qualquer y.);

L2: -[?x , eps, ?y , ?z , nop, nop]
(elimina-se a transição em vazio encontrada);

L3: +[?x , b , *n1 , nop, nop, nop]
(cria-se nova transição que consome b, partindo do estado x originalmente encontrado e chegando-se a um novo estado criado, n1);

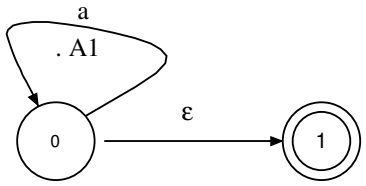
L4: +[*n1 , eps, *n2 , nop , nop, nop]
(cria-se uma nova transição em vazio, partindo-se de n1 previamente criado, e chegando-se a um novo estado criado pelo gerador *n2);

L5: +[*n2 , c , ?y , ?z , nop, nop]
(cria-se nova transição partindo-se do novo estado previamente criado, consumindo c e atingindo o estado y previamente encontrado).

Exemplo: reconhecimento da cadeia "aabbcc"

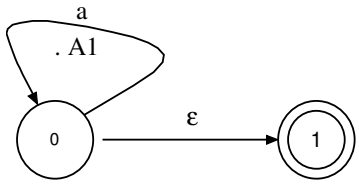
Ilustra-se passo a passo o reconhecimento de uma cadeia aabbcc.

O autômato inicial (máquina E0) está configurado da seguinte forma:



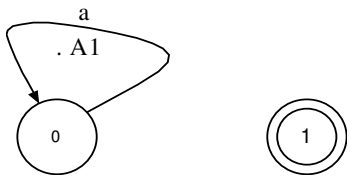
Após o consumo do primeiro símbolo a, há a execução da função adaptativa *.A1* . Durante a execução da função, as ações elementares são executadas, e os seguintes passos são executados:

- procura-se a transição em vazio (ação *L1*), encontrando-a do estado 0 ao estado 1:



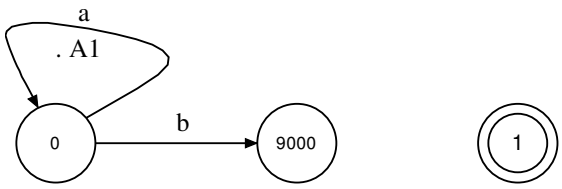
cadeia: - a b b c c
estado corrente: 0
variáveis: x:0, y:1

- elimina-se a transição encontrada (ação *L2*):



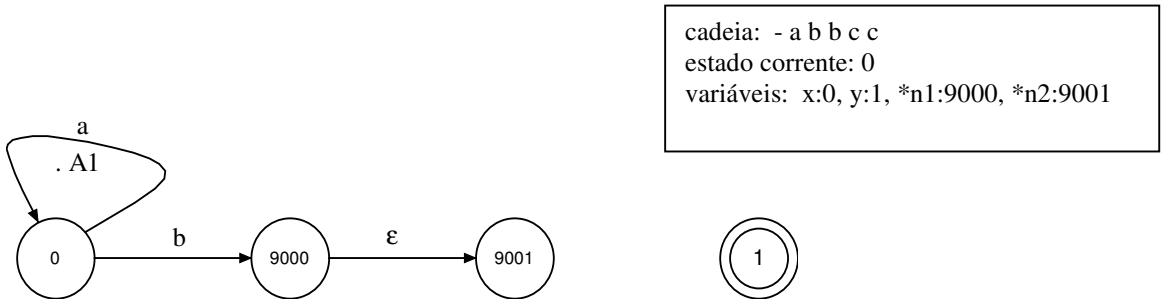
cadeia: - a b b c c
estado corrente: 0
variáveis: x:0, y:1

- acrescenta-se um novo estado cuja transição deverá consumir o símbolo b (ação *L3*). Neste caso foi criado o estado 9000:

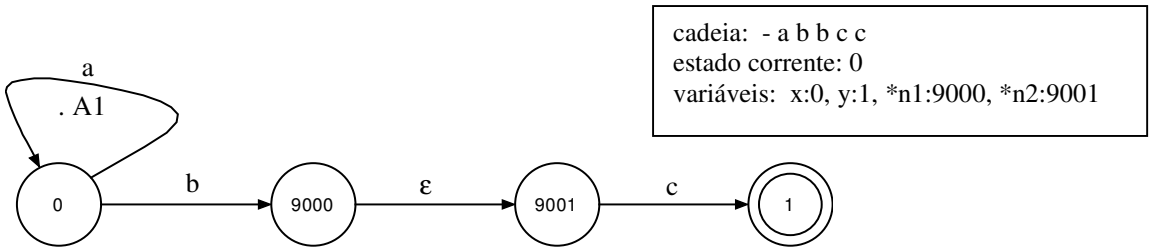


cadeia: - a b b c c
estado corrente: 0
variáveis: x:0, y:1, *n1:9000

- acrescenta-se uma transição ε (cadeia vazia) do estado criado para um novo estado (ação *L4*). Neste caso foi criado o estado 9001:

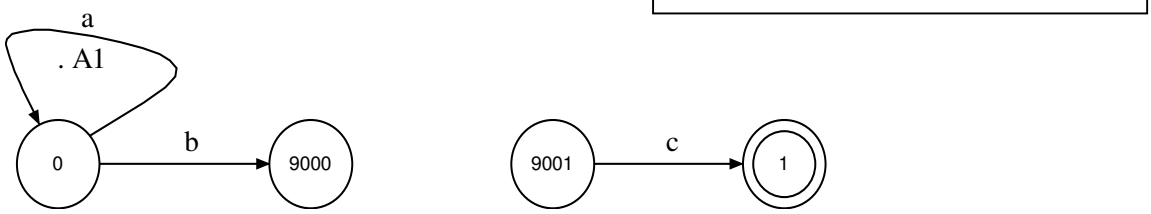


- acrescenta-se uma transição que consuma o símbolo *c* do novo estado (9001) ao estado inicial marcado com y (estado 1) (ação *L5*):

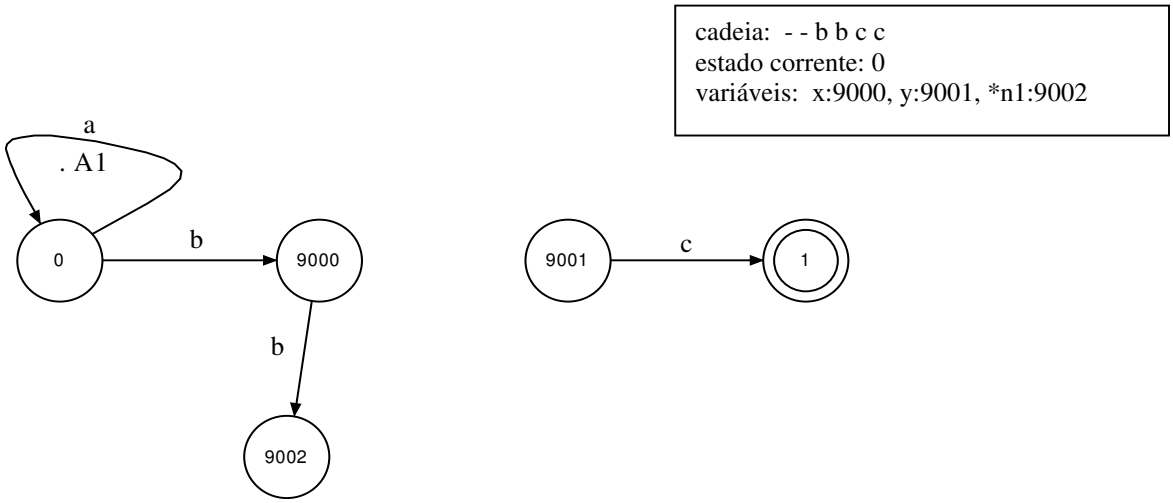


Com isso, prossegue-se a análise da cadeia no próximo a (-abbcc), sendo chamada novamente a ação adaptativa após o consumo do a (--bbcc), ocasionando as seguintes modificações:

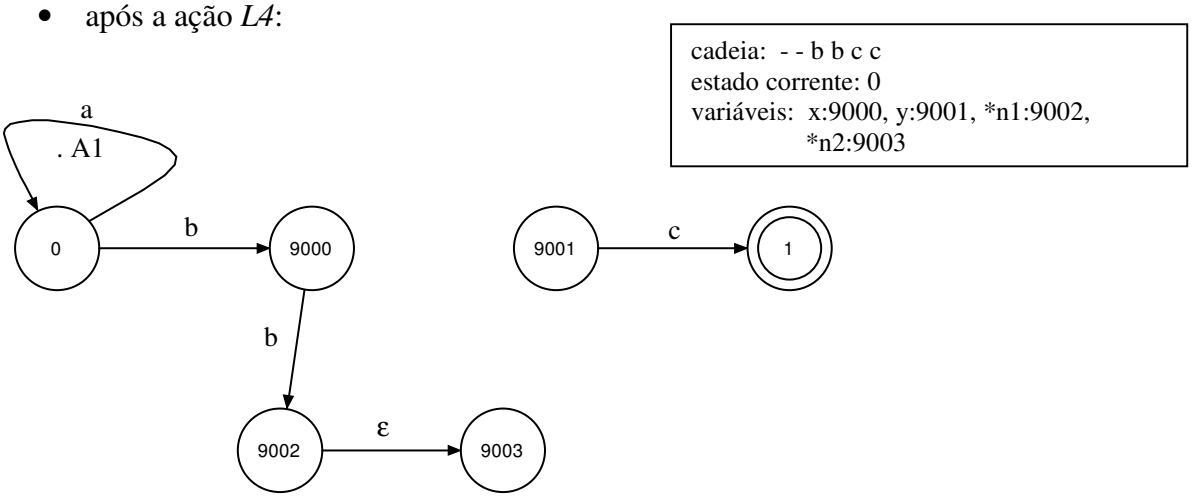
- após ações *L1* e *L2*:



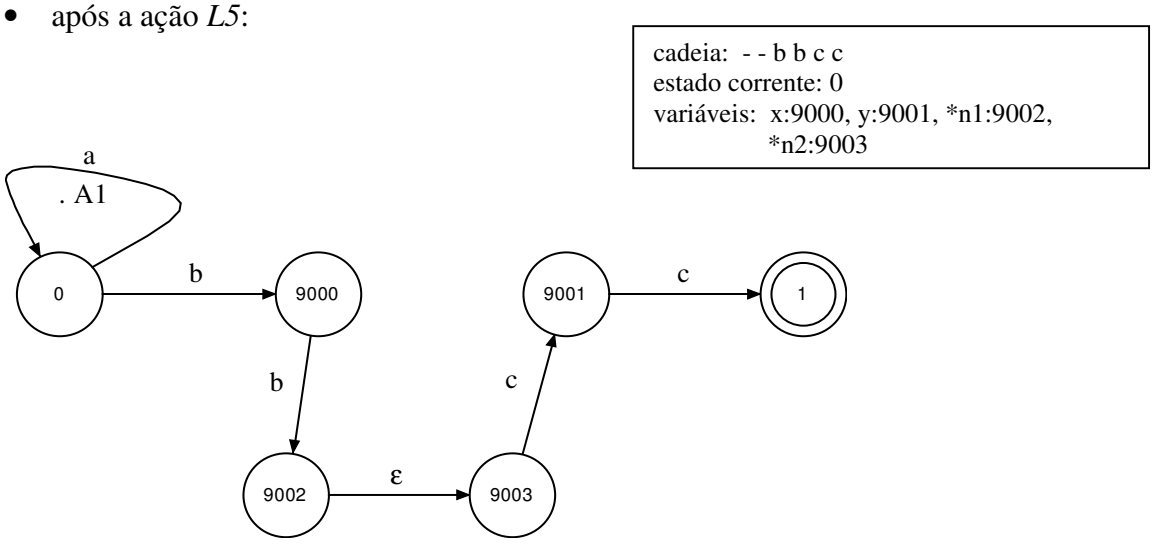
- após a ação *L3*:



- após a ação *L4*:



- após a ação *L5*:



Neste ponto, a cadeia a ser consumida é bbcc, e a análise prossegue a partir da nova máquina de estados gerada, ou seja, são processadas as transições que unem os estados 0-9000 (consumo do primeiro b), 9000-9002 (consumo do segundo b), 9002-9003 (transição em vazio), 9003-9001 (consumo do primeiro c), 9001-1 (consumo do segundo c). Ao atingir o estado 1, o autômato pára o reconhecimento em seu estado final, e a cadeia é aceita.

4 - PREPARAÇÃO DA GRAMÁTICA PARA A CRIAÇÃO DO *PARSER*

Para que um *parser* possa ser gerado automaticamente a partir de uma gramática de entrada, esta gramática será transformada em uma gramática equivalente, simplificada e ajustada de forma a se reduzir ao máximo as ocorrências de não-determinismos. A redução da ocorrência de não-determinismos simplifica a construção do parser e aumenta a eficiência no reconhecimento.

Neste capítulo, são apresentados o formato da gramática de entrada e a preparação da gramática para este fim.

4.1 Formato da gramática de entrada

A gramática de entrada utilizará produções que devem seguir uma das formas de apresentação descritas abaixo, sendo P um não-terminal que define uma regra de produção, Y e Z ações semânticas e μ uma seqüência de símbolos terminais e/ou não terminais, sendo $\mu \in (\Sigma \cup N)^*$:

$P \rightarrow P\{Y\}\ \mu\ \{Z\}$

$P \rightarrow \{Y\}\ \mu\ \{Z\}P$

$P \rightarrow \{Y\}\ \mu\ \{Z\}$

Como será visto a seguir, apesar de existirem outras possibilidades de representação, as produções da gramática de entrada serão analisadas de forma a categorizá-las em uma das três formas apresentadas acima. Além disso, serão aplicadas transformações à gramática, com o objetivo de obter uma gramática equivalente, convenientemente preparada para a construção do autômato/transdutor que atuará na geração da árvore sintática.

4.2 Tratamento gramatical

Nesta seção são apresentados os métodos para a preparação da gramática, com o objetivo de facilitar a construção do *parser* sintático.

4.2.1 Detecção de tipo de recursão

Conforme exposto em (NETO, 1993), identifica-se uma *recursão à esquerda* quando há uma referência direta ao não-terminal que define a regra na extremidade esquerda da expressão que forma o lado direito da regra, uma *recursão à direita* quando houver uma referência ao não-terminal que define a regra na extremidade direita dessa expressão, e uma *recursão central* quando a referência ao não-terminal que define a regra ocorrer em qualquer outro ponto da expressão. Cada uma dessas recursões, e também o caso em que nenhuma recursão é encontrada na expressão, é discutida a seguir, visando à preparação da gramática para a obtenção do *parser*. Para que isto seja possível, será necessário que se identifique cada uma das regras que compõem a gramática disponível inicialmente.

Para cada uma das regras de produção, devidamente numeradas, detecta-se o respectivo tipo de recursão, conforme caracterizado acima. Para isso, analisa-se a expressão que forma o lado direito de cada uma das regras, em busca de não-terminais que representem cada um dos casos acima mencionados.

Considerando-se que μ representa uma seqüência de símbolos terminais e/ou não terminais, sendo $\mu \in (N \cup \Sigma)^*$, que Y e Z representam ações semânticas, e que P é forçosamente um não-terminal (porque trata-se de linguagem livre de contexto), assume-se que:

a) $P \rightarrow P\{Y\}\ \mu\ \{Z\}$

Será classificado como regra contendo uma recursão à esquerda.

b) $P \rightarrow \{Y\}\ \mu\ \{Z\}P$

Será classificado como regra contendo uma recursão à direita.

c) $P \rightarrow \{Y\}\ \mu\ \{Z\}$

Representa os demais casos não classificados anteriormente, incluindo-se os casos de recursão central.

Uma vez detectado o tipo de recursão, passa-se à rotulação das produções. A rotulação possibilitará um registro da origem e transformação das regras, e será utilizada para a construção da árvore de derivação pelo parser.

4.2.2 Rotulação das produções

A inclusão de *rótulos* para os elementos de cada uma das regras tem o objetivo de fornecer informações importantes sobre a gramática, sendo essencial para a construção dos próximos passos de manipulação gramatical. Pode-se definir um *rótulo* como uma seqüência de símbolos que indicam o tipo de produção, o não-terminal ou terminal associado, e outras informações relevantes para o processo de construção do *parser* (por exemplo, o número correspondente à regra, o número que representa o correspondente estado do autômato a ser construído, etc).

Sendo $\beta_i \in (N \cup \Sigma) \cup \{"\varepsilon"\}$, onde $1 \leq i \leq n$, a construção dos rótulos é feita de acordo com as regras abaixo:

a) Recursões à esquerda

As construções identificadas como recursivas à esquerda:

$P \rightarrow P\{Y\}\ \beta_1 \beta_2 \ldots \beta_n \{Z\}$

serão rotuladas da seguinte forma:

- o não-terminal P não será rotulado;
- cada ação semântica será substituída por um não-terminal especial Ψ. O rótulo associado a Ψ representará a ação semântica substituída por ele;
- cada um dos elementos β_i estará associado um rótulo β_i' caso o elemento represente um terminal. Caso contrário não será associado um rótulo.
- o início do lado direito da regra de produção será rotulado com um colchete [, símbolo que deverá representar o início de uma regra;
- o final do lado direito da regra de produção será rotulado com um colchete], símbolo que deverá representar o final de uma regra. Este símbolo deverá ser precedido do não-terminal que define a regra, sua seqüência (número seqüencial definido no passo anterior), e o indicador □, que deverá representar o tipo de recursão esquerda.

Para a representação dos rótulos, indica-se uma seta ↑ na posição onde o rótulo deve ser incluído, e acima desta seta indica-se o rótulo.

A expressão acima, devidamente rotulada, é representada por:

$$\begin{array}{llllllllllllllllll} & & & & & & & [& & & & & \{Y\} & & \beta_1' & & \beta_2' & & & & \beta_n' & & \{Z\}\ P_i\ \square] \\ P & \rightarrow & \uparrow & P & \uparrow & \Psi & \uparrow & & \beta_1 & \uparrow & \beta_2 & \uparrow & \ldots & \beta_n & \uparrow & \Psi & \uparrow & \end{array}$$

(onde β_n' representa o rótulo devido ao elemento β_n e i representa o número de seqüência da regra em questão).

b) Recursões à direita

As construções identificadas como recursivas à direita:

$P \rightarrow \{Y\}\ \beta_1\ \beta_{2\ldots}\ \beta_n\ \{Z\}P$

serão rotuladas da seguinte forma:

- o não-terminal P terá rótulo nulo (inexistente);
- cada ação semântica será substituída por um não-terminal especial Ψ. O rótulo associado a Ψ representará a ação semântica substituída por ele;
- cada um dos elementos βi estará associado um rótulo β_i' caso o elemento represente um terminal. Caso contrário não será associado um rótulo.
- o início do lado direito da regra de produção será rotulado com um colchete [, símbolo que deverá representar o início de uma regra;
- o final do lado direito da regra de produção será rotulado com um colchete], símbolo que deverá representar o final de uma regra. Este símbolo deverá ser precedido do não-terminal que define a regra, sua seqüência (número seqüencial definido no passo anterior), e o indicador □, que deverá representar o tipo de recursão direita.

A expressão acima, devidamente rotulada, é representada por:

$$\begin{array}{llllllllllllllllll} & & & & & & & [& & \{Y\} & & \beta_1' & & \beta_2' & & & & \beta_n' & & \{Z\} & & P_i\ \square] \\ P & \rightarrow & \uparrow & \Psi & \uparrow & & \beta_1 & \uparrow & \beta_2 & \uparrow & \ldots & \beta_n & \uparrow & \Psi & \uparrow & & P & \uparrow & \end{array}$$

(onde β_n' representa o rótulo devido ao elemento β_n e i representa o número de seqüência da regra em questão).

c) Recursões à esquerda e à direita simultaneamente

Assume-se recursão pelo lado esquerdo:

[{Y} β_1' β_2' β_n' {Z} P_i □]

P → ↑ P ↑ Ψ ↑ β_1 ↑ β_2 ↑ ... β_n ↑ Ψ ↑ P ↑

d) Demais casos:

As construções identificadas como *demais casos,* e também aqui denominadas *caso geral*, recaem em uma das alternativas abaixo indicadas:

d.1) construções do tipo P → {Y} β_1 β_2 ... β_n {Z} serão rotuladas da seguinte forma:

- o não-terminal P terá rótulo nulo (inexistente);
- as ações semânticas {Y} e {Z} ... representarão os rótulos {Y} e {Z} respectivamente associados a um não-terminal especial Ψ;
- a cada um dos elementos β_i estará associado um rótulo β_i' caso o elemento represente um terminal. Caso contrário não será associado um rótulo.
- o início do lado direito da regra de produção será rotulado com um colchete [, símbolo que deverá representar o início de uma regra;
- o final do lado direito da regra de produção será rotulado com um colchete], símbolo que deverá representar o final de uma regra. Este símbolo deverá ser precedido do não-terminal que define a regra, sua seqüência (número seqüencial definido no passo anterior), e o indicador □, que deverá representar o tipo de recursão geral.

A expressão acima, devidamente rotulada, será representada por:

[{Y} β_1' β_2' β_n' {Z} P_i □]

P → ↑ Ψ ↑ β_1 ↑ β_2 ↑ ... β_n ↑ Ψ ↑

(onde β_n' representa o rótulo devido ao elemento β_n e i representa o número de seqüência da regra em questão).

d.2) construções onde há o não-terminal P entre os elementos $\beta_1 \beta_2 \ldots \beta_n$ figuram como construções de recursão central e são rotuladas de forma equivalente à construção indicada em d.1 .

d.3) construções do tipo $P \rightarrow \{Y\}\ \varepsilon\ \{Z\}$ serão rotuladas da mesma forma exposta em d.1, e produzem como resultado:

$$[\quad \{Y\} \quad \varepsilon \quad \{Z\}\ P_i\ \square]$$
$$P \rightarrow \uparrow \Psi \uparrow \quad \varepsilon \quad \uparrow \Psi \uparrow$$

4.2.3 Agrupamento de produções que definem um mesmo não-terminal

Para cada um dos tipos de recursão, havendo mais de uma alternativa, as correspondentes produções devem ser agrupadas de forma que uma única produção seja criada para cada uma delas. Este procedimento tem o objetivo de diminuir a ocorrência de transições em vazio geradas pelas diversas opções de um mesmo não-terminal. Para isso, definindo-se o termo μ_i para representar a seqüência de elementos β rotulados no item 4.2.2, e pertencentes à regra de produção cujo número i de seqüência está descrito em 4.2.1, os seguintes casos de agrupamento são possíveis:

a) Agrupamento de regras com recursão à esquerda

$$[\quad \{Y_1\} \quad \{Z_1\}P_1\ \square \quad \{Y_2\} \quad \{Z_2\}P_2\ \square$$
$$P \rightarrow \uparrow P \uparrow (\uparrow \Psi \uparrow \quad \mu_1 \quad \uparrow \Psi \uparrow \quad | \uparrow \Psi \uparrow \quad \mu_2 \quad \uparrow \Psi \uparrow \quad | \ldots$$

$$\{Y_k\} \quad \{Z_k\}P_k\ \square \quad]$$
$$\ldots \mid \uparrow \Psi \uparrow \quad \mu_k \quad \uparrow \Psi \uparrow \quad) \uparrow$$

Notar que o não-terminal P correspondente à recursão em cada uma das seqüências referenciadas por μ_i foi migrado para o lado esquerdo da associação.

b) Agrupamento de regras com recursão à direita

$$P \rightarrow \uparrow [\ \{Y_1\}\ \{Z_1\}P_1 \square\ |\ \{Y_2\}\ \{Z_2\}P_2 \square\ |\ \ldots\ |\ \{Y_k\}\ \{Z_k\}P_k \square\ ] \uparrow (\uparrow \Psi \uparrow\ \mu_1 \uparrow \Psi \uparrow\ |\ \uparrow \Psi \uparrow\ \mu_2 \uparrow \Psi \uparrow\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_k \uparrow \Psi \uparrow\ ) \uparrow P \uparrow$$

Notar que o não-terminal P correspondente à recursão em cada uma das seqüências referenciados por μ_i foi migrado para o lado direito da associação.

c) Agrupamento de regras com recursão à direita e à esquerda simultaneamente

Para os casos de recursão de ambos os lados, assume-se a recursão pelo lado esquerdo (recursão à esquerda):

$$P \rightarrow \uparrow [\ \{Y_1\}\ \{Z_1\}\ P_1 \square\ |\ \{Y_2\}\ \{Z_2\}\ P_2 \square\ |\ \ldots\ |\ \{Y_k\}\ \{Z_k\}\ P_k \square\ ] \uparrow P \uparrow (\uparrow \Psi \uparrow\ \mu_1 \uparrow \Psi \uparrow\ P_1 \uparrow\ |\ \uparrow \Psi \uparrow\ \mu_2 \uparrow \Psi \uparrow\ P_2 \uparrow\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_k \uparrow \Psi \uparrow\ P \uparrow\ ) \uparrow$$

d) Demais casos (intitula-se como *caso geral*)

$$P \rightarrow \uparrow [\ \{Y_1\}\ \{Z_1\}P_1 \square\ |\ \{Y_2\}\ \{Z_2\}P_2 \square\ |\ \ldots\ |\ \{Y_k\}\ \{Z_k\}P_k \square\ ] \uparrow (\uparrow \Psi \uparrow\ \mu_1 \uparrow \Psi \uparrow\ |\ \uparrow \Psi \uparrow\ \mu_2 \uparrow \Psi \uparrow\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_k \uparrow \Psi \uparrow\ ) \uparrow$$

Para casos de recursão central, também assume-se o caso geral:

$$P \rightarrow [\ \{Y_1\}\ (\ \uparrow \Psi \uparrow\ \mu_{1'}\ \uparrow\ P\ \uparrow\ \mu_{1''}\ \uparrow \Psi \uparrow\ \{Z_1\}P_1 \square\ |\ \uparrow \Psi \uparrow\ \mu_{2'}\ \uparrow\ \{Z_2\}P_2 \square\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_{k'}\ \uparrow\ P\ \uparrow\ \mu_{k''}\ \uparrow \Psi \uparrow\ \{Y_k\}\ )\ \uparrow\ \{Z_k\}P_k \square\ ]$$

No exemplo acima, o não-terminal P aparece como elemento intermediário entre as seqüências μ.

4.2.4 Remoção das recursões à direita e à esquerda

As associações já apresentadas podem ser resumidas em $P \rightarrow P\ e\ |\ d\ P\ |\ g$, onde e, d e g representam seqüências de terminais e/ou não-terminais, incluindo as respectivas ações semânticas, e são descritos por:

<u>seqüência e:</u>

$$\{Y_1\}\ \uparrow (\ \uparrow \Psi \uparrow\ \mu_1\ \uparrow \Psi \uparrow\ \{Z_1\}P_1 \square\ |\ \uparrow \Psi \uparrow\ \mu_2\ \uparrow \Psi \uparrow\ \{Z_2\}P_2 \square\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_k\ \uparrow \Psi \uparrow\ \{Y_k\}\ )\ \{Z_k\}P_k \square$$

<u>seqüência d:</u>

$$\{Y_1\}\ \uparrow (\ \uparrow \Psi \uparrow\ \mu_1\ \uparrow \Psi \uparrow\ \{Z_1\}P_1 \square\ |\ \uparrow \Psi \uparrow\ \mu_2\ \uparrow \Psi \uparrow\ \{Z_2\}P_2 \square\ |\ \ldots\ |\ \uparrow \Psi \uparrow\ \mu_k\ \uparrow \Psi \uparrow\ \{Y_k\}\ )\ \{Z_k\}P_k \square$$

<u>seqüência g:</u>

$$\{Y_1\} \quad \{Z_1\}P_1 \square \quad \{Y_2\} \quad \{Z_2\}P_2 \square$$
$$\uparrow (\uparrow \Psi \uparrow \quad \mu_1 \uparrow \Psi \uparrow \quad | \uparrow \Psi \uparrow \quad \mu_2 \uparrow \Psi \uparrow \quad | \ldots$$

$$\{Y_k\} \quad \{Z_k\}P_k \square$$
$$\ldots \quad | \uparrow \Psi \uparrow \quad \mu_k \uparrow \Psi \uparrow \quad)$$

As recursões de ambos os lados são semelhantes à seqüência e, e recursões centrais são semelhantes à seqüência g, com pequenas diferenças, conforme descrito em 4.2.3.

