**UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC**
**CENTRO TECNOLÓGICO - CTC**
**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE**
**BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO**

# GALS
# GERADOR DE ANALISADORES LÉXICOS E SINTÁTICOS

**Autor:** Carlos Eduardo Gesser
**Orientador:** Olinto José Varela Furtado
**Banca Examinadora:** Patrícia Vilain e José Eduardo de Lucca

**Palavras Chave:** Linguagens Formais, Compiladores, Gramáticas Regulares, Expressões Regulares, Autômatos Finitos, Gramáticas Livres de Contexto, Autômatos de Pilha, Analise Léxica, Análise Sintática.

Florianópolis, fevereiro de 2003

## Sumário

## Resumo

GALS é uma ferramenta para geração automática de analisadores léxicos e sintáticos, duas importantes fases do processo de compilação. Foi desenvolvida para ser uma ferramenta com propósitos didáticos, mas com características profissionais, podendo ser utilizada tanto no auxílio aos alunos da cadeira de Construção de Compiladores como possivelmente em outros projetos que necessitem processamento de linguagens.

## Abstract

GALS is a tool for automatic generation of lexical and sintatical analysers, two important phases of the compiling process. It was developed to be a tool with didactic pourposes, but with professional caracteristics, as it can be used to help compilers theory students and there is the possibility to be used any projects witch requires language processing.



## Palavras Chave:

Linguagens Formais, Compiladores, Gramáticas Regulares, Expressões Regulares, Autômatos Finitos, Gramáticas Livres de Contexto, Autômatos de Pilha, Analise Léxica, Análise Sintática.

# Introdução:

Analisadores Léxicos e Sintáticos são importantes partes de um compilador. Sua construção sem a utilização de ferramentas de auxílio pode levar a erros no projeto, além de ser um trabalho desnecessário.

Existem hoje diversas ferramentas comerciais semelhantes, porém a maioria delas gera apenas ou o Analisador Léxico ou o Sintático, o que obriga o usuário a conhecer duas ferramentas distintas e leva a incompatibilidades no momento da utilização em conjunto dos analisadores gerados. Muitas ferramentas geram analisadores segundo uma única técnica de análise sintática., o que limita seu valor didático. Praticamente todas geram analisadores em apenas uma linguagem objeto, diminuindo sua flexibilidade. Outro fator agravante é o fato de serem ferramentas de código fechado, o que impede que se possa provar formalmente o analisador gerado, já que não se tem acesso a seu código fonte.

Também existem ferramentas livres (gratuitas e de código aberto) para esta tarefa. A maioria apresenta as mesmas limitações citadas acima. Mas, por se tratarem de *software*, livre têm a vantagem de poderem ter seu código-fonte analisado, verificando-se assim se o que fazem é realmente aquilo que se propõe a fazer.

O GALS tenta superar estes problemas e se tornar uma ferramenta prática e versátil, com aplicabilidade tanto como ferramenta didática como para uso profissional, promovendo assim o uso das técnicas formais de análise.

# 1 – Fundamentação Teórica

## 1.1 – Teoria das Linguagens

### *1.1.1 – Símbolo*

Os símbolos de uma linguagem são os elementos mínimos que a compõe. Em linguagens humanas, são as palavras, em linguagens de máquina (*assembly*) são seus mnemônicos.

### *1.1.2 – Sentença*

Uma sentença (ou cadeia, ou *string*) é uma seqüência ordenada de símbolos de uma linguagem.

O tamanho de uma sentença é o número de símbolos que ela possui. Existe um caso especial de sentença, de tamanho zero, chamada de sentença vazia e denotada por 'ε', que é muito utilizada na teoria de linguagens, bem como durante a análise e processamento destas.

### *1.1.3 – Alfabeto*

Um alfabeto é um conjunto de símbolos, como o alfabeto romano: R = {a, b, c, ..., z}, ou o alfabeto grego: G = {α β, χ, ..., ω}.

O fechamento de um alfabeto, representado por '*', é definido como o conjunto de todas as sentenças possíveis de serem geradas a partir deste alfabeto, inclusive a sentença vazia. Assim, o fechamento do alfabeto romano, denotado por R*, seria: R* = {ε, a, aa, aaa, ..., b, bb, ..., ab, abc, ...}

Existe ainda o fechamento positivo ('$^+$'), que também é definido como o conjunto de todas as sentenças possíveis em um alfabeto, mas não incluindo a sentença vazia. Ou seja: $R^+$ = R* - {ε}.

### *1.1.4 – Linguagem*

Uma linguagem sobre um alfabeto é um conjunto de sentenças formadas a partir deste alfabeto. Trata-se de um subconjunto de seu conjunto fechamento.

As linguagens podem ser definidas, de maneira formal, através da enumeração de suas sentenças (mas isto apenas no caso de linguagens finitas), ou por meio de gramáticas, construções

mais sofisticadas, onde é possível representar de maneira finita, linguagens infinitas. As gramáticas também são a base para o reconhecimento de linguagens, isto é, analisar uma sentença e dizer se ela pertence ou não à linguagem.

### *1.1.5 – Gramática*

Gramáticas são utilizadas para representar uma linguagem. Cada gramática representa uma única linguagem, mas em geral, uma linguagem pode ser representada através de diversas gramáticas diferentes.

Uma gramática é um conjunto de Símbolos Terminais, que são exatamente os símbolos da linguagem representada, Símbolos Não-Terminais, que não fazem parte da linguagem, mas são utilizados na definição de Produções desta gramática, que dão a estrutura da linguagem.

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

- N é o conjunto dos símbolos não-terminais
- T é o conjunto dos símbolos terminais
- P é o conjunto das produções
- S é o símbolo inicial da gramática.

As produções são definidas por X ::= Y, onde $X \in (N \cup T)^+$, e $Y \in (N \cup T)^*$.

A seguir será dado um exemplo de gramática simples, para a linguagem dos números naturais maiores que zero:

N = { <número>, <digito>, <não zero> }

T = { 1, 2, 3, 4, 5, 6, 7 ,8 ,9 ,0 }

P = {

```
      <número>   ::= <número> <dígito>
                   | <não zero>
      <dígito>   ::= <não zero>
                   | 0
      <não zero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

}

S = <número>

Uma derivação em uma sentença é a substituição em uma parte desta por um outro conjunto de símbolos, de acordo com suas produções. A forma como as derivações podem ocorrer em uma gramática é definida por suas produções.

No exemplo anterior, para a sentença: 1 2 `<dígito>` 5, uma possível derivação seria 1 2 **0** 5. Pois existe uma produção que indica que é possível substituir o não terminal `<dígito>` pelo terminal 0.

### *1.1.6 – Classificação das Gramáticas*

As gramáticas foram agrupadas em quatro classes ou tipos, para facilitar seu estudo. Esta classificação foi feita por Chomsky, de acordo com capacidade representativa das gramáticas.

- Tipo 0 (Gramática sem Restrições): Esta classe engloba todas as gramáticas possíveis.
- Tipo 1 (Gramáticas Sensíveis ao Contexto): Possui a seguinte restrição:

  Para toda produção X ::= Y pertencente à gramática, $|X| \leq |Y|$ (o tamanho de X é menor que o tamanho de Y)
- Tipo 2 (Gramáticas Livres de Contexto): Possui a restrição das gramáticas tipo 1, e suas produções devem ser da forma A ::= Y, com $A \in N$, e $Y \in (N \cup T)^*$
- Tipo 3 (Gramáticas Regulares): Além das restrições das gramáticas tipo2, as tipo 3 só podem ter produções das formas:
  - A ::= aB, com A e B $\in$ N, e a $\in$ T, ou
  - A ::= a, com $A \in N$, e a $\in$ T.

Vale a pena notar que em gramáticas tipo 2 e 3, produções do tipo A ::= ε são permitidas, mesmo que elas quebrem a regra das gramáticas tipo 1 (o tamanho de ε é 0, que é menor que o tamanho do lado esquerdo da produção). Isto é possível graças a um teorema que prova que para toda gramática tipo 2 ou tipo 3 com este tipo de produção, pode-se escrever uma equivalente sem este tipo de produção.

### *1.1.7 – Classificação das Linguagens*

Assim como as gramáticas, as linguagens também estão agrupadas em quatro grupos, de acordo com a gramática que Pode representa-la. As linguagens do tipo 0 são aquelas que podem ser representadas por gramáticas do tipo 0. Linguagens do tipo 1, por gramáticas do tipo 1, o mesmo para os tipos 2 e 3.

# 1.2 – Compiladores

## *1.2.1 – Definições*

### 1.2.1.1 – Tradutor

Um tradutor é um mecanismo que faz a conversão de um texto em uma linguagem para uma outra linguagem, sem perder o sentido original. Um exemplo são os tradutores automáticos, que fazem a tradução (aproximada) de textos quaisquer em linguagens humanas (Português, Inglês, etc.).

### 1.2.1.2 – Compilador

Trata-se essencialmente de um Tradutor, mas neste caso, o texto a ser processado é um programa em uma linguagem de programação. O trabalho de um compilador é converter um programa em uma linguagem de alto nível em um programa em uma linguagem de baixo nível, ou nível intermediário.

### 1.2.1.3 – Montador (*Assembler*)

O Montador é o mais simples dos Tradutores. Ele converte um programa em linguagem de baixo nível (*assembly*) em linguagem de máquina.

### 1.2.1.4 – Interpretador

Também conhecido como Máquina Virtual, ele executa um programa em linguagem intermediária, e obtém resultados, simulando uma máquina real.

Ex: Máquina Virtual Java

### 1.2.1.5 – Tradutor/Interpretador

Combina as funções de um Tradutor e um Interpretador: à medida que vai traduzindo o programa, já o interpreta, obtendo seus resultados.

### 1.2.1.6 – Pré-Processador

Um caso de Tradutor onde as linguagens do texto de entrada e saída são do mesmo nível (em geral de alto nível). Ele executa pequenas modificações no texto, tais como expansão de Macros, inserção do conteúdo de outros arquivos de texto ao original, processamento primário, através de diretivas, passagem de opções ao compilador, etc.

C é uma linguagem que faz uso extensivo de um pré-processador. Outro caso é o dos primeiros compiladores de C++, que se tratavam na verdade de um pré-processador, que lia o programa C++ e gerava um programa C equivalente.

## *1.2.2 – O Processo de Compilação*

### 1.2.2.1 – Linguagens de Programação

As linguagens de programação de alto nível permitem ao programador dar instruções a uma máquina de maneira mais simples e prática do que diretamente na linguagem de máquina. Porém os processadores não entendam programas escritos em linguagens de programação de alto nível, é aí que entra o processo de compilação.

As linguagens, dos mais diversos tipos e paradigmas de programação são analisadas, em busca de erros, e informações para que em seguida seja gerado um programa em linguagem de máquina, que é o objetivo do processo.

A compilação de um programa é dividida em duas etapas: Análise e Síntese.

### 1.2.2.2 – Análise

Na etapa de análise, o texto do programa é examinado, e informações sobre ele são extraídas e utilizadas no processo, e a fim de se preencher as estruturas de dados necessárias à etapa de síntese. Nesta etapa, a maioria dos erros que poderiam estar presentes no programa são detectados, sendo possível abortar o processo de compilação, ou partir para um modo de recuperação de erros onde se tenta continuar a compilação para que se possa informar o maior número de erros simultaneamente ao usuário. Esta etapa é subdividida em três fazes de análise: Léxica, Sintática e Semântica, cada uma delas trata de um aspecto do programa fonte.

- **Análise Léxica**: os aspectos léxicos do programa são tratados nesta fase. Aqui entra em cena o analisador léxico, também conhecido como *scanner*. Ele lê o texto do programa e o divide em *tokens*, que são os símbolos básicos de uma linguagem de

programação. Eles representam palavras reservadas, identificadores, literais numéricos e de texto, operadores e pontuações. Comentários no texto do programa geralmente são ignorados pelo analisador.

Os analisadores léxicos serão vistos com mais detalhe posteriormente.

- **Análise Sintática**: etapa realizada pelo analisador sintático, ou *parser*. Este analisador trabalha lendo os *tokens* gerados pelo analisador léxico e os agrupando de acordo com a estrutura sintática da linguagem. O resultado final é chamado de árvore de derivação.

  Os analisadores sintáticos serão vistos com mais detalhe posteriormente.

- **Análise Semântica**: aspectos semânticos são tratados aqui. Estes aspectos levam em consideração o sentido do programa. É através da semântica que são extraídas as informações que possibilitam a posterior geração de código.

  Os aspectos semânticos não são facilmente especificáveis. Os principais mecanismos formais para a especificação dos aspectos semânticos são as gramáticas de atributos, as semânticas denotacionais e as semânticas de ações. Porém sua complexidade os torna inviáveis na prática, e acaba-se partindo para mecanismos semiformais, ou até mesmo informais.

  Uma implementação muito comum é a utilização de ações semânticas, que são inseridas no meio da especificação sintática sob a forma de sub-rotinas que são invocadas quando o *parser* as atinge.

Durante a etapa de análise, é montada a tabela de símbolos, que mantém informações sobre todos os símbolos do programa, e é utilizada também na geração de código.

### 1.2.2.3 – Síntese

Na etapa de síntese é que é gerado o programa objeto. Nesta etapa também ocorre o processo de otimização, que elimina partes inúteis e redundantes que são geradas durante a geração intermediária de código, e mesmo aquelas introduzidas pelo programador.

#### 1.2.2.4 – Tradução Dirigida pela Sintaxe

Este é um dos esquemas de compilação mais comum. O núcleo do compilador aqui é o Analisador Sintático. Ele fica acionando o Analisador Léxico, fazendo com que ele gere *tokens* para serem consumidos pelo sintático. Dentro da especificação sintática, estão inseridas ações semânticas, que são acionadas no momento que são encontradas.

A geração de código fica embutida no meio das ações semânticas. O código objeto vai sendo gerado a medida que o programa é processado.

#### 1.2.2.5 – Passos de Compilação

Um compilador implementado com o esquema anterior é chamado de Compilador de Um Passo, pois ele lê o arquivo fonte apenas uma vez. Porém pode ser interessante que existam etapas intermediárias na compilação. O léxico poderia ler o arquivo fonte e gerar um arquivo com os *tokens*, o sintático leria este arquivo e geraria um arquivo com a árvore de derivação total. Este arquivo seria a entrada para o analisador semântico, e assim por diante. Neste caso tem-se um compilador de vários passos.

O compilador *gcc* da GNU faz uso dessa técnica para promover sua portabilidade. Ele interrompe a compilação em um estágio intermediário, antes de gerar o código específico para a máquina alvo, gerando um arquivo com código intermediário. Então o restante da compilação (otimização e geração de código final) é efetuada por um módulo que é específico para cada plataforma.

### *1.2.3 – Analisador Léxico*

O analisador léxico faz a verificação do programa quanto aos aspectos léxicos. Estes aspectos são definidos por Gramáticas Regulares (Tipo 3), ou outro mecanismo equivalente, como expressões regulares e autômatos finitos , que serão abordadas mais tarde. Analisadores léxicos são implementados com base em Autômatos Finitos, que são o mecanismo formal de reconhecimento de linguagens regulares.

### 1.2.3.1 – Autômatos Finitos

Um autômato finito trabalha lendo uma *string* de símbolos e alterando seu estado interno de acordo com um controle. Se ao terminar a *string* o estado atual for um estado considerado final, o autômato aceita a entrada, senão rejeita.

Formalmente um autômato é uma Quíntupla: $F = (K, \Sigma, \delta, q_0, F)$, onde:

K – conjunto de estados do autômato;

$\Sigma$ – o alfabeto de entrada do autômato;

$\delta$ – função de $K \times \Sigma \rightarrow K$, que controla as transições entre os estados;

$q_0$ – estado inicial do autômato;

F – conjunto de estados finais.

O funcionamento de um autômato finito segue o seguinte algoritmo:

1. T ← q0
2. Se a entrada possui algum símbolo ainda, então
    a. A ← próximo símbolo da entrada
3. Senão
    a. Se T ∈ F então
        i. Aceita
    b. Senão
        i. Rejeita
4. Se (T, A) ∈ domínio de δ então
    a. T ← δ(T, A)
    b. Vá para 2
5. Senão
    a. Rejeita.

### 1.2.3.2 – Representações de Autômatos Finitos

Autômatos finitos podem ser representados por diagramas de transições, que são um grafo dirigido e rotulado, os vértices representam os estados e as arestas as transições. O estado inicial possui uma seta que o indica, e os finais possuem círculos duplos.

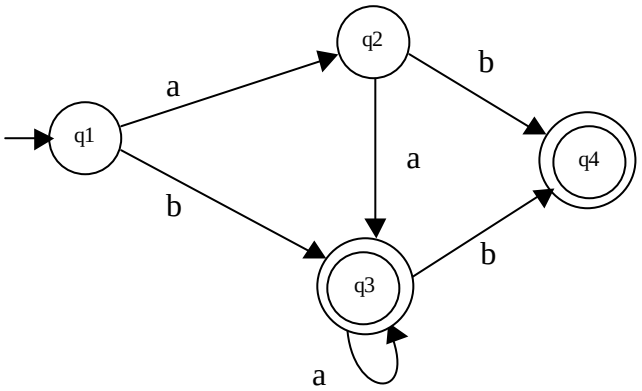



**Figura 1 - Autômato Finito**

Uma outra representação possível é através de uma tabela de transições. As colunas são rotuladas pelos símbolos do alfabeto de entrada, e as linhas pelos estados. A intersecção de uma linha e uma coluna indica o próximo estado. O estado inicial é indicado por uma seta, e os finais por um asterisco (*).

| δ | a | b |
|---|---|---|
| → q1 | q2 | q3 |
| q2 | q3 | q4 |
| * q3 | q3 | q4 |
| * q4 | - | - |

### 1.2.3.3 – Determinismo x Não-Determinismo

Se num autômato finito houver algum conflito, no caso de duas ou mais transições diferentes no mesmo estado com o mesmo símbolo na entrada, este autômato é dito não-determinístico.

A introdução desta característica não altera o poder de decisão dos autômatos finitos, mas algumas linguagens são mais “naturalmente” representadas por autômatos não-determinísticos. Porém o algoritmo para processar estes autômatos é muito mais complexo, o que desencoraja sua aplicação.

Os autômatos finitos determinísticos e os não-determinísticos são equivalentes. O seguinte algoritmo faz a conversão de um autômato finito não-determinístico $M = (K, \Sigma, \delta, q_0, F)$ em um autômato finito determinístico $M' = (K', \Sigma', \delta', q_0', F')$:

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

2 - $q_0' = [q_0]$.

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

4 - $\Sigma' = \Sigma$.

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

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

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

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

...

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

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

### 1.2.3.4 – Minimização

Um autômato finito é dito mínimo se:

1. Não possui estados INACESSÍVEIS, ou seja, estados que nunca são alcançáveis a partir do estado inicial;
2. Não possui estados MORTOS, ou seja, estados que não geram nenhum símbolo terminal;
3. Não possui estados EQUIVALENTES, que são estados que estão na mesma classe de equivalência.

**Classe de Equivalência:**

Um conjunto de estados $q_1, q_2, ..., q_j$ estão em uma mesma classe de equivalência se $\delta(q_1, a)$, $\delta(q_2, a)$,..., $\delta(q_j, a)$, para cada $a \in \Sigma$, resultam respectivamente nos estados $q_i, q_{i+1}, ..., q_n$ e estes pertencem a uma mesma classe de equivalência.

**Algoritmo para Construção das Classes de Equivalência**

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

**Algoritmo para Construção do Autômato Finito Mínimo**

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

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

Método:

1 - Elimine os estados inacessíveis.

2 - Elimine os estados mortos.

3 - Construa todas as possíveis classes de equivalência.

4 - Construa M' como segue:

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

b) $q_0' \rightarrow$ será a classes de equivalência que contiver $q_0$.

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

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

### 1.2.3.5 – ε-Transições

ε-transições são transições rotuladas por ε. A semântica dessas transições é de que o estado se altera, sem que seja consumido um símbolo da entrada.

Assim como o não-determinismo, as ε-transições não alteram o poder de decisão dos autômatos finitos, sendo os autômatos finitos com ε-transições equivalentes aos autômatos finitos livres de ε-transições.

### 1.2.3.6 – Expressões Regulares

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

a. φ é uma expressão que denota o conjunto vazio.
b. ε é uma expressão regular que denota o conjunto {ε}.
c. Para cada símbolo em a em A, a é uma expressão regular que denota o conjunto {a}.
d. Se r e s são expressões regulares que denotam os conjuntos R e S, respectivamente, então (r|s), (rs), e ($r^*$) são expressões regulares que denotam os conjuntos R∪S, RS e $R^*$, respectivamente.

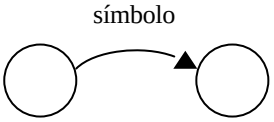
Na pratica, quando se escreve expressões regulares, pode-se omitir muitos parênteses se for assumido que o símbolo * tem precedência maior que a concatenação e |, e que a concatenação tem precedência maior que |.

Ex: ab*, ((abc)|(cba))*, aaa(a|a*)aaa.

### 1.2.3.7 – Equivalência entre Expressões Regulares e Autômatos Finitos

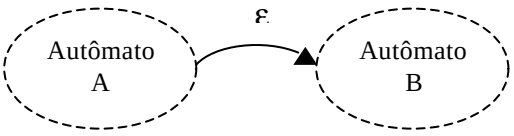
Pode-se facilmente obter um autômato finito não-determinístico a partir de uma expressão regular. A idéia principal é gerar uma série de pequenos autômatos, que são unidos via ε-transições.

Cada um dos símbolos da expressão dá origem a um autômato da seguinte forma:

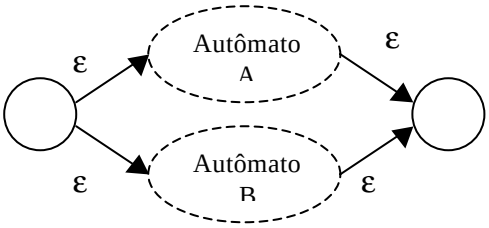



A partir daí, de acordo com a operação utilizada, os autômatos são unidos.

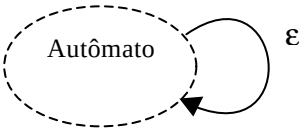
Concatenação:



União:




Fechamento




### 1.2.3.8 – Autômatos Finitos na Analise Léxica

A implementação utilizada em geral é utilizar a tabela de transições do autômato finito e um algoritmo que faz a análise léxica percorrendo a sentença e consultando a tabela. Uma outra possibilidade ainda é a de se programar o autômato diretamente, sabendo-se o estado atual, e o próximo símbolo da sentença, pode-se avançar para outro estado, prosseguindo-se dessa forma até acabar a sentença, ou não ser possível fazer a transição de estados.

O analisador léxico deve identificar os diferentes *tokens* da linguagem, e devolver ao analisador sintático um código para este *token.* É preciso então algum mecanismo para que o autômato finito faça isto. Geralmente é feito com que o autômato tenha vários pontos de saída, um para cada *token* diferente.

Para as palavras reservadas da lnguagem também é utilizada uma técnica diferente: sempre que um *token* de **identificador** é detectado, faz-se uma busca numa tabela por essa cadeia, se for encontrada, é retornado o *token* equivalente, senão, é retornado o *token* de identificador mesmo.

Na prática, o que se utiliza são geradores de analisadores léxicos, que são ferramentas que produzem automaticamente analisadores léxicos, normalmente a partir de uma especificação baseada em gramáticas regulares, expressões regulares ou autômatos finitos.

### *1.2.4 – Analisador Sintático*

O analisador sintático faz a verificação do programa quanto aos aspectos sintáticos. Estes aspectos são definidos por Gramáticas Livres de Contexto (Tipo 2). O mecanismo formal para o reconhecimento desta classe de gramáticas é o Autômato de Pilha. Existem duas estratégias para a implementação de analisadores sintáticos, que serão abordadas mais a fundo nas próximas seções. Estas estratégias são: a *Top-Down*, ou Descendente, e a *Bottom-Up*, ou Ascendente.

#### 1.2.4.1 – Autômatos de Pilha

O autômato de pilha é uma extensão do autômato finito onde existe uma pilha para manter o controle extra que é necessário para o gerenciamento de linguagens livres de contexto. Um autômato de pilha é uma Séptupla $P = (K, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, onde:

K - conjunto de estados do autômato;
$\Sigma$ - alfabeto de entrada;
$\Gamma$ - alfabeto da pilha;
$\delta$ - função de $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \rightarrow (K \times \Gamma^*)$ que controla as transições de estado e a pilha
$q_0$ - estado inicial do autômato;
$Z_0$ - símbolo inicial da pilha;
F – Conjunto de símbolos finais.

Os movimentos do autômato de pilha se dão através da função de mapeamento. A partir do estado atual, do próximo símbolo da entrada, e do topo da pilha, se obtém o próximo estado e os símbolos que serão empilhados no lugar do anterior. O símbolo da entrada é consumido no processo. No caso de a transição possuir $\varepsilon$ como símbolo da entrada, esta transição pode ocorrer qualquer que seja o símbolo da entrada, e ele não será consumido.

A notação gráfica dos autômatos de pilha é parecida com a notação dos autômatos finitos. O que muda são os rótulos das transições. Ao invés de apenas indicar o símbolo da entrada, é necessário indicar também o símbolo que deve estar no topo da pilha e a transformação a se fazer na pilha.

### 1.2.4.2 – Gramáticas Livres de Contexto e Arvore de Derivação

As gramáticas livres de contexto são a classe de gramáticas usadas para descrever os aspectos sintáticos das linguagens de programação em geral. A árvore de derivação é a representação gráfica da derivação de uma sentença a partir da gramática. Os nodos internos representam os símbolos não terminais da gramática, e as folhas os símbolos terminais, os *tokens* da linguagem.

### 1.2.4.3 – Ambigüidade

A ambigüidade ocorre quando uma mesma sentença possui duas ou mais árvores de derivação, ou equivalentemente, pode ser obtida a partir de duas ou mais derivações distintas. Esta característica deve ser evitada, pois o processo formal de compilação, através das técnicas usuais, exige que o programa seja gerado de maneira única, não podendo haver dupla interpretação na intenção do programador.

Muitas vezes a ambigüidade surge do modo como a gramática foi escrita, podendo ser removida se reescrevendo a gramática de outra maneira. Porém podem ocorrer casos de construções que são inerentemente ambíguas, não importando a forma com que se escreva a gramática.

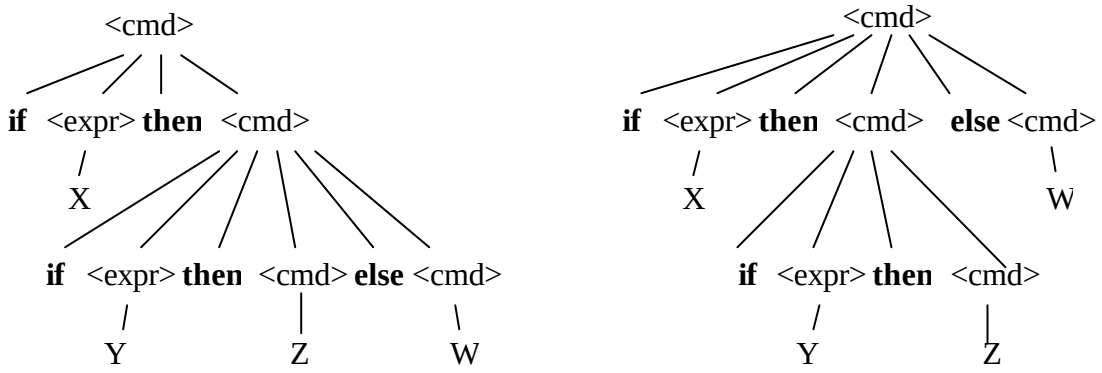
Um dos exemplos mais conhecidos é o do comando **if-then-else** das linguagens derivadas do ALGOL. Uma gramática livre de contexto para representa-lo seria:

<cmd> : : = **if** <expr> **then** <cmd>
<cmd> : : = **if** <expr> **then** <cmd> **else** <cmd>
<cmd> : : = “qualquer outro comando”
<expr> : : = “qualquer expressão”

O problema surge quando se tenta analisar esta sentença:

**if** X **then if** Y **then** Z **else** W

O ultimo else pertence ao if mais interno ou ao mais externo? Mostrando as duas possíveis árvores de derivação o problema torna-se mais aparente.




Neste caso especial, a solução normalmente usada é optar-se pela primeira árvore, onde o **else** sempre é relativo ao **if** mais próximo, facilitando assim a compreensão do programa.

Reconhecedores de gramáticas livres de contexto têm problemas com linguagens cuja gramática seja ambígua. O modo como ele é resolvido será visto mais tarde.

### 1.2.4.4 – Transformações em Gramáticas Livres de Contexto

Existe uma série de transformações que se pode fazer em gramáticas livres de contexto. Estas transformações não alteram a linguagem aceita pela gramática, mas retiram alguma característica indesejada, ou adicionam alguma característica que seja interessante, tudo dependendo da aplicação.

## 1.2.4.4.1 – Simplificação de Gramáticas Livres de Contexto

Pode-se diminuir o tamanho de uma gramática livre de contexto eliminando-se os símbolos que sejam inúteis na gramática. Existem dois tipos de símbolos inúteis, os símbolos improdutíveis, que são aqueles que não geram nenhum terminal, e os inalcançáveis, que são aqueles que não são deriváveis a partir do símbolo inicial.

Para se eliminar os símbolos inúteis, primeiro é preciso eliminar os símbolos improdutíveis, e depois os inalcançáveis.

**Algoritmo: Eliminação de Símbolos Improdutivos**

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

SP := T $\cup$ {$\varepsilon$}

Repita

Q := {X | X $\in$ N e X $\notin$ SP e (existe pelo menos uma produção X ::= $X_1 X_2$,...$X_n$ tal que $X_1 \in$ SP, $X_2 \in$ SP, ..., $X_n \in$ SP)};

SP := SP $\cup$ Q;

Até Q = $\varnothing$;

N'= SP $\cap$ N;

Se S $\in$ SP Então

P':= {p | p$\in$ P e todos os símbolos de p pertencem a SP}

Senão "L(G) = $\varnothing$";

P':= $\varnothing$;

Fim Se

Fim.

**Algoritmo: Eliminação de Símbolos Inalcançáveis**

Entrada: Uma GLC G=(N,T,P,S)

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

SA := {S}

Repita

M := {X | X $\in$ N$\cup$T e X $\notin$ SA e existe uma produção Y ::= $\alpha$X$\beta$ e Y $\in$ SA};

SA := SA $\cup$ M;

Até M = $\varnothing$;

N'= SA $\cap$ N;

T'= SA $\cap$ T;

P':= {p | p$\in$ P e todos os símbolos de p pertencem a SA};

Fim.

## 1.2.4.4.2 – Eliminação de ε-produções

As ε-produções são as produções que derivam ε. Este tipo de produção é aceita em gramáticas livres de contexto exatamente pelo fato de existir um algoritmo que transforma a gramática, tornando-a livre do ε.

O algoritmo é dividido em duas partes, primeiro ele monta o conjunto dos não-terminais que derivam, direta ou indiretamente ε, chamados de ε-não-terminais. Em seguida, de posse desse conjunto elimina-se os ε-não-terminais das demais produções.

**Algoritmo: Encontrar o Conjunto dos ε-não-terminais**

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

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

**Algoritmo: Eliminar Todos os ε-não-terminais**

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

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

Se S $\in$ E, então

Adicione a P’ as produções S’ ::= S | $\varepsilon$;

N’ := N $\cup$ {S’};

Senão

S’ := S;

N’ := N;

Fim Se

Fim.

## 1.2.4.4.3 – Eliminação de Produções Unitárias

Produções unitárias são as produções da forma <A> ::= a, com a sendo um não-terminal. O caso especial onde <A> ::= <A> é chamado de produção circular. Estas produções podem ser eliminadas diretamente, sem nenhum prejuízo para a gramática.

O algoritmo de retirada das produções simples requer que a gramática não possua produções circulares.

**Algoritmo: Eliminar Produções Unitárias**

Entrada: Uma GLC G=(N,T,P,S) sem produções circulares.

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

Para toda A $\in$ N faça:

$N_A$ := {B | A $\rightarrow$* B, com B $\in$ N};

Fim Para

P’ := $\varnothing$;

Para toda produção B ::= $\alpha \in$ P faça:

Se B ::= $\alpha$ não é uma produção unitária, então:

P’ := P’ $\cup$ {A ::= $\alpha$ | B $\in$ $N_A$};

Fim Se

Fim Para

Fim.

### 1.2.4.5 – Técnicas Para Análise Sintática

A análise sintática pode ser feita de duas formas:

- *Top-Down* ou Descendente
- *Bottom-Up* ou Ascendente

Na análise descendente, a árvore de derivação é construída a partir do símbolo inicial da gramática, através de derivações. Já a analise ascendente monta a árvore de derivação de maneira inversa: a partir dos *tokens*, chegando até o símbolo inicial através de reduções.

As duas técnicas serão analisadas em detalhe a partir de agora.

### 1.2.4.6 – Análise Descendente

A análise descendente é feita analisando a sentença (programa) e "tentando" montar uma árvore de derivação para ela. Para isso existem os analisadores com retrocesso (*backtracking*), e os analisadores preditivos.

O primeiro caso sempre tenta, a partir do símbolo inicial da gramática, efetuar derivações á esquerda e montar a árvore de derivação. No caso de não haver regra de derivação aplicável, ele desfaz a ultima derivação e continua. Se não houver possibilidade alguma de continuar, a sentença é rejeitada. Quando toda a sentença tiver sido processada, sua arvore de derivação está pronta.

Está técnica possui diversas desvantagens, como o excessivo tempo de processamento e a eventual dificuldade de se desfazer uma derivação no meio do processo de compilação.

Os analisadores preditivos impõem uma série de restrições à gramática que eles aceitam para que possam fazer a análise sem o retrocesso. No caso de não haver uma regra de transição, a sentença pode ser rejeitada, pois com certeza não haverá outra derivação possível. As gramáticas que se encaixam nestas restrições são chamadas de gramáticas LL.

#### 1.2.4.6.1 – Gramáticas LL

As gramáticas LL são gramáticas livres de contexto com as seguintes restrições:

1 – Não possuir recursão à esquerda

2 – Estar fatorada à esquerda

3 – Para os não-terminais com mais de uma regra de produção, os primeiros terminais deriváveis sejam capazes de identificar, univocamente, a produção que deve ser aplicada a cada instante.

Estas regras permitem que o analisador preditivo sempre possa "prever" qual a próxima derivação a ser feita de maneira não ambígua, levando em consideração apenas o próximo símbolo da entrada.

## 1.2.4.6.2 – Transformando uma gramática em LL

### *Eliminando a Recursão á Esquerda*

A remoção da recursão à esquerda é feita em dois passos: primeiro é feita uma transformação na gramática, para expor as recursões indiretas. Então são retiradas as recursões diretas.

**Algoritmo: Eliminar Recursões Indiretas**

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

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

A eliminação de recursões diretas é feita da seguinte maneira: para cada produção da forma $A ::= A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_n \mid ... \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_m$
onde nenhum $\beta_i$ começa com A, ela deve ser substituídas por:

$A ::= \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_m A'$

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

### *Fatoração*

Para fatorar uma gramática livre de contexto, deve-se alterar as produções envolvidas no não-determinismo da seguinte forma:

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

A ::= $\alpha$A'

A' ::= $\beta$ | $\gamma$

b) o não-determinismo indireto é retirado através de sua transformação em não-determinismo direto (através de derivações sucessivas) e posterior eliminação segundo o item (a)

### *A terceira restrição*

Não existe uma maneira formal de fazer uma gramática se adequar a esta restrição. Esta restrição geralmente está associada com a ambigüidade da gramática, que em alguns casos é impossível ser eliminada. Nestes casos, quando o analisador atingir o ponto ambíguo, haverá mais de uma possível derivação a ser feita, o que não é aceitável para os analisadores preditivos.

Para resolver isso, é necessário que se indique explicitamente qual das possíveis derivações deve ser feita. No caso do **if-then-else** por exemplo, deve-se fazer com que os **else** sempre sejam relativos aos **if** mais próximos.

## 1.2.4.6.3 – Análise Preditiva Tabular

Este tipo de analisador simula um autômato de pilha. Ele utiliza uma tabela para simular a função de mapeamento, uma pilha explicita para simular a pilha, e algumas variáveis internas para guardar o estado do autômato.

O algoritmo de controle do analisador sempre olha para o próximo símbolo na entrada, e para o símbolo no topo da pilha. Se o topo da pilha for um terminal, e igual ao da entrada, o símbolo é reconhecido, senão é gerado um erro. No caso de ser um não-terminal, ele tenta encontrar na tabela alguma transição, caso exista, a produção é jogada na pilha, caso contrario, um erro é gerado.

**Algoritmo para Análise Preditiva tabular**

Repita:

Faça X ser o topo da pilha.

Faça a ser o próximo símbolo da entrada

Se X é terminal ou $ então

Se X = a então

retire X do topo da pilha

retire a da entrada

Senão

erro

Fim Se

Senão (*X é não-terminal*)

Se tabela[X, a] = $X ::= Y_1Y_2...Y_k$ então

retire X da pilha

coloque $Y_kY_{k-1}...Y_2Y_1$ na pilha (*com Y1 no topo*)

Senão

erro

Fim Se

Fim Se

Até X = $ (*pilha vazia, análise concluída*)

## 1.2.4.6.4 – Construção da Tabela

Para a construção da tabela, primeiro é preciso que se calcule os conjuntos FIRST e FOLLOW.

### *FIRST(X)*

O conjunto FIRST de um símbolo de uma gramática é o conjunto de todos os símbolos que podem precedê-lo em uma sentença.

Algoritmo:

1. Se **a** for um terminal, então **FIRST(a) = { a }**.

2. Se existir uma produção **X** → ε na gramática, adiciona ε a **FIRST(X)**.
3. Se **X** → $\mathbf{Y_1Y_2...Y_k}$ é uma produção e, para algum i, todos $\mathbf{Y_1}$, $\mathbf{Y_2}$, ... $\mathbf{Y_{k-1}}$ derivam ε, então **FIRST(Yi)** está em **FIRST(X)**. Se todo **Yj(j** = 1, 2, ... k) deriva ε, então ε está em **FIRST(X)**.

## *FOLLOW(X)*

O conjunto FOLLOW de um símbolo é o conjunto dos símbolos que podem ser precedidos pelo símbolo em questão.

Algoritmo:

1. Se **S** é o símbolo inicial da gramática e **$** é o marcador de fim de sentença, então **$** está em **FOLLOW(S)**.
2. Se existe produção do tipo **A** → a **X**b, então todos os terminais de **FIRST(**b**)** fazem parte de **FOLLOW(X)**.
3. Se existe produção do tipo **A** → a **X**, ou **A** → a **X**b, sendo que b deriva ε em zero ou mais passos, então todos os terminais que estiverem em **FOLLOW(A)** fazem parte de **FOLLOW(X)**.

## *Geração da Tabela*

Na tabela, as colunas correspondem aos terminais, e as linhas aos não-terminais. As células da tabela contem as produções, cujas derivações são aplicadas quando o terminal da coluna esta na entrada, e o não-terminal da linha esta no topo da fila.

A construção da tabela M, é feita da seguinte forma:

1. Para cada produção <A> : := α existente na linguagem faça
    A. Para cada terminal a de FIRST(α), adicione esta produção a M[A, a].
    B. Se FIRST(α) contém ε, então adicione esta produção a M[A, b], para cada b em FOLLOW(A).

### 1.2.4.7 – Análise Ascendente

A idéia deste paradigma de análise sintática é construir a arvore de derivação a partir de suas folhas. O processo de reconhecimento consiste em transferir símbolos da sentença de entrada para a pilha, até que se tenha ali o lado direito de uma produção. Quando isso ocorre, esse lado direito é substituído (reduzido) pelo símbolo do lado esquerdo da produção. O processo segue até que a sentença chegue ao fim, e reste apenas o símbolo inicial da gramática na pilha, caso em que a sentença é aceita.

## 1.2.4.7.1 – Analisadores Sintáticos LR

Está é a classe de analisadores sintáticos ascendentes mais conhecida, eficiente e poderosa, do ponto de vista das estruturas que podem ser reconhecidas por ela. Uma gramática que pode ser reconhecida por um analisador LR é chamada de Gramática LR.

As gramáticas LR são capazes de reconhecer praticamente todas as estruturas sintáticas definidas por gramáticas livres de contexto não ambíguas. De fato, qualquer linguagem reconhecível por um analisador LL é também reconhecível por um analisador LR.

Os analisadores LR trabalham de maneira semelhante aos LL: usam uma pilha de controle e uma tabela de transições. Aqui a pilha é uma pilha de estados, e a tabela é dividida em duas partes: uma tabela de ações, indexada por estado atual(topo da pilha) e símbolo da entrada atual, e uma tabela de desvios, indexada por estado e por símbolo não terminal.

As ações que podem ser encontradas na tabela são as seguintes:

- **Empilhar S**, onde S é um estado:

  Neste caso, um símbolo da entrada é consumido, e o estado atual do analisador é alterado.
- **Reduzir** através da produção P:

  Aqui, o analisador concluiu que os símbolos analisados até o momento são o lado direito da produção P, e podem ser reduzidos a um símbolo não terminal (o lado esquerdo de P). Para tanto ele retira estados do topo da pilha de acordo a produção e empilha um novo estado, consultando a tabela de desvios.
- **Aceitar**:

Esta ação indica que a sentença foi reconhecida com sucesso.

- **Erro**:

  Um erro ocorre quando o analisador não encontra uma redução possível para o estado atual da pilha.

**Algoritmo para Análise LR**

Repita para sempre:

Faça S ser o estado no topo da pilha..

Faça a ser o próximo símbolo da entrada.

Se Ação[S, a] = empilhar S' Então

Empilhar S'.

Fim Se

Senão Se Ação[S, a] = reduzir P Então

Seja P a produção A ::= β.

Desempilhar |β| estados da pilha.

Seja S' o estado agora no topo da pilha.

Empilhar Desvio[S', A].

Fim Se

Senão Se Ação[S, a] = Aceitar

Para análise.

Fim Se

Senão

Erro.

Fim Se.

Fim Repita.

O ponto negativo desta classe de analisadores está no excessivo espaço ocupado por suas tabelas: em uma linguagem pequena, o número de estados pode chegar à casa dos milhares. Existem três técnicas para a geração da tabela de análise, variando-se nelas o tamanho final da tabela e a capacidade de análise da mesma frente a uma sentença qualquer. Estas técnicas são:

- LR Canônica – A mais abrangente de todas, mas que gera as maiores tabelas também.

- SLR – A mais simples de todas, gera tabelas compactas e o processo de geração também é simples. Porém falha em reconhecer certas estruturas que as outras técnicas reconhecem.
- LALR – Técnica intermediária. Possui um tamanho de tabela equivalente à SLR, com a capacidade de reconhecer quase tudo que a LR Canônica reconhece, ao custo de um processo de geração mais complexo.

## 1.2.4.7.2 – Construção da Tabela SLR

A construção desta tabela requer algumas definições.

### *Item LR*

Um item LR em uma gramática é uma produção com um **ponto** em alguma posição do lado direito. Este ponto é uma indicação de até aonde a produção já foi analisada. Uma produção como: A → abc geraria quatro itens:

A → .abc
A → a.bc
A → ab.c
A → abc.

Já a produção A→ε gera apenas o item A→ .

### *Função Fechamento( I )*

Seja I um conjunto de itens LR. O *fechamento(I)* é calculado da seguinte forma:

1. Todo item de I pertence *ao fechamento(I).*
2. Se A→α.Xβ está no conjunto *fechamento(I),* e X→ γ é uma produção, então adicione X→ .γ ao *fechamento(I).*

Estas regras devem ser repetidas até que não seja mais feita nenhuma alteração a *fechamento(I).*

## *Função Desvio( I, X )*

A função *desvio(I,X)* , onde I é o conjunto de itens e X um símbolo gramatical, é definida como o *fechamento* do conjunto de todos os itens $A \rightarrow \alpha X . \beta$ tais que $A \rightarrow \alpha . X\beta$ esteja em I.

## *Conjunto de Itens LR*

Para uma gramática G, o Conjunto Canônico de Itens LR, C, é obtido por:

C ← { fechamento( { S′→.S } ) }
Repita
    Para cada conjunto de itens I em C e cada símbolo gramatical X
        tal que desvio(I, X) não seja vazio e não esteja em C faça:
    Incluir desvio (I,X) a C
Até que não haja mais conjuntos de itens a serem incluídos a C

## *Construção da Tabela de Análise SLR*

Dada uma gramática G, obtém-se G', aumentando-se G com a produção $S' \rightarrow S\$$, onde S é o símbolo inicial de G. A partir de G', determina-se seu conjunto canônico C, a com o algoritmo seguinte, sua tabela de análise sintática SLR.

**Entrada:** Conjunto canônico de itens LR C de G'.
**Saída:** Tabela de análise SLR para G'.
**Método:**

Seja C = { $I_0$, $I_1$,..., $I_n$}. Os estados do analisador são 0, 1,..., n. A linha i da tabela é construída a partir do conjunto $I_i$, como segue:

As ações do analisador para o estado i são determinadas usando as regras:

1. se desvia($I_i$, a) = $I_j$, então faça AÇÃO[i, a] = empilha(j)
2. se $A \rightarrow \alpha .$ está em $I_i$, então para todo a em FOLLOW(A), faça AÇÃO[i, a] = reduz(n), sendo n o numero da produção $A \rightarrow \alpha$.
3. se $S' \rightarrow S.$ está em $I_i$, então faça AÇÃO[i, $] = aceita.

Se for gerado algum conflito, significa que a gramática não é SLR

As transições para o estado i são construídas da forma seguinte:

Se desvia($I_i$, A) = $I_j$, então TRANSIÇÃO[i, A] = j.

As posições não definidas na tabela constituem estados de erro.

### 1.2.5 – Resumo: Analisadores Preditivos versus Redutivos

Para uma melhor percepção das diferenças entre as duas técnicas e sua aplicabilidade, uma análise comparativa se faz necessária.

| | **Preditivo** | **Redutivo** |
|---|---|---|
| Construção Tabela | Processo Simples | Processo complexo |
| Tamanho Tabela | Relativamente Pequeno | Geralmente Grande |
| Generalidade | Mais restrito quanto à classe de gramáticas suportadas. Proíbe recursão à esquerda e gramáticas não fatoradas. | Muito mais abrangente (praticante todas as gramáticas livres de contexto) |
| Compreensão da Técnica | Simples e intuitiva. | Também é simples, mas não tão intuitiva. |
| Recuperação de Erros | A pilha sintática contém símbolos que predizem mas ainda não foram reconhecidos. O que facilita na determinação de possíveis reparos, e torna essa técnica mais simples nesse quesito. | A pilha sintática contém informação sobre o que já foi reconhecido. A decisão sobre caminhos alternativos que possam ser usados para reparar o erro não é trivial. |
| Uso através de um Gerador Automático | As transformações necessárias na gramática podem torna-la LL podem fazer com que ela fique ilegível | Obtém-se facilmente um analisador para quase qualquer gramática via um gerador LR |

# 2 – Geradores Automáticos de Analisadores

## 2.1 – Geradores de Analisadores Léxicos

### *2.1.1 – LEX*

O *LEX* é um dos mais tradicionais geradores de analisadores léxicos. Foi desenvolvido por Lesk e Schmidt em 1975. Trata-se originalmente de um programa UNIX que lê uma especificação léxica formada de expressões regulares e gera uma rotina de análise léxica para linguagem C. Existem atualmente versões para outros sistemas e que gerem analisadores para outras linguagens.

A especificação de entrada é formada de expressões regulares e comandos em linguagem de programação. O analisador é gerado de forma que quando uma dada expressão é reconhecida, os comandos associados com ela são executados. Isto pode facilitar o trabalho a princípio, mas juntar a especificação com o código objeto torna a leitura da primeira mais difícil. Além de que para se utilizar esta especificação para gerar um analisador em uma outra linguagem é preciso altera-la.

Por ser uma ferramenta muito conhecida e utilizada, é fácil encontrar documentação sobre ela para aprender como usa-la. Porém é preciso que o usuário esteja familiarizado com ferramentas de linha de comando

### *2.1.2 – Microlex*

Foi um projeto de conclusão de curso de Bacharelado em Ciências da Computação da UFSC, pelo aluno Rui Jorge Tramontin Jr. Faz parte de um projeto conjunto com o gerador de analisadores sintáticos **Brain**. O programa possui uma interface com o usuário que lhe permite criar/modificar o arquivo de entrada, bem como fazer testes com ele antes de gerar o analisador.

A forma de o usuário entrar com a especificação léxica da linguagem é através de expressões regulares, são definidas expressões para cada *token* da linguagem e o Microlex se encarrega da geração do autômato finito para a identificação destes.

O gerador final pode ser obtido em três linguagens: Object Pascal, C++ e Java, e ainda é possível escolher duas possíveis técnicas para o gerador: através de tabelas de análise, com uma rotina auxiliar, ou com o autômato programado diretamente na linguagem objeto.

Esta era a proposta inicial do projeto, mas ainda não foram implementadas todas as funcionalidades. Somente se pode gerar analisadores em Object Pascal, a geração para as outras linguagens não foi totalmente concluída.

## 2.2 – Geradores de Analisadores Sintáticos

### *2.2.1 – YACC*

O YACC, iniciais de *Yet Another Compiler-Compiler*, é um gerador de Analisadores Sintáticos que roda no ambiente UNIX. Escrito por Johnson em 1975. Aceita como entrada uma especificação das características sintáticas da linguagem, com ações semânticas embutias, e gera uma rotina em C para a análise sintática.

A saída do YACC consta de uma tabela de análise sintática LALR, uma rotina de controle, que executa em cima da tabela, e as ações semânticas, que foram especificadas durante a definição da linguagem no arquivo de entrada.

O analisador gerado é feito de modo a trabalhar em conjunto com uma rotina de análise léxica gerada pelo LEX, permitindo uma fácil integração entre os analisadores léxico (gerado pelo LEX), sintático e semântico (gerados pelo YACC). Todos são gerados no mesmo ambiente (UNIX) e na mesma linguagem (C).

A especificação é constituída de uma gramática livre de contexto. No fim de cada produção pode ser escrita a ação a ser tomada quando a produção for identificada. A ação é escrita diretamente na linguagem objeto, o que dificulta o uso de uma mesma especificação para a geração de compiladores em diferentes linguagens. Para que se possa utilizar o YACC é preciso que o usuário tenha experiência com o uso de programas de linha de comando.

### *2.2.2 – GAS*

Projeto de conclusão de curso de um aluno do Curso de Bacharelado em Ciências de Computação de UFSC, em 1991, o GAS é uma ferramenta didática para o aprendizado de Linguagens Formais e Compiladores.

A forma com que o usuário entra com a especificação da linguagem é através de uma gramática livre de contexto. O GAS então apresenta sua interface com o usuário que guia o processo de geração do analisador.

Quanto a classe de analisador gerada, é possível gerar analisadores LL(1), SLR(1), LALR(1) e LR(1). Esta característica lhe confere grande valor didático, pois assim torna-se

possível a estudantes de compiladores verificar as diferentes características das diferentes técnicas.

O GAS fornece ao usuário três possíveis linguagens objeto: Pascal, C e Modula 2. Isso é interessante pois aumenta seu valor como ferramenta, pois abre um leque maior de possibilidades ao usuário, permitindo a ele trabalhar com a linguagem que mais lhe agrade.

Como desvantagem do GAS, pode-se citar o fato sua interface ser um tanto quanto deficiente e defasada. O código-fonte do programa foi perdido, não sendo possível gerar uma nova versão. E apesar de serem oferecidas três opções de linguagem para o usuário, uma delas não funciona (C), e outra é pouquíssimo utilizada hoje em dia (Modula 2), restando apenas Pascal para se usar.

### *2.2.3 – Brain*

O Brain também foi um projeto de conclusão de curso do Curso de Bacharelado em Ciências de Computação de UFSC. Foi desenvolvido em conjunto por Alessandro Fragnani e Rodnei Rodhen, como parte de um projeto conjunto com o **Microlex**.

Possui uma interface com usuário que lhe permite passar a especificação sintática da linguagem (sob a forma de uma gramática livre de contexto) ao programa. Ele faz checagens na gramática para verificar se a classe que será gerada é suportada. Permite que se teste o gerador através de um simulador. E também é gerada uma documentação para o usuário em cima da gramática de entrada.

Quanto as classes de geradores, ele suportaria LL(1), LR(1), LALR(1) e SLR(1). Assim como o MicroLex, ele deveria gerar analisadores para Object Pascal, C++ e Java.

O projeto também não foi totalmente concluído, sendo que apenas analisadores LL(1) são gerados até o momento. A geração de analisadores em Object Pascal é satisfatória, vem inclusive sendo utilizada em alguns projetos. A geração do analisador em C++ gera um analisador com um *bug* em uma das tabelas.

## 2.3 – Estudo Comparativo

Os geradores de analisadores léxicos aqui apresentados, o LEX e o Microlex, trabalham em conjunto com geradores analisadores sintáticos aqui também apresentados, respectivamente o YACC e o Brain. A exceção é o GAS, que é um gerador de analisadores sintáticos e não possui um “parceiro” para a geração do analisador léxico. O fato de existirem duas ferramentas conjuntas facilita sua utilização, pois minimiza as incompatibilidades e diminui o trabalho do

usuário. Se ele quiser utilizar o GAS, terá que fazer o analisador léxico manualmente, ou tentar adaptar algum gerado por alguma outra ferramenta. Mas mesmo nos casos em que se tem duas ferramentas conjuntas, são duas ferramentas diferentes, e é preciso que o usuário aprenda a utilizar cada uma delas.

O par LEX/YACC é bastante conhecido e consagrado. Seu uso na geração de compiladores é extenso. São ferramentas que são de certa forma difíceis de se usar, principalmente por usuários menos experientes.

O GAS, mesmo não tendo um meio simples de obter um analisador léxico, é uma ferramenta excelente, de grande valor didático. Ele gera várias classes de analisadores sintáticos e em várias linguagens. Mas é um programa para DOS. Até quando será possível rodar ele no Windows? É possível que alguma versão futura do sistema operacional da Microsoft elimine totalmente o suporte a esse tipo de programa, tornando o GAS então inutilizável.

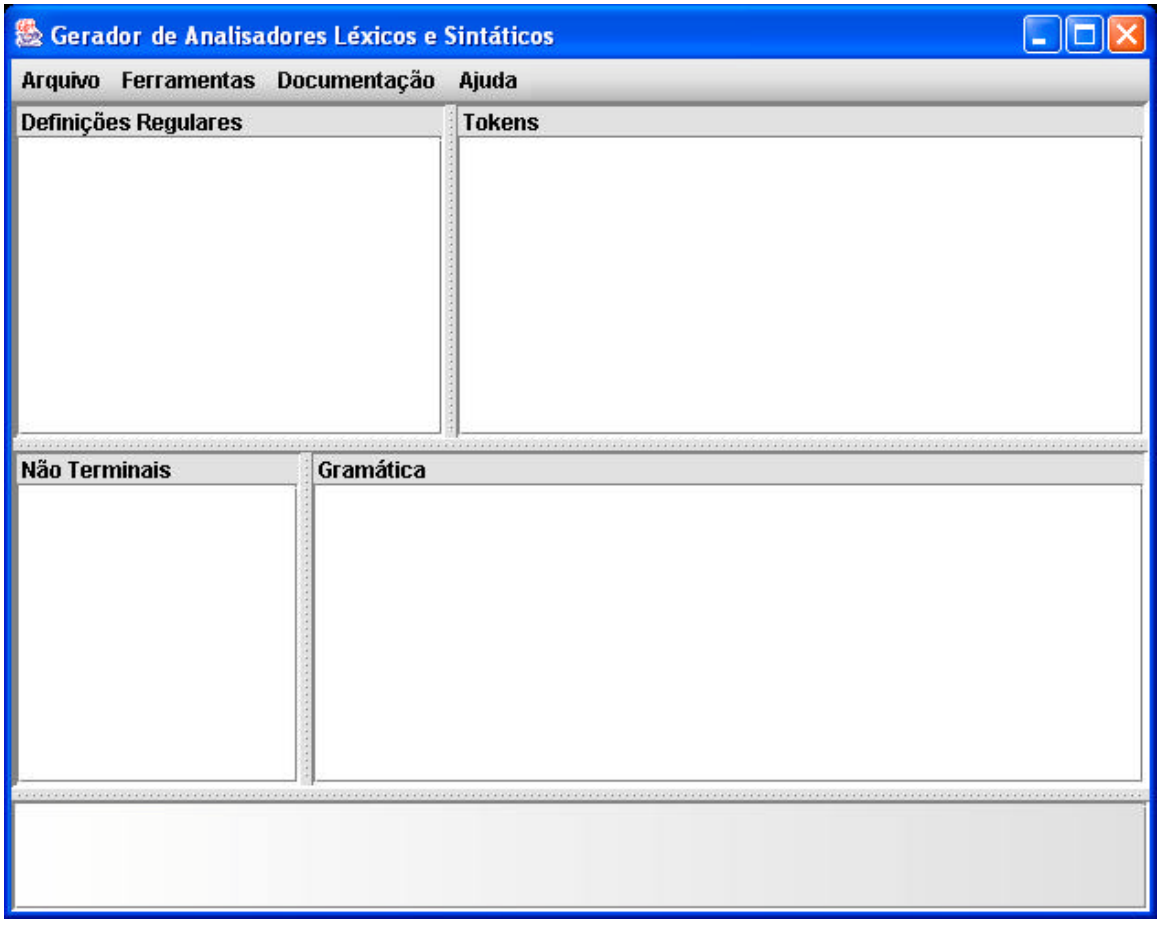
Por fim, o par Microlex/Brain surge como uma nova alternativa para a geração de analisadores léxicos e sintáticos. Os dois gerariam diversas classes de analisadores em diversas linguagens, porém o estado atual deles não é o que foi proposto inicialmente, eles estão um tanto quanto inacabados.

# 3 – GALS: Gerador de Analisadores Léxicos e Sintáticos

GALS é uma ferramenta para a geração automática de analisadores léxicos e sintáticos. Foi desenvolvido em Java, versão 1.4, podendo ser utilizado em qualquer ambiente para o qual haja uma máquina virtual Java. É possível gerar-se analisadores léxicos e sintáticos, através de especificações léxicas, baseadas em expressões regulares, e especificações sintáticas, baseadas em gramáticas livres de contexto. Pode-se fazer os analisadores léxico e sintático independentes um do outro, bem como fazer de maneira integrada. Existem três opções de linguagem para a geração dos analisadores: Java, C++ e Delphi. O GALS tenta ser um ambiente amigável, tanto para uso didático como para o uso profissional.

## 3.1 – Uso do Ambiente

É através do ambiente que o usuário entra com sua especificação para a geração do analisador. Está é a visão inicial que se tem:



Nesta interface são definidos aspectos léxicos e sintáticos de uma forma conjunta. Porém, dependendo da escolha que se faça, para gerar apenas o analisador léxico ou o sintático, o ambiente é rearranjado de modo adequado.

Os seguintes menus estão disponíveis para o usuário:

- Arquivo – Permite operações básicas, como abrir, salvar e criar novos arquivos, além de importar especificações sintáticas do GAS.
- Ferramentas – Possibilita modificar opções, verificar erros, simular os analisadores e gerar o código dos analisadores léxico e sintático.
- Documentação – Exibe as tabelas de análise.
- Ajuda – Ajuda para o usuário.

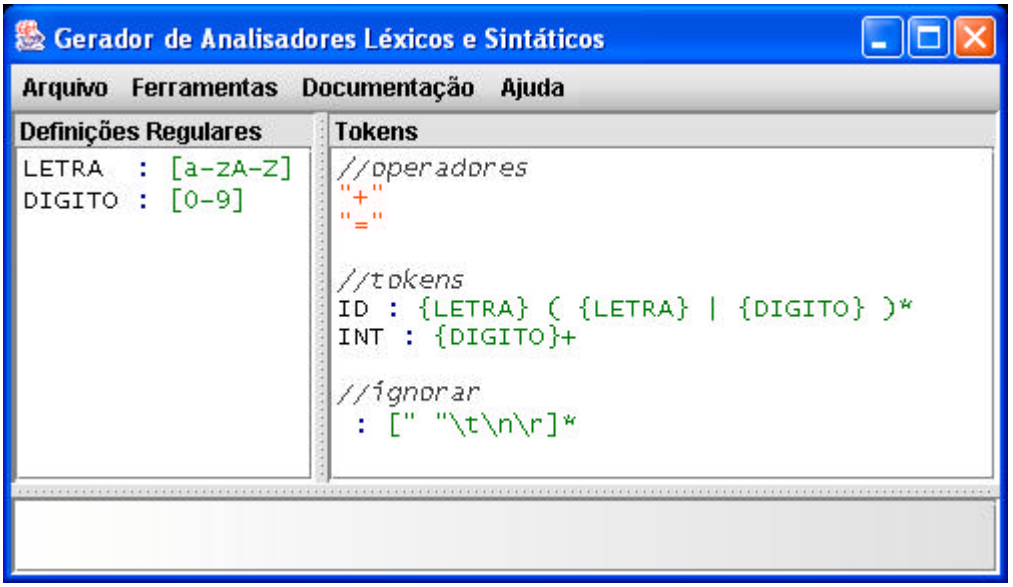
### *3.1.1 – Especificando a Entrada*

Existem três casos a serem explorados aqui: uma especificação para a geração de um analisador léxico apenas, uma para a geração de um sintático apenas, e uma especificação conjunta. Para cada um dos casos a interface se ajusta de forma adequada.

#### 3.1.1.1 – Especificando o Analisador Léxico

A especificação dos aspectos léxicos é composta de duas partes: especificação dos *tokens* que o analisador deve reconhecer, e a descrição opcional de definições léxicas, que são como macros que podem ser usadas na definição dos *tokens*.

Segue um pequeno exemplo




Têm-se aqui duas definições regulares: LETRA e DIGITO. Estas definições não influem em nada no analisador gerado, apenas facilitam a vida do usuário no sentido de que ele não precisa ficar repetindo uma expressão regular complexa na posterior definição dos *tokens*.

São definidos quatro *tokens* (“+”, “=”, ID e INT) e um conjunto de caracteres a serem ignorados pelo analisador. Como pode-se ver, existem maneiras diferentes para definir os *tokens*, e elas serão mostradas a serguir.

### 3.1.1.1.1 – Definições Regulares

Todas as definições regulares são da forma:

```
<IDENTIFICADOR> : <EXPRESSÃO REGULAR>
```

Só é permitida uma definição por linha. Identificadores repetidos são considerados erros, que são reportados ao usuário. Pode-se ainda inserir comentários de linha, que se iniciam com “`//`” e se estendem até o fim da linha.

### 3.1.1.1.2 - Tokens

A forma geral da definição de um token é:

```
<TOKEN> : <EXPRESSÃO REGULAR>
```

Só é permitida a definição de um *token* por linha. São aceitos comentários, da mesma forma que nas definições regulares. Aqui, TOKEN pode ser um identificador, ou uma seqüência de caracteres entre “”. Este é o caso geral, porem existem casos especiais que devem ser notados:

O primeiro deles é que a expressão regular pode ser omitida (junto com o `:`). Isto diz ao gerador que o formato do *token* coincide com sua identificação. Este caso é interessante principalmente para a definição de operadores e outros símbolos, em geral constituídos de caracteres especiais.

Um outro caso é que o identificador do *token* pode também ser omitido. Uma definição nesta forma indica ao gerador que se trata de uma expressão que deve ser reconhecida, porém nenhum *token* deve ser gerado, e que se deve depois disso procurar por uma outra expressão. A principal aplicação desta modalidade é a especificação de caracteres e expressões que devem ser ignorados pelo analisador, como comentários e espaços em branco.

Se em vez de `:` for utilizado `:!` para separar o identificador da expressão regular, então para este *token* não será feita a tentativa de encontrar outro *token* em caso de se chegar em um estado sem transições possíveis. Este comportamento é interessante em alguns casos, como comentários, onde um erro (como fim de arquivo inesperado) deve gerar um comentário inválido, e não fazer com que o analisador volte até o início do comentário e tente encontrar outro *token.*

Existe ainda um terceiro caso, que não segue o padrão anterior. Ele se presta à definição de casos especiais de um *token*. Seu uso mais claro é na definição de palavras reservadas, em geral casos especiais de identificadores. Poderia-se definir todas as palavras reservadas como tokens, mas isso geraria um autômato muito grande. Com esse mecanismo as palavras reservadas ficam em uma tabela separada. Então sempre que o analisador identificar o *token* base, faz uma procura na tabela de casos especiais, para ver se trata-se de um ou não.

A forma é esta:

```
<TOKEN> = <TOKEN BASE> : <PADRÃO>
```

Por exemplo: `BEGIN = ID : “begin”`.

## 3.1.1.1.3 – Expressões Regulares

As expressões regulares são a base da especificação léxica. Através delas são indicados os padrões que devem ser reconhecidos pelo analisador gerado. Sua sintaxe será apresentada agora.

Uma expressão regular é uma seqüência de caracteres. A expressão abc reconhece a seqüência de caracteres abc. Espaços em branco no meio de uma expressão são ignorados. Existem certos caracteres com significado especial, que dentro de uma expressão atuam como operadores ou caracteres de controle. Estes caracteres são:

```
"  \  |  *  +  ?  (  )  [  ]  {  }  .  ^  -
```

Para utilizar um desses caracteres dentro de uma expressão, sem seu sentido especial, pode-se precedê-lo por \ ou colocá-lo entre “”. Pode-se agrupar um número qualquer de caracteres entre “”. A exceção é ", que para ser colocado entre aspas deve ser dobrado, de maneira análoga ao ' nas *strings* de Pascal. Exemplos:

| Expressão | Padrão reconhecido |
|---|---|
| `\(` | `(` |
| `\\` | `\` |
| `"*"` | `*` |
| `"\(a? "` | `\(a?` |
| `""""` | `"` |

Quaisquer caracteres colocados entre aspas são tratados como caracteres normais, inclusive espaços em branco, que neste caso não são mais ignorados.

Além de caracteres especiais, existem os caracteres que não podem ser representados, como quebra de linha e tabulação. Para identificar estes caracteres são utilizadas as seguintes seqüências:

| Expressão | Padrão reconhecido |
|---|---|
| `\n` | Line Feed |
| `\r` | Carriage Return |
| `\s` | Espaço |
| `\t` | Tabulação |
| `\b` | Backspace |
| `\e` | Esc |
| `\XXX` | O caractere ASCII XXX (XXX é um número) |

Os caracteres [ , ] e ^ servem para definir classes de caracteres. Uma classe de caracteres é um conjunto de caracteres entre [ e ], ou entre [^ e ]. No primeiro caso, a classe reconhece qualquer caractere dentre os pertencentes a ela. No segundo caso, reconhece qualquer caractere,

exceto aqueles pertencentes à classe (complemento). Pode-se ainda especificar seqüências de caracteres dentro de classes, indicando os limites inferior e superior e separando-os por -. Exemplos:

| Expressão | Padrão reconhecido |
|---|---|
| `[abc]` | a, b ou c |
| `[^abc]` | Qualquer caractere, exceto a, b ou c |
| `[a-e]` | a, b ,c, d ou e |
| `[a-zA-Z]` | Qualquer letra |
| `[0-9]` | Qualquer dígito |

O operador | simboliza a união, indica que entre duas expressões, qualquer uma delas pode ser reconhecida.

`ab|cd` reconhece `ab` ou `cd`.

Pode-se utilizar parênteses para agrupar-se sub-expressões.

`a (b|c)` reconhece `ab` ou a`c`.

Os operadores *, + e ? são colocados após algum elemento da expressão e tem os seguintes significados:

* - O elemento pode aparecer repetidamente, zero ou mais vezes (fechamento reflexivo).
+ - O elemento pode aparecer repetidamente, pelo menos uma vez (fechamento aberto).
? – O elemento é opcional.

`a*` reconhece zero ou mais a's

`a+` reconhece um ou mais a's

`a?` reconhece um a ou nenhum a

O caractere . reconhece qualquer caractere, exceto quebra de linha.

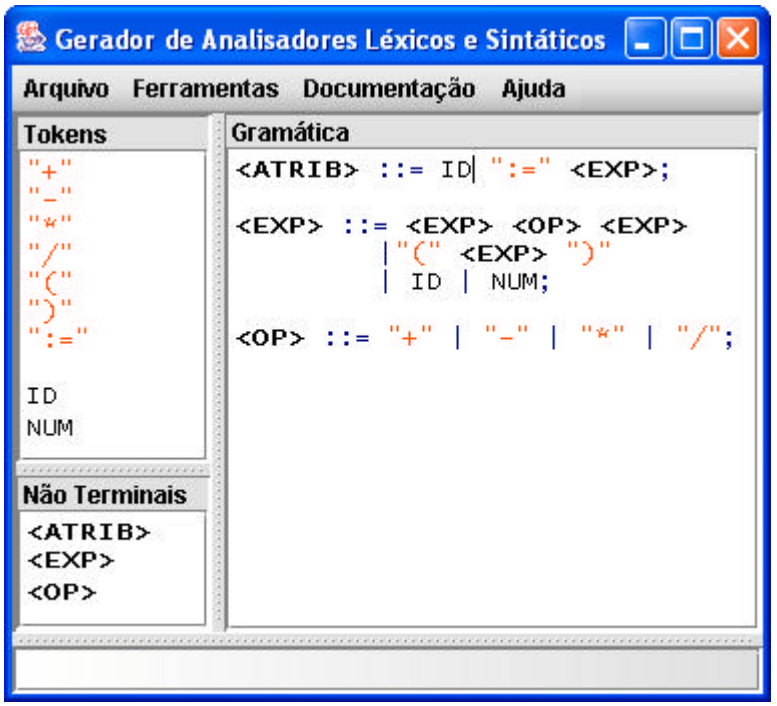
Por fim, na definição de *tokens* pode-se utilizar as definições regulares previamente definidas. seu uso se faz colocando o identificador da expressão entre { e }.

Um caso especial deve ainda ser notado. Apesar de / não ser um caractere especial, duas barras seguidas são o inicio de comentário. Se isto aparecer dentro de uma expressão fará com que o analisador acredite que a expressão acabou neste ponto e que o restante da linha pode ser ignorado. Se for preciso utilizar duas barras seguidas dentro de uma expressão, deve-se colocar um espaço entre elas.

### 3.1.1.2 – Especificando o Analisador Sintático

A especificação sintática é constituída de três partes: definição dos símbolos terminais (*tokens*), definição dos símbolos não-terminais, e definição das produções gramaticais. Em todos os casos são aceitos comentários de linha, iniciados com //.

Segue um exemplo de uma especificação:



#### 3.1.1.2.1 – Símbolos Terminais (Tokens)

Podem ser definidos por identificadores ou seqüências de caracteres entre aspas.

#### 3.1.1.2.2 – Símbolos Não-Terminais

São definidos por identificadores entre < e >. O primeiro símbolo é considerado o símbolo inicial da gramática.

#### 3.1.1.2.3 – Produções

Seguem uma forma semelhante à BNF:

```
<NT> ::= <LISTA DE SÍMBOLOS> | <LISTA DE SÍMBOLOS> | ... ;
```

Onde NT é um símbolo não-terminal e LISTA DE SÍMBOLOS é uma lista de símbolos terminais e não terminais e de ações semânticas (a serem vistas mais tarde).

Todos os símbolos utilizados aqui devem ter sido previamente definidos. O símbolo terminal ε é representado por î.

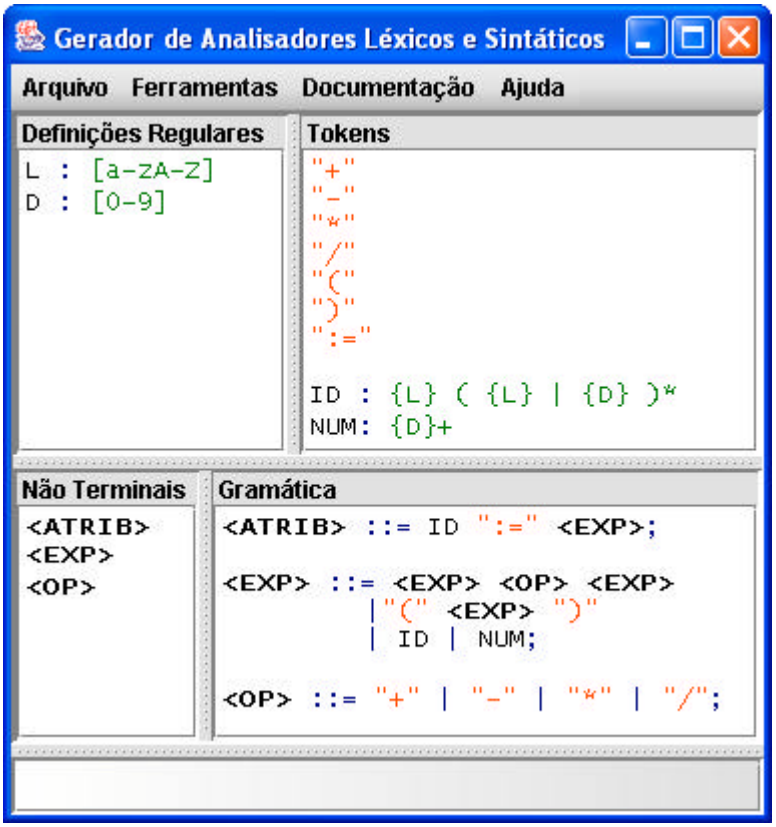
#### 3.1.1.2.4 – Ações Semânticas

Ações semânticas podem ser inseridas em qualquer ponto do lado direito das produções. São representadas por um # seguido de um número.

Durante a análise, quando uma destas ações é encontrada, o analisador semântico é acionado. São passados para ele o número da ação e o último *token* produzido pelo analisador léxico. Cabe ao usuário a disposição correta das ações para que possa ser feita a tradução dirigida pela sintaxe. A implementação das ações também é serviço do usuário.

### 3.1.1.3 – Especificando os Analisadores em Conjunto

Em uma especificação conjunta, as especificações léxica e sintática são ligadas pelos *tokens*. Os aspectos léxicos são definidos do mesmo modo que em uma especificação apenas léxica. O mesmo para os aspectos sintáticos, exceto pelos símbolos terminais, onde os *tokens* definidos na especificação léxica são tomados como símbolos terminais na especificação sintática. Um exemplo:




## *3.1.2 – Opções na geração do analisador*

### 3.1.2.1 – Opções Gerais

A primeira opção que se tem é referente ao que se deseja gerar: apenas o analisador léxico, apenas o sintático, ou ambos.

Em seguida pode-se escolher em qual linguagem será gerado o analisador. Pode-se optar por Java, C++ e Object Pascal.

Também é possível configurar-se os nomes das classes geradas. São geradas classes para cada tipo de analisador, e aqui se pode definir seus nomes. Se a linguagem for Java, pode-se escolher uma *package* para as classes. E em C++ pode-se escolher o *namespace* das classes.

### 3.1.2.2 – Opções do Analisador Léxico

O analisador léxico trabalha diretamente com o texto a ser analisado. Pode-se fazer com que o analisador trabalhe com ele de duas maneiras: lendo de uma *string*, ou de um *stream*. O primeiro caso é mais indicado para quando se está trabalhando com componentes gráficos,

quando se pode acessar diretamente todo o texto do programa. Já o segundo caso é indicado para quando se está lendo o programa diretamente de um arquivo, acessado em geral via *streams*.

#### 3.1.2.3 – Opções do Analisador Sintático

Para o analisador sintático, pode-se escolher que técnica de análise sintática será utilizada. As técnicas disponíveis são:

- Descendentes
  - Descendente Recursivo
  - LL
- Ascendentes
  - SLR
  - LALR
  - LR Canônico

### *3.1.3 – Documentação Gerada*

As tabelas de análise geradas podem ser vistas pelo usuário, como documentação. Todas elas são convertidas para o formato HTML e exibidas em um dialogo.

Do analisador léxico, é gerado um autômato finito a partir das expressões regulares. Pode-se ver este autômato em forma de tabela de transições.

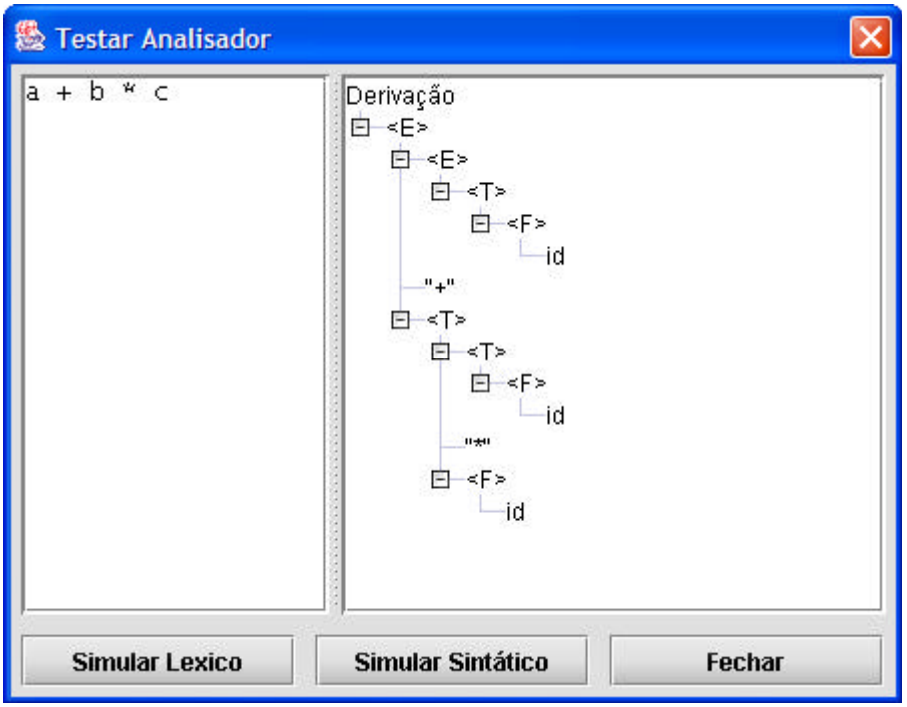
O analisador sintático gera mais documentação. Há os conjuntos FIRST e FOLLOW, e a tabela de análise sintática. No caso de o analisador ser um analisador LR (SLR, LALR, LR Canônico) também é mostrado o conjunto de estados a partir do qual é montada a tabela de análise.

### *3.1.4 - Simulador*

A ferramenta de simulação permite que o usuário possa testar suas especificações léxicas e sintáticas sem que precise gerar código e compilá-lo.

A simulação do analisador léxico mostra ao usuário a lista dos *tokens* produzidos a partir de uma entrada de teste com sua especificação. Para cada *token* é mostrado também seu valor léxico (lexema) e sua posição na entrada.


A simulação do analisador sintático gera a árvore de derivação para a sentença fornecida. A forma da entrada depende de a especificação do usuário ser apenas sintática ou léxica e sintática. No primeiro caso deve-se usar na sentença os símbolos terminais diretamente. No segundo caso, a entrada é primeiramente processada pelo simulador do analisador léxico, e os *tokens* gerados são então passados para o simulador de sintático.




### 3.1.4 – Importando Arquivos do GAS

No menu Arquivo existe a opção Importar Arquivo BNF. Este formato de arquivo é gerado pelo GAS – Gerador de Analisadores Sintáticos. Esta opção foi criada devido à grande quantidade de especificações sintáticas neste formato.

Existem algumas incompatibilidades entre aspectos léxicos do GAS e do GALS que devem ser notados. Em primeiro estão os identificadores. O GAS aceita qualquer seqüência de caracteres como identificador, exceto caracteres especiais, enquanto o GALS aceita apenas letras, dígitos e sublinhado ( _ ). Durante o processo de importação, os identificadores são transformados, para se adequar ao GALS: qualquer caractere que não seja letra ou digito é convertido para sublinhado.

Existe também uma diferença quanto à definição de símbolos terminais usando como identificador uma seqüência de caracteres quaisquer entre aspas. No GALS são sempre usadas aspas ("). Para representar este caractere dentro da seqüência deve-se utiliza-lo dobrado. No GAS, pode-se utilizar aspas, com qualquer caractere dentro, exceto aspas, ou apostrofe ('), com qualquer, apostrofe. A adaptação aqui é simples.

No fim do processo obtém-se uma especificação sintática no formato GALS, pronta para ser usada para gerar código.

## 3.2 – Análise do Código Gerado

Será feita agora a análise de partes do código gerado. Para tanto serão utilizados trechos de código em Java. O código gerado para outras linguagens é sempre análogo, e em casos de diferenças muito grandes, estas serão destacadas.

### *3.2.1 – Código Comum*

Certas classes e constantes são usadas tanto pelo analisador léxico quanto pelo sintático. Tem-se a classe Token, as classes de exceções, e as constantes e tabelas de analise.

#### 3.2.1.1 – A Classe Token

É uma classe simples, para guardar informações sobre cada *token.* Sua interface é:

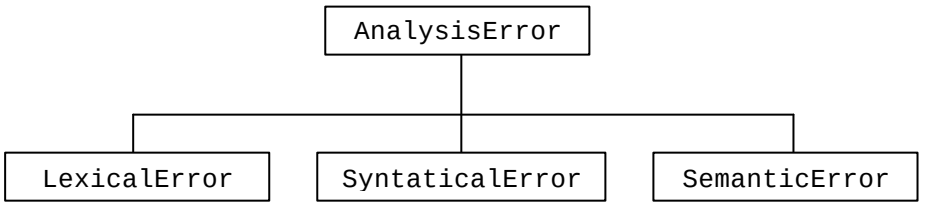
```
public class Token
{
      public Token(int id, String lexeme, int position);

      public int getId();
      public String getLexeme();
      public int getPosition();
}
```

Cada token possui um identificador (id), um valor léxico (lexeme) e uma posição no texto (position).

#### 3.2.1.2 – Classes de Exceções

As exceções representam os erros que acontecem no processo de análise. Estão agrupadas na seguinte hierarquia:



O analisador léxico é programado de modo a lançar LexicalError quando encontra uma situação de erro. O Analisador sintático lança SyntaticalError. SemanticError existe para manter a uniformidade, pois o analisador semântico não é gerado automaticamente. Quando o usuário for implementar suas ações semânticas, ele deve lançar esta exceção.

#### 3.2.1.3 – Constantes e Tabelas

Todas as constantes e tabelas ficam agrupadas num único módulo, que é implementado de modo diferente em cada linguagem. Em Java ficam dentro de uma interface, implementada pelas classes dos analisadoes. Em C++ ficam em um headerfile, e em Object Pascal em uma Unit.

Para os *tokens*, são geradas constantes. Estas não são utilizadas pelos analisadores, sua intenção é facilitar a implementação das ações semânticas. O usuário pode utilizá-las para identificar um dado *token*, em vez de fazer isso checando apenas o número do *token.*

### *3.2.2 – Analisador Léxico*

A classe gerada para o analisador léxico trabalha lendo caracteres e agrupando-os em *tokens*. Possui métodos para inicializar e alterar o texto processado.

Seu principal método no entanto é *nextToken*, que a cada chamada retorna um novo *token*, e quando não há mais *tokens* para serem retornados, retorna *null*. Deste modo, torna-se fácil o uso do analisador, bastando usa-lo de maneira iterativa. Será mostrada agora uma análise mais profunda deste método.

```
public Token nextToken() throws LexicalError
{
    if ( ! hasInput() )
        return null;

    int start = position;
    int state = 0;
    int endState = -1;
    int end = -1;

    while (hasInput())
    {
        state = nextState(nextChar(), state);
        if (state < 0)
```

```
                break;
            else
            {
                if (tokenForState(state) >= 0)
                {
                    endState = state;
                    end = position;
                }
            }
        }
        if (endState < 0)
            throw new LexicalError("Token inválido: " +
       input.substring(start, position-1), start);

        position = end;

        int token = tokenForState(endState);

        if (token == 0)
            return nextToken();
        else
        {
            String lexeme = input.substring(start, end);
            token = lookupToken(token, lexeme);
            return new Token(token, lexeme, start);
        }
    }
```

O primeiro passo é verificar se o fim do texto já não foi atingido. Neste caso é retornado *null*. Em caso contrário a análise pode prosseguir. A variável *start* indica onde este *token* atual começou. *State* mantém o estado do autômato finito, em cima do qual a análise léxica é feita. As variáveis *endState* e *end* guardam o ultimo estado final atravessado, e a posição no texto neste estado. Sua função é permitir sempre procurar pelo *token* mais longo possível.

Enfim entra-se no laço principal do método. Enquanto o fim do texto não for atingido, faz-se o estado avançar, de acordo com o estado atual e o próximo caractere da entrada. Se este for um estado inválido (não havia transição possível), o laço é quebrado. Senão, se este for um estado final, informações sobre ele são guardadas. Como sempre se tenta retornar o maior *token*

possível, a análise não pode parar neste ponto, somente quando não houver mais transições possíveis.

No caso de nenhum estado final ter sido visitado, um erro léxico é lançado. Neste ponto, o usuário poderia alterar o analisador para inserir algum código de recuperação de erros.

Por fim, verifica-se qual o *token* associado com o estado atual. O *token* de valor zero é associado com as expressões a serem ignoradas. Neste caso o método é chamado novamente, até que seja avaliado um *token* válido. Achando-se um *token* válido, ainda é preciso fazer a busca na tabela de casos especiais, para finalmente então fornecer o *token* efetivamente reconhecido.

### *3.2.3 – Analisador Sintático*

O analisador sintático faz uso de um analisador léxico (gerado automaticamente, ou pelo usuário) e de um analisador semântico. Sempre que um novo *token* é necessário, o método *nextToken* do analisador léxico é chamado. Da mesma forma, sempre que durante a análise uma ação semântica for alcançada, o analisador semântico é acionado.

Se a opção do usuário for gerar apenas um analisador sintático, será gerada para ele uma classe de analisador léxico, apenas com o método *nextToken*, vazio, cabendo ao usuário implementa-lo. É um método sem parâmetros e que deve retornar um novo *token* a cada chamada.

Também é gerada uma classe de analisador semântico, apenas com um método: *executeAction*, sem implementação. O analisador sintático invoca este método a cada ação semântica. São passados como parâmetro: o número da ação semântica, e o último token identificado.

Quanto ao analisador sintático, existem três casos a serem analisados. Cada método de análise (Descendente Recursivo, LL e LR) gera um algoritmo diferente, e eles serão analisados em separado. Para a técnica descendente recursivo, as produções são implementadas como métodos, que se chamam uns aos outros. Todas as técnicas possuem em comum: instâncias de um analisador léxico e um semântico, referências para o ultimo e o penúltimo *tokens* analisados e, com exceção da descendente recursivo, uma pilha. Para a pilha, em Java é utilizada a classe `Stack`, do pacote `java.util`. Em C++ é utilizada uma `std::stack`, pertencente à biblioteca padrão STL. E em ObjectPascal, uma `TList` é usada.

A referência para o ultimo *token* é utilizada durante a analise, para endereçar a tabela. A referencia para o penúltimo se faz necessária pelo analisador semântico. Quando uma ação semântica é identificada, é passado para o analisador semântico o ultimo *token* antes da ação. Este *token* porém não está mais disponível, o *token* atual já é o primeiro *token* depois da ação, e por isso guardar o penúltimo *token* se faz preciso.

Todos os analisadores são acionados do mesmo modo: através do método *parse*. Para este método são passadas instancias de analisadores léxico e semântico. Internamente, o método *parse* inicializa as instâncias de analisadores com as passadas como parâmetro e obtém o primeiro *token* do analisador léxico. Para os as técnicas com tabelas, é inicializa a pilha e o método *step* é chamado de dentro de um laço, até que a análise termine, ou um erro seja encontrado. O método *step* executa um passo da análise sintática. Para o descendente recursivo, é chamado o método correspondente ao símbolo inicial da gramática.

### 3.2.3.1 – Analisador LL

Será exibido o método *step*, que executa um passo da análise sintática.