Podem-se eliminar as auto-recursões fatorando-as em uma única expressão, adotando-se uma solução semelhante àquelas propostas por (NETO, 1993) e (IWAI, 2000):

$$[\quad \{Y_1\} \quad \{Z_1\}P_1 \square \quad \{Y_2\} \quad \{Z_2\}P_2 \square$$
$$P \rightarrow \uparrow (\uparrow (\uparrow \varepsilon \uparrow \backslash \uparrow \Psi \uparrow \quad \mu_1 \uparrow \Psi \uparrow \quad | \uparrow \Psi \uparrow \quad \mu_2 \uparrow \Psi \uparrow \quad | \ldots$$

$$\{Y_k\} \quad \{Z_k\}P_k \square \quad \{Y_1\} \quad \{Z_1\}P_1 \square$$
$$\ldots | \uparrow \Psi \uparrow \quad \mu_k \uparrow \Psi \uparrow \quad) \uparrow (\uparrow \Psi \uparrow \quad \mu_1 \uparrow \Psi \uparrow$$

$$\{Y_2\} \quad \{Z_2\}P_2 \square \quad \{Y_k\} \quad \{Z_k\}P_k \square$$
$$| \uparrow \Psi \uparrow \quad \mu_2 \uparrow \Psi \uparrow \quad | \ldots | \uparrow \Psi \uparrow \quad \mu_k \uparrow \Psi \uparrow \quad) \uparrow$$

$$\{Y_1\} \quad \{Z_1\}P_1 \square \quad \{Y_2\} \quad \{Z_2\}P_2 \square$$
$$(\uparrow \varepsilon \uparrow \backslash \uparrow \Psi \uparrow \quad \mu_1 \uparrow \Psi \uparrow \quad | \uparrow \Psi \uparrow \quad \mu_2 \uparrow \Psi \uparrow \quad | \ldots$$

$$\{Y_k\} \quad \{Z_k\}P_k \square \quad]$$
$$\ldots | \uparrow \Psi \uparrow \quad \mu_k \uparrow \Psi \uparrow \quad) \uparrow) \uparrow$$

4.2.5 Tratamento de não-determinismos

Durante o reconhecimento de sentenças pelo *parser*, poderão ocorrer *não-determinismos*, isto é, a partir de um símbolo da cadeia de entrada e de um estado do transdutor, existirem 2 ou mais estados distintos a se transitar. Esta

ocorrência não é desejável, porque implica na execução de caminhos alternativos para a cadeia de entrada, demandando tempo adicional e uma maior complexidade no processo de reconhecimento.

Conforme apontado em (NETO, 1993), não-determinismos poderão ocorrer por:

- presença de prefixos comuns explícitos;
- presença de não-terminal na extremidade esquerda de uma das opções de um agrupamento;
- presença explícita de cadeia vazia como uma das opções de um agrupamento;
- presença de uma construção cíclica na extremidade esquerda de uma das opções de um agrupamento;

A manipulação adequada para cada um dos casos acima mencionados, incluindo-se a existência de ações semânticas, é apresentada a seguir.

a) Presença de prefixos comuns explícitos

Prefixos são seqüências de terminais, não-terminais ou ações semânticas existentes no início de cada uma das opções de uma regra. A existência de prefixos comuns em opções de um agrupamento não é desejável, pois trajetórias de reconhecimento equivalentes são geradas, aumentando a ocorrência de não-determinismos. O agrupamento (fatoração) dos prefixos comuns ocasiona a geração de uma única trajetória de reconhecimento. Para o processo de arupamento dos prefixos comuns, é necessário que se calcule o *comprimento* de cada prefixo, o qual é definido pela quantidade de elementos (terminais, não-terminais ou ações semânticas) que o mesmo possui.

Os prefixos comuns às opções devem ser detectados, e devem ser fatorados os prefixos mais longos possíveis e, a partir destes, os prefixos comuns ao maior número de opções. No caso da existência de prefixos de mesmo comprimento ou de mesmo número de opções, não haverá uma ordem específica de escolha.

Enquanto existirem prefixos comuns a serem postos em evidência, deve-se repetir esta operação. Para um melhor entendimento, detalha-se a seguir a ocorrência de prefixos comuns.

Detalhamento:

Dadas duas expressões rotuladas S_1 e S_2 :

$$S_1 = \overset{X_0}{\uparrow}\ \mu_1\ \overset{X_1}{\uparrow}\ \mu_2\ \overset{X_2}{\uparrow}\ \ldots\ \overset{X_{n-1}}{\uparrow}\ \mu_n\ \overset{X_n}{\uparrow}\ \delta_1\ \overset{Y_1}{\uparrow}\ \delta_2\ \overset{Y_2}{\uparrow}\ \ldots\ \overset{Y_{p-1}}{\uparrow}\ \delta_p\ \overset{Y_p}{\uparrow}$$

$$S_2 = \overset{Z_0}{\uparrow}\ \mu_1\ \overset{Z_1}{\uparrow}\ \mu_2\ \overset{Z_2}{\uparrow}\ \ldots\ \overset{Z_{n-1}}{\uparrow}\ \mu_n\ \overset{Z_n}{\uparrow}\ \varphi_1\ \overset{W_1}{\uparrow}\ \varphi_2\ \overset{W_2}{\uparrow}\ \ldots$$

$$\ldots\ \overset{W_{q-1}}{\uparrow}\ \varphi_q\ \overset{W_q}{\uparrow}$$

Assumindo-se que:

- μ_i é um prefixo comum entre as expressões S_1 e S_2, e $\mu_i \in (N \cup \Sigma)$, sendo $1 \leq i \leq n$;
- δ_i é um prefixo da expressão S_1 , e $\delta_i \in (N \cup \Sigma)$, sendo $1 \leq i \leq p$, e que não tem um prefixo equivalente em mesma posição em S_2;
- φ_i é um prefixo da expressão S_2 , e $\varphi_i \in (N \cup \Sigma)$, sendo $1 \leq i \leq q$, e que não tem um prefixo equivalente em mesma posição em S_1;
- X_i e Z_i correspondam aos rótulos de cada um dos prefixos comuns μ_i das expressões S_1 e S_2, sendo $0 \leq i \leq n$;
- Y_j e W_k correspondam aos rótulos de cada um dos prefixos não equivalentes δ_i e φ_i respectivamente e das expressões S_1 e S_2, sendo $0 \leq j \leq p$, e $0 \leq k \leq q$;
- para os prefixos comuns μ_i, define-se um índice m correspondente ao índice obtido para o último rótulo em que $X_i = Z_i$, lendo-se as expressões S_1 e S_2 da esquerda para a direita, ou seja:

$$X_0\ X_1\ X_2\ \ldots\ X_m = Z_0\ Z_1\ Z_2\ \ldots\ Z_m\ , 0 \leq m \leq n$$

Colocando-se em evidência então os prefixos comuns, monta-se a expressão S_F , devidamente fatorada. Nesta expressão, são atribuídos os rótulos aos prefixos comuns que possuem rótulos idênticos em ambas as expressões (índice $0 \leq m$).

Uma cadeia composta pela concatenação dos rótulos não equivalentes (índice m+1 ≤ n) é agregada ao rótulo do primeiro elemento (terminal, não-terminal ou ação semântica) que constitui cada uma das opções entre parênteses na associação (δ_1 e φ_1) abaixo:

$$S_F = \uparrow \mu_1 \uparrow \mu_2 \uparrow \ldots \uparrow \mu_m \uparrow \mu_{m+1} \uparrow \mu_{m+2} \uparrow \ldots$$

$$\uparrow \mu_n \uparrow (\uparrow \delta_1 \uparrow \delta_2 \uparrow \ldots$$

$$\uparrow \delta_p \uparrow | \uparrow \varphi_1 \uparrow \varphi_2 \uparrow \ldots$$

$$\uparrow \varphi_q \uparrow) \uparrow$$

Com isso a expressão está devidamente fatorada, eliminando-se não-determinismos causados pela presença de prefixos comuns explícitos.

b) Presença de não-terminal na extremidade esquerda de uma das opções de um agrupamento

Na ocorrência de um não-terminal na extremidade esquerda de uma das opções, basta substituir o não-terminal pela expressão rotulada que o define, entre parênteses. Este procedimento reduz a quantidade de não-determinismos e aumenta a eficiência no reconhecimento, porque a existência de não-terminais em extremidade esquerda impõe a chamada de uma submáquina que represente o não-terminal, e a conseqüente inclusão de uma transição em vazio que desvie o reconhecimento para esta submáquina, o que não é desejável.

Detalhamento:

Dada uma expressão rotulada S, e um não-terminal K:

$$S = \uparrow K \uparrow \mu_1 \uparrow \mu_2 \uparrow \ldots \uparrow \mu_n \uparrow$$

Assumindo-se que φ' seja a expressão rotulada que define o não-terminal K, obtém-se a seguinte expressão final S_F:

		Y		X_0		X_1		X_2		X_{n-1}		X_n	
S_F	=	↑	(φ')	↑	μ_1	↑	μ_2	↑	...	↑	μ_n	↑	

A substituição do não-terminal K pela expressão rotulada que o define facilita a análise da expressão com a finalidade de fatorá-la e de eliminar outras ocorrências de não-determinismo.

c) Presença explícita de cadeia vazia como uma das opções de um agrupamento

Na ocorrência de cadeia vazia (ε), transforma-se a expressão em outra equivalente para efeito de eliminação de não-determinismos, fazendo-se desaparecer o símbolo ε, em virtude de ser a cadeia vazia o elemento neutro da operação de concatenação. A eliminação da cadeia vazia elimina uma transição em vazio, reduzindo a ocorrência de não-determinismos.

Detalhamento:

Dada uma expressão rotulada S:

		X_0		X_1		X_2		X_3		X_4		X_5		X_6		X_7	
S	=	↑	δ'	↑	(	↑	ε	↑	|	↑	μ'	↑	)	↑	φ'	↑	

Assumindo-se que:

- δ' , μ' , φ' correspondam a outras expressões rotuladas, sendo δ' e φ' fatores na expressão S;
- os rótulos inicial e final de δ' são respectivamente X_0 e X_1;
- os rótulos inicial e final de μ' são respectivamente X_4 e X_5;
- os rótulos inicial e final de φ' são respectivamente X_6 e X_7;

Transforma-se a expressão acima na seguinte expressão equivalente " S_F ":

X_0 X_1 X_2 X_3 X_6 X_7 X_0 X_1X_4 X_5X_6 X_7

S_F = (↑ δ' ↑ φ' ↑ | ↑ δ' ↑ μ' ↑ φ' ↑)

Esta expressão não contém a cadeia vazia, e portanto não haverá a necessidade da criação de uma transição em vazio posteriormente, na construção do *parser*, diminuindo a ocorrência de não-determinismos.

d) Presença de uma construção cíclica na extremidade esquerda de uma das opções de um agrupamento

Este tipo de construção dá origem à cadeia vazia, podendo encobrir prefixos comuns às cadeias geradas (NETO, 1993). Neste caso substitui-se o fator cíclico por uma expressão rotulada equivalente.

Detalhamento:

Dada uma expressão rotulada S:

X_0 X_1 X_2 X_3 X_4 X_5

S = ↑ (↑ ε ↑ \ ↑ μ' ↑) ↑

Sendo μ' correspondente a uma expressão rotulada, constrói-se a seguinte expressão equivalente S_F :

X_0X_1 X_2X_5 $X_0X_1X_2X_3$ X_4 X_3 X_4 X_5

S_F = ↑ ε ↑ | ↑ μ' ↑ (↑ ε ↑ \ ↑ μ' ↑) ↑

Desta forma, nota-se a presença de μ' como fator esquerdo ao agrupamento, e recai-se no item (c), onde há a presença da cadeia vazia, eliminando-se com isso o não-determinismo.

Este processo é interessante neste caso, porque durante a trajetória de reconhecimento de uma sentença da linguagem, identifica-se imediatamente a ocorrência de μ'. Antes da transformação (expressão S), haveria a necessidade de uma transição em vazio antes de se verificar a ocorrência de μ'.

Os tratamentos de não-determinismos propostos nos itens (a), (b), (c) e (d) acima deverão ser aplicados até que os mesmos sejam eliminados, ou que se recaia em alguma das transformações anteriormente obtidas. Neste último caso, há a existência de um não-determinismo que não pode ser eliminado. Quando isto acontece, reaplica-se um ou mais destes tratamentos no sentido de se obter um autômato mais adequado para que atue deterministicamente ao menos nos casos em que isto seja essencial, quando possível.

Com a gramática assim transformada, passa-se à construção do autômato de reconhecimento com base na qual é produzido o transdutor sintático, conforme descrito no capítulo 5.

5 - ANALISADOR SINTÁTICO

O analisador sintático ou *parser* será construído por meio de um transdutor, com base na gramática transformada. A linguagem de entrada do transdutor é o conjunto de todas as sentenças da linguagem definida pela gramática, e a linguagem de saída é o conjunto de todas as árvores sintáticas das sentenças da linguagem. Para cada sentença fornecida ao transdutor, a árvore sintática correspondente é gerada de acordo com a gramática inicialmente fornecida.

Para a construção do transdutor, são necessárias a identificação e atribuição (numeração) dos estados que irão compor o transdutor e a análise de cada um dos rótulos criados, os quais serão responsáveis pela geração da árvore sintática e pela indicação da execução das ações semânticas fornecidas previamente na gramática original. Desta forma, o transdutor atua como reconhecedor de sentenças da linguagem e fornece a árvore sintática correspondente a cada sentença.

5.1 Atribuição de estados

A atribuição, para efeito de identificação, dos números de estados que irão compor o autômato de reconhecimento, segue basicamente a idéia proposta em (NETO, 1993), porém considerando-se a existência das ações semânticas fornecidas na gramática original. O objetivo da atribuição de números de estados é preparar a gramática para a construção do *parser* sintático.

- a_i, b_i, c_i, d_i identificam expressões que compõem uma seqüência de elementos de uma regra de produção, devidamente rotulados, e $i \in$ □ o índice correspondente à expressão tratada, lembrando-se que as ações semânticas referenciadas anteriormente por Ψ estão incluídas nos conjuntos a_i, b_i, c_i, d_i;
- x, y ∈ □ são números de estado designados aos pontos (marcados com o símbolo □) indicados na expressão, durante o processamento dos passos a seguir;
- E é uma expressão qualquer, obtida no tratamento em questão;
- para a atribuição de números aos estados aplicam-se então as seguintes ações (os rótulos associados a cada um dos elementos indicados foram omitidos para a melhor visualização do processo de numeração):

a) Localiza-se os agrupamentos (seqüências de elementos entre parênteses) de uma regra de produção;

b) Isola-se um agrupamento completo e ainda não tratado;

c) Sendo r, s ∈ □ os números de estado previamente designados respectivamente aos pontos imediatamente à esquerda e imediatamente à direita do agrupamento entre parênteses tratado, de uma das formas abaixo:

$$E = \underset{r}{\square} (\square\ a_1\ \square\ | \ \ldots \ | \ \square\ a_m\ \square) \underset{s}{\square}$$

ou

$$E = \underset{r}{\square} (\square\ a_1\ \square\ | \ \ldots \ | \ \square\ a_m\ \square \setminus \square\ b_1\ \square\ | \ \ldots \ | \ \square\ b_n\ \square) \underset{s}{\square}$$

(observar que o símbolo □ indica a posição para a atribuição de número de estado.).

consideram-se os casos a seguir:

- Se existir r, fazer x = r, caso contrário associar a x um número correspondente a um novo estado, e designar x aos pontos extremos esquerdos de todas as opções internas ao agrupamento.
- Se existir s, fazer y = s, caso contrário associar a y um número correspondente a um novo estado e designar y aos pontos extremos direitos de todas as opções internas ao agrupamento.

d) Para agrupamentos do tipo:

$$E = \underset{r}{\square} (\square\ \varepsilon\ \square \setminus \square\ b_1\ \square\ | \ \ldots \ | \ \square\ b_n\ \square) \underset{s}{\square}$$

Cria-se nas extremidades das expressões b_i um único estado x = y, que será associado a cada uma das extremidades destas opções, resultando na expressão:

E = □ (□ ε □ \ □ b_1 □ | ... | □ b_n □) □
r x y y x y x s

e) Para outros casos que não se enquadrem nos casos anteriores, quando for necessário criar novos estados, geram-se sempre x e y distintos;

f) Numeram-se os estados extremos de cada uma das expressões que compõem o agrupamento, obtendo-se uma das configurações abaixo, conforme o caso:

E = □ (□ a_1 □ | ... | □ a_m □) □
r x y x y s

ou

E = □ (□ a_1 □ | ... | □ a_m □ \ □ b_1 □ | ... | □ b_n □) □
r x y x y y x y x s

sendo x = y :

E = □ (□ ε □ \ □ b_1 □ | ... | □ b_n □) □
r x y y x y x s

g) Na expressão obtida, propaga-se o estado x para o ponto externo imediatamente à esquerda do agrupamento, caso a este ponto não tenha sido atribuído nenhum estado:

E = □ (□ a_1 □ | ... | □ a_m □ \ □ b_1 □ | ... | □ b_n □) □
x x y x y y x y x

h) Na expressão obtida, propaga-se o estado y para o ponto externo imediatamente à direita do agrupamento, caso a este ponto não tenha sido atribuído nenhum estado:

E = □ (□ a_1 □ | ... | □ a_m □ \ □ b_1 □ | ... | □ b_n □) □
x x y x y y x y x y

i) Caso em ambos os pontos (à direita e à esquerda) do agrupamento ainda não tenha sido designado nenhum estado, efetuam-se as duas propagações acima indicadas, obtendo-se:

E	=	□	(	□	a_1	□	|	...	|	□	a_m	□	\	□	b_1	□	|	...	|	□	b_n	□	)	□
		x		x		y				x		y		y		x				y		x		y

j) No caso de existir uma única expressão a_i entre as opções, ou seja:

E	=	□	(	□	a_1	□	\	□	b_1	□	|	...	|	□	b_n	□	)	□
		x		x		y		y		x				y		x		y

e a expressão a_i tenha um agrupamento cíclico em sua extremidade esquerda:

a_i	=	□	(	□	c_1	□	|	...	|	□	c_p	□	\	□	d_1	□	|	...	|	□	d_q	□	)	□
		x																						

pode-se propagar o estado x para o interior deste agrupamento:

a_i	=	□	(	□	c_1	□	|	...	|	□	c_p	□	\	□	d_1	□	|	...	|	□	d_q	□	)	□
		x		x						x						x						x		

k) No caso de existir uma única expressão a_i entre as opções, ou seja:

E	=	□	(	□	a_1	□	\	□	b_1	□	|	...	|	□	b_n	□	)	□
		x		x		y		y		x				y		x		y

e a expressão a_i tenha um agrupamento cíclico em sua extremidade direita:

a_i	=	□	(	□	c_1	□	|	...	|	□	c_p	□	\	□	d_1	□	|	...	|	□	d_q	□	)	□
																								y

pode-se propagar o estado y para o interior deste agrupamento:

a_i	=	□	(	□	c_1	□	|	...	|	□	c_p	□	\	□	d_1	□	|	...	|	□	d_q	□	)	□
						y						y		y						y				y

l) No caso da ocorrência simultânea do exposto nos itens j e k, ou seja, quando a expressão a_i se confunde com o agrupamento cíclico:

$$a_i = \underset{x}{\square}(\square c_1 \square | \ldots | \square c_p \square \backslash \square d_1 \square | \ldots | \square d_q \square) \underset{y}{\square}$$

então serão permitidas ambas as propagações, conforme apresentado na expressão abaixo:

$$a_i = \underset{x}{\square}(\underset{x}{\square} c_1 \underset{y}{\square} | \ldots | \underset{x}{\square} c_p \underset{y}{\square} \backslash \underset{y}{\square} d_1 \underset{x}{\square} | \ldots | \underset{y}{\square} d_q \underset{x}{\square}) \underset{y}{\square}$$

m) Aplica-se, de forma análoga ao apresentado nos itens anteriores, a propagação dos estados previamente associados à esquerda e à direita a todos os agrupamentos internos e não cíclicos ainda não tratados;

n) Aos estados ainda não tratados (ou seja, aos quais ainda não foram designados números), deve-se atribuir novos números de estado não repetidos, de forma seqüencial, e diferentes dos já designados;

o) Os elementos de ação semântica Ψ são tratados como se fossem terminais para efeito de numeração de estados. Na construção do *parser* sintático, cada um destes elementos será tratado como uma transição em vazio no transdutor, cuja ação correspondente estará determinada pelo seu rótulo.

Com a conclusão da etapa de numeração dos estados da gramática preparada, passa-se à construção das transições que compõem o autômato/transdutor, conforme se descreve no item a seguir.

5.2 Construção do autômato de reconhecimento

Tendo-se as expressões numeradas, obtidas em 5.1, será necessária a interpretação das mesmas para a construção de um reconhecedor (autômato) para a linguagem cuja gramática foi fornecida inicialmente. O autômato deverá ser um autômato de pilha estruturado, de forma que possa permitir o empilhamento de estados de retorno quando ocorrerem chamadas a submáquinas. Os números atribuídos aos vários

pontos da expressão na etapa anterior serão utilizados como estados deste autômato, e os elementos existentes entre os números obtidos serão suas transições, que poderão corresponder a terminais, não-terminais, cadeias vazias e construções cíclicas. Para cada um destes elementos há um tratamento específico, detalhado a seguir.

Considerando que T é um trecho da expressão E anteriormente obtida, apresenta-se os seguintes possíveis casos:

- Para a ocorrência de cadeia vazia ε:

T = ε
x y

cria-se uma transição em vazio do estado x para o estado y:

$(\gamma, x, \alpha) \rightarrow (\gamma, y, \alpha)$

- Para a ocorrência de ação semântica Ψ:

T = Ψ
x y

cria-se uma transição em vazio do estado x para o estado y:

$(\gamma, x, \alpha) \rightarrow (\gamma, y, \alpha)$

- Para a ocorrência de terminal t:

T = t
x y

cria-se uma transição do estado x para o estado y, consumindo o terminal t:

$(\gamma, x, t\alpha) \rightarrow (\gamma, y, \alpha)$

- Para a ocorrência de não-terminal N:

T = N
x y

cria-se uma transição em vazio para o estado inicial n_0 da submáquina correspondente ao não-terminal N, empilhando-se o estado de retorno y na pilha do autômato para que possa ser executada a transição de retorno da submáquina:

$(\gamma , x, \alpha) \rightarrow (\gamma y , n_0, \alpha)$

- Para agrupamento cíclico do tipo:

T = □ (□ ε □ \ □ μ □) □

x y z t u v

devem ser criadas as seguintes transições:

$(\gamma , x, \alpha) \rightarrow (\gamma , y, \alpha)$

$(\gamma , z, \alpha) \rightarrow (\gamma , v, \alpha)$

$(\gamma , z, \alpha) \rightarrow (\gamma , t, \alpha)$

$(\gamma , t, \alpha) \rightarrow (\gamma , v, \alpha)$

$(\gamma , u, \alpha) \rightarrow (\gamma , t, \alpha)$

- Cada um dos estados n_f associados ao final de alguma das alternativas do não-terminal N anteriormente mencionado, cria-se uma transição de retorno ao estado y da submáquina chamadora:

$(\gamma y , n_f, \alpha) \rightarrow (\gamma , y, \alpha)$

- O estado que estiver na extremidade esquerda da expressão que define a submáquina N anteriormente apontada, será o estado inicial desta submáquina.
- Os estados inicial e final do autômato serão aqueles associados ao início e ao final da expressão que define a raiz da gramática em questão.

O conjunto de transições geradas neste processo define o autômato que reconhece a linguagem especificada pela gramática. Este autômato agora está pronto para ser transformado em um transdutor que terá como saída a árvore sintática ou de derivação para uma sentença da linguagem, conforme apresentado a seguir.

5.3 Construção do transdutor

Durante o processo de construção do autômato de reconhecimento demonstrado na etapa anterior, incluem-se transições adicionais àquelas de reconhecimento de uma determinada cadeia de entrada, de forma a se produzir na saída a árvore sintática correspondente ao reconhecimento desta cadeia.

Os rótulos incluídos nas etapas anteriores representam o histórico de reconhecimento, e fornecem informações que permitem a construção da correspondente árvore sintática. Para isso, aplica-se uma tabela para o mapeamento dos rótulos de modo que estes produzam como saída a correspondente árvore sintática, fornecida em formato texto, conforme descrito seguir.

5.3.1 Formato da árvore sintática de saída

O transdutor produzirá para uma cadeia de entrada, a correspondente árvore de saída no formato texto.

Considerando-se σ como um terminal (folha da árvore), γ_q, $1 \leq q \leq n$ uma sub-árvore ou uma folha, e p_w, $1 \leq w \leq n$ um não-terminal, a seqüência de caracteres que deverá representar a árvore sintática será constituída pelos seguintes símbolos, onde:

- ( ) ... denota uma folha associada à cadeia vazia;
- (σ) ... denota uma folha correspondente ao terminal σ;
- (γ_1 γ_2 ... γ_n X_1) denota X_1 como a raiz das sub-árvores;
- [... X_1) ... X_2) ... X_n) representa a mesma árvore que aquela representada por (((... X_1) ... X_2) ... X_n).

Observações

- [...] os colchetes são usados como delimitadores e podem ser omitidos.

A figura abaixo mostra a árvore sintática representada por
[(a) Q) (b) R) (c) (d) S)].

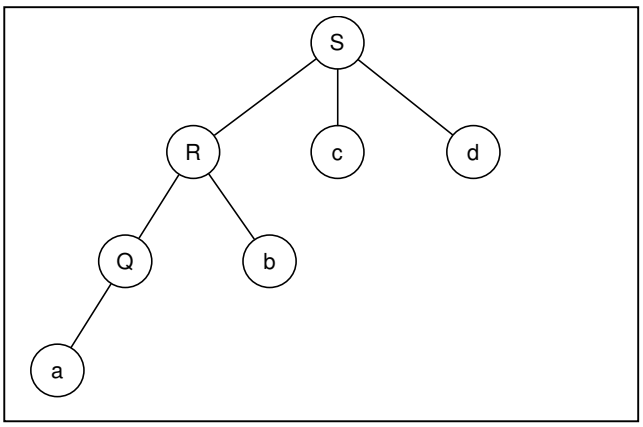


Fig. 4 – Árvore sintática correspondente à expressão
[(a) Q) (b) R) (c) (d) S)]

5.3.2 Geração da árvore sintática

À medida em que o autômato construído na etapa anterior é processado, os rótulos definidos para os elementos que compõem as regras da gramática deverão ser analisados de forma a produzirem símbolos na saída, ou empilhá-los/desempilhá-los para tratamento posterior. Para cada tipo de rótulo, pelo menos uma destas ações é aplicada.

A tabela abaixo define as ações a serem tomadas para cada rótulo, sendo composta por três colunas: *Rótulo*, indicando para qual rótulo deverá ser aplicada uma ação; *Saída*, indicando quais símbolos ou ações deverão ser aplicadas na saída; e *Pilha*, que representa a ação de empilhamento e desempilhamento de símbolos.

Os símbolos que aparecem na tabela podem representar terminais, não-terminais e ações semânticas. A seguir, apresenta-se uma breve explicação de cada um destes elementos:

ε ... representa a cadeia vazia;

σ ... define a ocorrência de um símbolo terminal;

P_i ... define a ocorrência de um não-terminal $P_i \in V - \Sigma$, e
P_i associado a alguma regra recursiva à esquerda;

P_i□ ... define a ocorrência de um não-terminal $P_i \in V - \Sigma$, e não associado a uma recursão;

P_i□ ... define a ocorrência de um não-terminal $P_i \in V - \Sigma$, e P_i associado a alguma regra recursiva à direita;

[ou] ... ocorrência de delimitadores utilizados na construção da árvore

{Ψ} ... indica uma ação semântica;

↓ ... indica que todo símbolo a seguir deverá ser empilhado;

↑ ... indica o desempilhamento dos símbolos a seguir;

π ... é um meta-caracter que simboliza a seqüência de todos os elementos da pilha de transdução compreendidas desde o seu topo até a ocorrência do símbolo anterior ao delimitador].

Rótulo	**Saída**	**Pilha**
ε	( )	(nenhuma ação)
σ	(σ)	(nenhuma ação)
P_i□	P_i□)	(nenhuma ação)
P_i□	P_i□) π	↑ π
P_i□	(	↓) P_i□
[	[(	↓]
]	π]	↑ π]
{Ψ}	executar a ação semântica Ψ	

Tabela II – Mapeamento dos rótulos

Para a aplicação desta tabela, durante o processamento do autômato gerado anteriormente, deve-se tomar, para cada transição, cada um dos elementos que figuram no rótulo do estado de destino, e obter, para tal elemento, uma correspondência na coluna *Rótulo* da tabela, aplicando-se a saída indicada, bem como o empilhamento/desempilhamento de símbolos da pilha de transdução, quando isto estiver indicado. As ações semânticas, quando existirem, deverão ser aplicadas.

A pilha de transdução é utilizada para o empilhamento e desempilhamento de símbolos durante o processo de reconhecimento.

A cada vez que é solicitado um desempilhamento de símbolos, cada símbolo previamente empilhado vai sendo removido do topo da pilha, sucessivamente, até que seja encontrado o caracter]. Na solicitação seguinte de desempilhamento, os símbolos remanescentes serão desempilhados até que seja encontrado um caracter] ou alcançado o fundo da pilha. A pilha de transdução, possui então, um conjunto de símbolos separados pelo separador].

Detalhando-se o exposto anteriormente, seja uma cadeia de símbolos armazenados na pilha até] exclusive, denominada Δ_i. O desempilhamento destes elementos produzirá a seqüência exposta acima, e é denominada π.