```
    private boolean step() throws AnalysisError
    {
        if (currentToken == null)
        {
            int pos = 0;
            if (previousToken != null)
                pos = previousToken.getPosition() +
previousToken.getLexeme().length();

            currentToken = new Token(DOLLAR, "$", pos);
        }

        int x = ((Integer)stack.pop()).intValue();
        int a = currentToken.getId();

        if (x == EPSILON)
        {
            return false;
        }
        else if (isTerminal(x))
        {
            if (x == a)
            {
                if (stack.empty())
                    return true;
                else
```

```
                {
                    previousToken = currentToken;
                    currentToken = scanner.nextToken();
                    return false;
                }
            }
            else
            {
                throw new SyntaticError(EXPECTED_MESSAGE[x],
currentToken.getPosition());
            }
        }
        else if (isNonTerminal(x))
        {
            if (pushProduction(x, a))
                return false;
            else
                throw new SyntaticError(ERROR_MESSAGE[x-
FIRST_NON_TERMINAL], currentToken.getPosition());
        }
        else if (isSemanticAction(x))
        {
            semanticAnalyser.executeAction(x-FIRST_SEMANTIC_ACTION,
previousToken);
            return false;
        }
        else
            return true;
    }
```

O primeiro passo é verificar se o *token* atual não é *null*. Se for, um *token* de fim de sentença é criado e tomado como atual. Isto é preciso para manter uniforme o tratamento de erros, no caso de erro no fim do programa.

Então o símbolo no topo da pilha é removido e verificado. Se ele for ε, não há mais nada neste passo. Se for um símbolo terminal, ele é confrontado com o *token* atual. Se forem iguais, e a pilha estiver vazia, a análise é terminada com sucesso. Senão mais um *token* da entrada é consumido. Se eles não forem iguais, é lançada uma exceção.

Se o símbolo retirado do topo da pilha for um símbolo não terminal, a tabela de análise é consultada e são empilhados na pilha os símbolos do lado direito da produção correspondente. Se não houver nenhuma produção associada, uma exceção é lançada.

O símbolo do topo da pilha pode ainda ser uma ação semântica. Neste caso o analisador semântico é acionado.

### 3.2.3.2 – Analisador LR

```
    private boolean step() throws AnalysisError
    {
        int state = ((Integer)stack.peek()).intValue();

        if (currentToken == null)
        {
            int pos = 0;
            if (previousToken != null)
                pos = previousToken.getPosition() +
previousToken.getLexeme().length();

            currentToken = new Token(DOLLAR, "$", pos);
        }

        int token = currentToken.getId();
        int[] cmd = SYNT_TABLE[state][token-1];

        switch (cmd[0])
        {
            case SHIFT:
                stack.push(new Integer(cmd[1]));
                previousToken = currentToken;
                currentToken = scanner.nextToken();
                return false;

            case REDUCE:
                int[] prod = PRODUCTIONS[cmd[1]];

                for (int i=0; i<prod[1]; i++)
                    stack.pop();
```

```
                int oldState = ((Integer)stack.peek()).intValue();
                stack.push(new Integer(SYNT_TABLE[oldState][prod[0]-
1][1]));
                return false;

            case ACTION:
                int action = FIRST_SEMANTIC_ACTION + cmd[1] - 1;
                stack.push(new Integer(SYNT_TABLE[state][action][1]));
                semanticAnalyser.executeAction(cmd[1], previousToken);
                return false;

            case ACCEPT:
                return true;

            case ERROR:
                throw new SyntaticError("Erro sintático",
currentToken.getPosition());
        }
        return false;
    }
```

Aqui é feito um processamento para a criação de um *token* fim de arquivo no caso de o *token* atual for *null*, assim como para o analisador LL.

Em seguida é verificada qual a ação associada na tabela com o estado no topo da pilha e o símbolo na entrada. Se o comando for SHIFT (empilha), um novo estado é empilhado, e um símbolo da entrada é consumido. REDUCE (reduzir) faz com que estados sejam desempilhados, e um novo empilhado, de acordo com a produção reduzida.

O comando ACTION é uma adaptação ao algoritmo LR para que este pudesse comportar ações semânticas. Na análise LR as ações semânticas são tratadas como produções, que derivam ε. O comando para ações semânticas é parecido com o de reduzir, a diferença é que neste caso o analisador semântico é acionado.

Por fim, tem-se ACCEPT (aceita), que indica que a análise acabou, e ERROR (erro), indicando um erro sintático.

## 3.3 – Aspectos de Implementação

### *3.3.1 – Processamento da Entrada*

#### 3.3.1.1 – Processamento dos Aspectos Léxicos

A especificação léxica é processada em duas partes: definições léxicas e declaração de *tokens*. Nos dois casos o processamento é semelhante. A entrada é separada por linhas, e cada linha é processada separadamente, de forma a obter o identificador e a expressão regular nela contidos.

As expressões regulares são então processadas individualmente, sendo construído um autômato finito determinístico a partir delas. Neste autômato cada estado final está associado com um *token*. Assim, quando este autômato for utilizado para a análise léxica pode-se saber qual o *token* reconhecido por ele apenas verificando em qual estado ele terminou.

Para o processamento das expressões foi construindo um analisador sintático LL, usando o próprio mecanismo de geração de código desta ferramenta (a geração de analisadores léxicos só foi iniciada após a geração de analisadores sintáticos estar mais avançada). Estas são as produções, já com as ações semânticas embutidas:

```
<exp_reg> ::= <exp_simp> #1 <resto_exp>;

<exp_simp> ::= <termo> #2 <rep_exp_simp>;

<resto_exp> ::= "|" <exp_simp> #3 <resto_exp>
              | î;

<rep_exp_simp> ::= <termo> #4 <rep_exp_simp>
                   | î;

<termo> ::= <fator> <op>;

<op> ::= "*" #5
       | "+" #6
       | "?" #7
       | î;
```

```
<fator> ::= "(" #8 <exp_reg> ")" #9
          | "[" <fim_classe>
          | "." #12
          | DEFINITION #14
          | CHAR #13;

<fim_classe> ::= "^" <item> <resto_classe> "]" #10
               | <item> <resto_classe> "]";

<item> ::= CHAR #13 <resto_intervalo>;

<resto_intervalo> ::= "-" CHAR #15
                    | î;

<resto_classe> ::= <item> <resto_classe> #11
                 | î;
```

A função das ações semânticas é checar erros, e converter a expressão regular em uma árvore. As árvores de cada expressão são então aglutinadas e a partir desta macro-árvore é então obtido o autômato finito.

### 3.3.1.1 – Processamento dos Aspectos Sintáticos

São três os componentes da especificação léxica: símbolos terminais, símbolos não-terminais e produções. O processamento dos dois primeiros é simples, uma vez que basta ler cada linha e guardar seu valor para obter as listas de terminais e não-terminais. Se também houver uma especificação léxica, a lista de terminais é obtida diretamente desta.

As produções são analisadas e delas é obtida uma gramática livre de contexto e a partir dela ser construído o analisador. As produções seguem a seguinte especificação:

```
<LISTA_PROD> ::= <PROD> <RESTO_LISTA_PROD>;

<RESTO_LISTA_PROD> ::= <LISTA_PROD>
                     | î;

<PROD> ::= <NT> #0 "::=" <RHS> #1 <RESTO_PROD> ";" ;
```

```
<RESTO_PROD> ::= "|" <RHS> #1 <RESTO_PROD>
               | î;

<RHS> ::= <T> #2 <RESTO_RHS>
        | <NT> #2 <RESTO_RHS>
        | <ACTION> #3 <RESTO_RHS>;

<RESTO_RHS> ::= <RHS>
              | î;

<T> ::= terminal #4;

<NT> ::= nao_terminal #4;

<ACTION> ::= action #5;
```

Aqui as ações semânticas tem o papel de obter as informações dos símbolos da gramática de entrada e agrupa-los em produções, sempre checando erros, como por exemplo símbolo não declarado.

### *3.3.2 – Realce de Sintáxe*

O editor fornece ao usuário o recurso de realce de sintaxe, muito comum em editores para linguagens de programação, e que o ajuda no entendimento do que está sendo escrito.

Para conseguir isto, é utilizado o próprio analisador léxico interno. Sempre que o texto é alterado, ele é analisado em busca de *tokens*. Para *tokens* diferentes são associados estilos diferentes, como uma cor, ou negrito. Durante este processamento os estilos do texto são alterados, indicando cada símbolo no texto com um estilo. Para melhorar a performance deste processamento, apenas a linha que foi modificada é analisada, uma vez que as demais não precisam de alterações.

# 4 – Conclusões

## 4.1 – Considerações Finais

Apesar deste trabalho não contribuir com nenhuma inovação na área de linguagens formais e compiladores, ele produziu uma ferramenta de grande valia, que permite a construção de analisadores léxicos e sintáticos, para compiladores ou qualquer outra atividade que precise de técnicas de análise léxica e sintática. Também promove o aprendizado das técnicas formais de análise, uma vez que o usuário pode ver as tabelas de análise e fazer simulações, visando validar sua especificação e com isso assimilando melhor os conceitos e métodos formais que compões a teoria das linguagens formais.

## 4.2 – Trabalhos Futuros

Como trabalhos futuros podem ser citados:

- Concluir a geração das tabelas para as técnicas LR Canônica e LALR.
- Permitir o uso de contextos de fim de expressão regular, permitindo que um *token* seja identificado apenas se seguido por uma expressão válida.
- Prover técnicas para a compressão das tabelas de análise, que em geral são muito esparsas.
- Construção de um simulador mais avançado, que permita a simulação passo a passo do analisador.
- Permitir a descrição das mensagens de erro diretamente de dentro da especificação.

# Bibliografia:


AHO, A. V., SETHI, R. e ULLMAN, J. D., **Compilers: Principles, Techniques and Tools**. Menlo Park: Addison-Wesley, 1986.

BARRET, W. A., COUCH, J. D., **Compiler Construction: Theory and Practice**. Chicago: Science Research Associates, 1979.

DEITEL, H. M., DEITEL, P. J., **Java, como Programar**. Porto Alegre: Bookman, 2001.

FISCHER, C. N., LeBLANC, R.J., **Crafting a Compiler**. Menlo Park: The Benjamin/Cummings Publishing, 1988.

FURTADO, O. J. V., **Apostila de Linguagens Formais e Compiladores**. Florianópolis: UFSC, 1992.

MORAIS, A. F., ROHDEN, R. **BRAIN: GERADOR DE ANALISADORES SINTÁTICOS**. Florianópolis, 2000.

PRICE, A. M. A., TOSCANI, S. S., **Implementação de Linguagens de Programação**. Porto Alegre: Sagra Luzzatto, 2001.

SCHEFFEL, R. M., **Apostila Compiladores I**. UNISUL.

SEBESTA, R. W. **Conceitos de Linguagens de Programação**. Porto Alegre: Bookman, 2000.

TRAMONTIN Jr., R. J. AMBIENTE PARA ENSINO E DESENVOLVIMENTO DE COMPILADORES: GERADOR DE ANALISADORES LÉXICOS. Florianópolis, 2000.

TREMBLAY, J. P., SORENSON, P. G., The Theory and Practice of Compiler Writing. New York: McGraw Hill, 1985.

The Lex & Yacc Page [on line]. Documento disponível na internet via WWW: <URL: http://www.combo.org/lex_yacc_page/>

Anexo I – Artigo

# GALS:
# Gerador de Analisadores Léxicos e Sintáticos

**Carlos Eduardo Gesser**
Curso de Ciências da Computação
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
gesser@inf.ufsc.br

## Resumo

GALS é uma ferramenta para geração automática de analisadores léxicos e sintáticos, duas importantes fases do processo de compilação. Foi desenvolvida para ser uma ferramenta com propósitos didáticos, mas com características profissionais, podendo ser utilizada tanto no auxílio aos alunos da cadeira de Construção de Compiladores como possivelmente em outros projetos que necessitem processamento de linguagens.

## Abstract

GALS is a tool for automatic generation of lexical and sintatical analysers, two important phases of the compiling process. It was developed to be a tool with didactic pourposes, but with professional caracteristics, as it can be used to help compilers theory students and there is the possibility to be used any projects witch requires language processing.

***Palavras Chave:***

Linguagens Formais, Compiladores, Gramáticas Regulares, Expressões Regulares, Autômatos Finitos, Gramáticas Livres de Contexto, Autômatos de Pilha, Analise Léxica, Análise Sintática.

# 1 – Introdução

Analisadores Léxicos e Sintáticos são importantes partes de um compilador. Sua construção sem a utilização de ferramentas de auxílio pode levar a erros no projeto, além de ser um trabalho desnecessário.

Existem hoje diversas ferramentas comerciais semelhantes, porém a maioria delas gera apenas ou o Analisador Léxico ou o Sintático, o que obriga o usuário a conhecer duas ferramentas distintas e leva a incompatibilidades no momento da utilização em conjunto dos analisadores gerados. Muitas ferramentas geram analisadores segundo uma única técnica de análise sintática., o que limita seu valor didático. Praticamente todas geram analisadores em apenas uma linguagem objeto, diminuindo sua flexibilidade. Outro fator agravante é o fato de serem ferramentas de código fechado, o que impede que se possa provar formalmente o analisador gerado, já que não se tem acesso a seu código fonte.

Também existem ferramentas livres (gratuitas e de código aberto) para esta tarefa. A maioria apresenta as mesmas limitações citadas acima. Mas, por se tratarem de *software*, livre têm a vantagem de poderem ter seu código-fonte analisado, verificando-se assim se o que fazem é realmente aquilo que se propõe a fazer.

O GALS tenta superar estes problemas e se tornar uma ferramenta prática e versátil, com aplicabilidade tanto como ferramenta didática como para uso profissional, promovendo assim o uso das técnicas formais de análise.

# 2 – Fundamentação Teórica

O processo de compilação é dividido em duas etapas: análise e síntese. Na primeira etapa é verificada a coerência do programa com relação à linguagem na qual ele esta escrito e a obtenção de informações necessárias para a etapa seguinte. Na etapa de síntese são concebidos os resultados da compilação, ou seja, é gerado o código do programa. Esta geração pode ser direta, ou com a geração de código intermediário. Em geral também são feitas otimizações no código gerado nesta etapa.

A etapa de análise é subdividida em três fases: análise léxica, análise sintática e análise semântica. Cada uma dessas fases se atém a um aspecto (léxico, sintático e semântico respectivamente) e verifica se o programa sendo processado (compilado) está de acordo com a especificação da linguagem correspondente para cada um destes aspectos.

Linguagens são especificadas formalmente através de gramáticas, ou mecanismos equivalentes. As gramáticas foram classificadas por Chomsky em quatro tipos: gramáticas sem restrições (tipo 0), gramáticas sensíveis ao contexto (tipo 1), gramáticas livres de contexto (tipo 2) e gramáticas regulares (tipo 3) [4]. Esta classificação foi feita levando em conta restrições impostas a cada tipo em relação à forma de suas produções, sendo as regulares as gramáticas mais restritas, e ao mesmo tempo as mais facilmente e eficientemente processáveis.

Aspectos léxicos de uma linguagem são definidos por gramáticas regulares ou expressões regulares, que são um formalismo equivalente. Para a definição dos aspectos sintáticos são utilizadas gramáticas livres de contexto, que são capazes de representar as construções sintáticas usuais das linguagens de programação e que gramáticas regulares não conseguem especificar. Os aspectos semânticos são muito mais complexos, o que faz com que em geral sejam usados, na prática, mecanismos semiformais.

Para a implementação de analisadores léxicos são utilizados autômatos finitos [1], mecanismos reconhecedores de linguagens regulares. O analisador léxico lê o programa fonte como um fluxo de caracteres e os agrupa em *tokens* para o analisador sintático. Nem todos os caracteres da entrada geram *tokens*. Caracteres como espaço em branco, tabulação e quebra de linha, e seqüências de caracteres que indicam comentários por exemplo, não contém nenhuma informação relevante para o processo de análise e são ignorados pelo analisador léxico. Caracteres inválidos, ou que não gerem um *token* válido fazem com que o analisador léxico detecte o erro, que é reportado como erro léxico.

Para a análise sintática existem duas classes de analisadores: a ascendente (ou *bottom-up*) e a descendente (ou *top-down*). A principal diferença entre as duas está no modo como elas tentam validar a entrada com relação à gramática usada para a especificação sintática da uma linguagem. As técnicas ascendentes partem da sentença (programa fonte) e, fazendo uma séria de reduções, tentam chegar ao símbolo inicial da

gramática. As descendentes fazem o caminho inverso: começam pelo símbolo inicial e, através de derivações tentam chegar até a sentença. Outra diferença é na relação capacidade *versus* recursos computacionais entre as diversas técnicas das duas classes de analisadores. Nenhuma das técnicas determinísticas existentes é capaz de reconhecer todas as gramática livre de contexto, porém as técnicas ascendentes são menos restritivas. Diversas gramáticas que são facilmente reconhecíveis por analisadores ascendentes não o são por analisadores descendentes. Por outro lado, a construção de analisadores descendentes é menos complexa e requer menos recursos computacionais, muito embora a performance de ambos é equivalente no que diz respeito ao tempo de análise. Como exemplo de técnicas de análise sintática pertencentes à classe de analisadores sintáticos ascendente pode-se citar as técnicas da Família LR (SLR, LALR e LR Canônico). Como técnicas descendentes pode-se citar: Descendente Recursivo e Preditivo Tabular(LL).

Uma implementação comum para a análise semântica é conhecida como Esquema de Tradução Dirigida pela Sintaxe. Nesta técnica são inseridas dentro da especificação sintática ações semânticas, que são funções que compõe o analisador semântico. Quando o analisador sintático atinge uma destas ações ele aciona o analisador semântico. Cabe ao analisador semântico verificar aspectos semânticos, como verificação de número de parâmetros em funções, validade de identificadores, coerência de tipos, além de colher informações que permitam o estabelecimento de um significado para o programa fonte através do código (real o intermediário) gerado

# 3 – Geradores de Analisadores Léxicos e Sintáticos

Existem ferramentas que podem gerar analisadores léxicos e sintáticos automaticamente, a partir de especificações formais. Em geral são utilizadas expressões regulares para a especificação léxica e gramáticas livres de contexto para a especificação sintática.

Existem diversas ferramentas, para as mais diversas linguagens. Aqui serão apresentadas duas das mais conhecidas e largamente utilizadas: Lex e Yaac [5].

## *Lex*

O *LEX* é um dos mais tradicionais geradores de analisadores léxicos. Foi desenvolvido por Lesk e Schmidt em 1975. Trata-se originalmente de um programa UNIX que lê uma especificação léxica formada de expressões regulares e gera uma rotina de análise léxica para linguagem C. Existem atualmente versões para outros sistemas e que geram analisadores para outras linguagens.

A especificação de entrada é formada de expressões regulares e comandos em linguagem de programação. O analisador é gerado de forma que quando uma dada expressão é reconhecida, os comandos associados com ela são executados. Isto pode facilitar o trabalho a princípio, mas juntar a especificação com o código objeto torna a leitura da primeira mais difícil. Além de que para se utilizar esta especificação para gerar um analisador em uma outra linguagem é preciso altera-la.

Por ser uma ferramenta muito conhecida e utilizada, é fácil encontrar documentação sobre ela para aprender como usa-la. Porém é preciso que o usuário esteja familiarizado com ferramentas de linha de comando. Além disso, visando ser genérica acabou tornando-se muito complexa e de difícil aprendizado.

## *Yacc*

O YACC, iniciais de *Yet Another Compiler-Compiler*, é um gerador de Analisadores Sintáticos que roda no ambiente UNIX. Escrito por Johnson em 1975. Aceita como entrada uma especificação das características sintáticas da linguagem, com ações semânticas embutias, e gera uma rotina em C para a análise sintática.

A saída do YACC consta de uma tabela de análise sintática LALR, uma rotina de controle, que executa em cima da tabela, e as ações semânticas, que foram especificadas durante a definição da linguagem no arquivo de entrada.

O analisador gerado é feito de modo a trabalhar em conjunto com uma rotina de análise léxica gerada pelo LEX, permitindo uma fácil integração entre os analisadores léxico (gerado pelo LEX), sintático e semântico (gerados pelo YACC). Todos são gerados no mesmo ambiente (UNIX) e na mesma linguagem (C).

A especificação é constituída de uma gramática livre de contexto. No fim de cada produção pode ser escrita a ação a ser tomada quando a produção for identificada. A ação é escrita diretamente na linguagem de implementação, o que dificulta o uso de uma

mesma especificação para a geração de compiladores em diferentes linguagens. Para que se possa utilizar o YACC é preciso que o usuário tenha experiência com o uso de programas de linha de comando.

# 4 – GALS

GALS é uma ferramenta para a geração automática de analisadores léxicos e sintáticos. Foi desenvolvido em Java, versão 1.4, podendo ser utilizado em qualquer ambiente para o qual haja uma máquina virtual Java. É possível gerar-se analisadores léxicos e sintáticos, através de especificações léxicas, baseadas em expressões regulares, e especificações sintáticas, baseadas em gramáticas livres de contexto. Pode-se fazer os analisadores léxico e sintático independentes um do outro, bem como fazer de maneira integrada. Existem três opções de linguagem para a geração dos analisadores: Java, C++ e Delphi. O GALS tenta ser um ambiente amigável, tanto para uso didático como para o uso profissional.

O uso do GALS se dá através de seu ambiente, onde o usuário especifica os aspectos léxicos e sintáticos de seus analisadores. O ambiente também fornece uma série de ferramentas como:

- Funções de arquivo – para salvar e abrir especificações e importar arquivos em formato BNF, gerados pelo GAS[1].
- Documentação – permite exibir tabelas geradas para os analisadores léxicos e sintáticos no formato HTML.
- Configuração – possibilita definir as opções do projeto, como linguagem objeto, técnica empregada para o analisador sintático, nomes das classes geradas, etc.
- Simulação dos Analisadores – que permite ao usuário testar sua especificação sem a necessidade de gerar código efetivamente.

[1] Gerador de analisadores sintáticos didático, desenvolvido na década de noventa por um aluno da UFSC

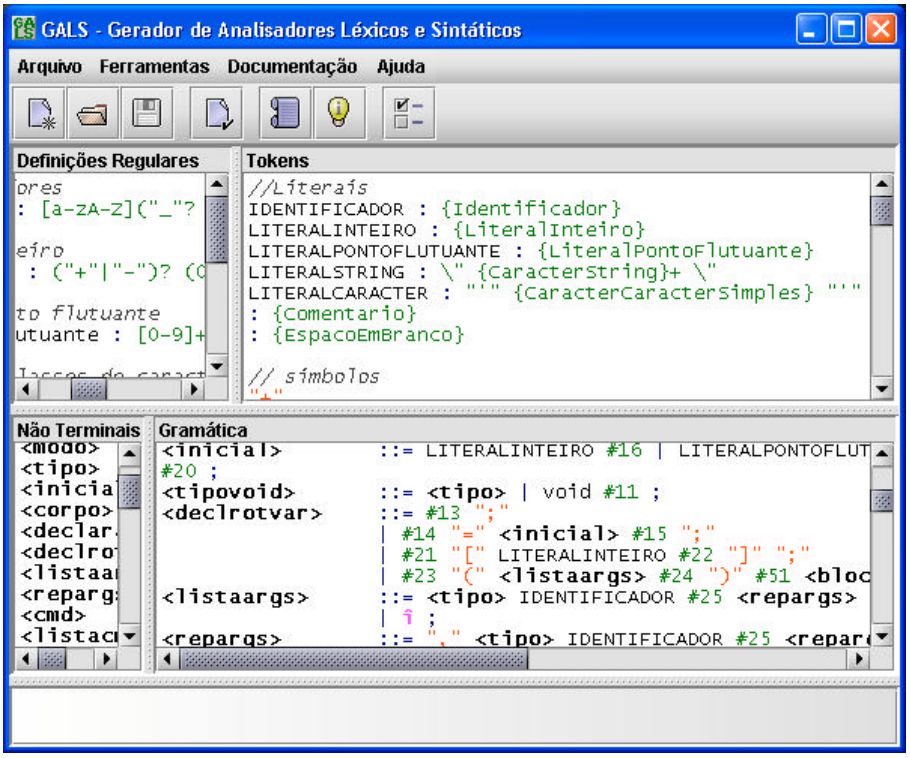



**Figura 2 – Tela Principal**

A especificação dos aspectos léxicos é uma declaração dos *tokens* que devem ser aceitos, esta declaração é feita atrtavés do uso de expressões regulares. Pode-se ainda fazer a definição prévia de expressões para uso posterior, o que facilita casos com expressões mais complexas. Os *tokens* podem ser declarados de diferentes formas: fornecendo um identificador e uma expressão regular, através de uma seqüência de caracteres entre aspas (esta seqüência identifica o *token*), ou ainda identificando um *token* como um caso particular de um outro já declarado, deste modo é construída uma tabela onde sempre que o *token* base for identificado, é verificado se ele não se trata de um dos casos especiais. Há ainda a declaração de expressões regulares que, quando identificadas são descartadas pelo analisador gerado; desta forma pode-se definir expressões para comentários, espaços em branco e outros que se deseje que sejam ignorados. O analisador gerado baseia-se em um autômato finito, com uma tabela de transições, ou com implementação específica. O analisador pode obter a entrada de uma *string* ou através de uma *stream*.

A especificação sintática se dá na forma de uma gramática livre de contexto, com a possibilidade da inserção de ações semânticas para permitir a análise semântica através de tradução dirigida pela sintaxe. Os símbolos não terminais são declarados explicitamente, assim como os terminais, exceto no caso de ser uma especificação léxica e sintática conjunta, onde os *tokens* do analisador léxico são considerados como símbolos terminais. Para o analisador gerado podem ser escolhidas cinco diferentes técnicas: Descendente Recursivo, LL, SLR, LALR e LR Canônica. No caso das técnicas descendentes, é verificado se a gramática não fere nenhuma das restrições impostas para que uma gramática seja LL(1) [3].

O código gerado é um conjunto de classes e tabelas. É gerada uma classe para cada analisador (léxico, sintático e semântico), classes auxiliares e classes de exceções. A classe gerada para o analisador semântico não possui implementação, apenas a interface que o analisador sintático espera. Cabe ao usuário inserir sua implementação.

# 5 – Conclusões

Apesar deste trabalho não propor inovações na área de linguagens formais e compiladores, ele contribui significativamente para melhoria das condições de ensino e desenvolvimento de compiladores. A ferramenta desenvolvida é de grande valia, permitindo a construção de analisadores léxicos e sintáticos, para compiladores ou qualquer outro sistema que precise realizar algum tipo de reconhecimento através das técnicas de análise léxica e sintática disponíveis . Também promove o aprendizado das técnicas formais de análise, uma vez que o usuário pode ver as tabelas de análise e fazer simulações, visando validar sua especificação; podendo assim, assimilar melhor os conceitos e métodos formais que compões a teoria das linguagens formais e a teoria de compiladores.

# 6 – Referências Bibliográficas

[1] AHO, A. V., SETHI, R. e ULLMAN, J. D., **Compilers: Principles, Techniques and Tools**. Menlo Park: Addison-Wesley, 1986.

[2] FISCHER, C. N., LeBLANC, R.J., **Crafting a Compiler**. Menlo Park: The Benjamin/Cummings Publishing, 1988.

[3] FURTADO, O. J. V., **Apostila de Linguagens Formais e Compiladores**. Florianópolis: UFSC, 1992.

[4] SCHEFFEL, R. M., **Apostila Compiladores I**. UNISUL.

[5] The Lex & Yacc Page [on line]. Documento disponível na internet via WWW: <URL: http://www.combo.org/lex_yacc_page/>

# Anexo II – Código Fonte

```
package gesser.gals;

import gesser.gals.generator.Options;
import gesser.gals.generator.OptionsDialog;
import gesser.gals.generator.parser.Grammar;
import gesser.gals.generator.scanner.FiniteAutomata;
import gesser.gals.util.LexSyntData;
import gesser.gals.util.MetaException;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.List;

import javax.swing.*;

import com.incors.plaf.kunststoff.KunststoffLookAndFeel;

public class MainWindow extends JFrame
{
	private static final ImageIcon GALS = new
ImageIcon(ClassLoader.getSystemResource("icons/gals.gif"));

	private static MainWindow singleton = null;

	public static MainWindow getInstance()
	{
		if (singleton == null)
		{
			singleton = new MainWindow();
		}
		return singleton;
	}

	JToolBar toolbar = new JToolBar();

	private JMenuBar menuBar = new JMenuBar();
	private JMenu grammar = new JMenu("Verificar");
	private JMenuItem lexTable = new JMenuItem(Actions.viewLexTable);
	private JMenuItem syntTable = new JMenuItem(Actions.showTable);
	private JMenuItem ff = new JMenuItem(Actions.ff);
	private JMenuItem useless = new JMenuItem(Actions.useless);
	private JMenuItem itemSet = new JMenuItem(Actions.showItemSet);

	private InputPane inPane = new InputPane();

	private boolean needRebuildFA = true;
	private boolean needRebuildGram = true;
	private FiniteAutomata fa = null;
	private Grammar gram = null;

	public void setChanged()
	{
		needRebuildFA = true;
		needRebuildGram = true;
	}

	private MainWindow()
	{
		super ("GALS - Gerador de Analisadores Léxicos e Sintáticos");

		setIconImage(GALS.getImage());

		createMenu();
```

```
            createToolBar();

            getContentPane().add(toolbar, BorderLayout.NORTH);
            getContentPane().add(inPane);



        pack();
        setSize(600,500);

        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e)
                {
                    Actions.close.actionPerformed(null);
                }
        });

            grammar.setVisible(false);
    }

    public Grammar getGrammar() throws MetaException
    {
            if (needRebuildGram || gram == null)
            {
                gram = inPane.getGrammar();
                needRebuildGram = false;
            }
            return gram;
      }

public FiniteAutomata getFiniteAutomata() throws MetaException
{
    if (needRebuildFA || fa == null)
    {
            fa = inPane.getFiniteAutomata();
            needRebuildFA = false;
    }
    return fa;
}

public Options getOptions()
{
    return OptionsDialog.getInstance().getOptions();
}

    public void updateData(LexSyntData lsd)
    {
            reset();

            if (lsd.getOptions() != null)
            {
                OptionsDialog.getInstance().setOptions(lsd.getOptions());
                setPanesMode(OptionsDialog.getInstance().getMode());
            }

            inPane.setData(lsd.getData());
    }

    public InputPane.Data getData()
    {
            return inPane.getData();
    }

    public void setData(InputPane.Data data)
    {
            inPane.setData(data);
    }

    public void reset()
    {
```

```
            inPane.reset();
            setPanesMode(OptionsDialog.getInstance().getMode());
    }

    public List getTokens() throws MetaException
    {
            List tokens = inPane.getTokens();

            while ( tokens.remove("\n") )
                    ;

            return tokens;
    }

    public void handleError(MetaException e)
    {
            inPane.handleError(e);
    }

    private void createMenu()
{
    JMenu file = new JMenu("Arquivo");

    file.add(new JMenuItem(Actions.new_));
    file.add(new JMenuItem(Actions.load));
    file.addSeparator();
    file.add(new JMenuItem(Actions.save));
    file.add(new JMenuItem(Actions.saveAs));
    file.addSeparator();
            file.add(new JMenuItem(Actions.importGAS));
            file.addSeparator();
    file.add(new JMenuItem(Actions.close));

    JMenu tools = new JMenu("Ferramentas");

    tools.add(new JMenuItem(Actions.verify));
            tools.add(new JMenuItem(Actions.genCode));
            tools.addSeparator();
            tools.add(new JMenuItem(Actions.simulate));
    tools.add(grammar);
            tools.add(useless);
            tools.addSeparator();
            tools.add(new JMenuItem(Actions.options));

    grammar.add(new JMenuItem(Actions.factored));
    grammar.add(new JMenuItem(Actions.recursion));
    grammar.add(new JMenuItem(Actions.condition3));

    JMenu doc = new JMenu("Documentação");

    doc.add(lexTable);
    doc.add(syntTable); //Sintatico - LL(1)
            doc.add(itemSet);     //Sintatico - LR(1)
    doc.add(ff);     //Sintatico

    /*
    JMenu transform = new JMenu("Transformações");

    transform.add(new JMenuItem(Actions.undo));
    Actions.undo.setEnabled(false);
    transform.addSeparator();
    transform.add(Actions.factorate);
    transform.add(Actions.removeRecursion);
    transform.add(Actions.removeUnitary);
    transform.add(Actions.removeUseless);
    transform.add(Actions.removeEpsilon);*/

            JMenu help = new JMenu("Ajuda");

    help.add(Actions.about);
```

```
                menuBar.add(file);
                //menuBar.add(transform);
                menuBar.add(tools);
                menuBar.add(doc);
                menuBar.add(help);

            setJMenuBar(menuBar);
        }

            private void createToolBar()
            {
                    toolbar.setFloatable(false);

                    toolbar.add(Actions.new_);
                    toolbar.add(Actions.load);
                    toolbar.add(Actions.save);

                    toolbar.addSeparator();

                    toolbar.add(Actions.verify);

                    toolbar.addSeparator();

                    toolbar.add(Actions.simulate);
                    toolbar.add(Actions.genCode);

                    toolbar.addSeparator();

                    toolbar.add(Actions.options);
            }

            public void setPanesMode(int mode)
            {
                    inPane.setMode(mode);

                    boolean lex  = mode == InputPane.LEXICAL || mode == InputPane.BOTH;
                    boolean synt = mode == InputPane.SYNTATIC || mode == InputPane.BOTH;
                    int type = OptionsDialog.getInstance().getOptions().parser;
                    boolean ll = type == Options.PARSER_LL || type == Options.PARSER_REC_DESC;

                    lexTable.setVisible(lex);

                    syntTable.setVisible(synt);
                    ff.setVisible(synt);
                    grammar.setVisible(synt && ll);

                    itemSet.setVisible( ! lex && ! ll);
            }

            public static void centralize(Component c)
            {
                    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
                    Point center = new Point(d.width/2, d.height/2);
                    c.setLocation(center.x-c.getWidth()/2, center.y-c.getHeight()/2);
            }

            public static void main(String[] args)
        {
            try
            {
                com.incors.plaf.kunststoff.KunststoffLookAndFeel kunststoffLnF = new
com.incors.plaf.kunststoff.KunststoffLookAndFeel();
                KunststoffLookAndFeel.setCurrentTheme(new
com.incors.plaf.kunststoff.KunststoffTheme());
                UIManager.setLookAndFeel(kunststoffLnF);
            }
            catch (javax.swing.UnsupportedLookAndFeelException e)
            {
                e.printStackTrace();
                System.exit(1);
            }
```

```
        MainWindow window = getInstance();
        centralize(window);

        window.show();
    }
}


package gesser.gals;

import gesser.gals.analyser.AnalysisError;
import gesser.gals.editor.BNFDocument;
import gesser.gals.editor.DefinitionsDocument;
import gesser.gals.editor.NTDocument;
import gesser.gals.editor.TokensDocument;
import gesser.gals.generator.OptionsDialog;
import gesser.gals.generator.parser.Grammar;
import gesser.gals.generator.scanner.FiniteAutomata;
import gesser.gals.parserparser.Parser;
import gesser.gals.scannerparser.LineParser;
import gesser.gals.util.MetaException;

import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

import javax.swing.*;
import javax.swing.event.UndoableEditEvent;
import javax.swing.event.UndoableEditListener;
import javax.swing.text.Document;
import javax.swing.text.StyledEditorKit;

/**
 * @author Gesser
 */

public class InputPane extends JPanel implements MouseListener, UndoableEditListener
{
        public static final int LEXICAL  = 0;
        public static final int SYNTATIC = 1;
        public static final int BOTH     = 2;

        private JEditorPane grammar      = new JEditorPane();
        private JEditorPane nonTerminals = new JEditorPane();
        private JEditorPane tokens       = new JEditorPane();
        private JEditorPane definitions  = new JEditorPane();

        private JPanel base = new JPanel(new BorderLayout());
        private JList errorList = new JList();

        private JPanel pnlGrammar            = createPanel(" Gramática", grammar, new
BNFDocument());
        private JPanel pnlNonTerminals = createPanel(" Não Terminais", nonTerminals, new
NTDocument());
        private JPanel pnlTokens             = createPanel(" Tokens", tokens, new
TokensDocument());
        private JPanel pnlDefinitions  = createPanel(" Definições Regulares", definitions,
new DefinitionsDocument());

        private int mode;

        public InputPane()
        {
                super (new BorderLayout());
```

```
		JSplitPane split = new JSplitPane(JSplitPane.VERTICAL_SPLIT, base, new
JScrollPane(errorList));
		split.setResizeWeight(0.9);
		add(split);

		setMode(BOTH);

		split.setDividerLocation(355);

		errorList.addMouseListener(this);
	}

	public Grammar getGrammar() throws MetaException
	{
		errorList.setListData(new Object[]{});

		if (mode != SYNTATIC && mode != BOTH)
			return null;

		List tokens = getTokens();
		List nt = new ArrayList();

		StringTokenizer ntTok = new StringTokenizer(nonTerminals.getText(), "\n",
true);
		while (ntTok.hasMoreTokens())
			nt.add(ntTok.nextToken());

		String gram = grammar.getText();


		Grammar g = new Parser().parse(tokens, nt, gram);
		return g;
	}

	public FiniteAutomata getFiniteAutomata() throws MetaException
	{
		errorList.setListData(new Object[]{});

		if (mode != LEXICAL && mode != BOTH)
			return null;

		LineParser lp = new LineParser();

		return lp.parseFA(definitions.getText(), tokens.getText());
	}

	public List getTokens() throws MetaException
	{
		List result = new ArrayList();
		if (mode == BOTH || mode == LEXICAL)
		{
			List tokens = getFiniteAutomata().getTokens();
			for (int i=0; i<tokens.size(); i++)
			{
				result.add(tokens.get(i));
				result.add("\n");
			}
		}
		else //mode == SYNTATIC
		{
			StringTokenizer tknzr = new StringTokenizer(tokens.getText(), "\n",
true);

			while (tknzr.hasMoreTokens())
				result.add(tknzr.nextToken());
		}
		return result;
	}

	public Data getData()
	{
```

```
                boolean lex = mode == LEXICAL || mode == BOTH;
                boolean synt = mode == SYNTATIC || mode == BOTH;

                return
                        new Data(
                                lex ? definitions.getText() : "",
                                tokens.getText(),
                                synt ? nonTerminals.getText() : "",
                                synt ? grammar.getText() : "");
        }

        public void setData(Data d)
        {
                definitions.setText(d.getDefinitions());
                tokens.setText(d.getTokens());
                nonTerminals.setText(d.getNonTerminals());
                grammar.setText(d.getGrammar());

                definitions.setCaretPosition(0);
                tokens.setCaretPosition(0);
                nonTerminals.setCaretPosition(0);
                grammar.setCaretPosition(0);

                MainWindow.getInstance().setChanged();
        }

        public void reset()
        {
                grammar.setText("");
                nonTerminals.setText("");
                tokens.setText("");
                definitions.setText("");
                OptionsDialog.getInstance().reset();
                MainWindow.getInstance().setChanged();
        }

        public void setMode(int mode)
        {
                this.mode = mode;

                switch (mode)
                {
                        case LEXICAL:  setLex(); break;
                        case SYNTATIC: setSynt(); break;
                        case BOTH:     setBoth(); break;
                }
        }

        private void setBoth()
        {
                base.removeAll();

                JSplitPane top = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
pnlDefinitions, pnlTokens);

                top.setResizeWeight(0.25);

                JSplitPane bottom = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
pnlNonTerminals, pnlGrammar);

                bottom.setResizeWeight(0.15);

                JSplitPane main = new JSplitPane(JSplitPane.VERTICAL_SPLIT, top, bottom);

                main.setResizeWeight(0.5);

                base.add(main);

                validate();
                repaint();
```

```
                top.setDividerLocation(0.25);
                bottom.setDividerLocation(0.15);
                main.setDividerLocation(0.5);
        }

        private void setSynt()
        {
                base.removeAll();

                JSplitPane left = new JSplitPane(JSplitPane.VERTICAL_SPLIT, pnlTokens,
pnlNonTerminals);

                left.setResizeWeight(0.5);

                JSplitPane main = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, left,
pnlGrammar);

                main.setResizeWeight(0.15);

                base.add(main);

                validate();
                repaint();

                left.setDividerLocation(0.5);
                main.setDividerLocation(0.15);
        }


        private void setLex()
        {
                base.removeAll();

                JSplitPane main = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
pnlDefinitions, pnlTokens);

                main.setResizeWeight(0.25);

                base.add(main);

                validate();
                repaint();

                main.setDividerLocation(0.25);
        }

        public void undoableEditHappened(UndoableEditEvent e)
        {
                Actions.UNDO_MAN.addEdit(e.getEdit());
                Actions.setSaved(false);
                MainWindow.getInstance().setChanged();
        }

        private JPanel createPanel(String caption, final JEditorPane comp, Document doc)
        {
                JPanel pnl = new JPanel(new BorderLayout());

                comp.setEditorKit(new StyledEditorKit());
                comp.setDocument(doc);

                comp.getDocument().addUndoableEditListener(this);

        comp.getKeymap().addActionForKeyStroke((KeyStroke)Actions.undo.getValue(Action.ACC
ELERATOR_KEY), Actions.undo);

        comp.getKeymap().addActionForKeyStroke((KeyStroke)Actions.redo.getValue(Action.ACC
ELERATOR_KEY), Actions.redo);

                pnl.add(new JLabel(caption), BorderLayout.NORTH);

                JPanel tmp = new JPanel(new BorderLayout());
```

```
        tmp.add(comp);

        JScrollPane scroll = new JScrollPane(tmp);
        scroll.getVerticalScrollBar().setUnitIncrement(10);
        scroll.getHorizontalScrollBar().setUnitIncrement(10);
        pnl.add(scroll);

        return pnl;
}


private static class ErrorData
{
        int index, position, mode;
        String message;

        ErrorData(String message, int index, int position, int mode)
        {
                this.message = message;
                this.index = index;
                this.position = position;
                this.mode = mode;
        }

        public String toString()
        {
                return message /*+ ", linha: "+index+", coluna "+position*/;
        }
}
public void handleError(MetaException e)
{
        AnalysisError ae = (AnalysisError) e.getCause();
        int line = e.getIndex();
        String msg = "";
        switch (e.getMode())
        {
                case MetaException.DEFINITION :
                        msg = "Erro em Definição Regular: ";
                        break;
                case MetaException.TOKEN :
                        msg = "Erro na Especificação de Tokens: ";
                        break;
                case MetaException.NON_TERMINAL :
                        msg = "Erro na Declaração dos Não-Terminais: ";
                        break;
                case MetaException.GRAMMAR :
                        msg = "Erro na Especificação da Gramática: ";
                        break;
        }
        msg += ae.getMessage();

        ErrorData ed = new ErrorData(msg, line, ae.getPosition(), e.getMode());

        ErrorData[] errors = {ed};

        errorList.setListData(errors);
        e.printStackTrace();
        Toolkit.getDefaultToolkit().beep();
        mouseClicked(null);
}

public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}

public void mouseClicked(MouseEvent e)
{
        ErrorData error = (ErrorData) errorList.getSelectedValue();

        if (error != null)
```

```
		{
			switch (error.mode)
			{
				case MetaException.DEFINITION:
					setPosition(definitions, error.index,
error.position);
					definitions.requestFocus();
					break;
				case MetaException.TOKEN :
					setPosition(tokens, error.index, error.position);
					tokens.requestFocus();
					break;
				case MetaException.NON_TERMINAL :
					setPosition(nonTerminals, error.index,
error.position);
					nonTerminals.requestFocus();
					break;
				case MetaException.GRAMMAR :
					grammar.getCaret().setDot(error.position);
					grammar.requestFocus();
					break;
			}
		}
	}

	private void setPosition(JEditorPane pane, int line, int col )
	{
		String text = pane.getText();
		int pos = 0;
		int strpos = 0;
		while (line>0)
		{
			while (strpos < text.length() && text.charAt(strpos) != '\n')
			{
				if (text.charAt(strpos) != '\r')
					pos++;
				strpos++;
			}
			strpos++;
			pos++;
			line--;
		}
		pos += col;
		pane.setCaretPosition(pos);
	}

	public static class Data
	{
		private String definitions = "";
		private String tokens = "";
		private String nonTerminals = "";
		private String grammar = "";

		public Data(String definitions, String tokens, String nonTerminals, String
grammar)
		{
			this.definitions = definitions;
			this.tokens = tokens;
			this.nonTerminals = nonTerminals;
			this.grammar = grammar;
		}

		public String getDefinitions()
		{
			return definitions;
		}

		public String getGrammar()
		{
			return grammar;
		}
```

```
		public String getNonTerminals()
		{
			return nonTerminals;
		}

		public String getTokens()
		{
			return tokens;
		}

		public void setGrammar(String string)
		{
			this.grammar = string;
		}

		public void setNonTerminals(String string)
		{
			this.nonTerminals = string;
		}
	}
}
package gesser.gals.editor;

import java.awt.Color;
import java.awt.Font;

import javax.swing.event.UndoableEditListener;
import javax.swing.text.*;

public abstract class SyntaxDocument extends DefaultStyledDocument
{
	public static final Font FONT = new Font("Lucida Console",Font.PLAIN, 13);

	protected static final SimpleAttributeSet NORMAL = new SimpleAttributeSet();
	static
	{
		StyleConstants.setForeground(NORMAL, Color.BLACK);
		StyleConstants.setBackground(NORMAL, Color.WHITE);
		StyleConstants.setFontFamily(NORMAL, FONT.getFamily());
		StyleConstants.setFontSize(NORMAL, FONT.getSize());
		StyleConstants.setBold(NORMAL, false);
		StyleConstants.setItalic(NORMAL, false);
	}

	protected static final SimpleAttributeSet STRING = new SimpleAttributeSet(NORMAL);
	protected static final SimpleAttributeSet OPERATOR = new
SimpleAttributeSet(NORMAL);
	protected static final SimpleAttributeSet REG_EXP = new
SimpleAttributeSet(NORMAL);
	protected static final SimpleAttributeSet ERROR = new SimpleAttributeSet(NORMAL);
	protected static final SimpleAttributeSet COMMENT = new
SimpleAttributeSet(NORMAL);


	private static void initAttributes()
	{
		StyleConstants.setBackground(REG_EXP, Color.WHITE);
		StyleConstants.setForeground(REG_EXP, new Color(0, 128, 0));
		StyleConstants.setBold(REG_EXP, false);
		StyleConstants.setItalic(REG_EXP, false);

		StyleConstants.setBackground(STRING, Color.WHITE);
		StyleConstants.setForeground(STRING, Color.RED);
		StyleConstants.setBold(STRING, false);
		StyleConstants.setItalic(STRING, false);

		StyleConstants.setBackground(OPERATOR, Color.WHITE);
		StyleConstants.setForeground(OPERATOR, new Color(0, 0, 128));
		StyleConstants.setBold(OPERATOR, false);
		StyleConstants.setItalic(OPERATOR, false);
```

```
                /*
                StyleConstants.setBackground(ERROR, new Color(255, 32, 32));
                StyleConstants.setForeground(ERROR, Color.WHITE);
                StyleConstants.setBold(ERROR, true);
                StyleConstants.setItalic(ERROR, false);
                */
                StyleConstants.setBackground(COMMENT, Color.WHITE);
                StyleConstants.setForeground(COMMENT, Color.DARK_GRAY);
                StyleConstants.setBold(COMMENT, false);
                StyleConstants.setItalic(COMMENT, true);
        }
        static  {initAttributes(); }

        protected abstract void apply(int startOffset, int endOffset, String input) throws
BadLocationException;

        public void insertString(int offset, String str, AttributeSet a) throws
BadLocationException
        {
                super.insertString(offset, str, a);

                int start = offset;
                int end = start+str.length();

                int length = getLength();

                String text = getText(0, length);

                start--;
                while (start >= 0 && text.charAt(start) != '\n')
                        start--;
                start++;

                while (end < length && text.charAt(end) != '\n')
                        end++;

                refresh(start, end, text);
        }

        public void remove(int offset, int length) throws BadLocationException
        {
                super.remove(offset, length);

                int start = offset;
                int end = start;

                length = getLength();

                String text = getText(0, length);

                start--;
                while (start >= 0 && text.charAt(start) != '\n')
                        start--;
                start++;

                while (end < length && text.charAt(end) != '\n')
                        end++;

                refresh(start, end, text);
        }

        private void refresh(int start, int end, String text)
                throws BadLocationException
        {
                UndoableEditListener[] listeners = getUndoableEditListeners();
                for (int i=0; i<listeners.length; i++)
                {
                        removeUndoableEditListener(listeners[i]);
                }
                apply(start, end, text);
                for (int i=0; i<listeners.length; i++)
```

```
                {
                        addUndoableEditListener(listeners[i]);
                }
        }
}

package gesser.gals.editor;

import gesser.gals.analyser.LexicalError;
import gesser.gals.analyser.Token;
import gesser.gals.generator.parser.Grammar;

import java.awt.Color;

import javax.swing.text.BadLocationException;
import javax.swing.text.SimpleAttributeSet;
import javax.swing.text.StyleConstants;

/**
 * @author Gesser
 *
 * To change this generated comment edit the template variable "typecomment":
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class BNFDocument extends NTDocument
{
        private static final SimpleAttributeSet ACTION_SEM = new
SimpleAttributeSet(NORMAL);
        private static final SimpleAttributeSet EPSILON = new SimpleAttributeSet(NORMAL);

        static
        {
                StyleConstants.setBackground(ACTION_SEM, Color.WHITE);
                StyleConstants.setForeground(ACTION_SEM, new Color(0, 128, 0));
                StyleConstants.setBold(ACTION_SEM, false);

                StyleConstants.setBackground(EPSILON, Color.WHITE);
                StyleConstants.setForeground(EPSILON, Color.MAGENTA);
                StyleConstants.setBold(EPSILON, true);
        }

        protected void apply(int startOffset, int endOffset, String input) throws
BadLocationException
        {
                if (startOffset >= endOffset)
                        return;

                scanner.setInput( input );
                scanner.setRange(startOffset, endOffset);
                scanner.setReturnComents(true);

                int oldPos = startOffset;
                Token t=null;
                boolean done = false;

                while (!done)
                {
                        int pos;

                        try
                        {
                                done = true;
                                t = scanner.nextToken();

                                while (t != null)
                                {
                                        pos = t.getPosition();
                                        int length = t.getLexeme().length();
```

```
                                        SimpleAttributeSet att = NORMAL;

                                        switch (t.getId())
                                        {
                                                case PIPE:
                                                case SEMICOLON:
                                                case DERIVES:
                                                        att = OPERATOR;

                                                        break;
                                                case TERM:
                                                        if (t.getLexeme().charAt(0) == '"')
                                                                att = STRING;
                                                        else if
(t.getLexeme().equals(Grammar.EPSILON_STR))
                                                                att = EPSILON;
                                                        else
                                                                att = NORMAL;

                                                        break;
                                                case NON_TERM:
                                                        att = NON_TERMINAL;
                                                        break;
                                                case -1:
                                                        att = COMMENT;
                                                        break;
                                                case ACTION:
                                                        att = ACTION_SEM;
                                                        length++;
                                                        break;
                                        }
                                        setCharacterAttributes(oldPos, pos-oldPos, NORMAL,
true);
                                        setCharacterAttributes(pos, length, att, true);
                                        oldPos = pos+length;

                                        t = scanner.nextToken();
                                }
                        }
                        catch (LexicalError e)
                        {
                                //modo panico
                                pos = e.getPosition();
                                setCharacterAttributes(oldPos, pos-oldPos, NORMAL, true);

                                oldPos = pos;

                                int length = 0;
                                for (int i=e.getPosition(); i < input.length() && "
\t\n\r".indexOf(input.charAt(i))==-1; i++)
                                        length++;

                                setCharacterAttributes(pos, length, ERROR, true);
                                oldPos = pos + length;

                                scanner.setPosition(e.getPosition() + length);
                                done = false;
                        }
                }
                setCharacterAttributes(oldPos, endOffset-oldPos, NORMAL, true);
        }
}

package gesser.gals.gas;

import gesser.gals.Actions;
import gesser.gals.FileFilters;
import gesser.gals.InputPane;
import gesser.gals.MainWindow;
import gesser.gals.InputPane.Data;
import gesser.gals.analyser.AnalysisError;
```

```
import gesser.gals.generator.Options;
import gesser.gals.generator.OptionsDialog;
import gesser.gals.generator.parser.Grammar;
import gesser.gals.util.LexSyntData;

import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import javax.swing.JButton;
import javax.swing.JDialog;
import javax.swing.JFileChooser;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.SwingConstants;

public class BNFImporter extends JDialog implements ActionListener
{
	private JButton prods = new JButton("Importar Produções para Projeto Atual");
	private JButton synt = new JButton("Gerar Analisador Sintático");
	private JButton full = new JButton("Gerar Analisador Léxico e Sintático");

	private int result;

	public BNFImporter()
	{
		super(MainWindow.getInstance(), "Importar Gramática BNF", true);

		getContentPane().setLayout(new BorderLayout(10, 10));

		getContentPane().add(
			new JLabel("<html><center>Arquivo processado com sucesso.<br>O que
você deseja fazer?</center></html>", SwingConstants.CENTER),
			BorderLayout.NORTH
			);

		JPanel btns = new JPanel(new GridLayout(0, 1, 5, 5));
		btns.add(prods);
		btns.add(synt);
		btns.add(full);
		getContentPane().add(btns);

		prods.addActionListener(this);
		synt.addActionListener(this);
		full.addActionListener(this);

		pack();
		setResizable(false);
		MainWindow.centralize(this);
		setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
	}

	public LexSyntData importGAS() throws FileNotFoundException, AnalysisError
	{
		if (!Actions.checkSaved())
			return null;

		Actions.FILE_CHOOSER.setFileFilter(FileFilters.BNF_FILTER);
		if (Actions.FILE_CHOOSER.showOpenDialog(MainWindow.getInstance()) ==
JFileChooser.APPROVE_OPTION)
		{
			File file = Actions.FILE_CHOOSER.getSelectedFile();

			GASScanner scanner = new GASScanner(new FileReader(file));
```

```
                GASParser  parser = new GASParser();
                GASTranslator translator = new GASTranslator();

                parser.parse(scanner, translator);

                Grammar g = translator.getGrammar();

                return getData(g);
        }
        else
                return null;
    }

    private LexSyntData getData(Grammar g)
    {
        show();

        switch (result)
        {
                case 0: return productionsImported(g);
                case 1: return newParser(g);
                case 2: return newFull(g);
                default : return null;
        }
    }

    private LexSyntData productionsImported(Grammar g)
    {
        Data data = MainWindow.getInstance().getData();

        data.setGrammar(g.toString());
        Options opts = OptionsDialog.getInstance().getOptions();

        if (! opts.generateParser)
        {
                opts.generateParser = true;
                StringBuffer bfrNT = new StringBuffer();
                String[] nt = g.getNonTerminals();
                for (int i=0; i<nt.length; i++)
                {
                        bfrNT.append(nt[i]).append("\n");
                }
                data.setNonTerminals(bfrNT.toString());
        }

        Actions.setSaved(false);

        return new LexSyntData(opts, data);
    }

    private LexSyntData newParser(Grammar g)
    {
        Actions.reset();