A seqüência de símbolos empilhados Δ_i formará a pilha de transdução Θ, que utilizará como separadores os elementos] , conforme ilustrado na figura abaixo:

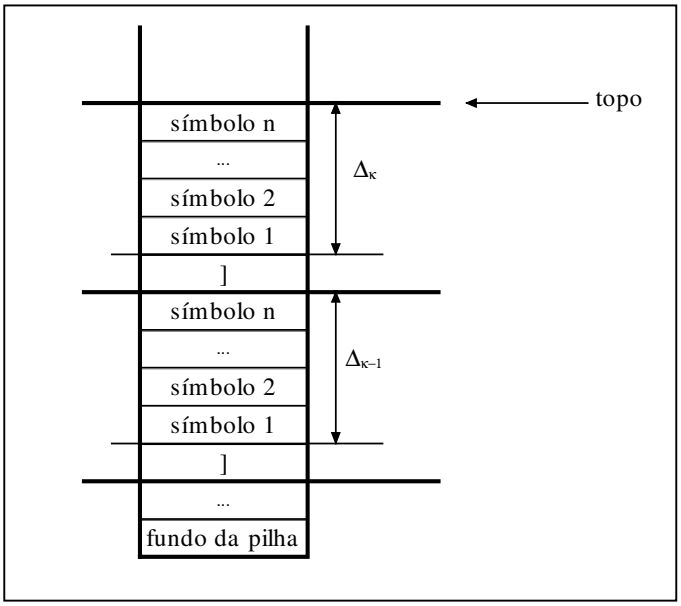


Fig. 5 - Pilha de transdução Θ

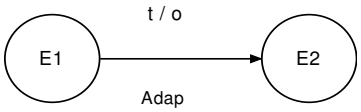
A cada operação de empilhamento, um símbolo qualquer será incluído na pilha, acima do seu topo corrente, e o ponteiro para o topo é então movido de forma coerente. A cada operação de desempilhamento, todos os símbolos que Δ_i representa, serão desempilhados.

Às operações de empilhamento e desempilhamento, estão associados os procedimentos Emp(*símbolo*) e Desempilha( ) respectivamente, no desenvolvimento da pilha de transdução à seguir. Ao ocorrer a ação de desempilhamento, os símbolos desempilhados são incluídos na linha de texto correspondente à árvore sintática.

5.3.3 Construção da pilha de transdução

A implementação da pilha de transdução é realizada através de ações adaptativas, onde a ação de desempilhamento de símbolos produz na saída (linha de texto que corresponde à árvore sintática) os elementos até a ocorrência do primeiro símbolo] , que corresponde, na tabela de mapeamento à seqüência ↑ π]. A ação de empilhamento será tratada de forma convencional. Para o efeito representado por ↑ π, propõe-se o desempilhamento ↑ π] e o posterior empilhamento de].

O funcionamento da pilha proposta será ilustrado através de uma execução passo-a-passo. Para isso, cada uma das etapas é apresentada, e ao final, é mostrado o código (ou conjunto de ações adaptativas) necessário para implementá-la. Adotou-se a seguinte representação para o autômato (transdutor) que representa a pilha de transdução:

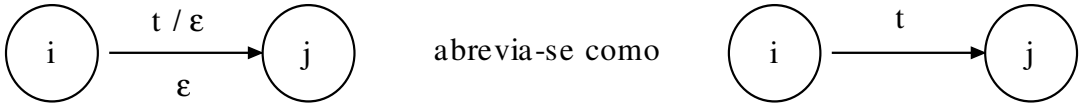


Nesta representação:

E1, E2	...	representam estados
t / o	...	representam respectivamente a transição (t) e a produção de caracteres na saída (o).
Adap	..	representa a ação adaptativa a ser executada.

Caso exista, a ação adaptativa Adap pode especificar uma ação anterior, uma ação posterior, ou ambas: Ant . , . Pos ou Ant . Pos respectivamente, sendo Ant e Pos chamadas de funções adaptativas.

Abreviatura:



Para a construção da pilha, criaram-se funções adaptativas com o objetivo de reajustar a configuração do autômato que a representa. Estas funções estão descritas na tabela abaixo e são detalhadas a seguir, através de um exemplo de funcionamento.

Nome da Função Adaptativa	***Descrição***	***Dependências***
Emp(x)	Empilhar o símbolo x na pilha de transdução.	ElCol , ElTran
ElCol	Reposicionar uma transição em que haja o consumo do símbolo] para o estado final de desempilhamento, e redefinir a próxima pilha através de uma transição com o token *top*.	AjTranColP2
ElTran	Eliminar a transição corrente	(não há)
AjTop	Localizar o estado a partir do qual há o marcador de topo da pilha e adicionar uma transição do estado inicial de desempilhamento para este estado	ElTran
AjTranColP2	Eliminar a transição corrente que aponta para o final da pilha.	(não há)

Tabela III - Descrição básica das funções adaptativas utilizadas na construção da pilha de transdução Θ

Na construção da pilha, assume-se que o empilhamento de símbolos será realizado pela chamada à uma função adaptativa Emp( ), que inclui transições no transdutor

que representa a pilha; o desempilhamento de símbolos é realizado através de um desvio do *parser* para este transdutor, que emitirá os símbolos da pilha no texto de saída. Ao término do desempilhamento, deverá haver o retorno do controle ao reconhecedor que atua como *parser*. Por esta razão, a pilha é construída como uma submáquina que devolve o controle do processamento ao término de sua execução. A seguir, detalha-se o funcionamento da pilha proposta, passo-a-passo:

a) Posição inicial da pilha

Para marcar o topo da pilha, considera-se uma transição que contenha um token *top* □ ∑. A criação desta transição possibilita a localização do topo da pilha nos passos a seguir. Assume-se os estados q_1 e q_f como inicial e final da pilha, respectivamente, conforme indicado na figura abaixo:

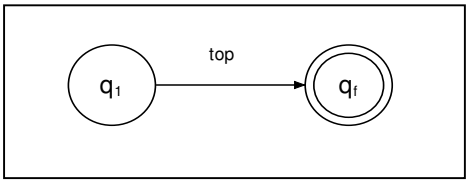


Fig. 6 – Aspecto inicial do autômato que representa a pilha

É importante notar que o *token* top apenas serve para a localização do topo da pilha, não sendo utilizado como símbolo a ser reconhecido na cadeia de entrada. Quando localizado, uma ação adaptativa criará uma transição do estado corrente ao estado q_f, que representa o retorno da submáquina "pilha" em questão.

b) Empilhamento de símbolos

A função Emp(σ) deverá promover um empilhamento do símbolo σ , realizando as seguintes operações:

- procurar a transição marcada com o token top, que indicará a posição corrente do topo da pilha;
- eliminar esta transição;
- incluir uma transição em vazio que deverá produzir como saída o símbolo σ passado como parâmetro, e associá-la a duas funções adaptativas: ElCol, a ser realizada antes da transição, e responsável pela finalização da pilha quando σ =],

e ElTran, a ser realizada após a transição, responsável por apontar o topo da pilha para o próximo símbolo a ser consumido. Ambas serão detalhadas a seguir, na execução de um desempilhamento.

A função de empilhamento na forma proposta em 3.2 é descrita por:

Emp(σ):

?[	?x,	top,	?y,	nop,	nop,	nop	]
-[	?x,	top,	?y,	nop,	nop,	nop	]
+[	?x,	eps ,	?y,	nop,	?p1,	ElCol.ElTran	]
+[	*new,	top,	?x,	nop,	nop,	nop	]

Onde:

?x, ?y	→	variáveis que irão conter os números de estado para as transições encontradas;
nop	→	representa nenhuma operação realizada;
eps	→	representa uma transição em vazio (ε);
*new	→	representa a criação de um novo estado;
?p1	→	representa o conteúdo do campo output encontrado.

A execução de Emp(a) produz então o seguinte autômato/transdutor:

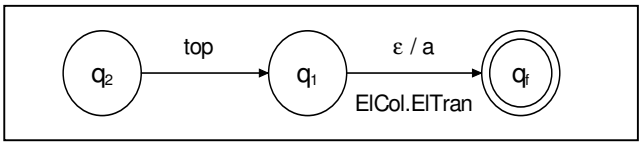


Fig. 7 – Autômato após a execução de Emp(a)

Notar que o indicador top agora aponta o primeiro elemento a ser desempilhado.

A execução sucessiva dos comandos Emp(b), Emp(]), Emp(]), Emp(v), produz a seguinte topologia:

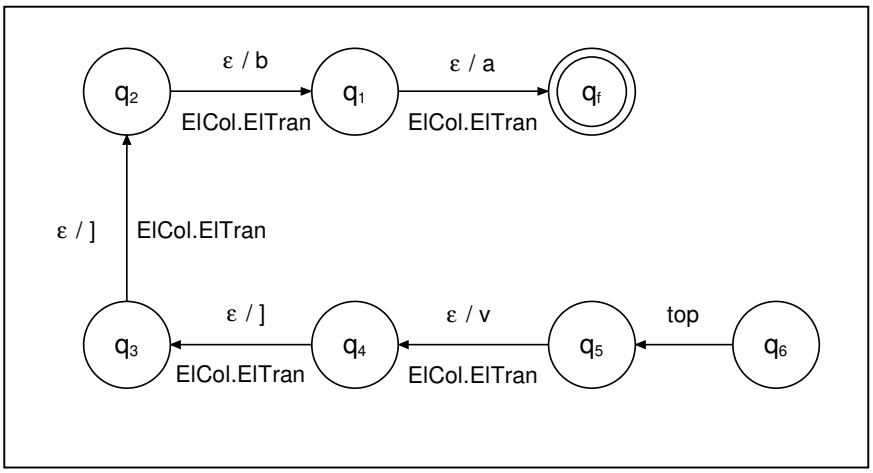


Fig. 8 – Autômato após a aplicação sucessiva da ação Emp para os símbolos a, b,],], v

Representando este autômato no formato da pilha de transdução Θ apresentada, monta-se a figura abaixo:

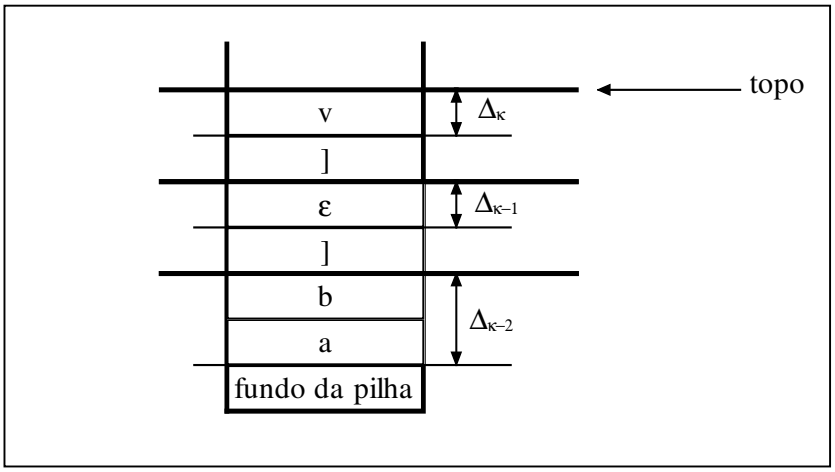


Fig. 9 - Pilha de transdução Θ referente ao empilhamento dos símbolos a, b,],], v

a) Desempilhamento de símbolos

A ação de desempilhamento é obtida a partir da chamada de uma submáquina cujo estado inicial é previamente ajustado para r_0. Esta submáquina é composta por uma transição em que há a chamada de uma função adaptativa posterior AjTop, conforme indicado na figura abaixo:

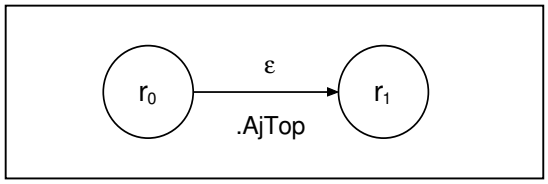


Fig. 10 – Submáquina de desempilhamento

A função adaptativa AjTop é responsável pelas seguintes ações:

- localizar o marcador de topo da pilha através da busca da transição com a marca top;
- eliminar esta transição;
- incluir uma transição do estado r_1 (ponto de início de leitura) para o estado para o qual a transição com a marca top apontava.

A descrição da função AjTop na forma proposta em 3.2 é:

AjTop:

?[?x, top, ?y, nop, nop, nop]
-[?x, top, ?y, nop, nop, nop]
+[r_1, eps, ?y, nop, nop, .ElTran]

onde:

?x, ?y → irão conter os números de estado para as transições encontradas;
. ElTran → chamada à função adaptativa posterior ElTran.

Para a execução de um desempilhamento, é necessária a chamada da submáquina de desempilhamento, que se inicia em r_0.

Como exemplo, para a pilha criada anteriormente, a chamada da submáquina de desempilhamento provocará inicialmente a chamada de AjTop conforme explicado, e a leitura prosseguirá a partir do estado r_1. Notar a eliminação da transição $q_6 \rightarrow q_5$, que marca o topo da pilha, e a inclusão da transição $r_1 \rightarrow q_5$ que conduzirá a leitura ao início do desempilhamento.

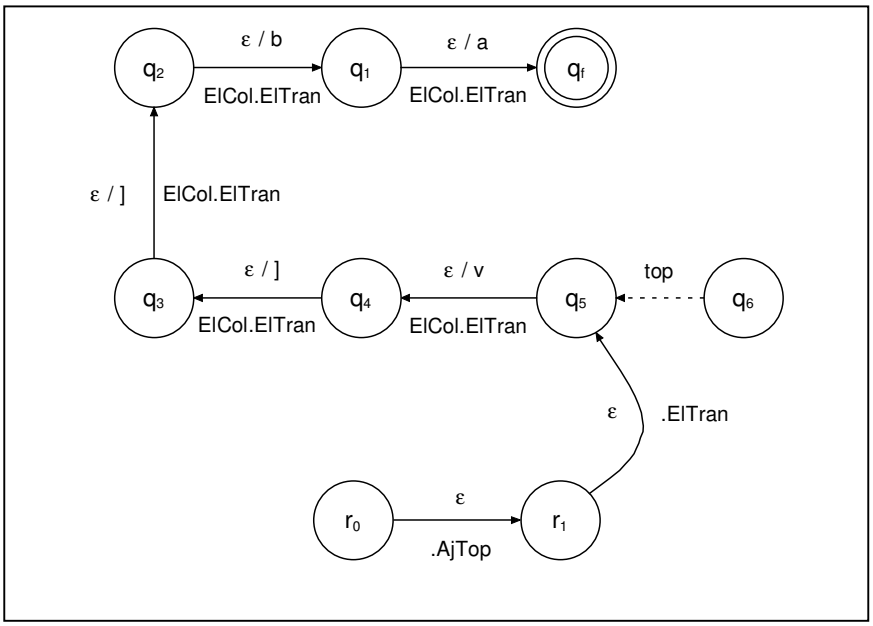


Fig. 11 - Pilha de transdução após a execução da transição $r_0 \rightarrow r_1$

Ao executar a transição $r_1 \rightarrow q_5$, há a eliminação da transição corrente (notar que esta transição está indicada em tracejado, indicando-se sua exclusão), ocasionada pela função adaptativa .ElTran, que pode ser construída da seguinte forma:

ElTran:

?[?x, ?z, ?sta, nop, ?k, nop]
-[?x, ?z, ?sta, nop, ?k, nop]

onde:

?x → qualquer origem;
?sta → estado corrente;
?z → qualquer transição;
?k → qualquer output (saída);

A configuração do autômato ao atingir o estado q_5 passa a ser:

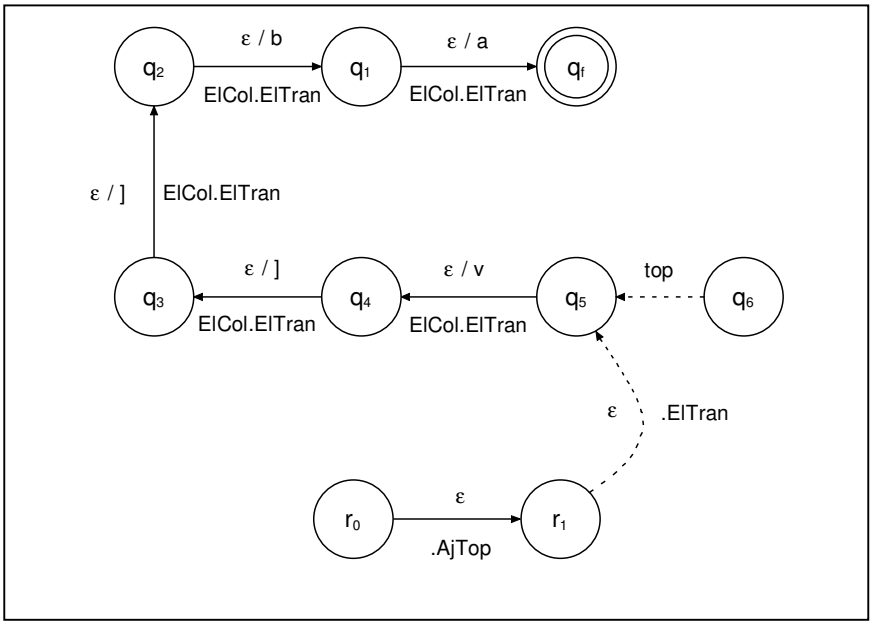


Fig. 12 - Pilha de transdução após a execução da transição $r_1 \rightarrow q_5$

Ao executar a transição $q_5 \rightarrow q_4$, há a chamada da função adaptativa ElCol. A função ElCol realiza as seguintes ações:

- procura uma transição em vazio a partir do estado corrente, em que haja a geração de um símbolo];
- ao encontrá-la, elimina esta transição ;
- inclui uma transição em vazio do estado corrente ao estado q_f (final da pilha), com chamada à função adaptativa AjTranColP2 (realizada após o consumo da transição).
- Incluir uma transição com o token *top*, do estado corrente ao estado de destino da transição eliminada no passo anterior, marcando-se o topo da pilha para a utilização num próximo desempilhamento.

Esta função é descrita por:

ElCol:

?[	?sta,	eps,	?y,	nop,	"]",	nop]
-[	?sta,	eps,	?y,	nop,	"]",	nop]
+[	?sta,	eps,	q_f,	nop,	nop,	.AjTranColP2]
+[	?sta,	top,	?y,	nop,	nop,	nop]

onde:

?y → estado de destino;

?sta → estado corrente;

A execução de ElCol sobre a pilha apresentada, não acarretará nenhuma alteração, uma vez que, no estado corrente, não há qualquer transição que consuma o símbolo]. Efetua-se então a emissão do símbolo v na saída, o qual havia sido anteriormente empilhado. Em seguida, há a chamada à função adaptativa ElTran, a qual elimina a transição corrente, e então a última transição ($q_5 \rightarrow q_4$) é eliminada, obtendo-se o seguinte autômato:

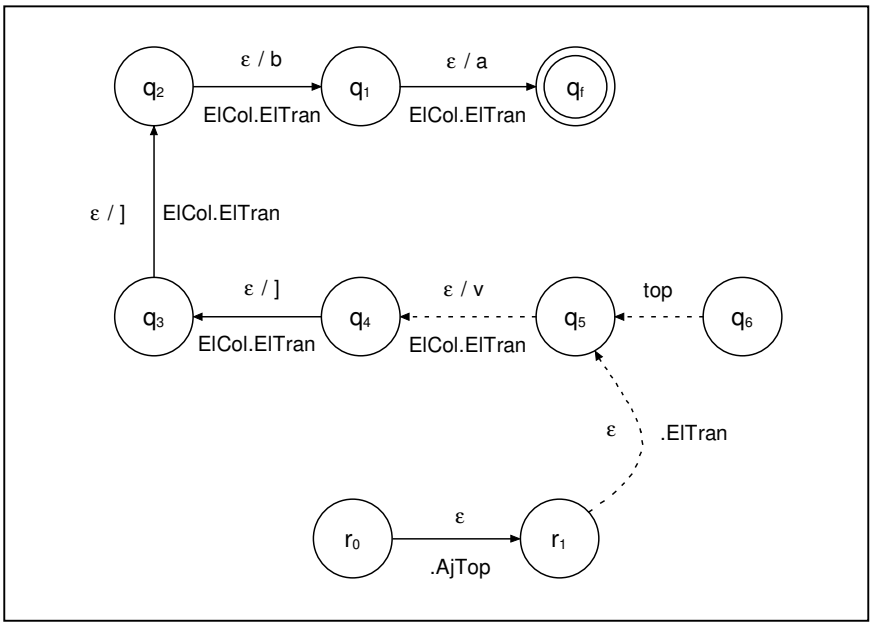


Fig. 13 - Pilha de transdução após a execução da transição $q_5 \rightarrow q_4$

Com isso, passa-se para a transição $q_4 \rightarrow q_3$ em que há novamente a chamada à função adaptativa ElCol. Desta vez, o símbolo] será encontrado, terminando a seqüência de desempilhamento. Os símbolos remanescentes na pilha somente poderão ser retirados na próxima solicitação de desempilhamento.

A execução de ElCol para a transição $q_4 \rightarrow q_3$ executará as seguintes operações:

- eliminação da transição $q_4 \rightarrow q_3$ (pois se trata de transição referenciando o separador);
- criação de uma transição em vazio $q_4 \rightarrow q_f$, que ativa a ação adaptativa AjTranColP2, que deverá ser executada após a chegada ao estado q_f ;
- criação de uma transição marcada com top $q_4 \rightarrow q_3$. Esta transição servirá de "memória" da posição de topo da pilha.

O diagrama do autômato que descreve a pilha fica então com o seguinte aspecto:

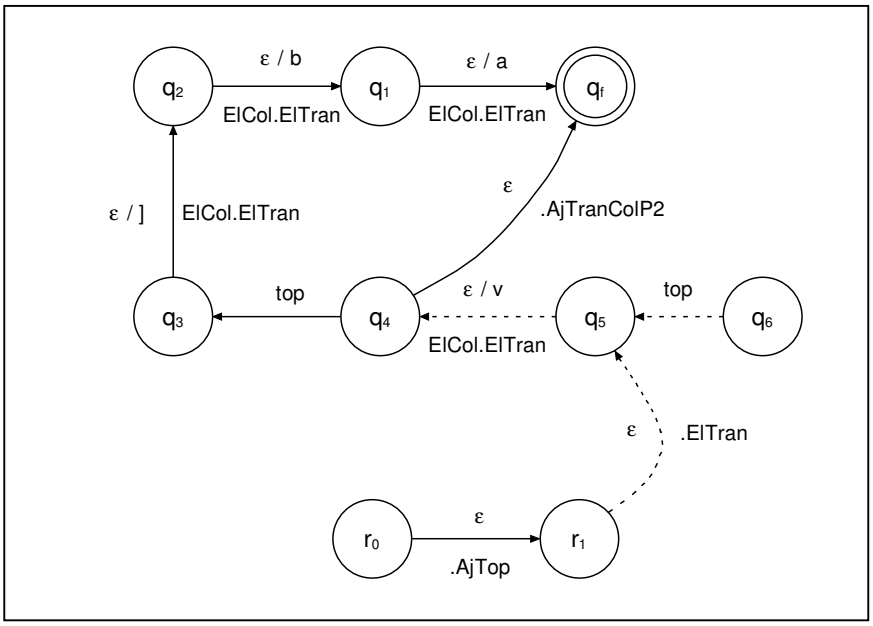


Fig. 14 - Pilha de transdução após a execução da transição $q_4 \rightarrow q_3$

Com isso, a transição $q_4 \rightarrow q_f$ passa a ser executada, e ao final, há a chamada de AjTranColP2. A função AjTranColP2 é responsável pela eliminação da transição corrente que tem como estado de destino o estado q_f. A função AjTranColP2 fica descrita como:

AjTranColP2:

?[?x, eps, q_f, nop, nop, nop]

-[?x, eps, q_f, nop, nop, nop]

onde:

?x → um estado de origem;

nop → nenhuma ação

O autômato fica com a seguinte configuração, após esta transição:

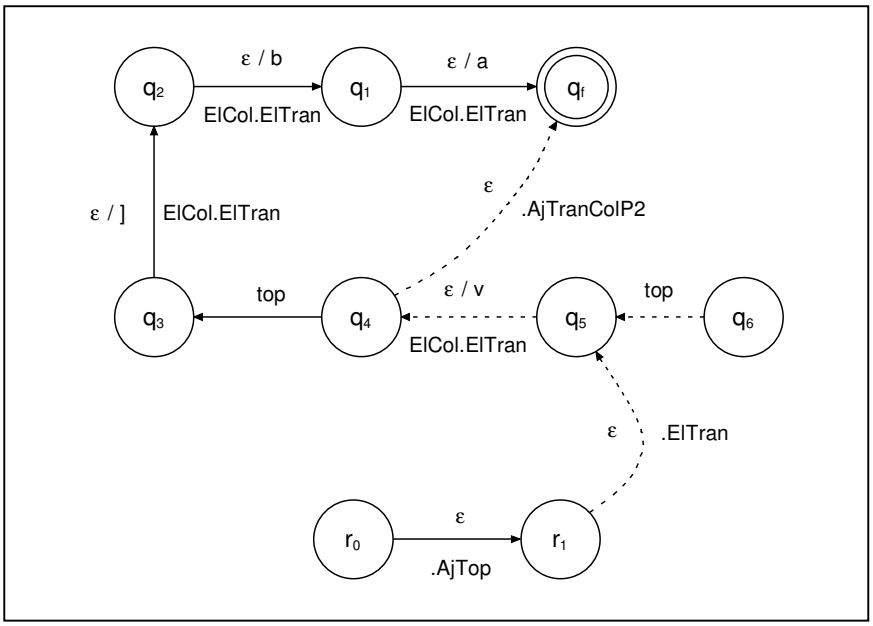


Fig. 15 - Pilha de transdução após a execução da transição $q_4 \rightarrow q_f$

Com isso, termina o desempilhamento até o separador, tendo sido emitido na saída (output) o símbolo v.

Ao ser realizado o desempilhamento seguinte, chama-se novamente a submáquina, a partir de seu estado r_0. Nesta transição, há a chamada da função adaptativa AjTop, responsável pela transição do estado r_1 (ponto de início de leitura) para a posição que a transição top apontava antes, resultando a seguinte configuração:

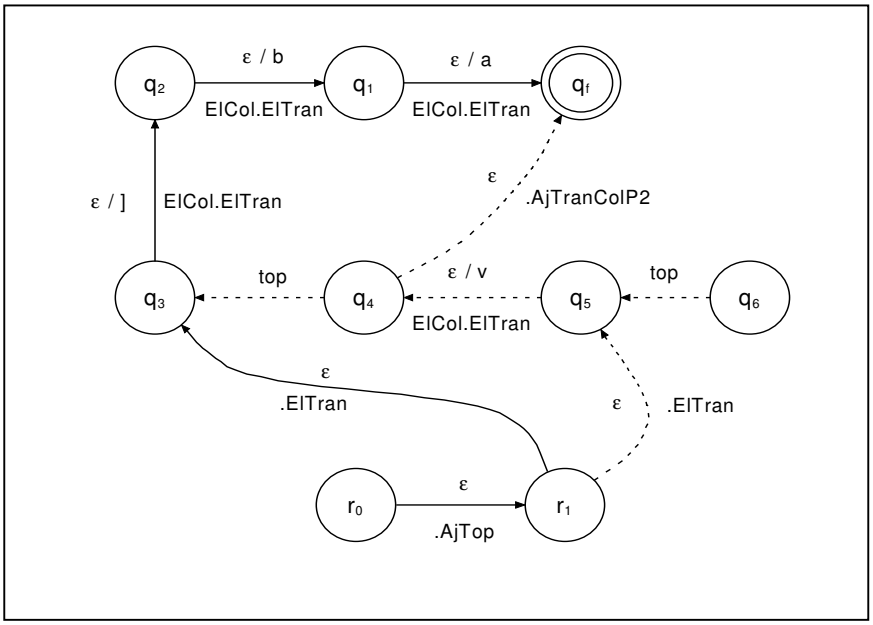


Fig. 16 - Pilha de transdução após a execução da transição $r_0 \rightarrow r_1$

Ao processar a transição $r_1 \rightarrow q_3$, há a eliminação da transição corrente causada pela execução da função adaptativa .ElTran, causando a seguinte configuração quando a leitura estiver iniciando a transição $q_3 \rightarrow q_2$:

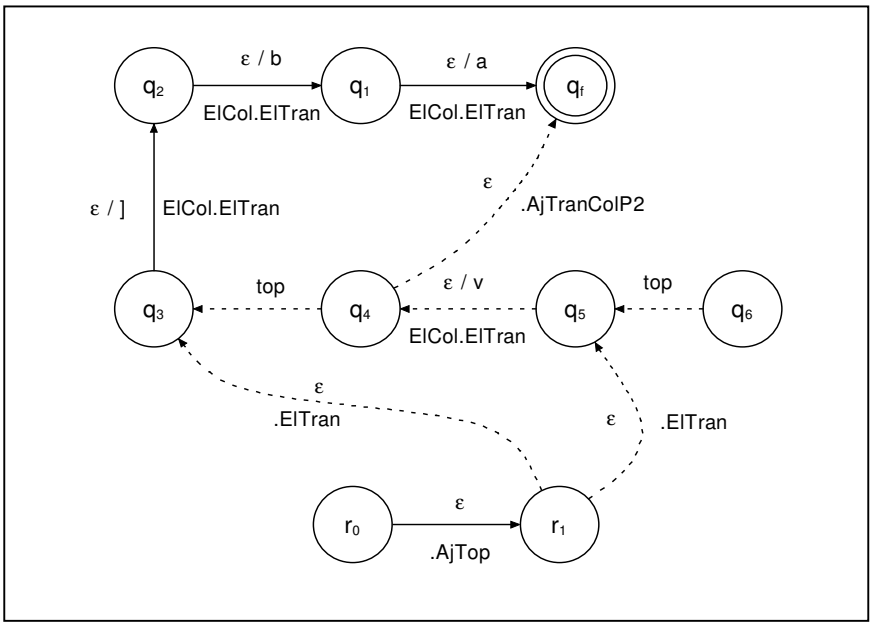


Fig. 17 - Pilha de transdução após a execução da transição $r_1 \rightarrow q_3$

Ao iniciar a transição $q_3 \rightarrow q_2$, há a chamada da função adaptativa ElCol, que irá:

- procurar uma transição em vazio a partir do estado corrente, em que haja a emissão de um separador];
- ao encontrá-la, eliminar esta transição;
- incluir uma transição em vazio do estado corrente para o estado q_f (final da pilha), ativando a função adaptativa posterior AjTranColP2 (realizada após o consumo da transição);
- Incluir uma transição do estado corrente para o estado anteriormente excluído para utilização no próximo desempilhamento, usando o token top, tendo-se como resultado a seguinte configuração:

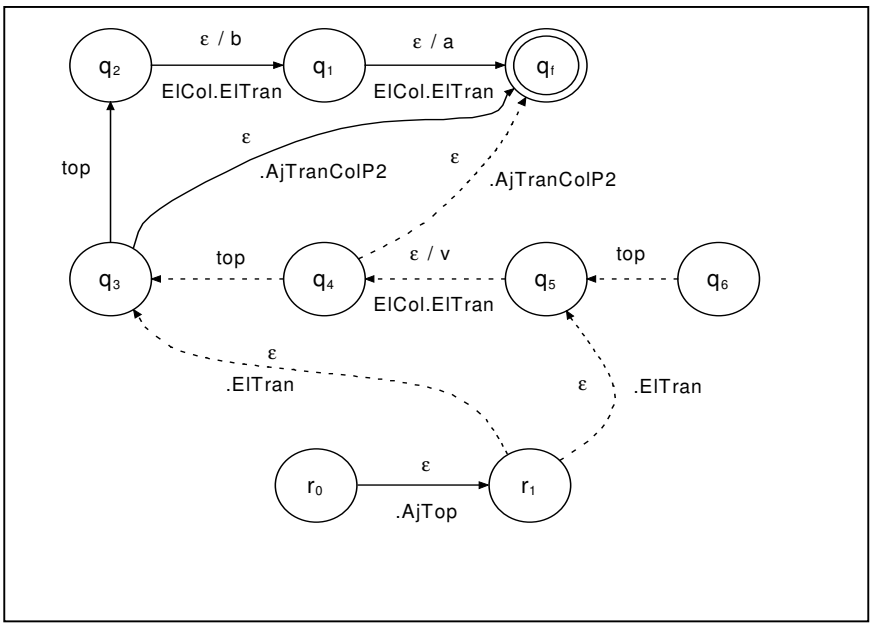


Fig. 18 - Pilha de transdução ao iniciar a execução da transição $q_3 \rightarrow q_2$

Ao processar a transição do estado q_3 ao estado q_f, executa-se a função adaptativa .AjTranColP2, que elimina esta mesma transição, finalizando o desempilhamento sem a produção de símbolos na saída, obtendo-se a seguinte configuração:

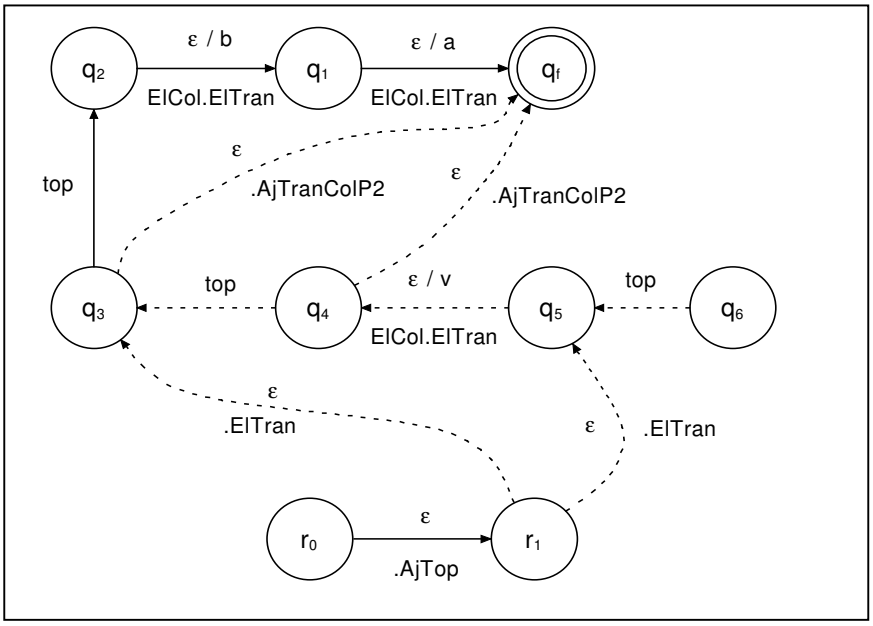


Fig. 19 - Pilha de transdução após a execução da transição $q_3 \rightarrow q_f$

Em um desempilhamento posterior, inicia-se novamente a pilha pelo estado r_0, e pela aplicação das funções adaptativas já vistas, efetuam-se os seguintes passos:

- execução de .AjTop (transição $r_0 \rightarrow r_1$):

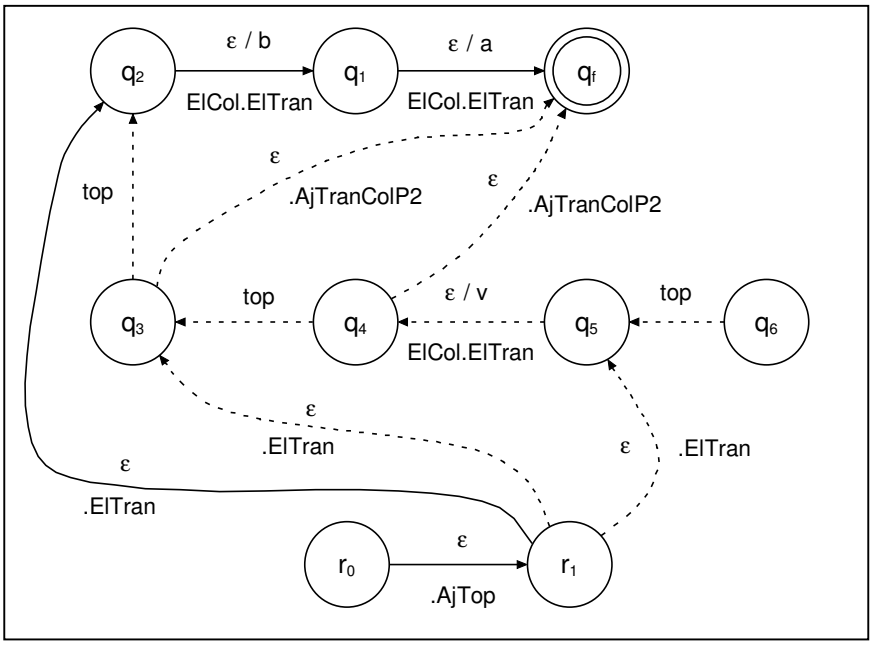


Fig. 20 - Pilha de transdução após a execução da transição $r_0 \rightarrow r_1$

- eliminação da transição inicial por .ElTran, emissão do caracter b na saída, e execução de ElCol (sem efeito) e ElTran (eliminação do estado corrente):

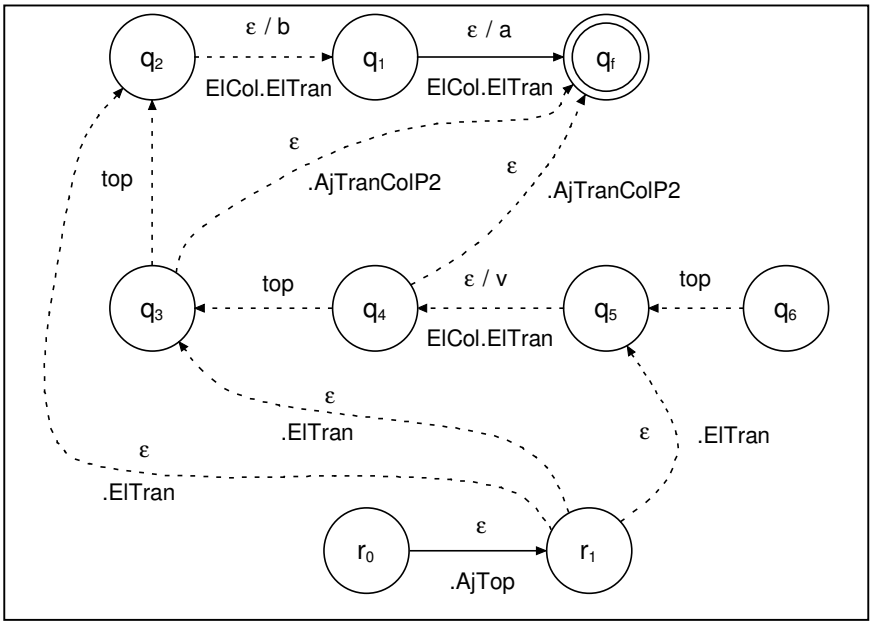


Fig. 21 - Pilha de transdução após a execução de $r_1 \rightarrow q_2$, $q_2 \rightarrow q_1$

- emissão do símbolo a na saída, e execução de ElCol e ElTran, tendo se ajustado o apontador do topo (top) da pilha:

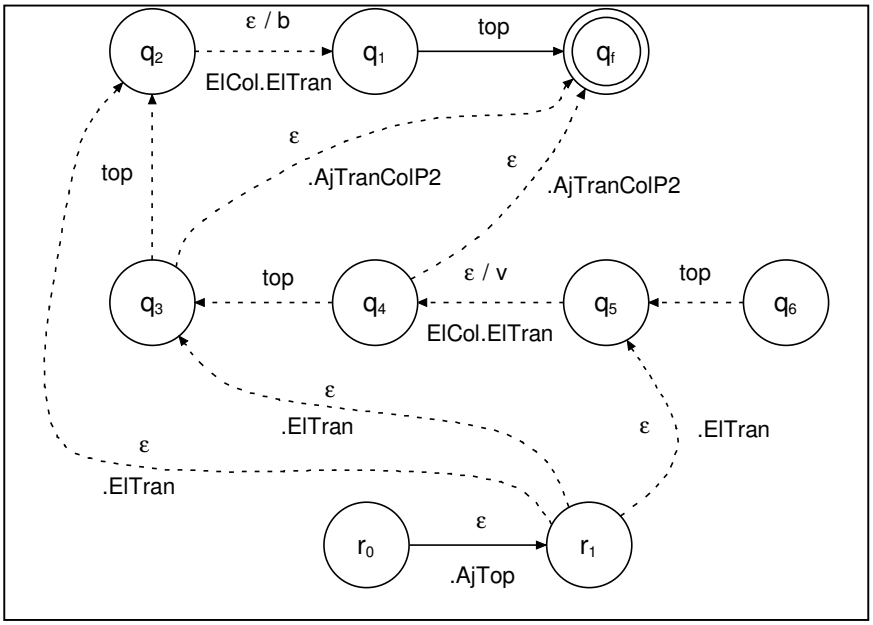


Fig. 22 - Pilha de transdução após a execução da transição $q_1 \rightarrow q_f$

Assim, o último desempilhamento é finalizado, com a emissão de b a.

A tabela abaixo representa, na notação adotada pela ferramenta *Adaptools*, a definição das funções apresentadas, e que será utilizada para a construção da pilha de transdução com as operações de empilhamento e desempilhamento, conforme discutido anteriormente. Neste caso, aos estados acima representados, foram atribuídos os seguintes números de estado para efeito de implementação:

r_0	....	90000
r_1	....	90200
q_f	....	100600
q_1	....	100500

É importante notar que os demais estados são criados durante o processo de empilhamento e desempilhamento.

A representa o autômato no qual ocorre a solicitação de empilhamento e desempilhamento, de acordo com o exemplo apresentado:

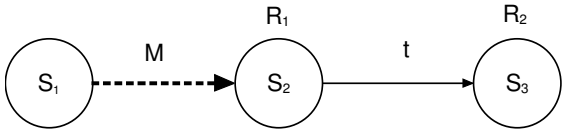
Head	Orig	Inpu	Dest	Push	Outp	Adap
A	0	eps	5	nop	emp(a),	Emp(a).
A	5	eps	6	nop	emp(b),	Emp(b).
A	6	eps	7	nop	emp(]),	Emp(]).
A	7	eps	8	nop	emp(]),	Emp(]).
A	8	eps	107	nop	emp(v),	Emp(v).
A	107	eps	90000	108	desempilhando(1)...	nop
A	108	eps	90000	109	desempilhando(2)...	nop
A	109	eps	90000	110	desempilhando(3)...	nop
A	110	eps	120	fin	--acabou!	nop
?Emp	?x	top	?y	nop	nop	nop
-Emp	?x	top	?y	nop	nop	nop
+Emp	?x	eps	?y	nop	?p1	ElCol.ElTran
+Emp	*new	top	?x	nop	nop	nop
PILHA	100500	top	100600	nop	nop	nop
PILHA	100600	eps	pop	nop	nop	nop
?AjTop	?x	top	?y	nop	nop	nop
-AjTop	?x	top	?y	nop	nop	nop
+AjTop	90200	eps	?y	nop	nop	.ElTran
?ElCol	?sta	eps	?y	nop	]	ElCol.ElTran
-ElCol	?sta	eps	?y	nop	]	ElCol.ElTran
+ElCol	?sta	eps	100600	nop		.AjTranColP2
+ElCol	?sta	top	?y	nop	nop	nop
?AjTranColP2	?x	eps	100600	nop		.AjTranColP2
-AjTranColP2	?x	eps	100600	nop		.AjTranColP2
Desempilha	90000	eps	90200	nop	- ajusta o topo...	.AjTop
?ElTran	?x	?z	?sta	nop	?k	ElCol.ElTran
-ElTran	?x	?z	?sta	nop	?k	ElCol.ElTran

Construída a pilha, poderão ser montadas as transições do autômato gerado no passo anterior, de forma que as mesmas possam produzir como saída a árvore sintática desejada. Para isso, aplica-se a *tabela de mapeamento* exposta em 5.3.2. A seguir, apresenta-se a maneira pela qual as transições do autômato gerado em 5.2 devem ser ajustadas de forma a transformá-lo no transdutor que gera a árvore sintática. Mais adiante, ilustra-se um exemplo prático das técnicas apresentadas até o momento.

5.3.4 Inclusão das transições que geram a árvore sintática

Passa-se agora à construção do transdutor responsável pela montagem da árvore sintática a partir de uma cadeia de entrada. Para isso, para cada um dos tipos de transições apresentadas, e para cada rótulo correspondente ao estado de destino, deverá ser aplicada a ação correspondente, de acordo com a tabela de mapeamento, apresentada em 5.3.2.

Dado o diagrama abaixo:



Sendo:

M	...	chamada a uma submáquina "M";
t	...	um terminal a ser consumido;
S_1, S_2, S_3	...	estados;
R_1, R_2	...	rótulos associados respectivamente aos finais de execução da submáquina M e do terminal t;

Durante o processamento de uma cadeia de entrada, haverá o empilhamento do estado de retorno da submáquina M (no caso S_2), e o consumo do terminal t. Antes de serem atingidos os estados S_2 e S_3, será realizada uma análise sobre os rótulos R_1 e R_2 correspondentes ao término de cada transição.

Para cada um dos símbolos existentes em cada um dos rótulos, será aplicada a tabela de mapeamento de 5.3.2, que indicará transições adicionais a serem incluídas antes da chegada aos estados S_2 e S_3. Estas transições serão responsáveis pela geração da cadeia de saída correspondente à árvore de derivação desejada a partir de uma dada cadeia de entrada. A representação da árvore é construída de acordo com o exposto em 5.3.1.

A seguir, ilustra-se um exemplo básico de criação do *parser*, onde são aplicados os métodos apresentados.

5.3.5 Analisador final – exemplo

Como exemplo, utilizaremos o conversor binário-decimal apresentado em 2.3.2, onde:

G = ({S, X}, {0, 1}, R, S)

R = { S → { } X{A}
X → { } 0 {B}
X → { } 1 {C}
X → X{ } 0 {D}
X → X{ } 1 {E}
}

A = {
{A} : imprimir valor
{B} : valor ← 0
{C} : valor ← 1
{D} : valor ← 2*valor
{E} : valor ← 2*valor+1
{ } : sem ação
}

Construção do *parser*

Passo 1 – regras devidamente rotuladas:

[{} {A}
S□□□]

00: S → ↑ Ψ ↑ X ↑ Ψ ↑

[{} 0 {B}
X□□□]
01: X → ↑ Ψ ↑ 0 ↑ Ψ ↑

[{} 1 {C}
X□□□]
02: X → ↑ Ψ ↑ 1 ↑ Ψ ↑

[{} 0 {D} X□□□]
03: X → ↑ X ↑ Ψ ↑ 0 ↑ Ψ ↑

[{} 1 {E}
X□□□]
04: X → ↑ X ↑ Ψ ↑ 1 ↑ Ψ ↑

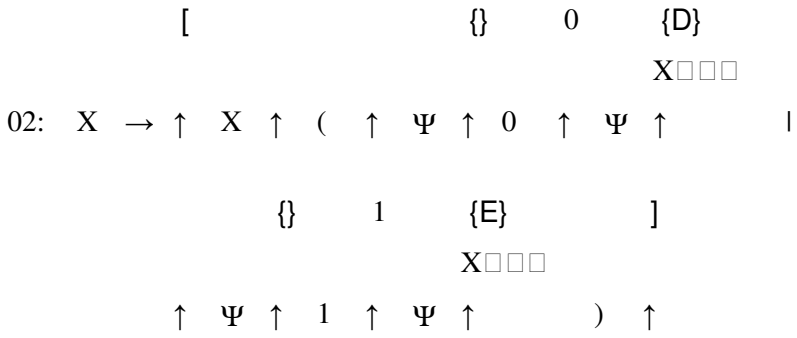
Passo 2 – Agrupamento por tipo de recursão

Como temos os tipos de recursão à esquerda e geral para o mesmo não-terminal X, o seguinte agrupamento é realizado, criando-se as regras 01 e 02 a partir das regras obtidas acima, e mantendo-se a regra 00, equivalente ao início da gramática. Notar que as regras anteriores 01 e 02 (recursão geral), e 03 e 04 (recursão à esquerda) foram respectivamente agrupadas e representadas pelas regras 01 e 02:

[{} {A}]
S□□□
00: S → ↑ (↑ Ψ ↑ X ↑ Ψ ↑) ↑

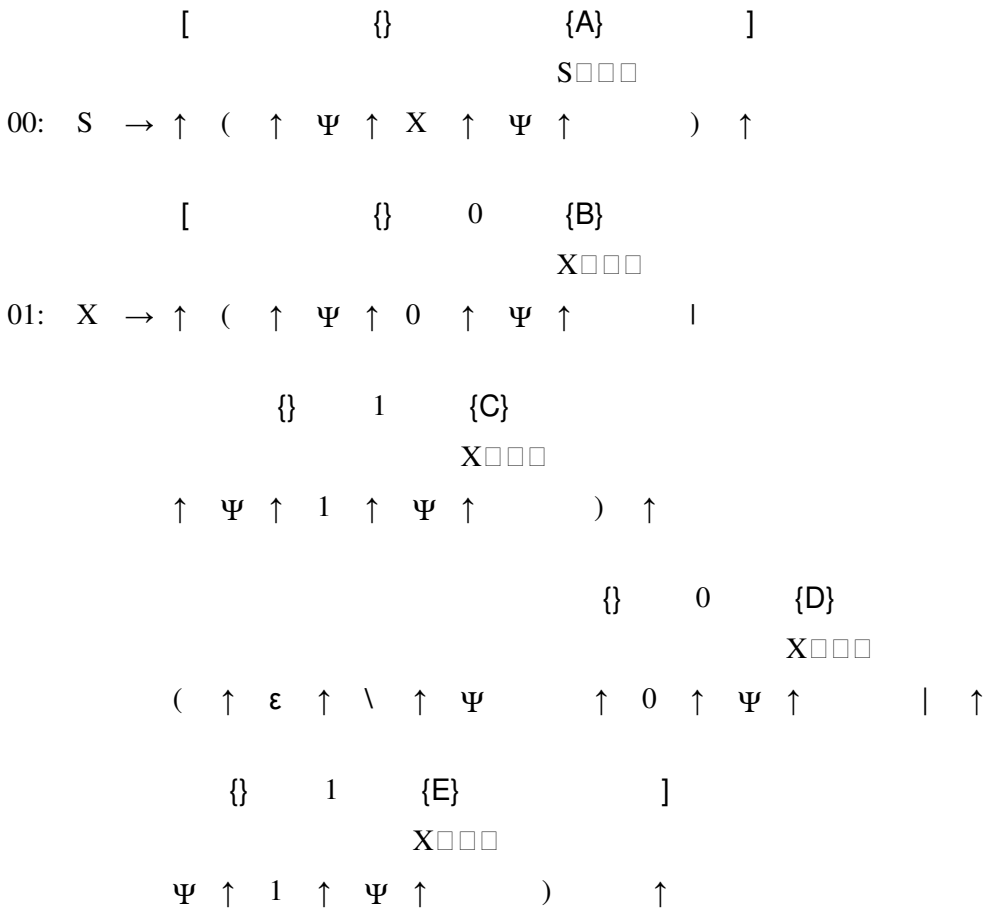
[{} 0 {B}
X□□□
01: X → ↑ (↑ Ψ ↑ 0 ↑ Ψ ↑ |

{} 1 {C}]
X□□□
↑ Ψ ↑ 1 ↑ Ψ ↑) ↑



Passo 3: agrupamento por nome de não-terminal

Neste passo, não-terminais de mesmo nome são agrupados, e portanto obtém-se uma única regra referente ao terminal X (regras 01 e 02), agora representado unicamente pela regra 01. Desta forma, obtém-se as seguintes regras:



Passo 4: ajuste das regras

Neste passo, os termos comuns são colocados em evidência, resultando na gramática descrita abaixo. Notar a colocação em evidência da ação semântica em um dos casos.

[{} {A}]

S□□□

00: S → ↑ (↑ Ψ ↑ X ↑ Ψ ↑) ↑

[{} 0 {B}

X□□□

01: X → ↑ (↑ Ψ ↑ (↑ 0 ↑ Ψ ↑ |

1 {C}

X□□□

↑ 1 ↑ Ψ ↑) ↑) ↑

{} 0 {D}

X□□□

(↑ ε ↑ \ ↑ Ψ ↑ 0 ↑ Ψ ↑ | ↑

{} 1 {E}]

X□□□

Ψ ↑ 1 ↑ Ψ ↑) ↑

Pode-se então substituir o não-terminal X em S, obtendo-se:

[[{} 0 {B}

X□□□

S → ↑ (↑ (↑ (↑ Ψ ↑ (↑ 0 ↑ Ψ ↑

1 {C} {}

X□□□

| ↑ 1 ↑ Ψ ↑) ↑) ↑ (↑ ε ↑ \ ↑ Ψ ↑

0 {D} {} 1 {E}]

X□□□ X□□□

0 ↑ Ψ ↑ | ↑ Ψ ↑ 1 ↑ Ψ ↑) ↑

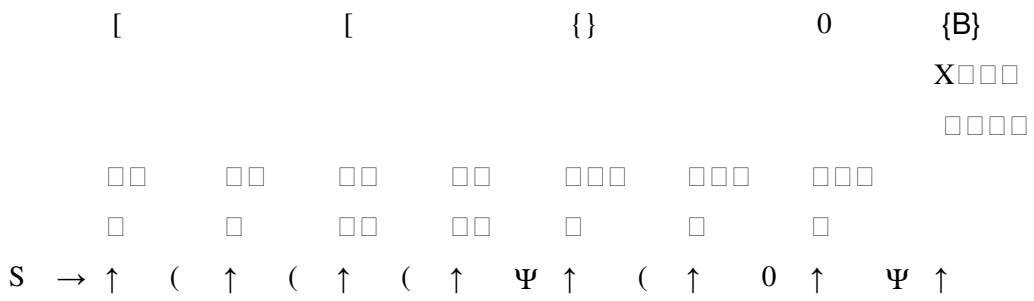
{A}]

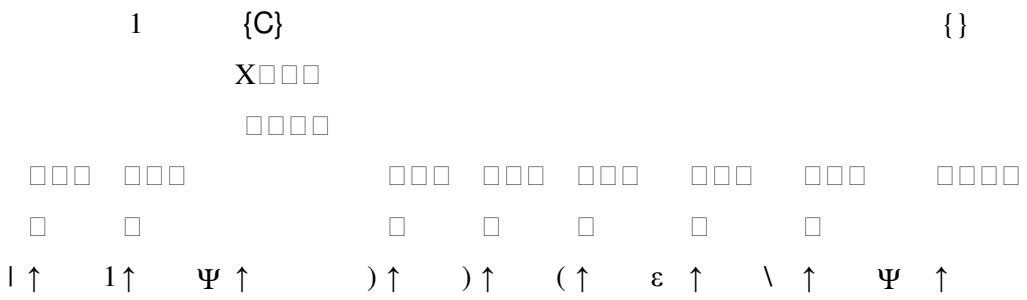
S□□□

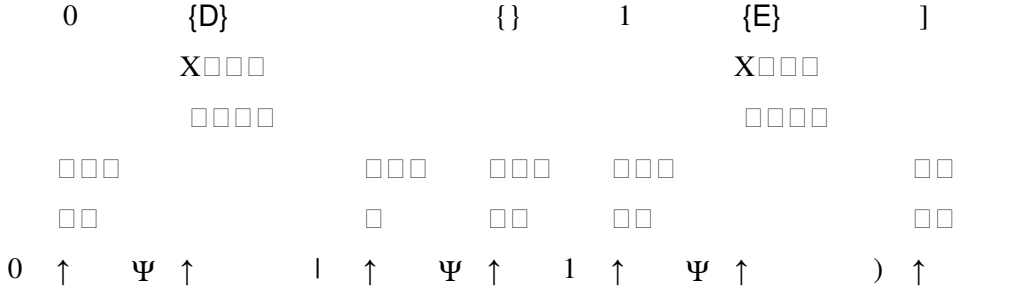
) ↑ Ψ ↑) ↑

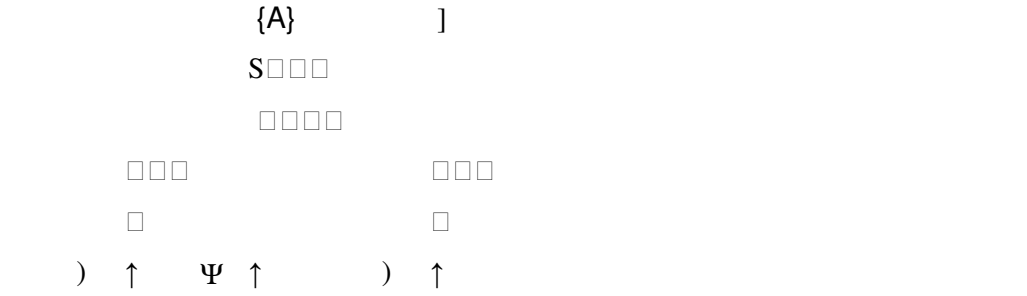
Passo 5: numeração de estados

Neste passo são atribuídos números de estado à gramática, diretamente anexados aos rótulos criados:









Passo 6: Montagem do autômato que reconheça a linguagem proposta

A partir da aplicação dos rótulos e números de estado conforme explicado anteriormente, obtém-se o autômato representado pela figura abaixo:

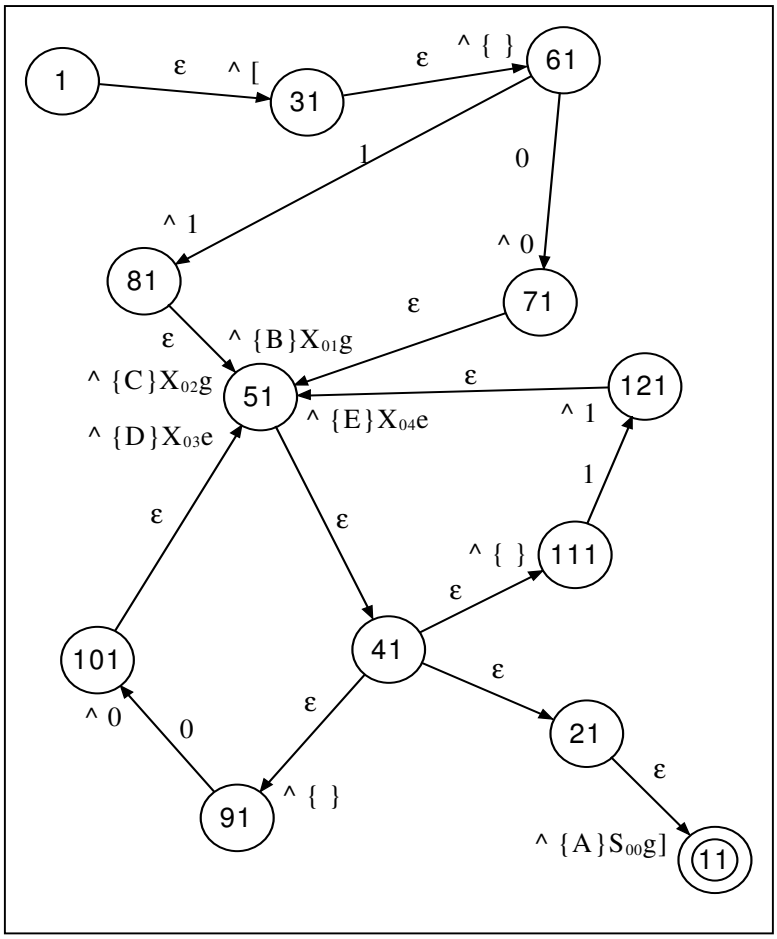


Fig. 23 – Autômato / Transdutor gerado para a gramática proposta

Todos os rótulos estão ligados ao destino de suas correspondentes transições. Para uma melhor visualização, o símbolo ^ foi colocado antes de cada rótulo.