        StringBuffer bfrT = new StringBuffer();
        String[] toks = g.getTerminals();
        for (int i=0; i<toks.length; i++)
        {
                bfrT.append(toks[i]).append("\n");
        }
        StringBuffer bfrNT = new StringBuffer();
        String[] nt = g.getNonTerminals();
        for (int i=0; i<nt.length; i++)
        {
                bfrNT.append(nt[i]).append("\n");
        }

        String gram = g.toString();
        InputPane.Data data = new InputPane.Data("", bfrT.toString(),
bfrNT.toString(), gram);
        Options opts = new Options();
```

```
                opts.generateScanner = false;

                return new LexSyntData(opts, data);
        }

        private LexSyntData newFull(Grammar g)
        {
                Actions.reset();

                StringBuffer bfrNT = new StringBuffer();
                String[] nt = g.getNonTerminals();
                for (int i=0; i<nt.length; i++)
                {
                        bfrNT.append(nt[i]).append("\n");
                }

                String gram = g.toString();
                InputPane.Data data = new InputPane.Data("", getTokens(g.getTerminals()),
bfrNT.toString(), gram);
                Options opts = new Options();

                return new LexSyntData(opts, data);
        }

        private String getTokens(String[] t)
        {
                List tokens = new ArrayList(Arrays.asList(t));
                StringBuffer bfr = new StringBuffer();

                bfr.append("//operadores\n");

                for (int i=0; i<tokens.size(); i++)
                {
                        String tok = (String) tokens.get(i);
                        if (tok.charAt(0) == '"')
                        {
                                bfr.append(tok).append("\n");
                                tokens.remove(i);
                                i--;
                        }
                }

                bfr.append("\n");

                bfr.append(KeyWordsSelector.process(tokens));

                return bfr.toString();
        }

        public void actionPerformed(ActionEvent e)
        {
                if (e.getSource() == prods)
                {
                        result = 0;
                        dispose();
                }
                else if (e.getSource() == synt)
                {
                        result = 1;
                        dispose();
                }
                else if (e.getSource() == full)
                {
                        result = 2;
                        dispose();
                }
        }
}
package gesser.gals.generator.java;

import gesser.gals.generator.Options;
```

```
import gesser.gals.generator.parser.Grammar;
import gesser.gals.generator.parser.ll.LLParser;
import gesser.gals.generator.parser.ll.NotLLException;
import gesser.gals.generator.parser.lr.Command;
import gesser.gals.generator.parser.lr.SLRParser;
import gesser.gals.generator.scanner.FiniteAutomata;
import gesser.gals.util.IntList;
import gesser.gals.util.ProductionList;

import java.util.*;

public class JavaCommonGenerator
{
	int[][][] lrTable = null;

	public Map generate(FiniteAutomata fa, Grammar g, Options options) throws
NotLLException
	{
		Map result = new HashMap();

		result.put("Token.java", generateToken(options));

		result.put("Constants.java", generateConstants(fa, g, options));
		if (fa != null)
			result.put("ScannerConstants.java", generateScannerConstants(fa,
options));
		if (g != null)
			result.put("ParserConstants.java", generateParserConstants(g,
options));

		result.put("AnalysisError.java", generateAnalysisError(options));
		result.put("LexicalError.java", generateLexicalError(options));
		result.put("SyntaticError.java", generateSyntaticError(options));
		result.put("SemanticError.java", generateSemanticError(options));

		return result;
	}

	private String generateToken(Options options)
	{
		StringBuffer result = new StringBuffer();

		String package_ = options.pkgName;
		if (package_ != null && !package_.equals(""))
			result.append("package " + package_ + ";\n\n");

		String cls =
		"public class Token\n"+
		"{\n"+
		"    private int id;\n"+
		"    private String lexeme;\n"+
		"    private int position;\n"+
		"\n"+
		"    public Token(int id, String lexeme, int position)\n"+
		"    {\n"+
		"        this.id = id;\n"+
		"        this.lexeme = lexeme;\n"+
		"        this.position = position;\n"+
		"    }\n"+
		"\n"+
		"    public final int getId()\n"+
		"    {\n"+
		"        return id;\n"+
		"    }\n"+
		"\n"+
		"    public final String getLexeme()\n"+
		"    {\n"+
		"        return lexeme;\n"+
		"    }\n"+
		"\n"+
		"    public final int getPosition()\n"+
```

```
        "    {\n"+
        "        return position;\n"+
        "    }\n"+
        "\n"+
        "    public String toString()\n"+
        "    {\n"+
        "        return id+\" ( \"+lexeme+\" ) @ \"+position;\n"+
        "    };\n"+
        "}\n"+
        "";

        result.append(cls);

        return result.toString();
}

private String generateAnalysisError(Options options)
{
        StringBuffer result = new StringBuffer();

        String package_ = options.pkgName;
        if (package_ != null && !package_.equals(""))
                result.append("package " + package_ + ";\n\n");

        String cls =
        "public class AnalysisError extends Exception\n"+
        "{\n"+
        "    private int position;\n"+
        "\n"+
        "    public AnalysisError(String msg, int position)\n"+
        "    {\n"+
        "        super(msg);\n"+
        "        this.position = position;\n"+
        "    }\n"+
        "\n"+
        "    public AnalysisError(String msg)\n"+
        "    {\n"+
        "        super(msg);\n"+
        "        this.position = -1;\n"+
    "    }\n"+
        "\n"+
        "    public int getPosition()\n"+
        "    {\n"+
        "        return position;\n"+
    "    }\n"+
        "\n"+
        "    public String toString()\n"+
        "    {\n"+
        "        return super.toString() + \", @ \"+position;\n"+
        "    }\n"+
        "}\n"+
        "";

        result.append(cls);

        return result.toString();
}

private String generateLexicalError(Options options)
{
        StringBuffer result = new StringBuffer();

        String package_ = options.pkgName;
        if (package_ != null && !package_.equals(""))
                result.append("package " + package_ + ";\n\n");

        String cls =
        "public class LexicalError extends AnalysisError\n"+
        "{\n"+
        "    public LexicalError(String msg, int position)\n"+
        "        {\n"+
```

```
        "        super(msg, position);\n"+
        "    }\n"+
        "\n"+
        "    public LexicalError(String msg)\n"+
        "    {\n"+
        "        super(msg);\n"+
        "    }\n"+
        "}\n"+
        "";

        result.append(cls);

        return result.toString();
}

private String generateSyntaticError(Options options)
{
        StringBuffer result = new StringBuffer();

        String package_ = options.pkgName;
        if (package_ != null && !package_.equals(""))
                result.append("package " + package_ + ";\n\n");

        String cls =
        "public class SyntaticError extends AnalysisError\n"+
        "{\n"+
        "    public SyntaticError(String msg, int position)\n"+
        "        {\n"+
        "        super(msg, position);\n"+
        "    }\n"+
        "\n"+
        "    public SyntaticError(String msg)\n"+
        "    {\n"+
        "        super(msg);\n"+
        "    }\n"+
        "}\n"+
        "";

        result.append(cls);

        return result.toString();
}

private String generateSemanticError(Options options)
{
        StringBuffer result = new StringBuffer();

        String package_ = options.pkgName;
        if (package_ != null && !package_.equals(""))
                result.append("package " + package_ + ";\n\n");

        String cls =
        "public class SemanticError extends AnalysisError\n"+
        "{\n"+
        "    public SemanticError(String msg, int position)\n"+
        "        {\n"+
        "        super(msg, position);\n"+
        "    }\n"+
        "\n"+
        "    public SemanticError(String msg)\n"+
        "    {\n"+
        "        super(msg);\n"+
        "    }\n"+
        "}\n"+
        "";

        result.append(cls);

        return result.toString();
}
```

```
	private String generateConstants(FiniteAutomata fa, Grammar g, Options options)
throws NotLLException
	{
		StringBuffer result = new StringBuffer();

		String package_ = options.pkgName;
		if (package_ != null && !package_.equals(""))
			result.append("package " + package_ + ";\n\n");

		String extInter = null;
		if (fa == null)
			extInter = "ParserConstants";
		else if (g == null)
			extInter = "ScannerConstants";
		else
			extInter = "ScannerConstants, ParserConstants";

		result.append(
		"public interface Constants extends "+extInter+"\n"+
		"{\n"+
		"    int EPSILON  = 0;\n"+
		"    int DOLLAR   = 1;\n"+
		"\n"+
		constList(fa, g)+
		"\n" );

	result.append("}\n");

		return result.toString();
	}

	private String generateScannerConstants(FiniteAutomata fa, Options options)
	{
		StringBuffer result = new StringBuffer();

		String package_ = options.pkgName;
		if (package_ != null && !package_.equals(""))
			result.append("package " + package_ + ";\n\n");

		result.append(
		"public interface ScannerConstants\n"+
		"{\n");

		result.append(genLexTables(fa, options));

		result.append("}\n");

		return result.toString();
	}

	private String generateParserConstants(Grammar g, Options options) throws
NotLLException
	{
		StringBuffer result = new StringBuffer();

		String package_ = options.pkgName;
		if (package_ != null && !package_.equals(""))
			result.append("package " + package_ + ";\n\n");

		result.append(
		"public interface ParserConstants\n"+
		"{\n");

		result.append(genSyntTables(g, options));

		result.append("}\n");

		return result.toString();
	}

	private String genLexTables(FiniteAutomata fa, Options options)
```

```
{
	String lexTable;

	switch (options.scannerTable)
	{
		case Options.SCANNER_TABLE_FULL:
			lexTable = lex_table(fa);
			break;
		case Options.SCANNER_TABLE_COMPACT:
			lexTable = lex_table_compress(fa);
			break;
		case Options.SCANNER_TABLE_HARDCODE:
			lexTable = "";
			break;
		default:
			//nunca acontece
			lexTable = null;
			break;
	}

	return
		lexTable+
		"\n"+
		token_state(fa)+
		(fa.hasContext() ?
		"\n"+
		context(fa) : "")+
		"\n"+
		(fa.getSpecialCases().length > 0 ?
		special_cases(fa)+
		"\n" : "")+
		scanner_error(fa)+
		"\n";
}

private String context(FiniteAutomata fa)
{
	StringBuffer result = new StringBuffer();

	result.append("    int[][] SCANNER_CONTEXT =\n"+
		      "    {\n");

	for (int i=0; i<fa.getTransitions().length; i++)
	{
		result.append("        {");
		result.append(fa.isContext(i)?"1":"0");
		result.append(", ");
		result.append(fa.getOrigin(i));
		result.append("},\n");
	}

	result.setLength(result.length()-2);
	result.append(
	"\n    };\n");

	return result.toString();
}

private String scanner_error(FiniteAutomata fa)
{
	StringBuffer result = new StringBuffer();

	result.append(
	"    String[] SCANNER_ERROR =\n"+
	"    {\n");

	int count = fa.getTransitions().length;
	for (int i=0; i< count; i++)
	{
		result.append("        \"");
```

```
                String error = fa.getError(i);
                for (int j=0; j<error.length(); j++)
                {
                        if (error.charAt(j) == '"')
                                result.append("\\\"");
                        else
                                result.append(error.charAt(j));
                }

                result.append("\",\n");
        }
        result.setLength(result.length()-2);
        result.append(
        "\n    };\n");

        return result.toString();
}

private String genSyntTables(Grammar g, Options options) throws NotLLException
{
        switch (options.parser)
        {
                case Options.PARSER_REC_DESC:
                case Options.PARSER_LL:
                        return genLLSyntTables(g, options.parser);
                case Options.PARSER_SLR:
                case Options.PARSER_LALR:
                case Options.PARSER_LR:
                        return genSLRSyntTables(g);
                default:
                        return null;
        }
}

private String genSLRSyntTables(Grammar g)
{
        lrTable = new SLRParser(g).buildIntTable();

        StringBuffer result = new StringBuffer(
                "    int FIRST_SEMANTIC_ACTION = "+g.FIRST_SEMANTIC_ACTION()+";\n"+
                "\n"+
                "    int SHIFT  = 0;\n"+
                "    int REDUCE = 1;\n"+
                "    int ACTION = 2;\n"+
                "    int ACCEPT = 3;\n"+
                "    int GO_TO  = 4;\n"+
                "    int ERROR  = 5;\n" );

        result.append("\n");

        result.append(emitSLRTable(g));

        result.append("\n");

        result.append(emitProductionsForLR(g));

        result.append("\n");

        result.append(emitErrorTableLR());

        return result.toString();
}
private Object emitProductionsForLR(Grammar g)
{
        StringBuffer result = new StringBuffer();

        ProductionList prods = g.getProductions();

        result.append("    int[][] PRODUCTIONS =\n");
        result.append("    {\n");
```

```
        for (int i=0; i<prods.size(); i++)
        {
                result.append("        { ");
                result.append(prods.getProd(i).get_lhs());
                result.append(", ");
                result.append(prods.getProd(i).get_rhs().size());
                result.append(" },\n");
        }
        result.setLength(result.length()-2);
        result.append("\n    };\n");

        return result.toString();
}

private String emitSLRTable(Grammar g)
{
        StringBuffer result = new StringBuffer();

        int[][][] tbl = lrTable;

        result.append("    int[][][] PARSER_TABLE =\n");
        result.append("    {\n");

        int max = tbl.length;
        if (g.getProductions().size() > max)
                max = g.getProductions().size();

        max = (""+max).length();

        for (int i=0; i< tbl.length; i++)
        {
                result.append("        {");
                for (int j=0; j<tbl[i].length; j++)
                {
                        result.append(" {");
                        result.append(Command.CONSTANTS[tbl[i][j][0]]);
                        result.append(", ");
                        String str = ""+tbl[i][j][1];
                        for (int k=str.length(); k<max; k++)
                                result.append(" ");
                        result.append(str).append("},");
                }
                result.setLength(result.length()-1);
                result.append(" },\n");
        }
        result.setLength(result.length()-2);
        result.append("\n    };\n");

        return result.toString();
}

private String genLLSyntTables(Grammar g, int type ) throws NotLLException
{
        StringBuffer result = new StringBuffer();

        if (type == Options.PARSER_LL)
        {
                int start = g.getStartSymbol();
                int fnt = g.FIRST_NON_TERMINAL;
                int fsa = g.getSymbols().length;

                String syntConsts =
                "    int START_SYMBOL = "+start+";\n"+
                "\n"+
                "    int FIRST_NON_TERMINAL    = "+fnt+";\n"+
        "    int FIRST_SEMANTIC_ACTION = "+fsa+";\n";

        result.append(syntConsts);

        result.append("\n");
```

```
                result.append(emitLLTable(new LLParser(g)));

                        result.append("\n");

                        result.append(emitProductionsForLL(g));

                        result.append("\n");

                        result.append(emitErrorTableLL(g));

                        return result.toString();
                }
                else if (type == Options.PARSER_REC_DESC)
                        return emitErrorTableLL(g);
                else
                        return null;
        }

        private String constList(FiniteAutomata fa, Grammar g)
        {
                StringBuffer result = new StringBuffer();

                List tokens = null;

                if (fa != null)
                        tokens = fa.getTokens();
                else if (g != null)
                        tokens = Arrays.asList(g.getTerminals());
                else
                        throw new RuntimeException("Erro Interno");

                for (int i=0; i<tokens.size(); i++)
                {
                        String t = (String) tokens.get(i);
                        if (t.charAt(0) == '\"')
                                result.append("    int t_TOKEN_"+(i+2)+" = "+(i+2)+";
"+"//"+t+"\n");
                        else
                                result.append("    int t_"+t+" = "+(i+2)+";\n");
                }

                return result.toString();
        }

        private String lex_table_compress(FiniteAutomata fa)
        {
                StringBuffer result = new StringBuffer();

                Map[] trans = fa.getTransitions();

                int[] sti = new int[trans.length+1];
                int count = 0;
                for (int i=0; i<trans.length; i++)
                {
                        sti[i] = count;
                        count += trans[i].size();
                }
                sti[sti.length-1] = count;

                int[][] st = new int[count][2];

                count = 0;
                for (int i=0; i<trans.length; i++)
                {
                        for (Iterator iter = trans[i].keySet().iterator(); iter.hasNext();)
                        {
                                Character ch = (Character) iter.next();
                                Integer itg = (Integer) trans[i].get(ch);

                                st[count][0] = ch.charValue();
                                st[count][1] = itg.intValue();
```

```
                        count++;
                }
        }

        result.append("    int[] SCANNER_TABLE_INDEXES = \n");
        result.append("    {\n");

        for (int i=0; i<sti.length; i++)
        {
                result.append("        ").append(sti[i]).append(",\n");
        }

        result.setLength(result.length()-2);
        result.append("\n    };\n\n");

        result.append("    int[][] SCANNER_TABLE = \n");
        result.append("    {\n");

        for (int i=0; i<st.length; i++)
        {
                result.append("        {").append(st[i][0]).append(",
").append(st[i][1]).append("},\n");
        }

        result.setLength(result.length()-2);
        result.append("\n    };\n");

        return result.toString();
    }

    private String lex_table(FiniteAutomata fa)
    {
        StringBuffer result = new StringBuffer();

        result.append("    int[][] SCANNER_TABLE = \n");
        result.append("    {\n");

        int count = fa.getTransitions().length;
        int max = String.valueOf(count).length();
        if (max == 1)
                max = 2;

        for (int i=0; i<count; i++)
        {
                result.append("        { ");
                for (char c = 0; c<256; c++)
                {
                        String n = String.valueOf(fa.nextState(c, i));
                        for (int j = n.length(); j<max; j++)
                                result.append(" ");
                        result.append(n).append(", ");
                }
                result.setLength(result.length()-2);
                result.append(" },\n");
        }
        result.setLength(result.length()-2);

        result.append("\n    };\n");

        return result.toString();
    }

    private String token_state(FiniteAutomata fa)
    {
        StringBuffer result = new StringBuffer();

        result.append("    int[] TOKEN_STATE = {");
        int count = fa.getTransitions().length;
        int max = String.valueOf(count).length();
        if (max == 1)
```

```
                max = 2;

        for (int i=0; i<count; i++)
        {
                int fin = fa.tokenForState(i);
                String n = String.valueOf(fin);
                for (int j = n.length(); j<max; j++)
                        result.append(" ");
                result.append(n).append(", ");
        }
        result.setLength(result.length()-2);
        result.append(" };\n");

        return result.toString();
}

private String special_cases(FiniteAutomata fa)
{
        int[][] indexes = fa.getSpecialCasesIndexes();
        FiniteAutomata.KeyValuePar[] sc = fa.getSpecialCases();

        StringBuffer result = new StringBuffer();

        int count = sc.length;

        result.append(
                "    int[] SPECIAL_CASES_INDEXES =\n"+
                "        { ");

        count = indexes.length;
        for (int i=0; i<count; i++)
        {
                result.append(indexes[i][0]).append(", ");
        }
        result.append(indexes[count-1][1]);
        result.append(" };\n\n");

        result.append(
                                "    String[] SPECIAL_CASES_KEYS =\n"+
                                "        {  ");
        count = sc.length;
        for (int i=0; i<count; i++)
        {;
                result.append("\"").append(sc[i].key).append("\", ");
        }
        result.setLength(result.length()-2);

        result.append(" };\n\n");

        result.append(
                                "    int[] SPECIAL_CASES_VALUES =\n"+
                                "        {  ");
        count = sc.length;
        for (int i=0; i<count; i++)
        {
                result.append(sc[i].value).append(", ");
        }
        result.setLength(result.length()-2);

        result.append(" };\n");

        return result.toString();
}

private String emitProductionsForLL(Grammar g)
{

        ProductionList pl = g.getProductions();
        String[][] productions = new String[pl.size()][];
        int max = 0;
        for (int i=0; i< pl.size(); i++)
```

```
        {
                IntList rhs = pl.getProd(i).get_rhs();
                if (rhs.size() > 0)
                {
                        productions[i] = new String[rhs.size()];
                        for (int j=0; j<rhs.size(); j++)
                        {
                                productions[i][j] = String.valueOf(rhs.get(j));
                                if (productions[i][j].length() > max)
                                        max = productions[i][j].length();
                        }
                }
                else
                {
                        productions[i] = new String[1];
                        productions[i][0] = "0";
                }
        }

        StringBuffer bfr = new StringBuffer();

        bfr.append("    int[][] PRODUCTIONS = \n");
        bfr.append("    {\n");

        for (int i=0; i< productions.length; i++)
        {
                bfr.append("        {");
                for (int j=0; j<productions[i].length; j++)
                {
                        bfr.append(" ");
                        for (int k = productions[i][j].length(); k<max; k++)
                                bfr.append(" ");
                        bfr.append(productions[i][j]).append(",");
                }
                bfr.setLength(bfr.length()-1);
                bfr.append(" },\n");
        }
        bfr.setLength(bfr.length()-2);
        bfr.append("\n    };\n");

        return bfr.toString();
}

private String emitLLTable(LLParser g)
{
        int[][] tbl = g.generateTable();
        String[][] table = new String[tbl.length][tbl[0].length];

        int max = 0;
        for (int i = 0; i < table.length; i++)
        {
                for (int j = 0; j < table[i].length; j++)
                {
                        String tmp = String.valueOf(tbl[i][j]);
                        table[i][j] = tmp;
                        if (tmp.length() > max)
                                max = tmp.length();
                }
        }

        StringBuffer bfr = new StringBuffer();

        bfr.append("    int[][] PARSER_TABLE =\n");
        bfr.append("    {\n");

        for (int i=0; i< table.length; i++)
        {
                bfr.append("        {");
                for (int j=0; j<table[i].length; j++)
                {
                        bfr.append(" ");
```

```
                        for (int k = table[i][j].length(); k<max; k++)
                                bfr.append(" ");
                        bfr.append(table[i][j]).append(",");
                }
                bfr.setLength(bfr.length()-1);
                bfr.append(" },\n");
        }
        bfr.setLength(bfr.length()-2);
        bfr.append("\n    };\n");

        return bfr.toString();
}

private String emitErrorTableLR()
{
        int count = lrTable.length;

        StringBuffer result = new StringBuffer();

        result.append(
        "    String[] PARSER_ERROR =\n"+
        "    {\n");

        for (int i=0; i< count; i++)
        {
                result.append("        \"Erro estado "+i+"\",\n");
        }

        result.setLength(result.length()-2);
        result.append(
        "\n    };\n");

        return result.toString();
}

private String emitErrorTableLL(Grammar g)
{
        String[] symbs = g.getSymbols();
        StringBuffer result = new StringBuffer();

        result.append(
        "    String[] PARSER_ERROR =\n"+
        "    {\n"+
        "        \"\",\n"+
        "        \"Era esperado fim de programa\",\n");

        for (int i=2; i< g.FIRST_NON_TERMINAL; i++)
        {
                result.append("        \"Era esperado ");
                for (int j=0; j<symbs[i].length(); j++)
                {
                        switch (symbs[i].charAt(j))
                        {
                                case '\"': result.append("\\\""); break;
                                case '\\': result.append("\\\\"); break;
                                default: result.append(symbs[i].charAt(j));

                        }
                }

                result.append("\",\n");
        }

        for (int i=g.FIRST_NON_TERMINAL; i< symbs.length; i++)
                result.append("        \""+symbs[i]+" inválido\",\n");

        result.setLength(result.length()-2);
        result.append(
        "\n    };\n");

        return result.toString();
```

```
	}
}

package gesser.gals.generator.java;

import gesser.gals.generator.Options;
import gesser.gals.generator.RecursiveDescendent;
import gesser.gals.generator.RecursiveDescendent.Function;
import gesser.gals.generator.parser.Grammar;
import gesser.gals.generator.parser.ll.NotLLException;
import gesser.gals.util.IntList;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;

public class JavaParserGenerator
{
	public Map generate(Grammar g, Options options) throws NotLLException
	{
		Map result = new HashMap();

		if (g != null)
		{
			String classname = options.parserName;

			String parser;

			switch (options.parser)
			{
				case Options.PARSER_REC_DESC:
					parser = buildRecursiveDecendantParser(g, options);
					break;
				case Options.PARSER_LL:
					parser = buildLLParser(g, options);
					break;
				case Options.PARSER_SLR:
				case Options.PARSER_LALR:
				case Options.PARSER_LR:
					parser = buildLRParser(g, options);
					break;
				default:
					parser = null;
			}

			result.put(classname+".java", parser);

			result.put(options.semanticName + ".java",
generateSemanticAnalyser(options));
		}

		return result;
	}

	private String buildRecursiveDecendantParser(Grammar g, Options parserOptions)
throws NotLLException
	{
		StringBuffer result = new StringBuffer();

		String package_ = (String) parserOptions.pkgName;

		result.append(emitPackage(package_));

		result.append(emitRecursiveDecendantClass(g, parserOptions));

		return result.toString();
	}

	private String buildLLParser(Grammar g, Options parserOptions)
	{
```

```
                StringBuffer result = new StringBuffer();

                String package_ = (String) parserOptions.pkgName;

                result.append(emitPackage(package_));

                result.append(emitImports());

                result.append(emitLLClass(g, parserOptions));

                return result.toString();
        }

        private String buildLRParser(Grammar g, Options parserOptions)
        {
                StringBuffer result = new StringBuffer();

                String package_ = parserOptions.pkgName;

                result.append(emitPackage(package_));

                result.append(emitImports());

                result.append(emitLRClass(g, parserOptions));

                return result.toString();
        }

        private String emitPackage(String package_)
        {
                if (package_ != null && !package_.equals(""))
                        return "package " + package_ + ";\n\n";
                else
                        return "";
        }

        private String emitImports()
        {
                return
                        "import java.util.Stack;\n"+
                        "\n";
        }

        private String emitLRClass(Grammar g, Options parserOptions)
        {
                StringBuffer result = new StringBuffer();

                String classname = (String)parserOptions.parserName;
                result.append("public class ").append(classname).append(" implements
Constants\n{\n");

                String scannerName = (String) parserOptions.scannerName;
                String semanName = (String) parserOptions.semanticName;

                String variables =
                "    private Stack stack = new Stack();\n"+
                "    private Token currentToken;\n"+
                "    private Token previousToken;\n"+
                "    private "+scannerName+" scanner;\n"+
                "    private "+semanName+" semanticAnalyser;\n"+
                "\n";

                result.append(variables);

                result.append(

                "    public void parse("+scannerName+" scanner, "+semanName+"
semanticAnalyser) throws AnalysisError\n"+
                "    {\n"+
                "        this.scanner = scanner;\n"+
                "        this.semanticAnalyser = semanticAnalyser;\n"+
```

```
                "\n"+
                "        stack.clear();\n"+
                "        stack.push(new Integer(0));\n"+
                "\n"+
                "        currentToken = scanner.nextToken();\n"+
                "\n"+
                "        while ( ! step() )\n"+
                "            ;\n"+
                "    }\n"+
                "\n"+
                "    private boolean step() throws AnalysisError\n"+
                "    {\n"+
                "        if (currentToken == null)\n"+
        "         {\n"+
                "            int pos = 0;\n"+
                "            if (previousToken != null)\n"+
                "                pos =
previousToken.getPosition()+previousToken.getLexeme().length();\n"+
                "\n"+
        "             currentToken = new Token(DOLLAR, \"$\", pos);\n"+
        "         }\n"+
        "\n"+
        "         int token = currentToken.getId();\n"+
                "        int state = ((Integer)stack.peek()).intValue();\n"+
                "\n"+
        "         int[] cmd = PARSER_TABLE[state][token-1];\n"+
                "\n"+
                "        switch (cmd[0])\n"+
                "        {\n"+
                "            case SHIFT:\n"+
                "                stack.push(new Integer(cmd[1]));\n"+
                "                previousToken = currentToken;\n"+
                "                currentToken = scanner.nextToken();\n"+
                "                return false;\n"+
                "\n"+
                "            case REDUCE:\n"+
                "                int[] prod = PRODUCTIONS[cmd[1]];\n"+
                "\n"+
                "                for (int i=0; i<prod[1]; i++)\n"+
                "                    stack.pop();\n"+
                "\n"+
                "                int oldState = ((Integer)stack.peek()).intValue();\n"+
                "                stack.push(new Integer(PARSER_TABLE[oldState][prod[0]-
1][1]));\n"+
                "                return false;\n"+
                "\n"+
                "            case ACTION:\n"+
                "                int action = FIRST_SEMANTIC_ACTION + cmd[1] - 1;\n"+
                "                stack.push(new
Integer(PARSER_TABLE[state][action][1]));\n"+
                "                semanticAnalyser.executeAction(cmd[1],
previousToken);\n"+
                "                return false;\n"+
                "\n"+
                "            case ACCEPT:\n"+
                "                return true;\n"+
                "\n"+
                "            case ERROR:\n"+
                "                throw new SyntaticError(PARSER_ERROR[state],
currentToken.getPosition());\n"+
                "        }\n"+
                "        return false;\n"+
                "    }\n"+
                "\n"

                );
                result.append("}\n");

                return result.toString();
        }
```

```
        private String emitLLClass(Grammar g, Options parserOptions)
        {
                StringBuffer result = new StringBuffer();

                String classname = parserOptions.parserName;
                result.append("public class ").append(classname).append(" implements
Constants\n{\n");

                String scannerName = (String) parserOptions.scannerName;
                String semanName = (String) parserOptions.semanticName;

                String variables =
                "    private Stack stack = new Stack();\n"+
                "    private Token currentToken;\n"+
                "    private Token previousToken;\n"+
                "    private "+scannerName+" scanner;\n"+
                "    private "+semanName+" semanticAnalyser;\n"+
                "\n";

                result.append(variables);

                result.append(emitLLFunctions(parserOptions));

                result.append("}\n");

                return result.toString();
        }

        private String emitLLFunctions(Options parserOptions)
        {
                StringBuffer result = new StringBuffer();

                result.append(emitTesters());

                result.append("\n");

                result.append(emitStep());

                result.append("\n");

                result.append(emitDriver(parserOptions));


                return  result.toString();
        }

        private String emitTesters()
        {
                return
                "    private static final boolean isTerminal(int x)\n"+
                "    {\n"+
                "        return x < FIRST_NON_TERMINAL;\n"+
                "    }\n"+
                "\n"+
                "    private static final boolean isNonTerminal(int x)\n"+
                "    {\n"+
                "        return x >= FIRST_NON_TERMINAL && x < FIRST_SEMANTIC_ACTION;\n"+
                "    }\n"+
                "\n"+
                "    private static final boolean isSemanticAction(int x)\n"+
                "    {\n"+
                "        return x >= FIRST_SEMANTIC_ACTION;\n"+
                "    }\n"+
                "";
        }

        private String emitDriver(Options parserOptions)
        {
                String scannerName = parserOptions.scannerName;
                String semanName   = parserOptions.semanticName;
```

```
                return
                "    public void parse("+scannerName+" scanner, "+semanName+"
semanticAnalyser) throws AnalysisError, LexicalError, SyntaticError, SemanticError\n"+
            "    {\n"+
                "        this.scanner = scanner;\n"+
                "        this.semanticAnalyser = semanticAnalyser;\n"+
                "\n"+
                "        stack.clear();\n"+
                "        stack.push(new Integer(DOLLAR));\n"+
                "        stack.push(new Integer(START_SYMBOL));\n"+
                "\n"+
                "        currentToken = scanner.nextToken();\n"+
                "\n"+
                "        while ( ! step() )\n"+
                "            ;\n"+
            "    }\n"+
                "";
        }

        private String emitStep()
        {
                return
                "    private boolean step() throws AnalysisError, LexicalError,
SyntaticError, SemanticError\n"+
                "    {\n"+
                "        if (currentToken == null)\n"+
         "        {\n"+
                "            int pos = 0;\n"+
                "            if (previousToken != null)\n"+
                "                pos =
previousToken.getPosition()+previousToken.getLexeme().length();\n"+
                "\n"+
         "             currentToken = new Token(DOLLAR, \"$\", pos);\n"+
         "        }\n"+
         "\n"+
                "        int x = ((Integer)stack.pop()).intValue();\n"+
                "        int a = currentToken.getId();\n"+
                "\n"+
                "        if (x == EPSILON)\n"+
                "        {\n"+
                "            return false;\n"+
                "        }\n"+
                "        else if (isTerminal(x))\n"+
                "        {\n"+
                "            if (x == a)\n"+
                "            {\n"+
                "                if (stack.empty())\n"+
                "                    return true;\n"+
                "                else\n"+
                "                {\n"+
                "                    previousToken = currentToken;\n"+
                "                    currentToken = scanner.nextToken();\n"+
                "                    return false;\n"+
                "                }\n"+
                "            }\n"+
                "            else\n"+
                "            {\n"+
                "                throw new SyntaticError(PARSER_ERROR[x],
currentToken.getPosition());\n"+
                "            }\n"+
                "        }\n"+
                "        else if (isNonTerminal(x))\n"+
                "        {\n"+
                "            if (pushProduction(x, a))\n"+
                "                return false;\n"+
                "            else\n"+
                "                throw new SyntaticError(PARSER_ERROR[x],
currentToken.getPosition());\n"+
                "        }\n"+
                "        else // isSemanticAction(x)\n"+
                "        {\n"+
```

```
                "            semanticAnalyser.executeAction(x-FIRST_SEMANTIC_ACTION,
previousToken);\n"+
                "            return false;\n"+
                "        }\n"+
                "    }\n"+
                "\n"+
                "    private boolean pushProduction(int topStack, int tokenInput)\n"+
                "    {\n"+
                "        int p = PARSER_TABLE[topStack-FIRST_NON_TERMINAL][tokenInput-
1];\n"+
                "        if (p >= 0)\n"+
                "        {\n"+
                "            int[] production = PRODUCTIONS[p];\n"+
                "            //empilha a produção em ordem reversa\n"+
                "            for (int i=production.length-1; i>=0; i--)\n"+
                "            {\n"+
                "                stack.push(new Integer(production[i]));\n"+
                "            }\n"+
                "            return true;\n"+
                "        }\n"+
                "        else\n"+
                "            return false;\n"+
                "    }\n"+
                "";
        }

        private String emitRecursiveDecendantClass(Grammar g, Options parserOptions)
throws NotLLException
        {
                RecursiveDescendent rd = new RecursiveDescendent(g);

                StringBuffer result = new StringBuffer();

                String classname = (String)parserOptions.parserName;
                result.append("public class ").append(classname).append(" implements
Constants\n{\n");

                String scannerName = (String) parserOptions.scannerName;
                String semanName = (String) parserOptions.semanticName;

                String variables =
                "    private Token currentToken;\n"+
                "    private Token previousToken;\n"+
                "    private "+scannerName+" scanner;\n"+
                "    private "+semanName+" semanticAnalyser;\n"+
                "\n";

                result.append(variables);

                result.append(
                "    public void parse("+scannerName+" scanner, "+semanName+"
semanticAnalyser) throws AnalysisError\n"+
                "    {\n"+
                "        this.scanner = scanner;\n"+
                "        this.semanticAnalyser = semanticAnalyser;\n"+
                "\n"+
                "        currentToken = scanner.nextToken();\n"+
                "        if (currentToken == null)\n"+
                "            currentToken = new Token(DOLLAR, \"$\", 0);\n"+
                "\n"+
                "        "+rd.getStart()+"();\n"+
                "\n"+
                "        if (currentToken.getId() != DOLLAR)\n"+
                "            throw new SyntaticError(PARSER_ERROR[DOLLAR],
currentToken.getPosition());\n"+
                "    }\n"+
                "\n"+
                "    private void match(int token) throws AnalysisError\n"+
                "    {\n"+
                "        if (currentToken.getId() == token)\n"+
                "        {\n"+
```

```
                "            previousToken = currentToken;\n"+
                "            currentToken = scanner.nextToken();\n"+
                "            if (currentToken == null)\n"+
                "            {\n"+
                "                int pos = 0;\n"+
                "                if (previousToken != null)\n"+
                "                    pos =
previousToken.getPosition()+previousToken.getLexeme().length();\n"+
                "\n"+
                "                currentToken = new Token(DOLLAR, \"$\", pos);\n"+
                "            }\n"+
                "        }\n"+
                "        else\n"+
                "            throw new SyntaticError(PARSER_ERROR[token],
currentToken.getPosition());\n"+
                "    }\n"+
                "\n");

                Map funcs = rd.build();

                for (int symb=g.FIRST_NON_TERMINAL; symb<g.FIRST_SEMANTIC_ACTION();
symb++)
                {
                        String name = rd.getSymbols(symb);
                        RecursiveDescendent.Function f = (Function) funcs.get(name);

                        result.append(
                                        "    private void "+name+"() throws
AnalysisError\n"+
                                        "    {\n"+
                                        "        switch (currentToken.getId())\n"+
                                        "        {\n" );

                        List keys = new LinkedList(f.input.keySet());

                        for (int i = 0; i<keys.size(); i++)
                        {
                                IntList rhs = (IntList) f.input.get(keys.get(i));
                                int token = ((Integer)keys.get(i)).intValue();

                                result.append(
                                                "            case "+token+": //
"+rd.getSymbols(token)+"\n");
                                for (int j=i+1; j<keys.size(); j++)
                                {
                                        IntList rhs2 = (IntList) f.input.get(keys.get(j));
                                        if (rhs2.equals(rhs))
                                        {
                                                token = ((Integer)keys.get(j)).intValue();
                                                result.append(
                                                "            case "+token+": //
"+rd.getSymbols(token)+"\n");
                                                keys.remove(j);
                                                j--;
                                        }
                                }

                                if (rhs.size() == 0)
                                        result.append(
                                                "                // EPSILON\n");

                                for (int k=0; k<rhs.size(); k++)
                                {
                                        int s = rhs.get(k);
                                        if (g.isTerminal(s))
                                        {
                                                result.append(
                                                "                match("+s+"); //
"+rd.getSymbols(s)+"\n");
                                        }
                                        else if (g.isNonTerminal(s))
```

```
                                        {
                                                result.append(
                                                "
"+rd.getSymbols(s)+"();\n");
                                        }
                                        else //isSemanticAction(s)
                                        {
                                                result.append(
                                                "
semanticAnalyser.executeAction("+(s-g.FIRST_SEMANTIC_ACTION())+", previousToken);\n");
                                        }
                                }

                                result.append(
                                                "                break;\n");
                        }

                        result.append(
                                                "            default:\n"+
                                                "                throw new
SyntaticError(PARSER_ERROR["+f.lhs+"], currentToken.getPosition());\n"+
                                                "        }\n"+
                                                "    }\n"+
                                                "\n");
                }

                result.append("}\n");

                return result.toString();
        }

        private String generateSemanticAnalyser(Options options)
        {
                StringBuffer result = new StringBuffer();

                String package_ = options.pkgName;
                if (package_ != null && !package_.equals(""))
                        result.append("package " + package_ + ";\n\n");

                String cls =
                "public class "+options.semanticName+" implements Constants\n"+
                "{\n"+
                "    public void executeAction(int action, Token token)     throws
SemanticError\n"+
                "    {\n"+
                "        System.out.println(\"Ação #\"+action+\", Token: \"+token);\n"+
                "    }  \n"+
                "}\n"+
                "";

                result.append(cls);

                return result.toString();
        }
}

package gesser.gals.generator.java;

import gesser.gals.generator.Options;
import gesser.gals.generator.scanner.FiniteAutomata;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

/**
 * @author Gesser
 */

public class JavaScannerGenerator
```

```
{
	boolean sensitive = true;
	boolean lookup = true;

	public Map generate(FiniteAutomata fa, Options options)
	{
		Map result = new HashMap();

		String classname = options.scannerName;

		String scanner;
		if (fa != null)
		{
			sensitive = options.scannerSensitive;
			lookup = fa.getSpecialCases().length > 0;
			scanner = buildScanner(fa, options);
		}
		else
			scanner = buildEmptyScanner(options);

		result.put(classname+".java", scanner);

		return result;
	}

	private String buildEmptyScanner(Options options)
	{
		StringBuffer result = new StringBuffer();

		String package_ = (String) options.pkgName;

		result.append(emitPackage(package_));

		String cls =
		"public class "+options.scannerName+" implements Constants\n"+
		"{\n"+
		"    public Token nextToken() throws LexicalError\n"+
		"    {\n"+
		"        return null;\n"+
		"    }\n"+
		"}\n"+
		"";

		result.append(cls);

		return result.toString();
	}

	private String buildScanner(FiniteAutomata fa, Options options)
	{
		String inType;
		String inInit;
		String inDef;
		if(options.input == Options.INPUT_STREAM)
		{
			inType = "java.io.Reader";
			inInit =
				"StringBuffer bfr = new StringBuffer();\n"+
			"	try\n"+
			"	{\n"+
			"	    int c = input.read();\n"+
			"	    while (c != -1)\n"+
			"	    {\n"+
			"		bfr.append((char)c);\n"+
			"		c = input.read();\n"+
			"	    }\n"+
			"	    this.input = bfr.toString();\n"+
			"	}\n"+
			"	catch (java.io.IOException e)\n"+
			"	{\n"+
			"	    e.printStackTrace();\n"+
```

```
                "       }\n"+
                "";
                inDef = "this(new java.io.StringReader(\"\"));";
        }
        else if(options.input == Options.INPUT_STRING)
        {
                inType = "String";
                inInit = "this.input = input;";
                inDef = "this(\"\");";
        }
        else
        {
                //nunca acontece
                inType = "";
                inInit = "";
                inDef  = "";
        }

        String package_ = options.pkgName;

        String cls =
        emitPackage(package_)+
        "public class "+options.scannerName+" implements Constants\n"+
        "{\n"+
        "    private int position;\n"+
        "    private String input;\n"+
        "\n"+
        "    public "+options.scannerName+"()\n"+
        "    {\n"+
        "        "+inDef+"\n"+
        "    }\n"+
        "\n"+
        "    public "+options.scannerName+"("+inType+" input)\n"+
        "    {\n"+
        "        setInput(input);\n"+
        "    }\n"+
        "\n"+
        "    public void setInput("+inType+" input)\n"+
        "    {\n"+
        "        "+inInit+"\n"+
        "        setPosition(0);\n"+
        "    }\n"+
        "\n"+
        "    public void setPosition(int pos)\n"+
        "    {\n"+
        "        position = pos;\n"+
        "    }\n\n"+

        mainDriver(fa)+
        "\n"+
        auxFuncions(fa, options)+

        "}\n"+
        "";

        return cls;
}

private String emitPackage(String package_)
{
        if (package_ != null && !package_.equals(""))
                return "package " + package_ + ";\n\n";
        else
                return "";
}

private String mainDriver(FiniteAutomata fa)
{
        return
        "    public Token nextToken() throws LexicalError\n"+
        "    {\n"+
```

```
                "        if ( ! hasInput() )\n"+
                "            return null;\n"+
                "\n"+
                "        int start = position;\n"+
                "\n"+
                "        int state = 0;\n"+
                "        int lastState = 0;\n"+
                "        int endState = -1;\n"+
                "        int end = -1;\n"+
                (fa.hasContext() ?
                "        int ctxtState = -1;\n"+
                "        int ctxtEnd = -1;\n" : "")+
                "\n"+
                "        while (hasInput())\n"+
                "        {\n"+
                "            lastState = state;\n"+
                "            state = nextState(nextChar(), state);\n"+
                "\n"+
                "            if (state < 0)\n"+
                "                break;\n"+
                "\n"+
                "            else\n"+
                "            {\n"+
                "                if (tokenForState(state) >= 0)\n"+
                "                {\n"+
                "                    endState = state;\n"+
                "                    end = position;\n"+
                "                }\n"+
                (fa.hasContext() ?
                "                if (SCANNER_CONTEXT[state][0] == 1)\n" +
                "                {\n" +
                "                    ctxtState = state;\n" +
                "                    ctxtEnd = position;\n" +
                "                }\n" : "")+
                "            }\n"+
                "        }\n"+
                "        if (endState < 0 || (endState != state &&
tokenForState(lastState) == -2))\n"+
                "            throw new LexicalError(SCANNER_ERROR[lastState], start);\n"+
                "\n"+
                (fa.hasContext() ?
                "        if (ctxtState != -1 && SCANNER_CONTEXT[endState][1] ==
ctxtState)\n"+
                "            end = ctxtEnd;\n"+
                "\n" : "" )+
                "        position = end;\n"+
                "\n"+
                "        int token = tokenForState(endState);\n"+
                "\n"+
                "        if (token == 0)\n"+
                "            return nextToken();\n"+
                "        else\n"+
                "        {\n"+
                "            String lexeme = input.substring(start, end);\n"+
                (lookup ?
                "            token = lookupToken(token, lexeme);\n" : "")+
                "            return new Token(token, lexeme, start);\n"+
                "        }\n"+
                "    }\n"+
                "";
        }

        private String auxFuncions(FiniteAutomata fa, Options options)
        {
                String nextState;

                switch (options.scannerTable)
                {
                        case Options.SCANNER_TABLE_FULL:
                                nextState =
                                        "    private int nextState(char c, int state)\n"+
```

```
                                        "    {\n"+
                                        "        int next = SCANNER_TABLE[state][c];\n"+
                                        "        return next;\n"+
                                        "    }\n";
                                break;
                        case Options.SCANNER_TABLE_COMPACT:
                                nextState =

                                        "    private int nextState(char c, int state)\n"+
                                        "    {\n"+
                                        "        int start =
SCANNER_TABLE_INDEXES[state];\n"+
                                        "        int end   = SCANNER_TABLE_INDEXES[state+1]-
1;\n"+
                                        "\n"+
                                        "        while (start <= end)\n"+
                                        "        {\n"+
                                        "            int half = (start+end)/2;\n"+
                                        "\n"+
                                        "            if (SCANNER_TABLE[half][0] == c)\n"+
                                        "                return SCANNER_TABLE[half][1];\n"+
                                        "            else if (SCANNER_TABLE[half][0] <
c)\n"+
                                        "                start = half+1;\n"+
                                        "            else  //(SCANNER_TABLE[half][0] >
c)\n"+
                                        "                end = half-1;\n"+
                                        "        }\n"+
                                        "\n"+
                                        "        return -1;\n"+
                                        "    }\n";
                                break;
                        case Options.SCANNER_TABLE_HARDCODE:
                        {
                                Map[] trans = fa.getTransitions();
                                StringBuffer casesState = new StringBuffer();
                                for (int i=0; i<trans.length; i++)
                                {
                                        Map m = trans[i];
                                        if (m.size() == 0)
                                                continue;

                                        casesState.append(
                                "            case "+i+":\n"+
                                "                switch (c)\n"+
                                "                {\n");

                                        for (Iterator iter = m.entrySet().iterator();
iter.hasNext(); )
                                        {
                                                Map.Entry entry = (Entry) iter.next();
                                                Character ch = (Character) entry.getKey();
                                                Integer it = (Integer) entry.getValue();
                                                casesState.append(
                                "                    case "+((int)ch.charValue())+": return
"+it+";\n");
                                        }

                                        casesState.append(
                                "                    default: return -1;\n"+
                                "                }\n");
                                }

                                nextState =
                                "    private int nextState(char c, int state)\n"+
                                "    {\n"+
                                "        switch (state)\n"+
                                "        {\n"+
                                casesState.toString()+
                                "            default: return -1;\n"+
                                "        }\n"+
```

```
                        "    }\n";
                }
                        break;
                default:
                        //nunca acontece
                        nextState = null;
        }

        return
        nextState+
        "\n"+
        "    private int tokenForState(int state)\n"+
        "    {\n"+
        "        if (state < 0 || state >= TOKEN_STATE.length)\n"+
        "            return -1;\n"+
        "\n"+
        "        return TOKEN_STATE[state];\n"+
        "    }\n"+
        "\n"+
        (lookup ?
        "    public int lookupToken(int base, String key)\n"+
        "    {\n"+
        "        int start = SPECIAL_CASES_INDEXES[base];\n"+
        "        int end   = SPECIAL_CASES_INDEXES[base+1]-1;\n"+
        "\n"+
        (sensitive?"":
        "        key = key.toUpperCase();\n"+
        "\n")+
        "        while (start <= end)\n"+
        "        {\n"+
        "            int half = (start+end)/2;\n"+
        "            int comp = SPECIAL_CASES_KEYS[half].compareTo(key);\n"+
        "\n"+
        "            if (comp == 0)\n"+
        "                return SPECIAL_CASES_VALUES[half];\n"+
        "            else if (comp < 0)\n"+
        "                start = half+1;\n"+
        "            else  //(comp > 0)\n"+
        "                end = half-1;\n"+
        "        }\n"+
        "\n"+
        "        return base;\n"+
        "    }\n"+
        "\n":"")+
        "    private boolean hasInput()\n"+
        "    {\n"+
        "        return position < input.length();\n"+
        "    }\n"+
        "\n"+
        "    private char nextChar()\n"+
        "    {\n"+
        "        if (hasInput())\n"+
        "            return input.charAt(position++);\n"+
        "        else\n"+
        "            return (char) -1;\n"+
        "    }\n"+
        "";
    }
}
package gesser.gals.simulator;

import gesser.gals.analyser.LexicalError;
import gesser.gals.analyser.Token;
import gesser.gals.generator.OptionsDialog;
import gesser.gals.generator.scanner.FiniteAutomata;

/**
 * @author Gesser
 */

public class FiniteAutomataSimulator implements BasicScanner
```

```
{
	private FiniteAutomata fa;
	private String input = "";
	private int position = 0;
	private boolean sensitive = true;

	public FiniteAutomataSimulator(FiniteAutomata fa)
	{
		this.fa = fa;
		sensitive = OptionsDialog.getInstance().getOptions().scannerSensitive;
	}

	public int analyse(String str)
	{
		int state = 0;

		for (int i=0; i<str.length(); i++)
		{
			state = fa.nextState(str.charAt(i), state);

			if (state <= 0)
				return -1;
		}
		return fa.tokenForState(state);
	}

	public void setInput(String text)
	{
		this.input = text;
		position = 0;
	}

	public Token nextToken() throws LexicalError
	{
		if ( ! hasInput() )
			return null;

		int start = position;

		int state = 0;
		int lastState = 0;
		int endState = -1;
		int end = -1;
		int ctxtState = -1;
		int ctxtEnd = -1;

		while (hasInput())
		{
			lastState = state;
			state = fa.nextState(nextChar(), state);

			if (state < 0)
			{
				break;
			}
			else
			{
				int tfs = fa.tokenForState(state);
				if (tfs >= 0)
				{
					endState = state;
					end = position;
				}
				if (fa.isContext(state))
				{
					ctxtState = state;
					ctxtEnd = position;
				}
			}
		}
```

```
                if (endState < 0 || (endState != state && fa.tokenForState(lastState) == -
2))
                        throw new LexicalError( fa.getError(lastState), start);

                if (ctxtState != -1 && fa.getOrigin(endState) == ctxtState)
                        end = ctxtEnd;
                position = end;

                int token = fa.tokenForState(endState);

                if (token == 0)
                        return nextToken();
                else
                {
                        String lexeme = input.substring(start, end);
                        token = lookupToken(token, lexeme);
                        return new Token(token, lexeme, start);
                }
        }

        public int lookupToken(int base, String key)
        {
                int start = fa.getSpecialCasesIndexes()[base][0];
                int end   = fa.getSpecialCasesIndexes()[base][1]-1;

                if (!sensitive)
                        key = key.toUpperCase();

                while (start <= end)
                {
                        int half = (start+end)/2;
                        int comp = fa.getSpecialCases()[half].key.compareTo(key);

                        if (comp == 0)
                                return fa.getSpecialCases()[half].value;
                        else if (comp < 0)
                                start = half+1;
                        else  //(comp > 0)
                                end = half-1;
                }

                return base;
        }

        private boolean hasInput()
        {
                return position < input.length();
        }

        private char nextChar()
        {
                if (hasInput())
                        return input.charAt(position++);
                else
                        return (char) -1;
        }
}

package gesser.gals.simulator;

import java.util.Stack;

import javax.swing.tree.DefaultMutableTreeNode;

import gesser.gals.analyser.*;
import gesser.gals.generator.parser.*;
import gesser.gals.generator.parser.ll.*;
import gesser.gals.util.IntList;
import gesser.gals.util.ProductionList;

public class LL1ParserSimulator
```

```
{
       public static final int EPSILON = 0;
    public static final int DOLLAR = 1;
    public static final int FIRST_TERMINAL = 2;
    public final int FIRST_NON_TERMINAL;
    public final int FIRST_SEMANTIC_ACTION;
    public final int LAST_SEMANTIC_ACTION;

    public final int START_SYMBOL;

    private Grammar grammar;
       private BasicScanner scanner;

       private int[][] table;
       private int[][] productions;

       private Stack stack = new Stack();
       private Token currentToken;

       String[] symb;
       DefaultMutableTreeNode node;

       public LL1ParserSimulator(LLParser parser)
       {
              this.grammar = parser.getGrammar();
              table = parser.generateTable();
              FIRST_NON_TERMINAL = grammar.FIRST_NON_TERMINAL;
              FIRST_SEMANTIC_ACTION = grammar.FIRST_SEMANTIC_ACTION();
              LAST_SEMANTIC_ACTION = grammar.LAST_SEMANTIC_ACTION();
              START_SYMBOL = grammar.getStartSymbol();

              ProductionList p = grammar.getProductions();
              productions = new int[p.size()][];
              for (int i=0; i< p.size(); i++)
              {
                     IntList rhs = p.getProd(i).get_rhs();
                     if (rhs.size() > 0)
                     {
                            productions[i] = new int[rhs.size()];
                            for (int j=0; j<rhs.size(); j++)
                                   productions[i][j] = rhs.get(j);
                     }
                     else
                            productions[i] = new int[]{0};
              }

              symb = grammar.getSymbols();
       }

       Stack nodeCount = new Stack();

       public boolean step() throws LexicalError, SyntaticError, SemanticError
       {
              if (currentToken == null)
              {
                     currentToken = new Token(DOLLAR, "$", 0);
              }
              int x = ((Integer)stack.pop()).intValue();
              int a = currentToken.getId();

              if (x == EPSILON)
              {
                     node.add(new DefaultMutableTreeNode("EPSILON"));

                     Integer itg = (Integer) nodeCount.pop();
                     while (itg.intValue() == 1)
                     {
                            node = (DefaultMutableTreeNode) node.getParent();
                            if ( nodeCount.size() > 0 )
                                   itg = (Integer) nodeCount.pop();
                            else
```

```
                                break;
                        }
                        nodeCount.push(new Integer(itg.intValue()-1));

                        return false;
                }
                else if (isTerminal(x))
                {
                        node.add(new DefaultMutableTreeNode(symb[a]));

                        Integer itg = (Integer) nodeCount.pop();
                        while (itg.intValue() == 1)
                        {
                                node = (DefaultMutableTreeNode) node.getParent();
                                if ( nodeCount.size() > 0 )
                                        itg = (Integer) nodeCount.pop();
                                else
                                        break;
                        }
                        nodeCount.push(new Integer(itg.intValue()-1));

                        if (x == a)
                        {
                                if (stack.empty())
                                        return true;
                                else
                                {
                                        currentToken = scanner.nextToken();
                                        return false;
                                }
                        }
                        else
                        {
                                node.add(new DefaultMutableTreeNode("ERRO SINTÁTICO: Era
esperado "+symb[x]));
                                throw new SyntaticError("Era esperado "+symb[x],
currentToken.getPosition());
                        }
                }
                else if (isNonTerminal(x))
                {
                        int p = table[x-FIRST_NON_TERMINAL][a-1];
                        if (p != -1)
                        {
                                int[] production = productions[p];
                                //empilha a produção em ordem reversa
                                for (int i=production.length-1; i>=0; i--)
                                {
                                        stack.push(new Integer(production[i]));
                                }
                                DefaultMutableTreeNode n = new
DefaultMutableTreeNode(symb[x]);
                                node.add(n);
                                node = n;
                                nodeCount.push(new Integer(production.length));
                                return false;
                        }
                        else
                        {
                                node.add(new DefaultMutableTreeNode("ERRO SINTÁTICO:
"+symb[a]+" inesperado"));
                                throw new SyntaticError(symb[a]+" inesperado",
currentToken.getPosition());
                        }
                }
                else if (isSemanticAction(x))
                {
                        node.add(new DefaultMutableTreeNode("#"+(x-
FIRST_SEMANTIC_ACTION)));

                        Integer itg = (Integer) nodeCount.pop();
```

```
                        while (itg.intValue() == 1)
                        {
                                node = (DefaultMutableTreeNode) node.getParent();
                                if ( nodeCount.size() > 0 )
                                        itg = (Integer) nodeCount.pop();
                                else
                                        break;
                        }
                        nodeCount.push(new Integer(itg.intValue()-1));

                        return false;
                }
                else
                {
                        //ERRO: impossivel
                        //assert false : "Erro Impossivel";
                        return false;
                }
        }

        public void parse(BasicScanner scnr, DefaultMutableTreeNode root) throws
LexicalError, SyntaticError, SemanticError
        {
                scanner = scnr;
                node = root;
                nodeCount.clear();
                stack.clear();

                stack.push(new Integer(DOLLAR));
                stack.push(new Integer(START_SYMBOL));

                currentToken = scanner.nextToken();

                while ( ! step() )
                        ; //faz nada
        }

        /**
     * @return TRUE se x eh um símbolo terminal
     */
    private final boolean isTerminal(int x)
    {
        return x >= 0 && x < FIRST_NON_TERMINAL;
    }

    /**
     * @return TRUE se x eh um símbolo não terminal
     */
    private final boolean isNonTerminal(int x)
    {
        return x >= FIRST_NON_TERMINAL && x < FIRST_SEMANTIC_ACTION;
    }

    /**
     * @return TRUE se x eh uma Ação Semântica
     */
    private final boolean isSemanticAction(int x)
    {
        return x >= FIRST_SEMANTIC_ACTION && x <= LAST_SEMANTIC_ACTION;
    }
}

package gesser.gals.generator.parser.lr;

import java.util.Stack;

import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.MutableTreeNode;

import gesser.gals.analyser.AnalysisError;
import gesser.gals.analyser.LexicalError;
```