Para possibilitar a construção do *parser*, transições adicionais são anexadas às transições do autômato indicado na figura acima, para que o mesmo atue conforme indicado na tabela II (item 5.3.2), realizando os devidos empilhamentos e desempilhamentos na pilha de transdução, com a correspondente produção da árvore sintática na saída.

Após a inclusão das transições adicionais, o transdutor fica então com o aspecto indicado na figura abaixo:

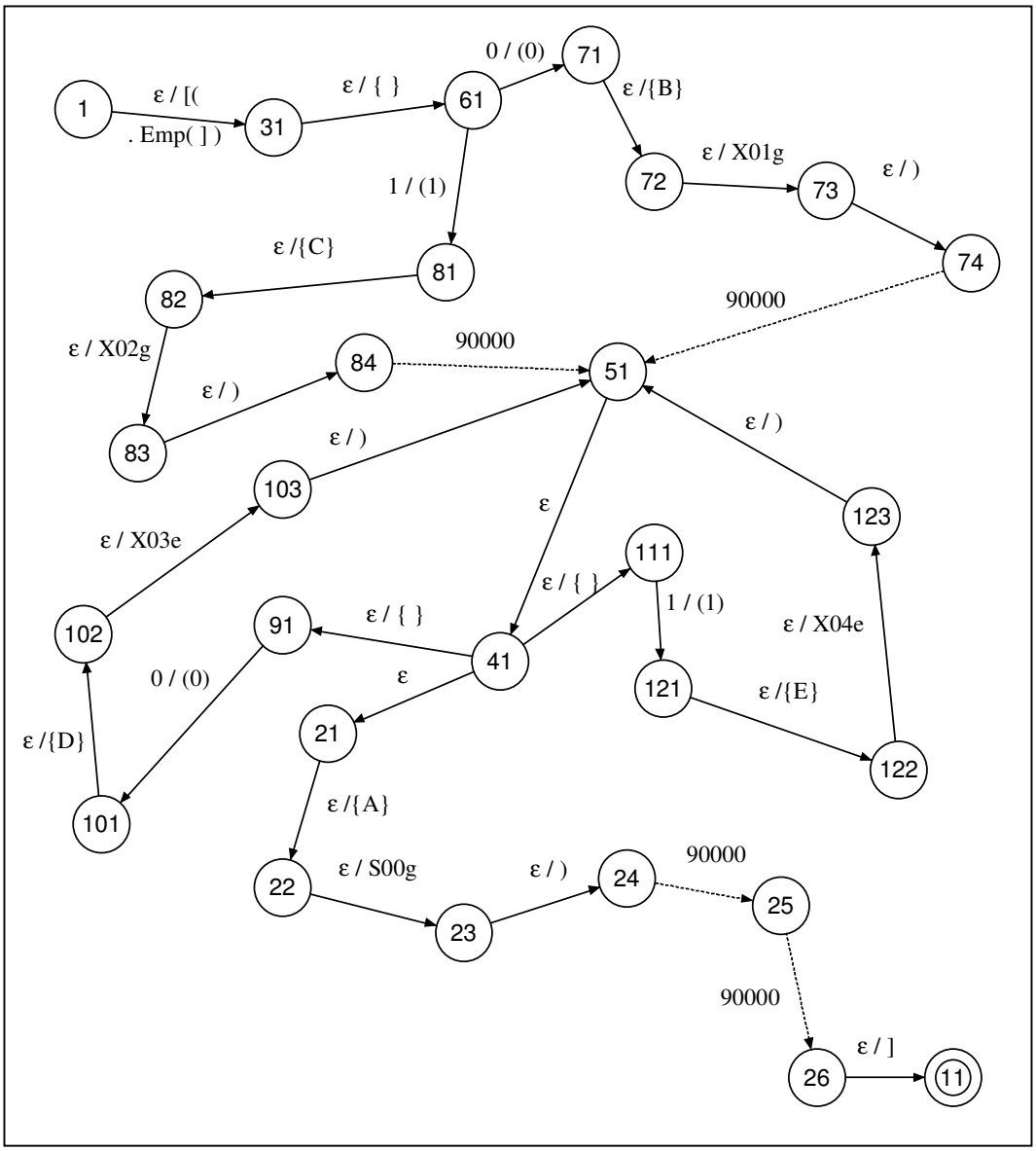


Fig. 24 – Autômato / Transdutor após a inclusão das transições adicionais

Na figura acima, ações semânticas estão delimitadas por { e }, sendo que { } representa nenhuma ação a ser aplicada. No caso deste exemplo, as ações indicadas por { } são as ações a serem aplicadas antes de cada regra da gramática original.

As transições pontilhadas representam chamadas à submáquinas. A submáquina 90000 representa a chamada ao desempilhamento de símbolos da pilha de transdução, cuja configuração original está indicada na figura abaixo.

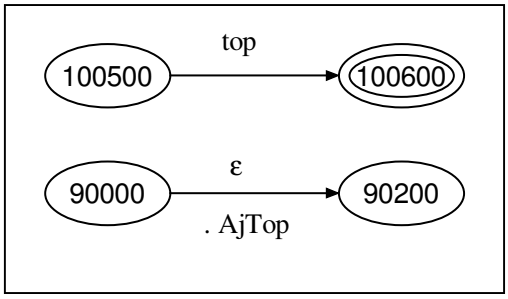


Fig. 25 – Configuração inicial da pilha de transdução

A seguir, ilustra-se o processamento do transdutor devidamente transformado.

Processando uma cadeia de entrada

A tabela abaixo mostra o reconhecimento e *parsing* da cadeia 101 aplicada ao autômato indicado na figura 24.

Símbolo	**Transição**	**Símbolo**	**Saída**	**Pilha**	**Ação Semântica**
[1] 0 1	1 → 31		[(	Emp(])	
	31 → 61		[(		{ }
	61 → 81	1	[((1)		
	81 → 82		[((1)		{C} (valor = 1)
	82 → 83		[((1) X□□□		
	83 → 84		[((1) X□□□)		
	84 → 51	submáquina	[((1) X□□□)	Desempilha	
	51 → 41		[((1) X□□□)		
	41 → 91		[((1) X□□□)		{ }
1 [0] 1	91 → 101	0	[((1) X□□□) (0)		
	101 → 102		[((1) X□□□) (0)		{D} (valor = 2)
	102 → 103		[((1) X□□□) (0) X□□□		
	103 → 51		[((1) X□□□) (0) X□□□)		
	51 → 41		[((1) X□□□) (0) X□□□)		
	41 → 111		[((1) X□□□) (0) X□□□)		{ }
1 0 [1]	111 → 121	1	[((1) X□□□) (0) X□□□) (1)		
	121 → 122		[((1) X□□□) (0) X□□□)		{E}

			(1)		(valor = 5)
	122 → 123		[((1) X□□□) (0) X□□□) (1) X□□□		
	123 → 51		[((1) X□□□) (0) X□□□) (1) X□□□)		
	51 → 41		[((1) X□□□) (0) X□□□) (1) X□□□)		
	41 → 21		[((1) X□□□) (0) X□□□) (1) X□□□)		
	21 → 22		[((1) X□□□) (0) X□□□) (1) X□□□)		{A} (imprimir 5)
	22 → 23		[((1) X□□□) (0) X□□□) (1) X□□□) S□□□		
	23 → 24		[((1) X□□□) (0) X□□□) (1) X□□□) S□□□)		
	24 → 25	submáquina	[((1) X□□□) (0) X□□□) (1) X□□□) S□□□)	Desempilha	
	25 → 26	submáquina	[((1) X□□□) (0) X□□□) (1) X□□□) S□□□)	Desempilha	
	26 → 11		[((1) X□□□) (0) X□□□) (1) X□□□) S□□□)]		

Tabela IV – *Parsing* de 101, incluindo-se a saída gerada e ações semânticas realizadas

O resultado impresso é 5, correspondendo ao número binário 101, fornecido como cadeia de entrada.

O *parsing* da cadeia 101 gera a saída [((1) X□□□) (0) X□□□) (1) X□□□) S□□□)]. Neste caso, S□□□ corresponde à raiz da árvore. O número 0 corresponde ao número da regra original da gramática (primeira regra, iniciando-se em zero), o símbolo □ mostra que a regra se refere a um não-terminal que se enquadra no caso geral. O nó S□□□ possui uma sub-árvore definida pelo nó X□□□. O nó X□□□ possui 2 sub-árvores: uma indicada por ((1) X□□□) (0) X□□□) e outra, indicada por (1). X□□□ é um nó correspondente à quarta regra, sendo recursivo à esquerda. X□□□ é um nó que representa a terceira regra, é recursivo à esquerda e tem como descendentes a sub-árvore ((1) X□□□), e o terminal (0). X□□□ é um nó que representa a segunda regra, e corresponde a uma regra geral (não-recursiva) , possuindo o terminal (1) como descendente. A árvore pode então ser construída, resultando na figura 8.

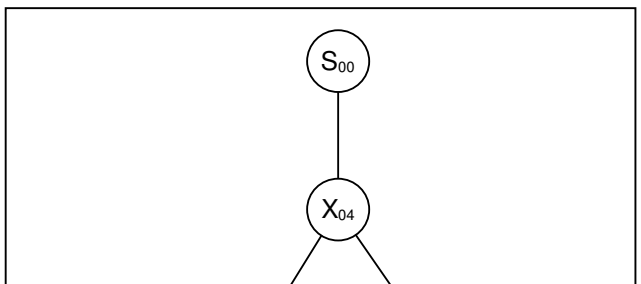


Fig. 26 – Árvore gerada para a cadeia 101

É importante salientar que, neste processo, não somente foi gerada a árvore de derivação, mas foi também realizado o reconhecimento da cadeia 101. Foi possível então, através da aplicação das técnicas apresentadas, a simplificação (quando possível) da gramática de entrada, e a construção do parser correspondente, que tem a função de reconhecer uma cadeia de entrada e também de produzir, na saída, uma árvore de derivação correspondente ao reconhecimento desta cadeia.

6 - ASPECTOS DE IMPLEMENTAÇÃO

6.1 Descrição

A ferramenta proposta para a implementação dos referidos algoritmos utiliza como entrada uma gramática a ser tratada, e produz como saída um autômato que, quando processado pela ferramenta *Adaptools* para uma dada cadeia de entrada, produz a árvore sintática correspondente a esta cadeia. A estrutura funcional geral do software está exposta na figura abaixo:

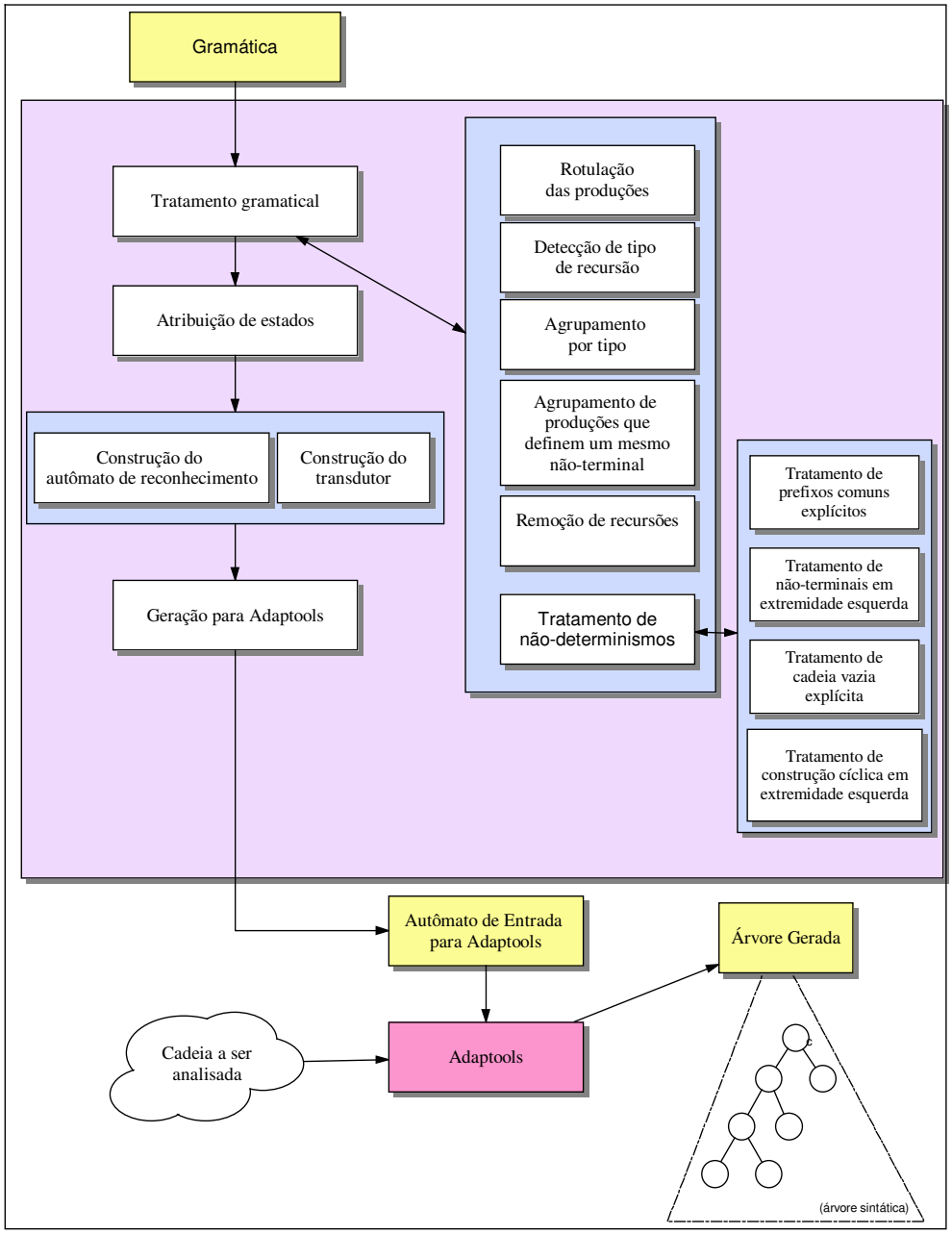


Fig. 27 - Estrutura funcional geral do software

A apresentação do software desenvolvido está ilustrada na figura abaixo:

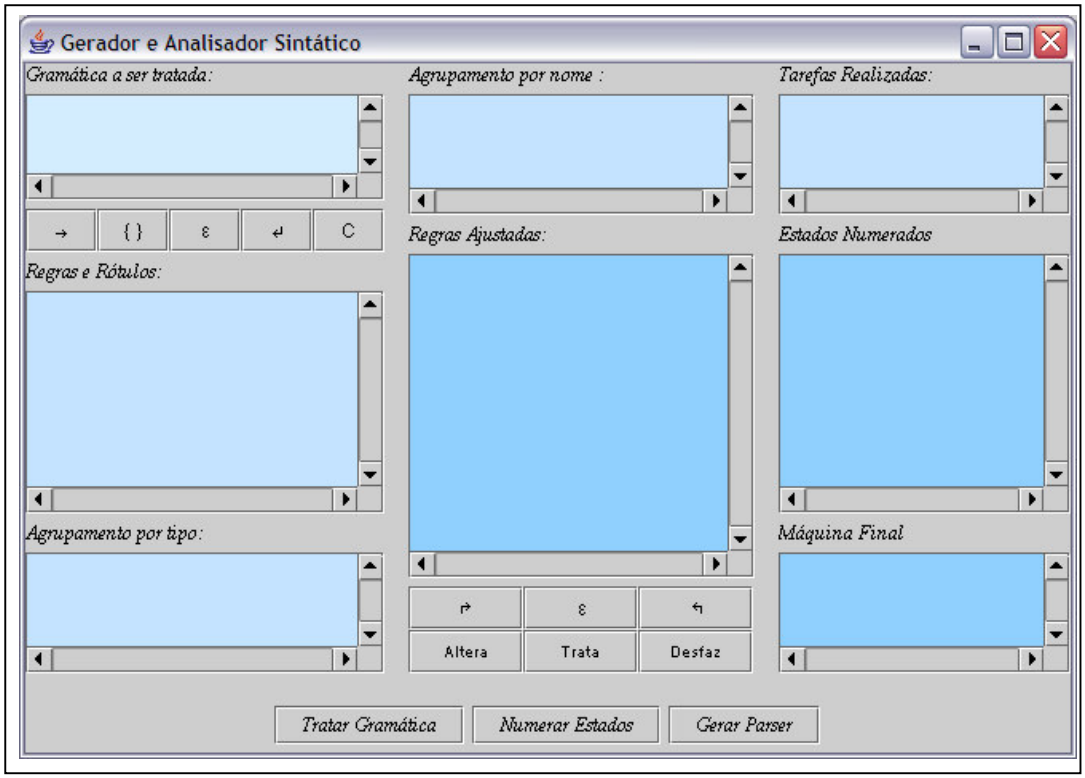


Fig. 28 – Apresentação do software

O software é responsável por:

- realizar a análise e transformação da gramática de entrada, intitulada na figura como *Tratamento Gramatical*. Esta transformação envolve a aplicação dos métodos estudados: rotulação das produções (4.2.1, 4.2.2), detecção de tipo de recursão (4.2.2), agrupamento por tipo (4.2.3), agrupamento de produções que definem um mesmo não-terminal (4.2.3), remoção de recursões (4.2.4) e tratamento de não-determinismos (4.2.5). A aplicação do tratamento de não-determinismos envolve 4 tarefas básicas: tratamento de prefixos comuns explícitos (4.2.5.a), tratamento de não-terminais em extremidade esquerda (4.2.5.b), tratamento de cadeia vazia explícita (4.2.5.c), tratamento de construção cíclica em extremidade esquerda (4.2.5.d);
- atribuir estados à gramática transformada (5.1);
- construir o autômato/transdutor correspondente (5.2 e 5.3);

- gerar código equivalente para o *Adaptools* utilizando-se a forma de representação exposta em (3.2) e incluindo-se a construção da pilha de transdução definida em (5.3.3).

6.2 Formato utilizado para a apresentação dos algoritmos

Optou-se pelo detalhamento algorítmico dos passos mais relevantes, priorizando-se o seu entendimento pelo leitor. Desta forma, alguns passos pertinentes unicamente à implementação de código estão omitidos.

A estrutura utilizada para a exposição dos passos a serem implementados está ilustrada na figura abaixo. Para estruturas que envolvam dependência optou-se pela representação da dependência através da posição de sua descrição e por um número que representa o nível de dependência entre parênteses. Assim, o trecho abaixo:

- [5]<passo 1>
  - [6]<passo 2>
- [5]<passo 3>

Indica que <passo 2> depende de <passo 1>, já o <passo 3> está ao mesmo nível do <passo 1> (nível 5), ocorrendo após a sua execução. Como exemplo, pode-se citar uma estrutura de controle de loop para o <passo 1> (while, do...while, for, etc), ou mesmo uma condição (if).

Algoritmo: **<nome do algoritmo em negrito>**

Função: <descrição de sua função>

Entradas:

- <descrição das entradas>

Saídas:

- <descrição das saídas>

Descrição:

<descrição básica do algoritmo, através de passos em que a posição e o número entre colchetes indica a sua dependência hierárquica>

Fig. 29 – Estrutura utilizada para detalhamento algorítmico

6.3 Estrutura básica de formação de regras

A modificação e transformação das regras iniciais e o processo de criação e migração dos rótulos correspondentes aos elementos de cada regra expostos nos capítulos 4 e 5 geram cadeias de rótulos e de elementos de comprimento variável, razão pela qual o formato escolhido para o trabalho com cada uma das regras de produção é uma estrutura dinâmica baseada em cadeias (Strings) dispostas em seqüência e implementada através da linguagem Java.

Esta seqüência de cadeias está montada estruturalmente conforme o ilustrado na figura abaixo:

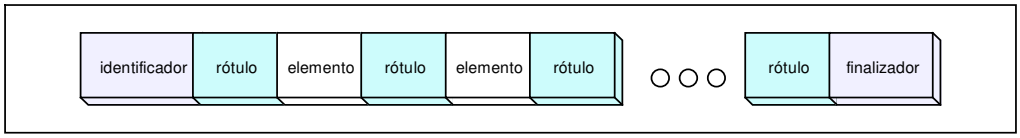


Fig. 30 – Estrutura das regras de produção

Onde:

- Bloco "identificador" : refere-se ao não-terminal que define a regra;
- Bloco "rótulo": corresponde a um dado rótulo a ser inserido/tratado nesta posição (conforme exposto no capítulo 4)
- Bloco "elemento": corresponde ao terminal / não-terminal presente nesta posição. Eventualmente este bloco poderá conter uma abertura ou um fechamento de parênteses, resultante da associação de uma ou mais regras;
- Bloco "finalizador": utilizado como referência de final da regra.

A referência de final consta também ao término de cadeia referente a cada bloco. Assim será possível manter o sistema dinâmico o suficiente para a variação de comprimento de cadeias referentes aos blocos de rótulo.

<u>Exemplo:</u>

A regra abaixo, apresentada em 4.2.2.a :

		[				{Y}		β_1'		β_2'			β_n'		{Z}P_i□]
P	→	↑	P	↑	Ψ	↑	β_1	↑	β_2	↑	...	β_n	↑	Ψ	↑

Será armazenada da seguinte forma:

- identificador: P
- rótulo: [
- elemento: P
- rótulo: <nulo>
- elemento: Ψ
- rótulo: {Y}
- elemento: β_1
- rótulo: β_1'
- elemento: β_2
- rótulo: β_2'

....

- elemento: β_n
- rótulo: β_n'
- elemento: Ψ
- rótulo: {Z}P_i□]
- elemento: <finalizador>

Notar que cada seqüência de cadeias representa uma única regra de produção. A gramática completa é definida, portanto, através de uma matriz de cadeias de comprimento variável, onde cada linha da matriz representa uma regra, e cada coluna da matriz será um identificador, elemento ou rótulo. Nos algoritmos apresentados a seguir, esta matriz está apresentada como <u>*matriz de regras*</u>.

6.4 Rotulação das regras de produção

Para a rotulação das regras de produção, aplica-se o algoritmo construir descrito abaixo:

Algoritmo: **construir**
Função: Transcrição das regras no formato de entrada para o formato de trabalho

Entradas:

- vetor contendo as regras de produção originais

Saídas:

- matriz de regras novas carregada com os elementos correspondentes no formato padrão
- vetor de tipos carregado

Descrição:

- [0]para cada regra no formato original, referenciada a partir de um indexador n :
  - [1]procurar o símbolo → e separar os lados esquerdo e direito da regra
  - [1]atribuir o lado esquerdo obtido ao campo *identificador* da regra nova;
  - [1]detectar o tipo de regra de produção à partir do tipo de recursão *esquerda*, *direita* ou *outros*
  - [1]montar o primeiro rótulo da regra nova com o símbolo [
  - [1]para todos os elementos do lado direito da regra :
    - [2]copiar o elemento para a regra nova
    - [2]se elemento for um terminal
      então
      - [3]anexar à regra nova o próprio elemento como rótulo

      senao
      - [3]não anexar rótulo à regra nova
  - [1]anexar como rótulo final da regra os seguintes elementos unidos num único rótulo:

    - [2]não-terminal que a define
    - [2]indexador da regra original n
    - [2]um caracter denotando o tipo de recursão, detalhado abaixo:
        - □ → para recursão à esquerda
        - □ → para recursão à direita
        - □ → para o caso geral (outros)
    - [2]símbolo]
- [1]finalizar a regra nova

Neste algoritmo, a detecção de tipo de recursão segue o exposto em 4.2.2, assumindo-se que:

- Será considerada recursão à esquerda quando o primeiro elemento do lado direito da produção for idêntico ao elemento não-terminal que define a regra:
  $P \rightarrow P\{Y\}\ \beta_1 \beta_2 \ldots \beta_n \{Z\}$
- Será considerada recursão à direita quando o último elemento do lado direito da produção for idêntico ao elemento não-terminal que define a regra:

$P \rightarrow \{Y\}\ \beta_1 \beta_{2\ldots} \beta_n\ \{Z\}P$

- Casos que não se enquadrem no exposto acima serão considerados como *caso geral* pelo algoritmo.

6.5 Descritivo dos algoritmos

Alguns dos algoritmos desenvolvidos a partir dos procedimentos propostos nos capítulos 4 e 5 têm maior relevância e estão descritos a seguir.

6.5.1 Manipulação da gramática

Para algoritmos de tratamento gramatical, estão sendo detalhados os algoritmos correspondentes aos procedimentos "4.2.3 – Agrupamento de Produções que Definem um Mesmo Não-Terminal", "4.2.4 – Eliminação das Auto-Recursões à Direita e à Esquerda" e "4.2.5 – Tratamento de Não-Determinismos". Os algoritmos "4.2.1 - Detecção de Tipo de Recursão" e "4.2.2 - Rotulação das Produções" estão representados pelo algoritmo construir (item 6.4 acima).

6.5.1.1 Agrupamento de produções

Este algoritmo é baseado em 4.2.3, nomeia-se agruparTipo, e é responsável pelo agrupamento de regras de mesmo tipo de recursão e de mesmo não-terminal. O processo é realizado a partir de um laço que deverá tratar cada uma das regras, analisá-la, e partir em busca de outras regras definidas pelo mesmo não-terminal. Ao final, o conjunto de regras fornecido como entrada é atualizado com as novas regras criadas.

Algoritmo: **agruparTipo**

Função: Agrupar regras de produção definidas pelo mesmo não-terminal e que tenham o mesmo tipo de recursão

Entradas:

- matriz sRegras de regras de produção previamente ajustadas
- vetor cTipo de tipos de recursão associados a cada uma das linhas de sRegra

Saídas:

- matriz sRegras ajustada com produções agrupadas por tipo, para cada não-terminal
- vetor cTipo[ ] atualizado de acordo com a transformação

Descrição:

- [0]seja uma matriz sTemp temporária, de auxílio à criação das novas regras
- [0]para cada uma das regras em sRegras
  - [0]se esta regra ainda não foi verificada

  então
    - [1]marcar esta regra como já verificada
    - [1]montar a primeira regra nova em sTemp com os primeiros campos (identificador e rótulo)
    - [1]se cTipo referencia esta regra como ‘E’ (recursão esquerda)

      então
        - [2]anexar o nome do não-terminal como elemento em sTemp

- [1]anexar um parênteses de abertura ‘(‘ como elemento em sTemp
- [1]anexar um rótulo em branco como rótulo em sTemp
- [1]para cada elemento da regra original analisada
  - [2]anexar o elemento correspondente em sTemp;
  - [2]anexar o rótulo correspondente em sTemp.
- [1]eliminar o indicador final ‘]’ do rótulo de sTemp
- [1]para cada uma das regras restantes em sRegras, a partir da regra corrente
  - [2]se esta regra ainda não foi verificada
    então
    - [3]se esta regra for uma regra de mesmo tipo e referenciada pelo mesmo não-terminal de sTemp
      então
      - [4]marcar esta regra como já verificada
      - [4]anexar o elemento ‘|’ (divisor) em sTemp
      - [4]para cada coluna desta regra, respeitando o posicionamento de cada tipo
        - [5]anexar o elemento correspondente em sTemp;
        - [5]anexar o rótulo correspondente em sTemp.
      - [4]eliminar o indicador final ‘]’ do rótulo de sTemp
- [1]anexar um parênteses de fechamento ‘)’ como elemento em sTemp
- [1]se cTipo referencia esta regra como ‘D’ (recursão à direita)
  então
  - [2]anexar elemento que define a regra corrente em sTemp
- [1]atualizar o rótulo final de sTemp, acrescentando o símbolo ‘]’

- [0]copiar todos os elementos da matriz sTemp para a matriz sRegras, atualizando-a.

6.5.1.2 Remoção das auto-recursões eliminávéis

Este algoritmo é baseado em 4.2.3, nomeia-se agruparRegras, e é responsável pelo agrupamento de várias regras de mesmo não-terminal e diferentes tipos de recursão em uma única expressão.

Algoritmo: **agruparRegras**
Função: Agrupar regras de produção de mesmo não-terminal
Entradas:

- matriz sRegras
- vetor cTipo (tipos de recursão para cada regra)

Saídas:

- matriz sRegras devidamente agrupada

Descrição:

- [0]seja uma matriz sTemp[ ] [ ] temporária, de auxílio à criação das novas regras
- [0]seja um vetor sParteE, correspondente à montagem temporária de parte da regra final que tenha recursão à esquerda
- [0]seja um vetor sParteD, correspondente à montagem temporária de parte da regra final que tenha recursão à direita
- [0]seja um vetor sParteO, correspondente à montagem temporária de parte da regra final que corresponda a outros casos (geral)
- [0]para cada uma das regras em sRegras
    - [1]se esta regra ainda não foi verificada

    então
        - [2]marcar esta regra como já verificada
        - [2]montar a primeira regra nova em sTemp com os primeiros campos (identificador e rótulo)
        - [2]limpar índices e conteúdos de sParteE, sParteD e sParteO
        - [2]se cTipo referencia esta regra como recursão à direita (‘D’)
        entao

- [3]anexar o parênteses de abertura ‘(‘ como elemento em sParteD;
- [3]anexar o símbolo ‘ε’ como elemento em sParteD
- [3]anexar o símbolo ‘\’ como elemento em sParteD;
- [3]para cada um dos elementos da regra corrente em sRegras
  - [4]anexar elementos e rótulos em sParteD
- [3]anexar o parênteses de fechamento ‘)’ como elemento em sParteD

- [2]se cTipo referencia esta regra como recursão à esquerda (‘E’)
  então
  - [3]anexar o parênteses de abertura ‘(‘ como elemento em sParteE;
  - [3]anexar o símbolo ‘ε’ como elemento em sParteE
  - [3]anexar o símbolo ‘\’ como elemento em sParteE;
  - [3]para cada um dos elementos da regra corrente em sRegras
    - [4]anexar elementos e rótulos em sParteE
  - [3]anexar o parênteses de fechamento ‘)’ como elemento em sParteE
- [2]se cTipo referencia esta regra como (‘O’)
  então
  - [3]para cada um dos elementos da regra corrente em sRegras
    - [4]anexar elementos e rótulos em sParteO
- [2]para todas as regras restantes em sRegras a partir da regra corrente
  - [3]se esta regra ainda não foi verificada
    então
    - [4]se esta regra trata o mesmo não-terminal da regra tratada pela regra corrente
      então
      - [5]marcar esta regra como já verificada

- [5]se cTipo referencia esta regra como recursão à direita (‘D’)

  entao

  - [6]anexar o parênteses de abertura ‘(‘ como elemento em sParteD;
  - [6]anexar o símbolo ‘ε’ como elemento em sParteD
  - [6]anexar o símbolo ‘\’ como elemento em sParteD;
  - [6]para cada um dos elementos da regra corrente em sRegras
    - [7]anexar elementos e rótulos em sParteD
  - [6]anexar o parênteses de fechamento ‘)’ como elemento em sParteD

- [5]se cTipo referencia esta regra como recursão à esquerda (‘E’)

  então

  - [6]anexar o parênteses de abertura ‘(‘ como elemento em sParteE;
  - [6]anexar o símbolo ‘ε’ como elemento em sParteE
  - [6]anexar o símbolo ‘\’ como elemento em sParteE;
  - [6]para cada um dos elementos da regra corrente em sRegras
    - [7]anexar elementos e rótulos em sParteE
  - [6]anexar o parênteses de fechamento ‘)’ como elemento em sParteE

- [5]se cTipo referencia esta regra como (‘O’)

  - [6]para cada um dos elementos da regra corrente em sRegras
    - [7]anexar elementos e rótulos em sParteO

- [2]anexar ‘[‘ como rótulo em sTemp
- [2]para cada elemento de sParteD
  - [3]anexar elemento e rótulo à sTemp
- [2]para cada elemento de sParteO
  - [3]anexar elemento e rótulo à sTemp
- [2]para cada elemento de sParteE
  - [3]anexar elemento e rótulo à sTemp
- [2]ajustar o rótulo final incluindo um ‘]’ como rótulo em sTemp
- [2]incrementar o contador de regras em sTemp para trabalhar a próxima regra

- [0]copiar sTemp para sRegras

6.5.1.3 Tratamento de não-determinismos

Este algoritmo é baseado no exposto em 4.2.5, e é construído a partir das regras geradas no algoritmo anterior. A simplificação das regras, a partir da detecção dos conjuntos delimitadores dos parênteses mais internos destas regras é definida pelo algoritmo simplificarRegras. A partir desta simplificação, devem ser executados 4 outros algoritmos que realizam outras tarefas de simplificação, a saber:

- algoritmo simplificarRegrasPrefixosComuns
- algoritmo simplificarRegrasSubstituirNaoTerminais
- algoritmo simplificarRegrasSubstituirEpsilon
- algoritmo simplificarRegrasEliminarConstrucaoCiclica

6.5.1.3.1 Simplificação das regras

Algoritmo: simplificarRegras

Função: Simplificar as Regras, colocando termos comuns em evidência, substituindo não-terminais, substituindo construções com épsilon e eliminando construções cíclicas no lado esquerdo de opções.

Entradas:

- matriz sRegras de regras de produção

Saídas:

- matriz sRegras ajustada com produções simplificadas

Descrição:

- [0]seja uma matriz sTemp temporária, de auxílio à criação das novas regras;
- [0]seja um vetor inteiro iParAbInterno, que conterá os posicionamentos de abertura de parênteses mais internos para cada regra
- [0]seja um vetor inteiro iParFeInterno, que conterá os posicionamentos de fechamento de parênteses mais internos para cada regra

(o objetivo das variáveis iParAbInterno e iParFeInterno é marcar os conjuntos mais internos entre parênteses em cada produção)

- [0]seja um flag bPrimeiroFechamento, que deverá indicar quando o fechamento de parênteses será uma primeira ocorrência após uma abertura de parênteses
- [0]para cada uma das regras em sRegras
  - [1]inicializamos os vetores iParAbInterno e iParFeInterno
  - [1]para cada um dos elementos da regra corrente
    - [2]se o elemento for um ‘(‘
      então
      - [3]se o flag bPrimeiroFechamento não estiver ativado
        então
        - [4]ativar o flag bPrimeiroFechamento
      - [3]armazenar a posição do elemento ‘(‘ em iParAbInterno

      senão
      - [3]se o elemento for um ‘)’
        então
        - [4]se o flag bPrimeiroFechamento estiver ativado

        então
        - [5]armazenar a posição do elemento ‘)’ em iParFeInterno
        - [5]incrementar indexador de iParFeInterno e iParAbInterno
        - [5]desativar o flag bPrimeiroFechamento
  - [1]finalizar as listas de parênteses iParAbInterno e iParFeInterno obtidas
  - [1]executar o algoritmo simplificarRegrasPrefixosComuns
    - entradas:
      - a regra corrente da matriz sRegras;
      - o vetor iParAbInterno, indicador dos delimitadores ‘(‘ dos conjuntos mais internos
      - o vetor iParFeInterno, indicador dos delimitadores ‘)’ dos conjuntos mais internos
    - saídas:
      - atualização da regra corrente

        - vetores iParAbInterno e iParFeInterno atualizados
- [0]executar o algoritmo simplificarRegrasSubstituirNaoTerminais
    - entradas:
        - matriz completa de sRegras
    - saídas:
        - matriz completa de sRegras com não-terminais substituídos pelas produções equivalentes
- [0]executar o algoritmo simplificarRegrasSubstituirEpsilon
    - parâmetros de entrada:
        - matriz completa de sRegras
    - saídas:
        - matriz completa de sRegras com opções ‘ε’ substituídas pelas produções equivalentes
- [0]executar o algoritmo simplificarRegrasEliminarConstrucaoCiclica
    - parâmetros de entrada:
        - matriz completa de sRegras
    - saídas:
        - matriz completa de sRegras com construções cíclicas à esquerda substituídas pelas produções equivalentes

6.5.1.3.2 Eliminação de prefixos comuns

O algoritmo a seguir realiza uma análise em uma regra fornecida como parâmetro, eliminando a ocorrência de prefixos comuns. O processo é realizado da seguinte forma:

- para cada trecho isolado contido entre parênteses mais internos
- divide-se todas as partes contidas entre divisores ‘|’
- procura-se a ocorrência de trechos comuns entre estas partes, armazenando-se em um vetor iPartesTriplas todas as combinações entre as partes e o comprimento de equivalência das combinações, denotado pelo indicador de posição do final da equivalência. Após a definição, elege-se qual das partes gera um conjunto que provoca um maior número de combinações, e, entre estas qual a combinação de maior cadeia, colocando-a posteriormente em evidência.
- Exemplo: dadas 3 partes
- parte 0: abcde
- parte 1: abtyr
- parte 2: abcct
- a lista de combinações será:
- iPartesTriplas[0][1] = 2 elementos (‘a’ e ‘b’)
- iPartesTriplas[0][2] = 3 elementos (‘a’, ‘b’, ‘c’)
- iPartesTriplas[1][2] = 2 elementos (‘a’ e ‘b’)
- a combinação de maior ocorrência é a que refere-se a iPartesTriplas[0], ou seja, duas ocorrências;
- para as combinações possíveis de iPartesTriplas[0], a de maior cadeia é a que a combina com a parte 2, ou seja: iPartesTriplas[0][2], e será a primeira a ser posta em evidência, resultando-se:
- parte 0: abc(de | ct)
- parte 1: abtyr

- a lista de combinações será:

- iPartesTriplas[0][1] = 2 elementos (‘a’ e ‘b’)
- a combinação de maior ocorrência é a que se refere a iPartesTriplas[0], ou seja, uma ocorrência
- para as combinações possíveis de iPartesTriplas[0], a de maior cadeia é a que combina com 1, ou seja iPartesTriplas[0][1] (a única), resultando-se:
- parte 0: ab(tyr | c(del|ct))
- Observação: este processo é realizado até que se acabem as ocorrências em comum, lembrando que todas as partes são recriadas e renumeradas para cada uma das ocorrências

Algoritmo: **simplificarRegrasPrefixosComuns**

Função: Simplificar uma Regra, colocando termos comuns em evidência.

Entradas:

- matriz sRegras de regras de produção
- vetor iParAbInterno de posicionamentos de parênteses mais internos que abrem
- vetor iParFeInterno de posicionamento de parênteses mais internos que fecham

Saídas:

- matriz sRegras ajustada com produções simplificadas

Descrição:

- [0]seja um vetor sRegraTemp[], de auxílio à montagem de uma nova regra
- [0]seja um vetor sAbFe[], de auxílio ao isolamento de um trecho entre parênteses
- [0]seja uma matriz sAbFePartes[][], que irá armazenar cada uma das partes separadas por '|', entre parênteses
- [0]seja um vetor sAbFePartesComprimento[], que conterá o comprimento de cada uma das partes
- [0]seja iMaior uma variável inteira que deverá armazenar o maior número de comparações obtidas entre cadeias
- [0]seja iMaiorTemp uma variável inteira que deverá armazenar temporariamente o maior número de comparações obtidas entre cadeias
- [0]seja iIndiceMaior um indicador do maior indice
- [0]seja iMaisLongo uma variável inteira que deverá armazenar a posição de cadeia mais longa
- [0]seja iMaisLongoTemp uma variável inteira que deverá armazenar temporariamente a posição de cadeia mais longa
- [0]para cada um dos trechos entre parênteses
  - [1]carregar o trecho em sAbFe
  - [1]para cada um dos elementos em sAbFe
    - [2]se não encontrar separador

então

- [3]copiar elemento para sAbFePartes, na linha atual

senão

- [3]copiar finalizador para sAbFePartes, na linha e coluna atual
- [3]incrementar o indexador de linha de sAbFePartes
- [3]atribuir zero ao indexador de colunas de sAbFePartes

- [1]copiar finalizador para sAbFePartes, na última coluna a ser carregada
- [1]copiar finalizador para sAbFePartes, na próxima entrada (linha) a ser carregada
- [1]copiar o comprimento da parte atual em sAbFePartesComprimento
- [1]se encontrou separador no laço anterior

  então

  - [2]faça
    - [3]para cada uma das partes em sAbFePartes
      - [4]para cada elemento de cada parte
        - [5]para cada elemento das partes restantes
          - [6]se o elemento da parte possuir posição de comprimento menor que o elemento da parte restante

            então

            - [7]para cada um dos elementos da parte e da parte restante
              - [8]se estes elementos forem iguais

                então

                - [9]carregar iPartesTriplas com a posição da eqüivalência na linha colunas correspondentes às partes comparadas

                senão

                - [9]finalizar esta comparação
        - [5]marcar o finalizador para iPartesTriplas
    - [3]separar a linha que gerou o maior número de combinações de iPartesTriplas:

- [3]carregar iMaior com zero
- [3]para cada uma das linhas de iPartesTriplas
  - [4]carregar iMaiorTemp com zero
  - [4]para cada uma das colunas de iPartesTriplas iniciando em linha+1
    - [5]se existir um conteúdo diferente de zero
      então
      - [6]somar um à iMaiorTemp
      - [6]se iMaiorTemp > iMaior
      então
        - [7]carregar iMaior com iMaiorTemp
        - [7]carregar iIndiceMaior com o indexador da linha atual em iPartesTriplas
- [3]separar na linha de maior número de combinações, a combinação que gera a cadeia mais longa:
- [3]carregar iMaisLongo com zero
- [3]carregar iMaisLongoTemp com zero
- [3]para cada uma das colunas de iAbFePartes, na linha iIndiceMaior e iniciando na coluna iIndiceMaior+1
  - [4]se existir um conteúdo diferente de zero
    então
    - [5]se o conteúdo for maior que o conteúdo de iMaisLongoTemp
      então
      - [6]carregar iMaisLongo com o indexador da coluna atual
      - [6]carregar iMaisLongoTemp com o conteúdo
- [3]montar a cadeia que une os elementos. A equivalência entre as partes referenciadas por iIndiceMaior e iMaisLongo estará garantida até a posição indicada por iAbFePartes[iIndiceMaior][iMaisLongo] :
- [3]para um contador de zero até a posição indicada por iAbFePartes[iIndiceMaior][iMaisLongo]

- [4]anexar o conteúdo de sAbFePartes[iIndiceMaior] em sAbFePartesTemp
- [4]se o conteúdo for um rótulo
  então
  - [5]ajustar o rótulo conforme regras da associação

- [3]anexar '(' em sAbFePartesTemp
- [3]anexar rótulo em branco em sAbFePartesTemp
- [3]copiar a parte diferente da cadeia referenciada por iIndiceMaior:
- [3]para um contador da posição indicada por iAbFePartes[iIndiceMaior][iMaisLongo]+1 até o comprimento da cadeia referenciada por iIndiceMaior
  - [4]se for o primeiro rótulo
    então
    - [5]ajustar o rótulo conforme regras da associação
  - [4]anexar o conteúdo de sAbFePartes[iIndiceMaior] em sAbFePartesTemp
- [3]se o loop anterior não gerou nenhum passo
  então
  - [4]anexar epsilon em sAbFePartesTemp
  - [4]anexar rótulo em branco em sAbFePartesTemp
- [3]anexar '|' em sAbFePartesTemp
- [3]anexar um rótulo em branco em sAbFePartesTemp
- [3]copiar a parte diferente da cadeia referenciada por iMaisLongo:
- [3]para um contador da posição indicada por iAbFePartes[iIndiceMaior][iMaisLongo]+1 até o comprimento da cadeia referenciada por iMaisLongo
  - [4]se for o primeiro rótulo
    então
    - [5]ajustar o rótulo conforme regras da associação
  - [4]anexar o conteúdo de sAbFePartes[iMaisLongo] em sAbFePartesTemp

- [3]se o loop anterior não gerou nenhum passo
  então
  - [4]anexar epsilon em sAbFePartesTemp
  - [4]anexar rótulo em branco em sAbFePartesTemp
- [3]anexar ‘)’ em sAbFePartesTemp
- [3]anexar um rótulo em branco em sAbFePartesTemp
- [3]anexar um finalizador em sAbFePartesTemp
- [3]neste momento, tendo ajustado a nova seqüência, assumir que esta será referenciada por iIndiceMaior e eliminar a seqüência referenciada por iMaisLongo, ajustando-se todas as partes:
- [3]para cada um dos elementos em sAbFeParteTemp
  - [4]copiar o elemento para sAbFePartes[iIndiceMaior]
- [3]copiar o finalizador para sAbFePartes[iIndiceMaior]
- [3]copiar o comprimento de iIndiceMaior em sAbFePartesComprimento
- [3]reordenar a lista sAbFePartes, eliminando a parte referenciada por iMaisLongo

- [2]enquanto encontrar alguma equivalência entre as partes. (fim)
- [2]atualizamos as modificações, substituindo o trecho entre sAbFe com o novo trecho:
- [2]para cada uma das linhas de sAbFePartes
  - [3]para cada uma das colunas de sAbFePartes
    - [4]copiar elemento para sAbFe
  - [3]copiar "|" para sAbFe
- [2]copiar finalizador para sAbFe
- [2]atualizar sRegra (regras originais), substituindo o trecho entre iParAbInterno e iParFeInterno com o trecho em sAbFe:
- [2]para cada um dos elementos de sRegra até a posição referenciada por iParAbInterno
  - [3]copiar sRegra para sRegraTemp
- [2]ajustar o *gap* a ser acrescentado pela movimentação dos parênteses:

- [2]para cada um dos elementos de sAbFe
  - [3]copiar sAbFe para sRegraTemp
- [2]copiar finalizador para sRegraTemp
- [2]para cada um dos elementos em sParAbInterno e sParFeInterno
  - [3]ajustar o conteúdo acrescendo o gap calculado acima
- [2]para cada um dos elementos de sRegraTemp
  - [3]copiar o elemento para sRegra

6.5.1.3.3 **Substituição de não-terminais**

O algoritmo a seguir realiza a substituição de não-terminais em extremidade esquerda, ou seja, que constem após '|' ou '(' .

Algoritmo: **simplificarRegrasSubstituirNaoTerminais**

Função: Simplificar uma Regra, substituindo não-terminais.

Entradas:

- matriz sRegras de regras de produção

Saídas:

- matriz sRegras ajustada

Descrição:

- [0]seja um vetor sRegraTemp[], de auxílio à montagem de uma nova regra
- [0]seja uma matriz inteira iElegiveis[][] onde a linha representa a regra e cada coluna representa uma posição de não-terminal elegivel para a substituição nesta regra
- [0]seja uma variável inteira iIndiceRegraNaoTerminal
- [0]para cada uma das regras em sRegras
  - [1] determina-se os não-terminais elegíveis para substituição:
  - [1]carregar iElegiveisColuna com zero
  - [1]para cada um dos elementos da regra em sRegras
    - [2]se o elemento for um não-terminal

      então
      - [3]se o elemento for elegível para a substituição

então

- [4]carregar em iElegiveis, na linha referente a esta regra e em uma nova coluna a posição deste elemento

- [1]copiar finalizador para a lista de elegiveis

- [0]substituição dos não-terminais elegíveis pelas correspondentes regras:
- [0]para cada uma das regras
  - [1]se existem não-terminais elegíveis

    então
    - [2]para todos os não-terminais elegíveis
      - [3]procurar a regra referente ao não-terminal em sRegras e armazenar seu indice em iIndiceRegraNaoTerminal
      - [3]para todos os elementos da regra atual de 0 até a posição anterior ao não-terminal elegível
        - [4]copiar o elemento da regra atual (sRegra) em sRegraTemp
      - [3]copiar o elemento ‘(‘ em sRegraTemp
      - [3]para cada um dos elementos da regra referenciada por iIndiceRegraNaoTerminal
        - [4]copiar o elemento da regra referenciada em sRegraTemp
        - [4]atualizar o *gap*
      - [3]copiar o elemento ‘)’ em sRegraTemp
      - [3]para todos os elementos da regra atual da posição posterior ao não-terminal elegível até o final
        - [4]copiar o elemento da regra atual (sRegra) em sRegraTemp
      - [3]copiar a nova regra (atualizada) em sRegras:
      - [3]para cada um dos elementos de sRegraTemp
        - [4]copiar o elemento para sRegras
      - [3]para todos os elementos de iElegíveis na linha atual
        - [4]somar o gap calculado ao elemento

6.5.1.3.4 Substituição de cadeia vazia

O algoritmo a seguir realiza a substituição de cadeia vazia, quando esta ocorre como uma das opções de um agrupamento. Pela sua extensibilidade e para facilitar o entendimento, este algoritmo foi descrito em grandes etapas as quais representam o significado de um conjunto de operações.

Algoritmo: **simplificarRegrasSubstituirEpsilon**

Função: Simplificar uma Regra, substituindo opções que contém ε.

Entradas:

- matriz sRegras de regras de produção

Saídas:

- matriz sRegras ajustada com produções simplificadas

Descrição:

- [0]seja um vetor sTemp, de auxílio à montagem de trechos a substituir
- [0]seja uma matriz sPartesInternasEpsilon, que conterá em cada linha cada uma das partes separadas por 'l' de um conjunto de elementos que contém o 'ε'
- [0]seja um vetor sFatorEsquerdo que irá conter os elementos correspondentes a um fator esquerdo (por exemplo, uma associação entre parênteses ou um único elemento) a uma associação que tenha o elemento epsilon
- [0]seja um vetor sFatorDireito que irá conter os elementos correspondentes a um fator direito (por exemplo, uma associação entre parênteses ou um único elemento) a uma associação que tenha o elemento epsilon
- [0]seja uma variável inteira iPosAbertura
- [0]seja uma variável inteira iPosFechamento
- [0]para cada uma das regras em sRegras
    - [1]para cada um dos elementos
        - [2]se o elemento for um 'ε'

          então
            - [3]percorrer todos os elementos à esquerda até encontrar um parênteses de abertura que não seja correspondente a qualquer fechamento à esquerda e armazenar em iPosAbertura

- [3]percorrer todos os elementos à direita até encontrar um parênteses de fechamento que não seja correspondente à qualquer abertura à esquerda e armazenar em iPosFechamento
- [3]construir a matriz sPartesInternasEpsilon, carregando cada linha com cada conjunto de elementos separados entre ‘|’ e compreendidos entre iPosAbertura e iPosFechamento.
- [3]se existem iPosAbertura e iPosFechamento
  então
  - [4]construir o sFatorEsquerdo à esquerda do conjunto entre iPosAbertura e iPosFechamento
  - [4]construir o sFatorDireito à direita do conjunto entre iPosAbertura e iPosFechamento
  - [4]limpar o vetor sTemp
  - [4]se existe um fator esquerdo
    então
    - [5]para cada um dos elementos da regra corrente até o elemento anterior ao início do fator esquerdo
      - [6]copiar elemento para sTemp

    senão
    - [5]para cada um dos elementos da regra corrente até o elemento anterior a iPosAbertura
      - [6]copiar elemento para sTemp
  - [4]para cada uma das partes internas em sPartesInternasEpsilon
    - [5]se o primeiro elemento desta parte for um epsilon
      então
      - [6]se existir fator esquerdo
        então
        - [7]copiar o rótulo do fator esquerdo em sTemp
        - [7]copiar os elementos do fator esquerdo em sTemp

        senão
        - [7]copiar o rótulo antes de iPosAbertura em sTemp

- [6]anexar ao rótulo final de sTemp (até o momento), o rótulo anterior e posterior à epsilon
- [6]copiar cada elemento após epsilon (pode ser uma associação) em sTemp
- [6]se existir fator direito

  então
  - [7]copiar rótulo do fator direito em sTemp
  - [7]copiar os elementos do fator direito em sTemp

  senão
  - [7]copiar o rótulo após iPosFechamento em sTemp
- [6]se próximo elemento em sPartesInternasEpsilon não for o finalizador

  então
  - [7]copiar "|" em sTemp
  - [7]copiar rótulo em branco em sTemp

senão
- [6]se existir fator esquerdo

  então
  - [7]copiar o rótulo do fator esquerdo em sTemp
  - [7]copiar os elementos do fator esquerdo em sTemp

  senão
  - [7]copiar o rótulo antes de iPosAbertura em sTemp
- [6]anexar ao rótulo final de sTemp (até o momento), o rótulo anterior ao elemento em sPartesInternasEpsilon
- [6]copiar cada elemento após este elemento (pode ser uma associação) em sTemp
- [6]se existir fator direito

  então
  - [7]copiar rótulo do fator direito em sTemp
  - [7]copiar os elementos do fator direito em sTemp

  senão
  - [7]copiar o rótulo após iPosFechamento em sTemp

        - [6]se próximo elemento em sPartesInternasEpsilon não for o finalizador

          então

            - [7]copiar ‘|’ em sTemp
            - [7]copiar rótulo em branco em sTemp
    - [5]copiar os elementos de sFatorEsquerdo em sTemp
    - [5]copiar os elementos de sFatorDireito em sTemp
- [4]//finalização da regra montada em sTemp:
- [4]para cada um dos elementos a partir de iPosFechamento+2 (elemento após ‘(‘)
    - [5]copiar elemento em sTemp
- [4]finalizar sTemp
- [4]copiar a nova regra montada em sTemp para a regra correspondente em sRegras

6.5.1.3.5 Eliminação de construções cíclicas

O algoritmo a seguir realiza a substituição de construções cíclicas à esquerda de uma opção. Pela sua extensibilidade e para facilitar o entendimento, este algoritmo foi descrito em grandes etapas que definem o significado de um conjunto de operações.

Algoritmo: **simplificarRegrasEliminarConstrucaoCiclica**

Função: Simplificar uma Regra, alterando construções cíclicas à esquerda pelo seu equivalente.

Entradas:

- matriz sRegras de regras de produção

Saídas:

- matriz sRegras ajustada com produções simplificadas

Descrição:

- [0]seja um vetor sTemp, de auxílio à montagem de trechos a substituir
- [0]seja um vetor sElementoCiclico, para armazenar o elemento cíclico
- [0]seja um inteiro iPosAbertura
- [0]seja um inteiro iPosFechamento
- [0]seja um inteiro iPosEpsilon
- [0]para cada uma das regras em sRegras
  - [1]para cada um dos elementos
    - [2]se o elemento for um conjunto " ε, rot, \, rot "
      então
      - [3]carregar iPosEpsilon com a posição de ε
      - [3]percorrer cada elemento à esquerda até encontrar um ‘(‘, armazenando sua posição em iPosAbertura
      - [3]se o conjunto a partir de iPosAbertura for inicio de uma das opções ou for início de regra
        então

- [4]percorre-se cada elemento à direita até encontrar um ‘)’, copiando cada elemento após "\" para sElementoCiclico
- [4]finalizar sElementoCiclico
- [4]carregar iPosFechamento com a posição do ‘)’ encontrado
- [4]para cada elemento da cadeia original até iPosAbertura-1
    - [5]copiar elemento em sTemp
- [4]atualizar o último elemento (rótulo) de sTemp anexando o rótulo antes de ε
- [4]copiar o ε em sTemp
- [4]copiar o rótulo de ε em sTemp
- [4]atualizar o último elemento (rótulo) de sTemp anexando o rótulo após iPosFechamento
- [4]copiar ‘|’ a sTemp
- [4]copiar o rótulo a sTemp, que deverá ser formado pelo rótulo antes de ε, após ε e do início da cadeia cíclica
- [4]copiar os elementos de sElementoCiclico em sTemp
- [4]copiar um ‘(‘ em sTemp
- [4]copiar rótulo em branco em sTemp
- [4]copiar o ε em sTemp
- [4]copiar ‘\’ em sTemp
- [4]copiar os elementos de sElementoCiclico em sTemp
- [4]copiar ‘)’ em sTemp
- [4]copiar o rótulo após iPosFechamento em sTemp
- [4]copiar todos os outros elementos restantes da regra original a sTemp
- [4]para todos os elementos de sTemp
    - [5]copiar elemento para a regra atual de sRegras, atualizando-a

6.5.2 Obtenção do analisador sintático

Nesta seção são apresentados alguns algoritmos relevantes para a construção do *parser* sintático.

6.5.2.1 Atribuição de estados

Neste algoritmo aplica-se a atribuição de estados, de acordo com as regras apresentadas em 5.1 .

Algoritmo: **numerarEstados**

Função: Realizar a atribuição de estados a cada uma das regras

Entradas:

- matriz sRegras de regras de produção

Saídas:

- matriz sRegras com a inclusão dos números de estados

Descrição:

- [0] seja um vetor inteiro iParAb para armazenamento dos índices de abertura de parênteses
- [0] seja um vetor inteiro iParFe[] para armazenamento dos índices de fechamento de parênteses
- [0] seja uma pilha stkPar utilizada para o empilhamento de parênteses
- [0] seja iNovoEstado uma variável que representa o número de um novo estado a ser atribuído
- [0] sejam r, s, x, y números de estados a serem trabalhados de forma semelhante ao exposto em 5.1
- [0] para cada uma das regras em sRegras
  - [1] para cada um dos elementos da regra selecionada
  - [1]localizar os agrupamentos entre parênteses da expressão:
    - [2] se encontrar um parênteses de abertura

      então
      - [3] carregar o vetor iParAb com a posição do parênteses de abertura, empilhando-se esta posição em stkPar

- [2] se encontrar um parênteses de fechamento
  então
  - [3] armazenar a posição em iParFe, utilizando-se como índice a posição armazenada no topo da pilha sktPar.
  - [3]Retirar elemento do topo de stkPar

- [1] selecionar um agrupamento completo ainda não tratado:
- [1] para todos os agrupamentos obtidos, delimitados por iParAb e iParFe
  - [2] atribuir valor 1 a iNovoEstado
  - [2] obter o número de estado antes da posição de abertura do agrupamento selecionado
  - [2] se este número de estado existir
    então
    - [3] carregar r com o número de estado
      senão
    - [3] carregar r com zero
  - [2] obter o número de estado após posição de fechamento do agrupamento selecionado
  - [2] se este número de estado existir
    então
    - [3] carregar s com o número de estado
      senão
    - [3] carregar s com zero
  - [2] se existir r
    então
    - [3] x=r
      senao
    - [3] atribui-se a x um novo estado:
    - [3] x=iNovoEstado
    - [3] iNovoEstado = iNovoEstado + 10
  - [2] se existir s
    então

- [3] y=s

  senao
- [3] atribui-se a y um novo estado:
- [3] y=iNovoEstado
- [3] iNovoEstado = iNovoEstado + 10

- [2] designar x aos pontos extremos esquerdos de todas as opções internas ao agrupamento
- [2] designar y aos pontos extremos direitos de todas as opções internas ao agrupamento
- [2] de acordo com o exposto em 5.1, a existência de construção cíclica (\) leva à atribuição de y aos estados antes e depois da construção cíclica, e à atribuição invertida aos elementos a seguir à construção, isto é, atribui-se y (esq) e x(dir), ao inves de x (esq) e y (dir) para tanto, realiza-se esta verificação
- [2] numera-se os estados extremos de cada uma das expressões que compõem o agrupamento
- [2] na expressão obtida, propaga-se o estado x para o ponto externo imediatamente à esquerda do agrupamento, caso a este ponto não tenha sido atribuído nenhum estado
- [2] na expressão obtida, propaga-se o estado y para o ponto externo imediatamente à direita do agrupamento, caso a este ponto não tenha sido atribuído nenhum estado
- [2] aplica-se, de forma análoga ao apresentado nos itens anteriores, a propagação dos estados previamente associados à esquerda e à direita a todos os agrupamentos internos e não cíclicos ainda não tratados

Obs: os elementos que indicam ações semânticas são considerados símbolos terminais neste processo.

6.5.2.2 Mapeamento das regras

O algoritmo abaixo realiza o mapeamento das regras.

Algoritmo: **mapearRegras**

Função: Realizar a conversão das regras de produção para autômato, conforme descrito em 5.2. Pela sua extensibilidade e para facilitar o entendimento, este algoritmo foi descrito em grandes etapas as quais representam o significado de um conjunto de operações.

Entradas:

- matriz sRegras de regras de produção

Saídas:

- produção do autômato que representa o *parser*, com todos os tratamentos necessários ao reconhecimento de cadeia.

Descrição:

- [0] seja sLinhaAdaptools uma matriz bidimensional de tipo String que deverá ser carregada com as linhas para o Adaptools
- [0] atribuir a sLinhaAdaptools a primeira linha, equivalente à versão da ferramenta:
- sLinhaAdaptools[0][0]="[Version] 2";
- [0] criar estados iniciais para cada uma das regras de produção para que estes estados sejam usados como possíveis empilhamentos de retorno.
- [0] para cada uma das regras de produção
  - [1] para cada um dos elementos
    - [2]buscar um agrupamento cíclico
    - [2] se encontrar um agrupamento cíclico
      então
      - [3] marcar o início do rótulo esquerdo ao agrupamento encontrado
      - [3] marcar o final do rótulo direito ao agrupamento encontrado
  - [1] montar as transições para os conjuntos que contenham construções cíclicas, detectados no passo anterior:

- [1] detectar os elementos x, y, z, t, u e v descritos em 5.2
- [1] montar as transições correspondentes através da chamada ao algoritmo montarLinha (*):

  (γ , x, α) → (γ , y, α)
  (γ , z, α) → (γ , v, α)
  (γ , z, α) → (γ , t, α)
  (γ , t, α) → (γ , v, α)
  (γ , u, α) → (γ , t, α)

- [1] detecta-se os demais elementos:
- [1] para cada um dos elementos
  - [2] obter o rótulo e número de estado (x) à esquerda do elemento
  - [2] obter o rótulo e número de estado (y) à direita do elemento
  - [2] se o elemento for uma ocorrência de um 'ε'
    então
    - [3] criar uma transição em vazio de x para y
    - [3] montar a transição correspondente através da chamada ao algoritmo montarLinha (*)
  - [2] se o elemento for um terminal ou ação semântica
    então
    - [3] criar uma transição de x para y consumindo o terminal
    - [3] montar a transição correspondente através da chamada ao algoritmo montarLinha (*)
  - [2] se o elemento for um não-terminal
    então
    - [3] criar uma transição em vazio partindo do estado atual x para o estado inicial da submáquina associada ao não-terminal (para isso, deve-se procurar o estado inicial que corresponda ao início da definição desta submáquina)
    - [3] montar a transição correspondente através da chamada ao algoritmo montarLinha (*), salvando o estado y na pilha do autômato correspondente à esta transição

- [1] para cada uma das regras
  - [2] se não for a primeira regra
    então
    - [3] criar uma transição em vazio que execute o retorno ao estado y da submáquina “chamadora”
    - [3] montar a transição correspondente através da chamada ao algoritmo montarLinha (*)

    senao
    - [3] criar um desvio para o estado final
    - [3] montar a transição correspondente através da chamada ao algoritmo montarLinha (*)
    - [3] executar o algoritmo “montarPilha” que deverá montar a pilha de transdução, conforme descrito em 5.3.3.

(*) Observação: O algoritmo montarLinha é responsável pela montagem de uma transição do *parser* no *Adaptools*, incluindo a esta transição, transições adicionais que possibilitem a construção da árvore sintática de saída, ou seja, que transformam o autômato de reconhecimento em um transdutor que produza a árvore de saída, conforme descrito em 5.3.4.

6.5.2.3 Montagem das linhas

Algoritmo: **montarLinha**

Função: Este algoritmo recebe como parâmetros os elementos necessários para a construção de uma transição no *Adaptools*. A partir destes elementos, constrói a transição, porém incluindo transições adicionais para a construção da árvore sintática de saída, conforme descrito em 5.3. Pela sua extensibilidade e para facilitar o entendimento, este algoritmo foi descrito em grandes etapas as quais representam o significado de um conjunto de operações.

Entradas:

- sLinhaAdap: matriz que representa cada uma das linhas a serem montadas
- sRegra: nome da regra
- sNumEstEsq: número de estado à esquerda do elemento
- sRotEsq: rótulo à esquerda do elemento
- sSimbolo: símbolo (elemento)
- sNumEstDir: número de estado à direita do elemento
- sRotDir: rótulo à direita do elemento
- sPush: símbolo a ser empilhado no *Adaptools*
- sAdap: ação adaptativa a ser executada no *Adaptools*

Saídas:

- produção do autômato que representa o *parser*, com todos os tratamentos necessários ao reconhecimento de cadeia e produção de árvore sintática.

Descrição:

- [0] seja sNovoEstEsq um novo número de estado à esquerda do símbolo/elemento, criado para o processamento dos rótulos. A partir deste estado, e até o estado sNumEstDir, serão incluídas transições que permitam o processamento do rótulo através de empilhamentos e desempilhamentos que produzam como saída a árvore sintática desejada.
- [0] se não existir um rótulo sRotDir
  então

- [1] se o elemento sSimbolo for uma ação semântica
  então
  - [2] produzir uma transição em vazio, de sNumEstEsq a sNumEstDir
  senão
  - [2] produzir uma transição que consome sSimbolo, de sNumEstEsq a sNumEstDir

senao

- [1] cria-se um novo número de estado à esquerda do símbolo, intermediário entre sNumEstEsq e sNumEstDir, denominado sNovoEstEsq, conforme descrito acima
- [1] se sPush não for "nop", isto é, se houver algum empilhamento de estado de retorno
  então
  - [2] cria-se o consumo do símbolo a partir de uma transição do estado sNumEstEsq ao estado sNumEstDir, carregando-se sPush com sNovoEstEsq (isto porque o processamento do rótulo deverá se dar após o processamento da submáquina)

  senão
  - [2] se sSimbolo não for ação semântica
    então
    - [3] cria-se o consumo do símbolo a partir de uma transição do estado sNumEstEsq ao estado sNovoEstEsq

    senão
    - [3] corrigir o novo número de estado esquerdo para o estado original, pois não há a necessidade do novo estado:
    - [3] sNovoEstEsq = sNumEstEsq
- [1] para a montagem das linhas referentes aos rótulos, aplica-se o descrito em 5.3.4, ou seja, apesar das transições já terem sido trabalhadas anteriormente, o que se faz aqui é a criação das saídas de acordo com os rótulos:
- [1] A- analisar cada rótulo do estado de destino (sRotDir) da regra, buscando seus rótulos elementares, que podem conter os seguintes elementos:

        - ε
        - qualquer terminal
        -]
        - [
        - não-terminais com os respectivos indicadores de recursão central, esquerda, direita
        - ações semânticas
- [1] para cada um dos elementos analisados no rótulo representado por sRotDir
    - [2] se o elemento for uma ação semântica
      então
        - [3] apresenta-se a ação semântica na saída, incluindo-se para isto uma transição em vazio a partir de sNovoEstEsq até sNovoEstEsq + 1
    - [2] se o elemento for um épsilon
      então
        - [3] apresenta-se "( )" na saída, incluindo-se para isto uma transição em vazio a partir de sNovoEstEsq até sNovoEstEsq + 1
    - [2] se o elemento for um terminal qualquer
      então
        - [3] apresenta-se "(terminal)" na saída, incluindo-se para isto uma transição em vazio a partir de sNovoEstEsq até sNovoEstEsq + 1
    - [2] se o elemento for um não-terminal
      então
        - [3] se for recursivo à esquerda
          então
            - [4] apresenta-se o não-terminal, o símbolo '□' de recursão à esquerda e ')' na saída
        - [3] se for "outros casos" (geral)
          então

- [4] apresenta-se o não-terminal, o símbolo '□' e ')' na saída incluindo-se para isto transições em vazio a partir de sNovoEstEsq
- [4] processa-se o desempilhamento "pi" na pilha de transdução. Para isso, desempilha-se pi,] e empilha-se] , através de chamada à ação adaptativa correspondente ao empilhamento na pilha de transdução para o empilhamento, e desvio para a submáquina da pilha para o desempilhamento (conforme descrito em 5.3.3)

- [3] se for recursivo à direita
  então
  - [4] apresenta-se '(' na saída, incluindo-se para isto transições em vazio a partir de sNovoEstEsq
  - [4] empilha-se ')', o não terminal e o símbolo '□' de recursão à direita, incluindo-se para isto transições em vazio a partir de sNovoEstEsq

  senão
- [3] se elemento for um '['
  então
  - [4] produz-se "[(" na saída, incluindo-se para isto transições em vazio a partir de sNovoEstEsq
  - [4] empilha-se o símbolo ']' , utilizando-se chamada à ação adaptativa correspondente ao empilhamento na pilha de transdução (conforme descrito em 5.3.3)
- [3] se elemento for um ']'
  então
  - [4] produz-se um desempilhamento "pi" através de uma chamada à submáquina de desempilhamento (conforme descrito em 5.3.3)
  - [4] produz-se ']' na saída, incluindo-se para isto transições em vazio a partir de sNovoEstEsq

- [1] cria-se uma transição em vazio para o estado direito, representado por sNumEstDir, para se finalizar a transição tratada neste método

6.6 Exemplo de funcionamento

Para melhor ilustrar a apresentação da ferramenta e para um melhor entendimento, optou-se pela execução do exemplo apresentado em 5.3.3.

Na figura abaixo, carrega-se o software com a gramática do conversor binário-decimal na área de texto intitulada “Gramática a ser analisada”. Todos os outros campos de texto são gerados automaticamente, e o campo de texto “Regras Ajustadas” pode ser alterado e retrabalhado pelo usuário, a fim de se melhorar a apresentação da regra gerada automaticamente.

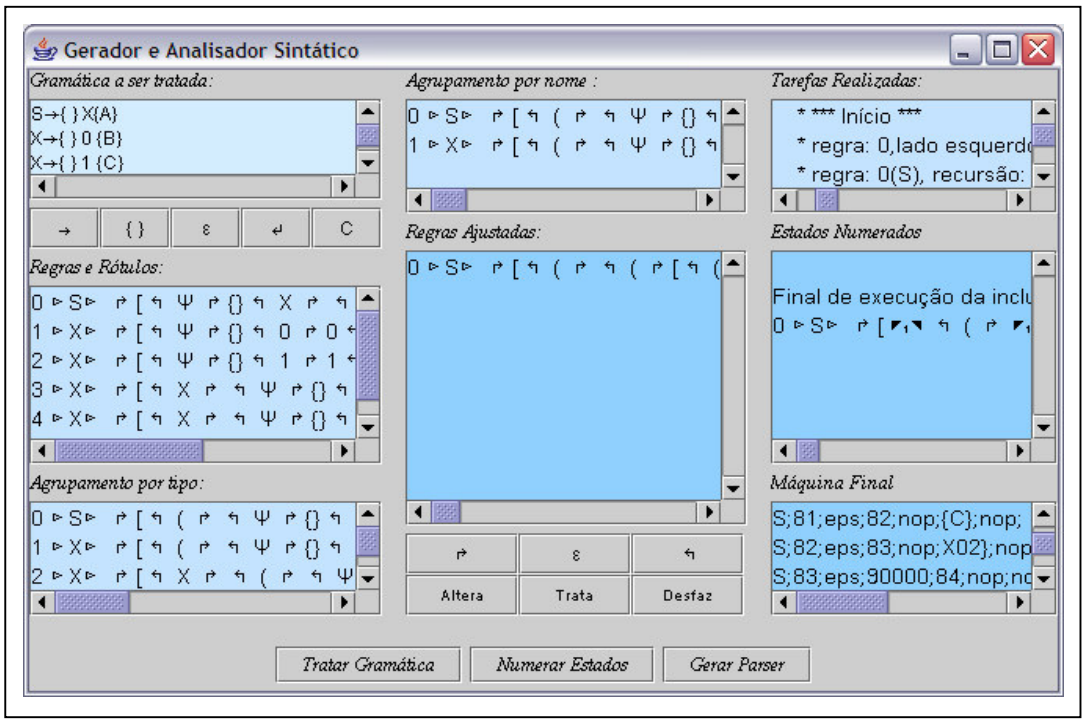


Fig. 31 – Exemplo de execução

Para este exemplo, foram obtidos os seguintes resultados, lembrando-se que os símbolos □ e □ representam a indicação da existência de um rótulo e número de estado correspondentes ao símbolo/elemento imediatamente à esquerda. As etapas apresentadas foram extraídas diretamente da ferramenta. Por esta razão, alguns tipos de caracteres podem ser diferentes daqueles apresentados até o momento.

Campo 1: gramática a ser tratada

Preenchido pelo usuário com a gramática apresentada em 5.3.3:

S→{ } X{A}
X→{ } 0 {B}
X→{ } 1 {C}
X→X{ } 0 {D}
X→X{ } 1 {E}

Fig. 32 – Gramática a ser tratada

Campo 2: regras e rótulos

Preenchido automaticamente pelo software a partir do passo anterior, com a rotulação adequada das regras.:

0 □ S□ □ [□ Ψ □ {} □ X □ □ Ψ □ {A}S□□□] □
1 □ X□ □ [□ Ψ □ {} □ 0 □ 0 □ Ψ □ {B}X□□□] □
2 □ X□ □ [□ Ψ □ {} □ 1 □ 1 □ Ψ □ {C}X□□□] □
3 □ X□ □ [□ X □ □ Ψ □ {} □ 0 □ 0 □ Ψ □ {D}X□□□] □
4 □ X□ □ [□ X □ □ Ψ □ {} □ 1 □ 1 □ Ψ □ {E}X□□□] □

Fig. 33 – Regras e rótulos

Campo 3: agrupamento por tipo

A partir do passo anterior, o software altera automaticamente as regras, definindo apenas uma regra para cada tipo de recursão (esquerda, geral, direita), resultando assim em apenas 3 regras: a regra 00 é mantida, a regra 01 representa o agrupamento das regras anteriores 01 e 02 (recursão geral), e a regra 02 representa o agrupamento das regras anteriores 03 e 04 (recursão à esquerda):

0 □ S□ □ [□ (□ □ Ψ □ {} □ X □ □ Ψ □ {A}S□□□ □) □] □
1 □ X□ □ [□ (□ □ Ψ □ {} □ 0 □ 0 □ Ψ □ {B}X□□□ □ | □ □ Ψ
□ {} □ 1 □ 1 □ Ψ □ {C}X□□□ □) □] □
2 □ X□ □ [□ X □ □ (□ □ Ψ □ {} □ 0 □ 0 □ Ψ □ {D}X□□□ □ |
□ □ Ψ □ {} □ 1 □ 1 □ Ψ □ {E}X□□□ □) □] □
Fig. 34 – Agrupamento por tipo

Campo 4: agrupamento por nome

Após o agrupamento por tipo, apenas uma regra para cada não-terminal é gerada, podendo reunir diversos tipos de recursão em uma única regra. Notar a geração automática de uma única regra para o não-terminal X (regra 01), que representa a junção das regras anteriores 01 e 02:

0 □ S□ □ [□ (□ □ Ψ □ {} □ X □ □ Ψ □ {A}S□□□ □) □] □
1 □ X□ □ [□ (□ □ Ψ □ {} □ 0 □ 0 □ Ψ □ {B}X□□□ □ | □ □ Ψ
□ {} □ 1 □ 1 □ Ψ □ {C}X□□□ □) □ □ (□ □ ε □ □ \ □ □ Ψ
□ {} □ 0 □ 0 □ Ψ □ {D}X□□□ □ | □ □ Ψ □ {} □ 1 □ 1 □
Ψ □ {E}X□□□ □) □] □
Fig. 35 – Agrupamento por nome

Campo 5: regras ajustadas

Após o passo anterior, as regras são ajustadas com o menor número de não-terminais possível. Devido à complexidade do ajuste e das etapas anteriores, dá-se ao usuário a possibilidade de melhoria ou alteração no resultado das regras ajustadas, através do botão *altera* (que ocasiona a aceitação do campo 5 alterado pelo usuário, após confirmação), do botão *trata* (que quando possível re-simplifica a gramática), e do botão *desfaz* (que possibilita desfazer qualquer alteração indesejada). O resultado deste campo está apresentado na figura abaixo.

Notar que apenas uma regra representa então a gramática transformada.

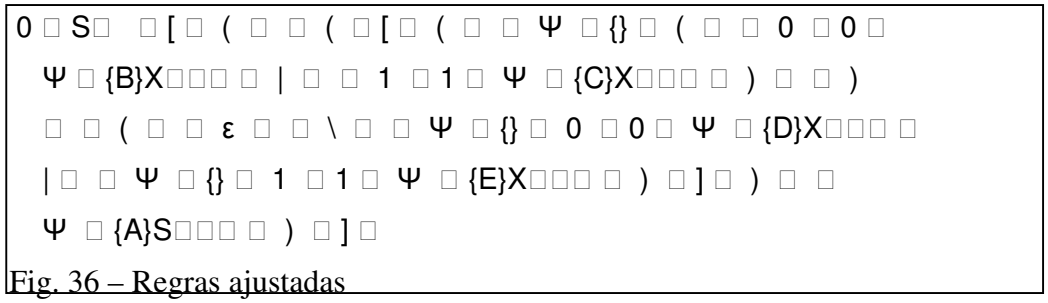


Fig. 36 – Regras ajustadas

Campo 6: tarefas realizadas

Neste campo é apresentado um resumo das tarefas realizadas passo-a-passo, para efeito de verificação e para efeito didático. Devido à extensibilidade do conteúdo deste campo, optou-se pela apresentação de um pequeno trecho para a simulação/exemplo desta seção:

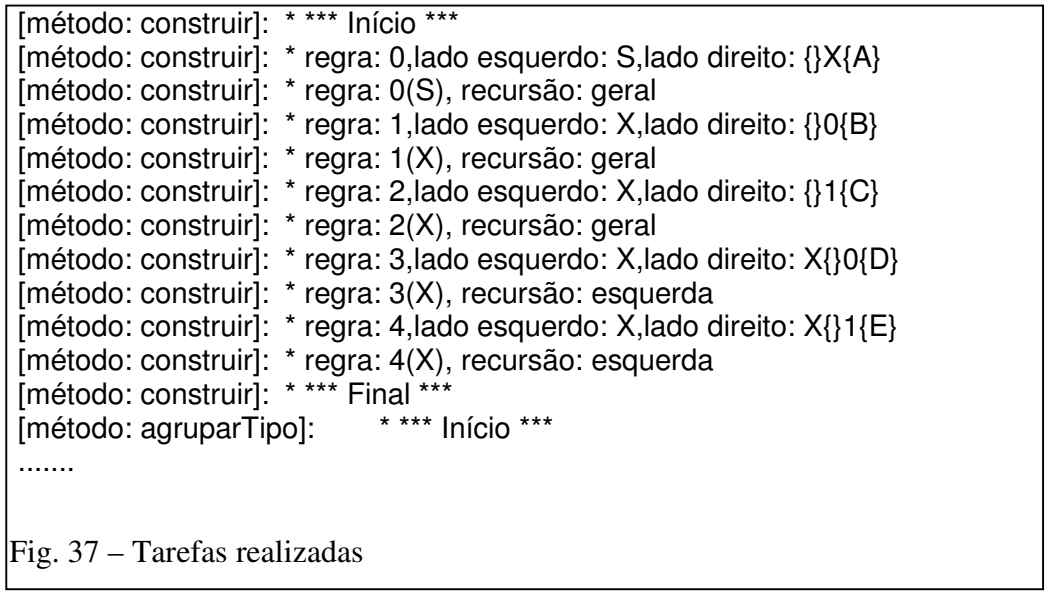


Fig. 37 – Tarefas realizadas

Campo 7: estados numerados

A partir das regras ajustadas, os números de estados são incluídos, e aparecem como anexos ao rótulo de cada elemento da regra. O algoritmo é aplicado no sentido de se diminuir ao máximo o número total de estados:

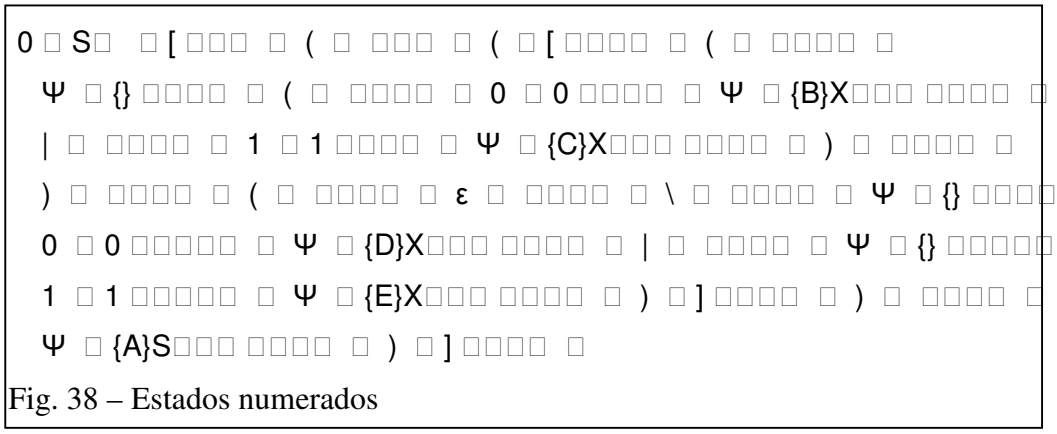


Fig. 38 – Estados numerados

Campo 8: Máquina final

Com a obtenção das regras finais devidamente numeradas, o programa constrói o código a ser inserido no *Adaptools*. Devido à extensibilidade do conteúdo deste campo, optou-se pela apresentação de um pequeno trecho da máquina final, para efeito de ilustração:

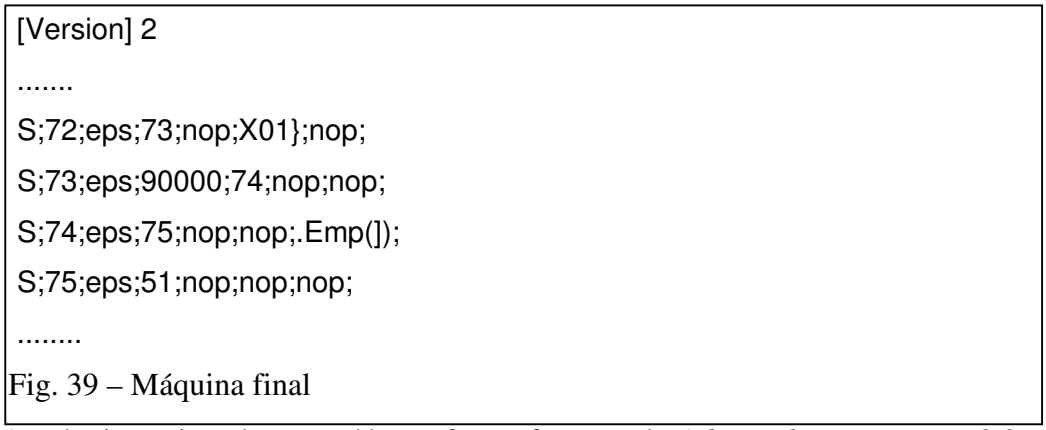


Fig. 39 – Máquina final

A máquina acima é construída conforme formato do *Adaptools*, exposto em 3.2, onde nop representa *nenhuma ação*, e eps representa *épsilon*. Desta forma, este trecho representa a aplicação dos seguintes passos, detalhados linha a linha:

- S;72;eps;73;nop;X01};nop;

Criação de uma transição em vazio na máquina S, do estado 72 ao estado 73, sem o empilhamento de símbolos, com a emissão dos caracteres X01} na saída, e sem a aplicação de qualquer função adaptativa.

- S;73;eps;90000;74;nop;nop;

Criação de uma transição em vazio na máquina S, do estado 73 ao estado 90000, com o empilhamento do número de estado 74 (para retorno), sem emissão de caracteres na saída e sem a aplicação de qualquer função adaptativa. Notar que nesta linha está sendo realizado o desempilhamento da pilha Δ constante no topo da pilha de transdução Θ definida anteriormente em 5.3.2 e 5.3.3. Este desempilhamento provocará a emissão, na saída, da seqüência π, conforme a aplicação da tabela de mapeamento. Ao final do desempilhamento de Δ, o processamento do autômato retornará ao estado 74.

- S;74;eps;75;nop;nop;.Emp(]);

Criação de uma transição em vazio na máquina S, do estado 74 ao estado 75, sem o empilhamento de símbolos, sem a emissão de caracteres na saída, e com a aplicação da função adaptativa .Emp(]), que realizará o empilhamento de] na pilha de transdução Θ . Este empilhamento está detalhado anteriormente, em 5.3.3 .

- S;75;eps;51;nop;nop;nop;

Criação de uma transição em vazio na máquina S, do estado 75 ao estado 51, sem o empilhamento de símbolos, sem a emissão de caracteres na saída, e sem a aplicação de qualquer função adaptativa.

7 - CONSIDERAÇÕES FINAIS

Em 6.6 ilustra-se, a título de exemplo, o funcionamento da ferramenta, ou seja, partindo-se de regras de produção acrescidas de ações semânticas, há o devido ajuste

e simplificação da gramática, e um *parser* final é criado em formato de código para a sua execução no *Adaptools*.

Para a aplicação dos métodos e algoritmos propostos durante o desenvolvimento, foram criados diversos métodos na linguagem Java, com o objetivo de que fossem fracamente acoplados em sua construção.

A construção do software foi realizada através da aplicação direta dos algoritmos propostos passo a passo, da forma mais simples possível, resultando um código relativamente simples, devidamente comentado e que utiliza como estruturas de dados apenas matrizes, vetores e pilhas, inseridos em *loops* de controle incremental simples na maioria dos casos, sendo assim um código de fácil entendimento para manutenção, futuras implementações e adaptações por terceiros.

Durante a construção da ferramenta e na realização dos testes, constatou-se que, algumas vezes durante a geração automática *das regras ajustadas* (caixa de texto da aplicação – ver exemplo no item 6.6), seria desejável uma intervenção do usuário, para melhorar a representação de uma regra. Por exemplo, nas regras geradas automaticamente, há a possibilidade da existência de grupos equivalentes entre parênteses, ou mesmo de estados desnecessários obtidos por associações. Por esta razão, permitiu-se ao usuário a modificação das regras ajustadas antes da atribuição de números de estado e da conseqüente geração do *parser*. Para este propósito, criou-se o botão *Altera,* que permite a alteração das regras, e o botão *Trata*, que permite uma nova aplicação dos algoritmos de tratamento gramatical, porém desta vez atuando sobre a gramática modificada pelo usuário. Isto proporcionou um maior dinamismo ao sistema, bem como liberdade ao usuário na modificação de regras, durante o trabalho de simplificação.

Constatou-se também que a geração do *parser,* em alguns casos, incluiu transições repetidas. Estas transições foram geradas pelo processo de construção do autômato de reconhecimento descrito em 5.2 a partir da numeração de estados simplificada, descrita em 5.1. Isto porque o processo de numeração de estados prioriza a obtenção do menor conjunto possível de estados, resultando transições equivalentes em pontos diferentes da gramática. Este problema foi resolvido através da inclusão de cada uma das transições e respectivos estados *origem* e *destino* em uma matriz de controle interna. Durante o processo de construção do reconhecedor, a inclusão de qualquer

nova transição é realizada na matriz de controle somente se esta transição já não tiver sido incluída anteriormente.

Durante os testes realizados, foi constatado então o funcionamento satisfatório da ferramenta e dos métodos propostos, aplicados a gramáticas livres de contexto com ações semânticas.

7.1 Futuras Implementações e Pesquisas Sugeridas

Embora neste trabalho tenha sido abordada a definição, construção, e implementação de um *parser* aplicado a linguagens livres de contexto com ações semânticas, sugere-se uma pesquisa futura na aplicação dos métodos para a sua adequação à linguagens dependentes de contexto com o uso de técnicas adaptativas. Em (IWAI, 2000), são propostos métodos similares aos expostos neste trabalho, porém considerando-se as gramáticas adaptativas. Neste caso, as ações adaptativas são incluídas na gramática original e são transformadas de forma semelhante às ações semânticas, mas, ao contrário destas, sempre ocorrem no início de uma regra.

A aplicação das ações adaptativas dá-se durante o reconhecimento da cadeia de entrada, porém estas ações atuam sobre a gramática original (antes da transformação), o que impõe um retrabalho excessivo sobre a gramática original. Como futura implementação, sugere-se a proposta de um mecanismo formal de transformação das ações adaptativas (que atualmente agem sobre a gramática original) para que passem a atuar na gramática transformada, permitindo-se assim, a inclusão de adaptatividade de forma equivalente ao proposto nesta dissertação.

Adicionalmente, sugere-se uma melhoria na ferramenta externa que processa o *parser* criado (*Adaptools)*, de forma que a mesma trate não-determinismos. Este foi um fator que, de certa forma, atrapalhou a implementação dos testes realizados, uma vez que o autômato de reconhecimento deve ser determinístico para que o *Adaptools* possa processá-lo.

No anexo há uma breve exposição dos algoritmos apresentados em (IWAI, 2000), comparando-se, quando possível, aos algoritmos deste trabalho. Deve-se lembrar, no entanto, que os algoritmos, da forma como agora estão propostos, não incluem a adequação das ações adaptativas para que atuem sobre as regras transformadas.

A utilização de gramáticas adaptativas viabilizaria uma série de outras aplicações. Como conhecimento adicional, sugere-se (BASSETO; NETO, 1999) onde é proposta a construção de um compositor musical adaptativo, (COSTA; HIRAKAWA; NETO, 2002) onde é aplicada adaptatividade ao reconhecimento de padrões, (NETO, 1994), (NETO, 2001) e (NETO; PARIENTE, 2002). Para o tratamento de linguagens naturais (que podem ser trabalhadas com gramáticas adaptativas), sugere-se Allen (1955) e Grishman (1986) que exploram o conceito e aplicações da linguagem natural, Black (1998) que apresenta o método de Government Binding aplicado ao tratamento de linguagens naturais, e (MENEZES, C. E. D.; NETO, 2002), onde é realizada a construção de um etiquetador morfológico baseado em autômatos adaptativos.

7.2 Contribuições

Como principais contribuições deste trabalho, pode-se apontar:

- A construção de um método para a geração automática de *parsers* com a inclusão de ações semânticas;
- A transformação deste método em algoritmos que possibilitem a construção de uma ferramenta de geração automática;
- A verificação da inclusão de ações adaptativas no método apresentado;
- A construção de uma ferramenta de trabalho significativa, tanto para o auxílio na construção de *parsers*, quanto para a sua utilização na preparação de gramáticas;
- A viabilidade da utilização do método apresentado para fins educacionais, através de exemplos apresentados passo-a-passo na ferramenta construída;
- A validação de metodologias propostas em teses anteriores através da aplicação, evolução e extensão dos algoritmos e métodos de (NETO, 1993), (IWAI, 2000), (PEREIRA, 1999);

7.3 Conclusões

Embora existam diversos métodos para a construção de *parsers* a partir de gramáticas livres de contexto, este trabalho foi baseado em um método alternativo, devido à sua simplicidade e aplicabilidade para fins didáticos. O método aqui desenvolvido é, pois, um método prático que permite a extensão do autômato automaticamente gerado em (NETO, 1993) e (NETO; PARIENTE, 1999), através da inclusão de chamadas a ações semânticas incluídas na gramática originalmente fornecida.

No início do trabalho, não havia uma ferramenta que implementasse os recursos apontados em (NETO, 1993), e que permitisse a visualização didática das fases de sua execução, gerando automaticamente os analisadores sintáticos desejados. Durante a realização deste trabalho, os métodos apresentados anteriormente foram estudados exaustivamente, e constatou-se que a sua implementação prática na forma de uma ferramenta seria viável.

A ferramenta gerou transdutores simples e compatíveis com os originais, sendo capazes de simultaneamente reconhecer a cadeia de entrada, gerar a árvore sintática e indicar a execução das ações semânticas.

O método e a ferramenta contribuem de forma a auxiliar o os pesquisadores do LTA, que poderão estudá-los ou utilizá-los como parte de seus trabalhos de pesquisa.

A produção (e aceitação) deste trabalho em artigo e evento (RICCHETTI; NETO, 2005a), (RICCHETTI; NETO, 2005b), em que há a exposição do método, atestaram a sua relevância como trabalho de pesquisa e desenvolvimento.

A construção da ferramenta tornou possível a aplicação do método no ensino de linguagens formais.

Este trabalho poderá ser o ponto de partida de uma série de trabalhos futuros, dentre eles, a inclusão de adaptatividade e o estudo de métodos para o tratamento de gramáticas livres e dependentes de contexto.

ANEXO

Neste anexo é apresentada a preparação de uma gramática adaptativa (IWAI, 2000), comparando-a com o método desenvolvido neste trabalho.

Extensão do método proposto para o tratamento de gramáticas adaptativas

1 Detecção de tipo de recursão

Considerando-se que α e β representam qualquer seqüência de terminais e/ou não-terminais, {A} representa uma ação adaptativa qualquer, e que P é não-terminal, assume-se que:

- $P \rightarrow \{A\}\ P\ \alpha$ define uma recursão à esquerda;
- $P \rightarrow \{A\}\ \alpha\ P$ define uma recursão à direita;
- $P \rightarrow \{A\}\ \alpha\ P\ \beta$ define um caso geral (sem recursão);
- $P \rightarrow \{A\}\ \alpha$ define um caso geral (sem recursão);
- $P \rightarrow \square$ define um caso geral (sem recursão);

2 Rotulação das produções

A inclusão de rótulos para os elementos de cada uma das regras é muito semelhante à inclusão de rótulos proposta em 4.2.2. Sendo $\beta_i \in (N \cup \Sigma) \cup \{"\varepsilon"\}$, onde $1 \leq i \leq n$, a construção dos rótulos é feita de acordo com as regras abaixo:

a) Recursões à esquerda

As construções identificadas como recursivas à esquerda:

$P \rightarrow \{A\}\ P\ \beta_1 \beta_2 \ldots \beta_n$

Serão rotuladas da seguinte forma:

- a ação adaptativa {A} será copiada como seu próprio rótulo;
- o não-terminal P terá rótulo nulo (inexistente);
- cada um dos elementos β_i não será rotulado se β_i representar um não-terminal, e será rotulado como β_i caso represente um terminal
- o início do lado direito da regra de produção será rotulado com [;

- o final do lado direito da regra de produção será rotulado com], precedido do não-terminal que define a regra, sua seqüência e o indicador □.
- A expressão acima devidamente rotulada é representada por:

$$\begin{array}{lllllllllllllll} & & [& & \{A\} & & & & \beta_1' & & \beta_2' & & & \beta_n' P_i \square] \\ P & \rightarrow & \uparrow & \{A\} & \uparrow & P & \uparrow & \beta_1 & \uparrow & \beta_2 & \uparrow & \ldots & \beta_n & \uparrow & \end{array}$$

Notar que este caso difere do apresentado anteriormente, pois mantém a ação adaptativa como elemento da regra, e que não há a aplicação de qualquer ação adaptativa ao final da regra. Prova-se, no trabalho de Iwai(2000), que um mesmo efeito é obtido quando a ação adaptativa posterior é referenciada no início da regra.

b) Recursões à direita

As construções identificadas como recursivas à direita:

$P \rightarrow \{A\}\ \beta_1 \beta_2 \ldots \beta_n\ P$

Serão rotuladas da seguinte forma:

- a ação adaptativa {A} será copiada como seu próprio rótulo;
- o não-terminal P terá rótulo nulo (inexistente);
- cada um dos elementos β_i não será rotulado se β_i representar um não-terminal, e será rotulado como β_i caso represente um terminal
- o início do lado direito da regra de produção será rotulado com um colchete [;
- o final do lado direito da regra de produção será rotulado com um colchete], precedido do não-terminal que define a regra, sua seqüência e o indicador □.
- A expressão acima devidamente rotulada é representada por:

$$\begin{array}{lllllllllllllll} & & [& & \{A\} & & \beta_1' & & \beta_2' & & & & \beta_n' & & P_i \square] \\ P & \rightarrow & \uparrow & \{A\} & \uparrow & \beta_1 & \uparrow & \beta_2 & \uparrow & \ldots & \beta_n & \uparrow & & P & \uparrow \end{array}$$

Permanecem as observações do item a) .

c) Outros casos:

As construções identificadas como *demais casos* recaem em uma das alternativas abaixo indicadas:

c.1) $P \rightarrow \{A\}\ \beta_1 \beta_2 \ldots \beta_n$

- Os elementos são rotulados conforme descrito anteriormente
- o final do lado direito da regra de produção será rotulado com um colchete], precedido do não-terminal que define a regra, sua seqüência e o indicador □ .
- A expressão acima devidamente rotulada é representada por:

	[		{A}		β_1'		β_2'			β_n' P_i □]
P	→	↑	{A}	↑	β_1	↑	β_2	↑	...	β_n	↑

c.2) $P \rightarrow \{A\}\ \varepsilon$

A expressão acima devidamente rotulada é representada por:

	[		{A}		εP_i □]	
P	→	↑	{A}	↑	ε	↑

c.3) $P \rightarrow$ □

Representa-se por:

	[		P_i □]	
P	→	↑	□	↑

c.4) sendo $\alpha, \beta \in V_C$ (símbolos de contexto), e E um não-terminal, consideram-se as seguintes possibilidades:

c.4.1) $P \rightarrow \{A\}\ \alpha\ E$

Neste caso considera-se o empilhamento de α, e rotula-se da seguinte forma:

	[		{A}		□α		P_i □]	
P	→	↑	{A}	↑	□α	↑	E	↑

c.4.2) $\alpha P \rightarrow \{A\}\ E$

Neste caso considera-se o desempilhamento de α, e rotula-se da seguinte forma:

[{A} □α P_i □]

αP → ↑ {A} ↑ □α ↑ E ↑

c.4.3) αP → {A} β E

Neste caso considera-se o desempilhamento de α e o empilhamento de β, e rotula-se da seguinte forma:

[{A} □α □ β P_i □]

αP → ↑ {A} ↑ □α ↑ □ β ↑ E ↑

c.4.4) P ← {A} α E

Neste caso considera-se o empilhamento de α, de forma análoga ao caso (c.4.1), e rotula-se da seguinte forma:

[{A} □α P_i □]

P → ↑ {A} ↑ □α ↑ E ↑

3 Agrupamento de produções que definem um mesmo não-terminal

As produções que contêm mais de uma alternativa para cada um dos tipos de recursão apontados devem ser agrupadas de forma que uma única produção deverá existir para cada um deles. De forma análoga ao exposto em 4.2.3, e nomeando-se o lado direito de cada uma das regras de produção por μ_i, teremos os seguintes casos de agrupamento (IWAI, 2000):

a) Agrupamento de regras com recursão à esquerda

[$\{A_1\}$ P_1 □ $\{A_2\}$ P_2 □

P →↑ P ↑ (↑ $\{A_1\}$ ↑ μ_1 ↑ | ↑ $\{A_2\}$ ↑ μ_2 ↑ | ...

$\{A_k\}$ P_k □]

... | ↑ $\{A_k\}$ ↑ μ_k ↑) ↑

Notar que não há qualquer ação adaptativa posterior indicada explicitamente.

b) Agrupamento de regras com recursão à direita

$$P \rightarrow \uparrow [\uparrow \{A_1\} \uparrow \mu_1 \uparrow P_1 \square \uparrow | \uparrow \{A_2\} \uparrow \mu_2 \uparrow P_2 \square \uparrow | \ldots | \uparrow \{A_k\} \uparrow \mu_k \uparrow P_k \square \uparrow] \uparrow P \uparrow$$

c) Demais produções (intitula-se como *geral*)

$$P \rightarrow \uparrow [\uparrow \{A_1\} \uparrow \mu_1 \uparrow P_1 \square \uparrow | \uparrow \{A_2\} \uparrow \mu_2 \uparrow P_2 \square \uparrow | \ldots | \uparrow \{A_k\} \uparrow \mu_k \uparrow P_k \square \uparrow] \uparrow$$

d) Agrupamento de regras dependentes de contexto

d.1) Para regras do tipo:

$\alpha_i P \rightarrow \mu_i$

(lembrando que μ_i representa a cadeia completa do lado direito, incluindo empilhamentos e/ou desempilhamentos de símbolos de contexto).

obtém-se:

$$P \rightarrow \uparrow [\uparrow \{A_1\} \uparrow \square\alpha_1 \uparrow \mu_1 \uparrow P_1 \square \uparrow | \uparrow \{A_2\} \uparrow \square\alpha_2 \uparrow \mu_2 \uparrow P_2 \square \uparrow | \ldots | \uparrow \{A_k\} \uparrow \square\alpha_k \uparrow \mu_k \uparrow P_k \square \uparrow] \uparrow$$

Caso haja recursão à direita ou à esquerda para o caso acima indicado, prevalecem as regras anteriormente descritas, lembrando-se que permanecem os empilhamentos e/ou desempilhamentos referentes ao símbolos de contexto:

d.2) Agrupamento de regras com recursão à esquerda

Para regras do tipo:

$\alpha_i P \rightarrow P\mu_i$

Realiza-se o seguinte agrupamento:

[$\{A_1\}$ $\square\alpha_1$ P_1 $\square$ $\{A_2\}$ $\square\alpha_2$ P_2 $\square$

$P \rightarrow \uparrow P \uparrow (\uparrow \{A_1\} \uparrow \square\alpha_1 \uparrow \mu_1 \uparrow | \uparrow \{A_2\} \uparrow \square\alpha_2 \uparrow \mu_2 \uparrow | \ldots$

$\{A_k\}$ $\square\alpha_k$ P_k $\square$]

$\ldots | \uparrow \{A_k\} \uparrow \square\alpha_k \uparrow \mu_k \uparrow) \uparrow$

<u>d.3) Agrupamento de regras com recursão à direita</u>

Para regras do tipo:

$\alpha_i P \rightarrow \mu_i P$

Agrupa-se da seguinte forma:

[$\{A_1\}$ $\square\alpha_1$ P_1 $\square$ $\{A_2\}$ $\square\alpha_2$ P_2 $\square$

$P \rightarrow \uparrow (\uparrow \{A_1\} \uparrow \square\alpha_1 \uparrow \mu_1 \uparrow | \uparrow \{A_2\} \uparrow \square\alpha_2 \uparrow \mu_2 \uparrow | \ldots$

$\{A_k\}$ $\square\alpha_k$ P_k $\square$]

$\ldots | \uparrow \{A_k\} \uparrow \square\alpha_k \uparrow \mu_k \uparrow) \uparrow P \uparrow$

<u>d.4) Regras referenciadas por setas à esquerda</u>

Para regras do tipo:

$P \leftarrow P\mu_i$

Adotam-se as mesmas regras propostas para as regras que utilizam a seta direita "→".

4 Remoção das recursões à direita e à esquerda

Ocorrem de forma análoga ao descrito em 4.2.4, ou seja, a partir de

P → P e | d P | g, onde e, d e g representam seqüências de terminais e/ou não-terminais, inseridos em recursões à esquerda, à direita e geral, obtém-se uma única expressão:

```
      [                          {A1}     P1□              {A2}     P2□
P →↑ ( ↑ ( ↑ ε ↑ \ ↑ {A1} ↑     μ1 ↑     | ↑ {A2} ↑     μ2 ↑     | ...

                {Ak}     Pk□                  {A1}     P1□
... | ↑ {Ak} ↑     μk ↑     ) ↑ ( ↑ {A1} ↑     μ1 ↑

                {A2}     P2□                  {Ak}     Pk□
| ↑ {A2} ↑     μ2 ↑     | ... | ↑ {Ak} ↑     μk ↑     ) ↑

                     {A1}     P1□              {A2}     P2□
( ↑ ε ↑ \ ↑ {A1} ↑     μ1 ↑     | ↑ {A2} ↑     μ2 ↑     | ...

                  {Ak}     Pk□          ]
... | ↑ {Ak} ↑     μk ↑     ) ↑ ) ↑
```

5 Tratamento de não-determinismos

A possível eliminação de não-determinismos é obtida através de processo semelhante ao exposto em 4.2.5.

6 Construção do analisador sintático

A aplicação dos passos também é semelhante aos apresentados para a gramática com ações semânticas. A numeração de estados e a construção das transições do autômato de reconhecimento ocorre de forma idêntica, havendo pequena alteração na definição das regras de mapeamento, como pode ser observado na tabela abaixo.

Rótulo	**Saída**	**Pilha**
ε	( )	(nenhuma ação)
σ	(σ)	(nenhuma ação)
P_i□	P_i□)	(nenhuma ação)
P_i□	P_i□) π	↑ π
P_i□	(	↓) P_i□
[	[(	↓]
]	π]	↑ π]
□ α	□ α	
□ α	□ α	
{A}	executar a ação adaptativa A	

Tabela V – Mapeamento para gramática adaptativa

Os símbolos que aparecem na tabela podem representar terminais, não-terminais, ações adaptativas e símbolos de contexto. Para o tratamento da pilha de transdução, são utilizados os símbolos ↑, ↓, π. A seguir, apresenta-se uma breve explicação de cada um destes elementos:

ε ... representa a cadeia vazia;

σ ... define a ocorrência de um símbolo terminal;

P_i□ ... define a ocorrência de um não-terminal P_i associado a alguma regra recursiva à esquerda;

P_i□ ... define a ocorrência de um não-terminal P_i não associado a uma recursão;

P_i□ ... define a ocorrência de um não-terminal P_i associado a alguma regra recursiva à direita;

[ou] ... ocorrência de delimitadores utilizados na construção da árvore

□ α ... indica a inserção de informações de contexto na geração de uma sentença;

□ α ... indica a remoção de informações de contexto na geração de uma sentença;

{A}	...	indica uma ação adaptativa A;
↓	...	indica que todo símbolo a seguir deverá ser empilhado;
↑	...	indica o desempilhamento dos símbolos a seguir;
π	...	é um meta-caracter que simboliza a seqüência de todos os elementos da pilha de transdução compreendidas desde o seu topo até a ocorrência do símbolo anterior ao delimitador].

Através da aplicação da tabela acima, constrói-se o *parser* dependente de contexto. Porém, devem-se notar as ressalvas feitas em 7.2, para viabilizar a aplicação dos referidos métodos na ferramenta proposta.

8 - REFERÊNCIAS

AHO A.V.; ULLMAN, J.D. **The theory of parsing, translation and compiling.** Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1972. v.1.

AHO A. V.; ULLMAN, J.D. **The theory of parsing, translation and compiling.** Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1973. v.2.

ALLEN, J. **Natural language understanding.** 2.ed. California: The Benjamin Cummings Publishing Company, Inc., 1995. 654p.

APPEL, A. **Modern compiler implementation in C.** 1.ed. Cambridge, MA: The Press Syndicate of The University of Cambridge, 1997. 398p.

BASSETO, B.A.; NETO, J.J. A stochastic musical composer based on adaptive algorithms. In: CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 10., PUC-Rio, 1999. **SBC-99.** Rio: [s.n.], 1999. p.105-113.

BLACK, C.A. **A step-by-step introduction to the government and binding theory of syntax.** Mexico: Mexico Branch and University of North Dakota, 1998.

BARNES, B.H. A programmer's view of automata. **ACM Computing Surveys**, v.4, n.2, 1972.

BONFANTE, A.G.; NUNES, M.G.V.N. The implementation process of a statistical parser for brazilian portuguese, 2001. **IME-USP São Carlos**

BURSHTEYN, B. Generation and recognition of formal languages by modifiable grammars. **ACM SIGPLAN Notices**, v.25, n.12, p.45-53, 1990.

CHOMSKY, N. **Syntactic Structures**. 2ed., 1957, Walter de Gruyter Inc. 117p., ISBN: 3110172798

CONWAY, M.E. Design of a separable transition diagram compiler. **Communications of the ACM**, v.6, n.7, pp. 396-408, 1963.

COSTA, E.R.; HIRAKAWA, A.R., NETO, J.J. An adaptive alternative for syntactic pattern recognition. In: INTERNATIONAL SYMPOSIUM ON ROBOTICS AND AUTOMATION - ISRA 2002., 3rd ., Toluca, Mexico, Sept. 2002. **Proceedings.** Mexico: 2002, p.409-413.

FREITAS, A.V.; NETO, J.J. Uma ferramenta para construção de aplicações multilinguagens de programação. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN - CACIC 2001., El Calafate, Argentina, 2001. **Anais.** Argentina: 2001.

GRISHMAN, R. **Computational linguistics: an introduction.** Cambridge, MA: Cambridge University Press, 1986.

HOPCROFT, J. E.; ULLMAN, J. D. **Formal languages and their relations to automata.** New York, NY: Addison-Wesley, 1979a.

IWAI, M.K. **Um formalismo gramatical adaptativo para linguagens dependentes de contexto**. 2000. 191p. : Tese (Doutorado) – Escola Politécnica, Universidade de São Paulo. São Paulo, 2000.

LOMET, D.B. A formalization of transition diagram systems. **Journal of the ACM**, v.20, n.2, p. 235-257, 1973.

MENEZES, C.E.D.; NETO, J. J. Um método para a construção de analisadores morfológicos, aplicado à língua portuguesa, baseado em autômatos adaptativos. In: V PROPOR, ENCONTRO PARA O PROCESSAMENTO COMPUTACIONAL DE PORTUGUÊS ESCRITO E FALADO, Brasil, Atibaia, 19-22 Nov. 2000. **Anais.**

MENEZES, C.E.D.; NETO, J.J. Um método híbrido para a construção de etiquetadores morfológicos, aplicado à língua portuguesa, baseado em autômatos adaptativos. In: CONFERÊNCIA IBEROAMERICANA EN SISTEMAS, CIBERNÉTICA E INFORMÁTICA., Orlando, Florida, 19-21 Jul. 2002. **Anais**.

MENEZES, P.B. **Linguagens formais e autômatos.** 4.ed. Porto Alegre: Sagra Luzzato Editora, 2000. 165p.

NETO, J.J.;MAGALHÃES, M.E.S. Um gerador Automático de Reconhecedores Sintáticos para o SPD. In:VIII SEMISH, Florianópolis, 1981.**Anais**.

NETO, J.J. **Introdução à compilação.** Rio de Janeiro: Livros Técnicos e Científicos Editora S.A., 1987. 214p.

NETO, J.J. Uma Solução Adaptativa para Reconhecedores Sintáticos. **Anais** da Escola Politécnica, Departamento de Engenharia de Eletricidade, 1988b.

NETO, J.J.; KOMATSU, W. Compilador de gramáticas descritas em notação de wirth modificada. **Anais EPUSP**, engenharia de eletricidade, Série B., vol. 1, p.477-518, 1988

NETO, J.J. **Contribuições à metodologia de construção de compiladores.** 1993. 272p. : Tese (Livre-Docência) - Escola Politécnica, Universidade de São Paulo. São Paulo, 1993.

NETO, J.J. Adaptive automata for context-sensitive languages. **SIGPLAN NOTICES**, New York, v. 29, n.9, p. 115-124, Sept. 1994.

NETO, J.J.; IWAI, M.K. Adaptive automata for syntax learning. In: Conferencia Latinoamericana de Informatica - CLEI 98, Quito, Equador, 1998. **Memorias.** Quito. p. 135-149.

NETO, J.J.; PARIENTE, C.B.; LEONARDI, F., Compiler Construction – A Pedagogical Approach. **ICIE**, 1999

NETO, J.J. Adaptive rule-driven devices - general formulation and case study. In: INTERNATIONAL CONFERENCE ON IMPLEMENTATION AND APPLICATION OF AUTOMATA – CIAA 2000, 5th., London, Ontario, Canada, July 2000. **Lecture Notes in Computer Science 2088.** Berlin: Springer-Verlag, 2001. p.340-342.

NETO, J.J.; PARIENTE, C.A.B. Adaptive automata - a revisited proposal. In: INTERNATIONAL CONFERENCE ON IMPLEMENTATION AND APPLICATION OF AUTOMATA – CIAA 2002, 7th., Tours, France, Jul. 2002. **Lecture Notes in Computer Science 2608.** Berlin: Springer-Verlag, 2002. p.158-168

PAGAN, F.G. **Formal Definition of Programming Languages**, Prentice Hall, 1981

PAPADIMITRIOU, C.H., LEWIS, H.R. **Elements of the theory of computation.** 2.ed. New Jersey: Prentice-Hall, Inc., 1998, 351p.

PEREIRA, J.C.D.; NETO, J.J. Um ambiente de desenvolvimento de reconhecedores sintáticos baseado em autômatos adaptativos. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO – SBLP97, 2., Campinas, São Paulo, 1997. **Anais.** São Paulo: SBLP, 1997. p.139-150.

PEREIRA, J.C.D. **Ambiente integrado de desenvolvimento de reconhecedores sintáticos baseado em autômatos adaptativos.** 1999. 162p. : Dissertação (Mestrado) - Escola Politécnica, Universidade de São Paulo. São Paulo, 1999.

PISTORI, H. ; NETO, J.J. Adaptree - Proposta de um algoritmo para indução de árvores de decisão baseado em técnicas adaptativas. In: CONFERÊNCIA LATINO AMERICANA DE INFORMÁTICA – CLEI 2002., Montevideo, Uruguai, Novembro 2002. **Anais.** Montevideo, 2002.

PISTORI, H.; NETO, J.J. A free software for the development of adaptive automata. In: INTERNATIONAL FORUM ON FREE SOFTWARE - WORKSHOP ON FREE SOFTWARE – WSL, 4th, Porto Alegre, Brasil, June 5th-7th, 2003a. **Anais.** Porto Alegre, 2003.

PISTORI, H**. Tecnologia adaptativa em engenharia de computação: estado da arte e aplicações.** 2003. 172p. : Tese (Doutorado) – Escola Politécnica, Universidade de São Paulo. São Paulo, 2003.

PISTORI, H. ; NETO, J.J.; COSTA, E.R. Utilização de tecnologia adaptativa na detecção da direção do olhar. **SPC Magazine**, v.2., n.2, p.22-29, 2003.

PRICE, A. M.A.; TOSCANI, S.S. **Implementação de linguagens de programação: compiladores.** 2.ed. Porto Alegre: Editora Sagra Luzzatto, 2004.

RICCHETTI, P.M.; NETO, J.J. A practical method for the implementation of syntactic parsers. In: WSEAS INTERNATIONAL CONFERENCE ON AUTOMATION & INFORMATION - ICAI'05, 6th , Buenos Aires, Argentina, March 1st-3th , 2005. **Proceedings**. Buenos Aires, 2005a.

RICCHETTI, P.M.; NETO, J.J. A practical method for the implementation of syntactic parsers. In: WSEAS INTERNATIONAL CONFERENCE ON AUTOMATION & INFORMATION - ICAI'05, 6th , Buenos Aires, Argentina, March 1st-3th , 2005. **WSEAS Transactions.** Buenos Aires, 2005b.

ROCHA, R.L.A.; NETO, J.J. Uma proposta de linguagem de programação funcional com características adaptativas. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACION, 9., La Plata, Argentina, 6-10 de Outubro, 2003. **Anais.** La Plata, Argentina: 2003.

TREMBLAY, J.P.; SORENSON, P.G. **The theory and practice of compiler writing.**, New York, NY: McGraw-Hill, 1985.

WIRTH, NIKLAUS. **Compiler construction.** 1sted. Addison Wesley Longman Limited, 1996. 172p.