```
import gesser.gals.analyser.SemanticError;
import gesser.gals.analyser.SyntaticError;
import gesser.gals.analyser.Token;
import gesser.gals.simulator.BasicScanner;
import gesser.gals.util.ProductionList;


public class LRParserSimulator
{
	private Stack stack = new Stack();

	private BasicScanner scanner;
	private Token currentToken = null;
	private Token previousToken = null;

	private Command[][] table;
	private int[][] productions;
	private int semanticStart;

	private String[] symbols;
	private Stack nodeStack = new Stack();

	public static final int DOLLAR = 1;

	public LRParserSimulator(SLRParser parser)
	{
		table = parser.buildTable();
		semanticStart = parser.semanticStart;
		ProductionList pl = parser.g.getProductions();
		productions = new int[pl.size()][2];

		symbols = parser.g.getSymbols();

		for (int i=0; i<pl.size(); i++)
		{
			productions[i][0] = pl.getProd(i).get_lhs();
			productions[i][1] = pl.getProd(i).get_rhs().size();
		}
	}

	public void parse(BasicScanner scanner, DefaultMutableTreeNode root) throws
SemanticError, SyntaticError, SyntaticError, LexicalError
	{
		this.scanner = scanner;

		nodeStack.clear();

		stack.clear();
		stack.push(new Integer(0));

		currentToken = scanner.nextToken();

		try
		{
			while ( ! step() )
				; //faz nada
			root.add((MutableTreeNode)nodeStack.pop());
		}
		catch(AnalysisError e)
		{
			for (int i=0; i<nodeStack.size(); i++)
				root.add((MutableTreeNode)nodeStack.get(i));
			root.add(new DefaultMutableTreeNode(e.getMessage()));

			e.printStackTrace();
		}
	}

	private boolean step() throws SyntaticError, SemanticError, LexicalError
	{
		int state = ((Integer)stack.peek()).intValue();
```

```
		if (currentToken == null)
		{
			int pos = 0;
			if (previousToken != null)
				pos =
previousToken.getPosition()+previousToken.getLexeme().length();

			currentToken = new Token(DOLLAR, "$", pos);
		}

	int token = currentToken.getId();

		Command cmd = table[state][token-1];

		switch (cmd.getType())
		{
			case Command.SHIFT:
				stack.push(new Integer(cmd.getParameter()));

				nodeStack.push(new
DefaultMutableTreeNode(symbols[currentToken.getId()]));

				previousToken = currentToken;
				currentToken = scanner.nextToken();
				return false;

			case Command.REDUCE:
				int[] prod = productions[cmd.getParameter()];

				Stack tmp = new Stack();
				for (int i=0; i<prod[1]; i++)
				{
					stack.pop();
					tmp.push(nodeStack.pop());

				}
				int oldState = ((Integer)stack.peek()).intValue();
				stack.push(new Integer(table[oldState][prod[0]-
1].getParameter()));

				DefaultMutableTreeNode node = new
DefaultMutableTreeNode(symbols[prod[0]]);
				while (!tmp.isEmpty())
				{
					node.add((MutableTreeNode)tmp.pop());
				}
				nodeStack.push(node);
				return false;

			case Command.ACTION:
				int action = semanticStart + cmd.getParameter() - 1;
				stack.push(new
Integer(table[state][action].getParameter()));
				nodeStack.push(new
DefaultMutableTreeNode("#"+cmd.getParameter()));
				//semanticAnalyser.executeAction(cmd.getParameter(),
previousToken);
				return false;
			/*
			case Command.GOTO:
				break;
			*/
			case Command.ACCEPT:
				return true;

			case Command.ERROR:
				throw new SyntaticError("Erro sintático",
currentToken.getPosition());
		}
		return false;
```

```
        }
}

package gesser.gals.generator.parser;

import gesser.gals.HTMLDialog;
import gesser.gals.util.BitSetIterator;
import gesser.gals.util.IntList;
import gesser.gals.util.ProductionList;
import java.util.*;

/**
 * A classe Grammar representa as Gramáticas Livres de Contexto, utilizadas
 * pelos análisadores sintáticos
 *
 * @author Carlos Eduardo Gesser
 */

public class Grammar implements Cloneable
{
    public static final int EPSILON = 0;
    public static final int DOLLAR = 1;
    public static final int FIRST_TERMINAL = EPSILON+2;

    public static final String EPSILON_STR = "î";

    protected String[] symbols;
    public int FIRST_NON_TERMINAL = 0;
    public int FIRST_SEMANTIC_ACTION() { return symbols.length; };
    public int LAST_SEMANTIC_ACTION() { return
FIRST_SEMANTIC_ACTION()+SEMANTIC_ACTION_COUNT; };
    public int SEMANTIC_ACTION_COUNT = 0;
    protected int startSymbol;

    public BitSet[] firstSet;
    public BitSet[] followSet;

    protected ProductionList productions = new ProductionList();

    /**
     * Contrói um objeto do tipo Grammar
     *
     * @param t símbolos terminais
     * @param n símbolos não terminais
     * @param p produções
     * @param startSymbol súimbolo inicial da gramática
     */
    public Grammar(String[] t, String[] n, ProductionList p, int startSymbol)
    {
        setSymbols(t, n, startSymbol);
        setProductions(p);
        fillFirstSet();
        fillFollowSet();
    }

       /**
     * Contrói um objeto do tipo Grammar
     *
     * @param t símbolos terminais
     * @param n símbolos não terminais
     * @param p produções
     * @param startSymbol súimbolo inicial da gramática
     */
       public Grammar(List t, List n, List p, int start)
       {
              String[] T = new String[t.size()];
       System.arraycopy(t.toArray(), 0, T, 0, T.length);
       String[] N = new String[n.size()];
       System.arraycopy(n.toArray(), 0, N, 0, N.length);
              ProductionList P = new ProductionList();
              P.addAll(p);
```

```
            setSymbols(T, N, start);
        setProductions(P);
        fillFirstSet();
        fillFollowSet();
       }


/**
 * Preenche os símbolos e inicializa arrays;
 *
 * @param t símbolos terminais
 * @param n símbolos não terminais
 */
private void setSymbols(String[] t, String[] n, int startSymbol)
{
    symbols = new String[t.length + n.length + 2];
    FIRST_NON_TERMINAL = t.length + 2;
    symbols[EPSILON] = EPSILON_STR;
    symbols[DOLLAR] = "$";
    for (int i = 0, j = FIRST_TERMINAL; i < t.length; i++, j++)
        symbols[j] = t[i];

    for (int i = 0, j = FIRST_NON_TERMINAL; i < n.length; i++, j++)
        symbols[j] = n[i];

    this.startSymbol = startSymbol;
}

/**
 * @param p produções
 */
private void setProductions(ProductionList p)
{
    productions.add(p);
    int max = 0;
    for (int i=0; i<productions.size(); i++)
    {
            productions.getProd(i).setGrammar(this);
            for (int j=0; j<productions.getProd(i).get_rhs().size(); j++)
                    if (productions.getProd(i).get_rhs().get(j) > max)
                            max = productions.getProd(i).get_rhs().get(j);
    }
    SEMANTIC_ACTION_COUNT = max - FIRST_SEMANTIC_ACTION();
}

/**
 * @return TRUE se x eh um símbolo terminal
 */
public final boolean isTerminal(int x)
{
    return x < FIRST_NON_TERMINAL;
}

/**
 * @return TRUE se x eh um símbolo não terminal
 */
public final boolean isNonTerminal(int x)
{
    return x >= FIRST_NON_TERMINAL && x < FIRST_SEMANTIC_ACTION();
}

public final boolean isSemanticAction(int x)
{
    return x >= FIRST_SEMANTIC_ACTION();
}

    public ProductionList getProductions()
    {
            return productions;
    }
```

```
	public String[] getSymbols()
	{
		return symbols;
	}

	public String[] getTerminals()
	{
		String[] terminals = new String[FIRST_NON_TERMINAL-2];
		System.arraycopy(symbols,2,terminals,0,terminals.length);
		return terminals;
	}

	public String[] getNonTerminals()
	{
		String[] nonTerminals = new String[FIRST_SEMANTIC_ACTION() -
FIRST_NON_TERMINAL];

	System.arraycopy(symbols,FIRST_NON_TERMINAL,nonTerminals,0,nonTerminals.length);
		return nonTerminals;
	}

	public int getStartSymbol()
	{
		return startSymbol;
	}

	/**
	 * Cria uma nova produção. Se a produção criada já existe na gramática,
	 * null é retornado.
	 *
	 * @param lhs lado esquerdo da produção
	 * @param rhs lado direito da produção
	 *
	 * @return produção gerada, ou null se esta já existir
	 * */
	public Production createProduction(int lhs, int[] rhs)
	{
		Production p = new Production(this, lhs, rhs);
		for (int i = 0; i < productions.size(); i++)
			if (productions.getProd(i).equals( p ))
				return null;

		return p;
	}

	/**
	 * Cria uma nova produção. Se a produção criada já existe na gramática,
	 * null é retornado.
	 *
	 * @param lhs lado esquerdo da produção
	 * @param rhs lado direito da produção
	 *
	 * @return produção gerada, ou null se esta já existir
	 * */
	public Production createProduction(int lhs, IntList rhs)
	{
		Production p = new Production(this, lhs, rhs);
		for (int i = 0; i < productions.size(); i++)
			if (productions.getProd(i).equals( p ))
				return null;

		return p;
	}

	public Production createProduction(int lhs)
	{
		return new Production(this, lhs, new IntList());
	}

	protected boolean isEpsilon(IntList x, int start)
```

```
    {
            for (int i=start; i<x.size(); i++)
                    if (! isSemanticAction(x.get(i)))
                            return false;
            return true;
    }

    protected boolean isEpsilon(IntList x)
    {
            return isEpsilon(x, 0);
    }

    /**
 * @return BitSet indicando os symbolos que derivam Epsilon
 */
private BitSet markEpsilon()
{
    BitSet result = new BitSet();

    for (int i = 0; i < productions.size(); i++)
    {
        Production P = productions.getProd(i);
        if (isEpsilon(P.get_rhs()))
            result.set(P.get_lhs());
    }
    for (int i=FIRST_SEMANTIC_ACTION(); i <= LAST_SEMANTIC_ACTION(); i++)
           result.set(i);

    boolean change = true;;
    while (change)
    {
        change = false;
        boolean derivesEpsilon;
        for (int i = 0; i < productions.size(); i++)
        {
            Production P = productions.getProd(i);
            derivesEpsilon = true;
            for (int j = 0; j < P.get_rhs().size(); j++)
            {
                derivesEpsilon = derivesEpsilon && result.get(P.get_rhs().get(j));
            }
            if (derivesEpsilon && !result.get(P.get_lhs()))
            {
                change = true;
                result.set(P.get_lhs());
            }
        }
    }
    return result;
}

    private static final BitSet EMPTY_SET = new BitSet();
    static { EMPTY_SET.set(EPSILON); }
    public BitSet first(int symbol)
    {
            if (isSemanticAction(symbol))
                    return EMPTY_SET;
            else
                    return firstSet[symbol];
    }

    public BitSet first(IntList x)
    {
            return first(x, 0);
    }

    public BitSet first(IntList x, int start)
    {
            BitSet result = new BitSet();

            if (isEpsilon(x, start))
```

```
                result.set(EPSILON);
            else
            {
                int k = x.size();
                while (isSemanticAction(x.get(start)))
                    start++;

                BitSet f = (BitSet)first(x.get(start)).clone();
            f.clear(EPSILON);
            result.or(f);
            int i=start;
            while (i < k-1 && first(x.get(i)).get(EPSILON))
            {
                i++;
                f = (BitSet)first(x.get(i)).clone();
                f.clear(EPSILON);
                result.or(f);
            }
            if (i == k-1 && first(x.get(i)).get(EPSILON))
                result.set(EPSILON);
            }
            return result;
        }

        /**
     * Calcula os conjuntos FIRST de todos os símbolos de Gramática
     */
    private void fillFirstSet()
    {
        BitSet derivesEpsilon = markEpsilon();
        firstSet = new BitSet[symbols.length];
        for (int i = 0; i < firstSet.length; i++)
        {
            firstSet[i] = new BitSet();
        }

        for (int A = FIRST_NON_TERMINAL; A < FIRST_SEMANTIC_ACTION(); A++)
        {
            if (derivesEpsilon.get(A))
                firstSet[A].set(EPSILON);
        }
        for (int a = FIRST_TERMINAL; a < FIRST_NON_TERMINAL; a++)
        {
            firstSet[a].set(a);
            for (int A = FIRST_NON_TERMINAL; A < FIRST_SEMANTIC_ACTION(); A++)
            {
                boolean exists = false;
                for (int i = 0; i < productions.size(); i++)
                {
                    Production P = productions.getProd(i);
                    if (P.get_lhs() == A && !isEpsilon(P.get_rhs()) && P.firstSymbol() ==
a)
                    {
                        exists = true;
                        break;
                    }
                }
                if (exists)
                    firstSet[A].set(a);
            }
        }
        boolean changed;
        do
        {
            changed = false;
            for (int i = 0; i < productions.size(); i++)
            {
                Production P = productions.getProd(i);
                BitSet old = (BitSet)firstSet[P.get_lhs()].clone();
                firstSet[P.get_lhs()].or(first(P.get_rhs()));
                if (!changed && !old.equals(first(P.get_lhs())) )
```

```
                changed = true;
            }
        }
        while (changed);
    }

       /**
     * Calcula os conjuntos FOLLOW de todos os símbolos não terminais de Gramática
     */
    private void fillFollowSet()
    {
        followSet = new BitSet[symbols.length];
        for (int i = 0; i < followSet.length; i++)
        {
            followSet[i] = new BitSet();
        }
        followSet[startSymbol].set(DOLLAR);
        boolean changes;
        do
        {
            changes = false;
            for (int i = 0; i < productions.size(); i++)
            {
                Production P = productions.getProd(i);
                for (int j=0;j<P.get_rhs().size(); j++)
                {
                    if (isNonTerminal(P.get_rhs().get(j)))
                    {
                        BitSet s = first(P.get_rhs(), j+1);
                        boolean deriveEpsilon = s.get(EPSILON);

                        if( P.get_rhs().size() > j+1 )
                        {
                            s.clear(EPSILON);
                            BitSet old = (BitSet)followSet[P.get_rhs().get(j)].clone();
                            followSet[P.get_rhs().get(j)].or(s);
                            if (!changes && !followSet[P.get_rhs().get(j)].equals(old))
                                changes = true;
                        }

                        if (deriveEpsilon)
                        {
                            BitSet old = (BitSet)followSet[P.get_rhs().get(j)].clone();
                            followSet[P.get_rhs().get(j)].or(followSet[P.get_lhs()]);
                            if (!changes && !followSet[P.get_rhs().get(j)].equals(old))
                                changes = true;
                        }
                    }
                }
            }
        }
        while (changes);
    }

    /**
     * Gera uma representação String dos conjuntos First e Follow
     * @return First e Follow como uma String
     */
    public String stringFirstFollow()
    {
        StringBuffer result = new StringBuffer();
        for (int i = FIRST_NON_TERMINAL; i < firstSet.length; i++)
        {
            StringBuffer bfr = new StringBuffer();
            bfr.append("FIRST(").append(symbols[i]).append(") = { ");
            for (int j = 0; j < firstSet[i].size(); j++)
            {
                if (firstSet[i].get(j))
                    bfr.append("").append(symbols[j]).append(" ");
            }
            bfr.append("}");
```

```
            result.append(bfr).append('\n');
        }
        for (int i = FIRST_NON_TERMINAL; i < followSet.length; i++)
        {
            StringBuffer bfr = new StringBuffer();
            bfr.append("FOLLOW(").append(symbols[i]).append(") = { ");
            for (int j = 0; j < followSet[i].size(); j++)
            {
                if (followSet[i].get(j))
                    bfr.append(symbols[j]).append(" ");
            }
            bfr.append("}");
            result.append(bfr).append('\n');
        }
        return result.toString();
    }

    public String ffAsHTML()
    {
       StringBuffer result = new StringBuffer();

              result.append(
                     "<HTML>"+
                     "<HEAD>"+
                     "<TITLE>First &amp; Follow</TITLE>"+
                     "</HEAD>"+
                     "<BODY><FONT face=\"Verdana, Arial, Helvetica, sans-serif\">"+
                     "<TABLE border=1 cellspacing=0>");

              result.append(
                     "<TR align=center>"+
                     "<TD bgcolor=black><FONT color=white><B>SÍMBOLO</B></FONT></TD>"+
                     "<TD bgcolor=black><FONT color=white><B>FIRST</B></FONT></TD>"+
                     "<TD bgcolor=black><FONT color=white><B>FOLLOW</B></FONT></TD>"+
                     "</TR>");

              for (int i = FIRST_NON_TERMINAL; i < FIRST_SEMANTIC_ACTION(); i++)
        {
              result.append("<TR align=center>");

                     result.append("<TD nowrap
bgcolor=#F5F5F5><B>"+HTMLDialog.translateString(symbols[i])+"</B></TD>");

                     StringBuffer bfr = new StringBuffer("  ");
            for (int j = 0; j < firstSet[i].size(); j++)
            {
                if (firstSet[i].get(j))
                    bfr.append(symbols[j]).append(", ");
            }
            bfr.setLength(bfr.length()-2);

            result.append("<TD nowrap
bgcolor=#F5F5F5>"+HTMLDialog.translateString(bfr.toString())+"</TD>");

            bfr = new StringBuffer("  ");
            for (int j = 0; j < followSet[i].size(); j++)
            {
                if (followSet[i].get(j))
                    bfr.append(symbols[j]).append(", ");
            }
            bfr.setLength(bfr.length()-2);

            result.append("<TD nowrap
bgcolor=#F5F5F5>"+HTMLDialog.translateString(bfr.toString())+"</TD>");

            result.append("</TR>");
        }

              result.append(
                     "</TABLE>"+
                     "</FONT></BODY>"+
```

```
                "</HTML>");

        return result.toString();
}

/**
 * Remove os estados improdutivos da gramática
 * @throws EmptyGrammarException se o símbolo inicial for removido
 */
protected void removeImproductiveSymbols() throws EmptyGrammarException
{
    BitSet SP = getProductiveSymbols();

    updateSymbols(SP);
}

/**
 * Remove os estados inúteis, os inprodutívos e os inalcansáveis
 * @throws EmptyGrammarException se o símbolo inicial for removido
 */
public void removeUselessSymbols() throws EmptyGrammarException
{
    removeImproductiveSymbols();
    removeUnreachableSymbols();
    //removeRepeatedProductions();
}

   /**
    * Elimina as produções repetidas da gramática.
    */
   private void removeRepeatedProductions() throws EmptyGrammarException
   {/*
           BitSet repeated = new BitSet();
           sortProductions();

           Production p = productions[0];
           for (int i = 1; i < productions.length; i++)
           {
                   Production local = productions[i];
                   if (local.equals(p))
                           repeated.set(i);
                   p = local;
           }

           //retira as produçoes que não possuem símbolos úteis
    Production[] P = new Production[productions.length];
    int k = 0;
    for (int i=0;i<productions.length;i++)
    {
        if (! repeated.get(i))
            P[k++] = productions[i];
    }
    productions = new Production[k];
    for (int i=0; i< productions.length; i++)
        productions[i] = P[i];*/
   }

/**
 * Calcula as produções cujo lado esquerdo é <code>symbol</code>
 * @return BitSet indicando essas produções
 */
public BitSet productionsFor(int symbol)
{
   BitSet result = new BitSet();
    for (int i = 0; i < productions.size(); i++)
    {
        if (productions.getProd(i).get_lhs() == symbol)
            result.set(i);
    }
    return result;
}
```

```
    /**
     * Transforma as recursões à esquerda indiretas em recusões diretas
     * @param prods produções para serem processadas
     * @return lista de produçoes sem recursão indireta
     */
    private ProductionList transformToFindRecursion(ProductionList prods)
    {
        ProductionList prodList = new ProductionList();
        prodList.addAll(prods);
        for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++ )
        {
            for (int j=FIRST_NON_TERMINAL; j<i; j++)
            {
                for (int it = 0; it < prodList.size(); it++)
                {
                    Production P = (Production)prodList.get(it);
                    if (P.get_lhs() == i && P.firstSymbol() == j)
                    {
                        prodList.remove(it);
                        it--;
                        IntList actions = new IntList();
                        for (int k = 0; k < P.get_rhs().size() &&
isSemanticAction(P.get_rhs().get(k)); k++)
                                                    actions.add(P.get_rhs().get(k));

                        for (int it2 = 0; it2 < prodList.size(); it2++)
                        {
                            Production P2 = (Production)prodList.get(it2);
                            if (P2.get_lhs() == j)
                            {
                                int[] rhs = new int[P2.get_rhs().size() +
P.get_rhs().size()-1];
                                int k = 0;
                                for ( ; k<actions.size(); k++)
                                     rhs[k] = actions.get(k);
                                int m = k;
                                for ( k = 0 ; k<P2.get_rhs().size(); k++)
                                    rhs[k + m] = P2.get_rhs().get(k);
                                m = m + k - (actions.size() + 1);
                                for ( k = actions.size() + 1; k<P.get_rhs().size(); k++)
                                    rhs[k + m] = P.get_rhs().get(k);

                                Production newProduction = createProduction(P.get_lhs(),
rhs);
                                if (newProduction != null)
                                    prodList.add(newProduction);
                            }
                        }
                    }
                }
            }
        }
        return prodList;
    }

    /**
     * Remove as recursões á esquerda da gramática.
     * Primeiramente transforma a gramática para que as recursões
     * indiretas se tornem diretas. Em seguida remove as recursões
     * diretas
     */
    public void removeRecursion()
    {
        productions = transformToFindRecursion(productions);
        removeDirectRecursion();
    }

    /**
     * Remove as recursões á esquerda da gramática.
     *  É preciso que não existam recursões indiretas
```

```
     */
    private void removeDirectRecursion()
    {
        for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++)
        {
            BitSet recursive = productionsFor(i);
            BitSet prods = productionsFor(i);
            int newSymbol = -1;
            for (BitSetIterator iter = new BitSetIterator(recursive); iter.hasNext();)
            {
                int x = iter.nextInt();
                if (productions.getProd(x).get_lhs() !=
productions.getProd(x).firstSymbol())
                    iter.remove();
            }
            if (recursive.length() > 0)
            {
                newSymbol = createSymbol(addTail(symbols[i]));
                for (BitSetIterator iter = new BitSetIterator(prods); iter.hasNext();)
                {
                    int x = iter.nextInt();
                    Production P = productions.getProd(x);
                    if (recursive.get(x))
                    {
                        P.get_rhs().remove(0);
                        P.get_rhs().add(newSymbol);
                        P.set_lhs(newSymbol);
                    }
                    else
                    {
                       P.get_rhs().add(newSymbol);
                    }
                }
            }
            if (newSymbol != -1)
                productions.add( createProduction(newSymbol) );
        }
        fillFirstSet();
        fillFollowSet();
        sort();
    }

        private int createSymbol(String s)
        {
               for (Iterator i = productions.iterator(); i.hasNext();)
               {
                      Production p = (Production) i.next();
                      IntList rhs = p.get_rhs();
                      for (int j=0; j<rhs.size(); j++)
                             if (isSemanticAction(rhs.get(j)))
                                    rhs.set(j, rhs.get(j) + 1);
               }
               String[] newSymbols = new String[symbols.length+1];
               System.arraycopy(symbols,0,newSymbols,0,symbols.length);
               symbols = newSymbols;
               symbols[symbols.length-1] = s;

               return symbols.length-1;
        }

        /**
         * Verifica se o símbolo a deriva o simbolo b em 0 ou mais passos.
         *
         * @param a índice do primeiro símbolo
         * @param b índice do segundo símbolo
         */
        private boolean derives(int a, int b)
        {
               if (a == b)
                      return true;
```

```
		BitSet src = new BitSet();

		src.set(b);

		for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++)
		{
			for (BitSetIterator it = new BitSetIterator(src); it.hasNext(); )
			{
				int cur = it.nextInt();
				if (derivesDirectly(i, cur) && !src.get(i))
				{
					src.set(i);
					i = -1;
					//break;
					continue;
				}
			}
		}

		return src.get(a);
	}

	/**
	 * Verifica se o símbolo a deriva o simbolo b diretamente.
	 *
	 * @param a índice do primeiro símbolo
	 * @param b índice do segundo símbolo
	 */
	private boolean derivesDirectly(int a, int b)
	{
		BitSet derivesEpsilon = markEpsilon();

		for (int i=0; i<productions.size(); i++)
		{
			Production p = productions.getProd(i);

			if (p.get_lhs() == a)
			{
				if (p.get_rhs().size() == 1)
				{
					if (p.get_rhs().get(0) == b)
						return true;
				}
				else
				{
					IntList rhs = p.get_rhs();

					for (int j=0; j<rhs.size(); j++)
					{
						if (rhs.get(j) == b)
						{
							boolean allEpsilon = true;
							for (int k=0; k<j; k++)
							{
								if (!
derivesEpsilon.get(rhs.get(k)))
									allEpsilon = false;
							}
							for (int k=j+1; k<rhs.size(); k++)
							{
								if (!
derivesEpsilon.get(rhs.get(k)))
									allEpsilon = false;
							}
							if (allEpsilon)
								return true;
						}
					}
				}
			}
		}
```

```
            return false;
    }

/**
 * Remove as produçoes Unitárias.
 * Estas produções são aquelas da forma A ::= X, onde X é um não-terminal.
 */
public void removeUnitaryProductions()
{
            ProductionList prods = new ProductionList();
    // as produções que NÃO são ciclos são adicionadas a prods
    for (int i = 0; i < productions.size(); i++)
    {
        Production p = productions.getProd(i);
        if (p.get_rhs().size() != 1 || p.get_rhs().get(0) != p.get_lhs())
            prods.add(p);
    }

    BitSet[] N = new BitSet[symbols.length];

    for (int i=FIRST_NON_TERMINAL; i < N.length; i++)
    {
           N[i] = new BitSet();
           for (int j=FIRST_NON_TERMINAL; j<FIRST_SEMANTIC_ACTION(); j++)
                  if (derives(i, j))
                         N[i].set(j);
    }

    productions.clear();

    for (int i=0; i<prods.size(); i++)
    {
           Production p = prods.getProd(i);
           if (p.get_rhs().size() != 1 || !isNonTerminal(p.get_rhs().get(0)))
           {
                  for (int j=FIRST_NON_TERMINAL; j<N.length; j++)
                  {
                         if (N[j].get(p.get_lhs()))
                         {
                                Production np = createProduction(j,p.get_rhs());
                                if (np != null)
                                       productions.add(np);
                         }
                  }
           }
    }

    //TODO: terminar este algoritimo
    sort();
}

/**
 * Remove as Epsilon-Produções da Gramática
 */
public void removeEpsilon()
{
            BitSet E = markEpsilon();
    ProductionList prods = new ProductionList();

    for (int i=0; i<productions.size(); i++)
    {
           Production p = productions.getProd(i);
           if ( ! isEpsilon( p.get_rhs() ) )
           {
                  boolean derivesEpsilon = true;
            for (int j = 0; j < p.get_rhs().size(); j++)
            {
                derivesEpsilon = derivesEpsilon && E.get(p.get_rhs().get(j));
            }
            if (! derivesEpsilon)
                         prods.add(p);
```

```
                }
            }

            for (int it = 0; it < prods.size(); it++)
            {
                Production p = prods.getProd(it);

                if (! isEpsilon( p.get_rhs() ))//?INUTIL?
                {
                    int i=0;
                    while (i < p.get_rhs().size())
                    {
                        //procura pelo epsilon-NT
                        for (; i<p.get_rhs().size(); i++)
                        {
                            if (!isSemanticAction(p.get_rhs().get(i)) &&
E.get(p.get_rhs().get(i)))
                                break;
                        }
                        if (i < p.get_rhs().size())
                        {
                           Production pNew = derivationAt(p, i);
                           if (pNew != null && !prods.contains(pNew))
                                   prods.add(pNew);
                            i++;
                        }
                    }
                }
            }
            if (E.get(startSymbol))
            {
             //   String newSymbol = ;
//
  //          String[] s = new String[symbols.length+1];
    //        System.arraycopy(symbols, 0, s, 0, symbols.length);
      //      symbols = s;
        //    int newPos = symbols.length-1;
          //  symbols[newPos] = newSymbol;

                int newPos = createSymbol(addTail(symbols[startSymbol]));

                prods.add(createProduction(newPos, new int[]{startSymbol}));
                prods.add(createProduction(newPos));
                startSymbol = newPos;

                fillFirstSet();
                fillFollowSet();
            }
            productions = prods;

            sort();
        }

        private Production derivationAt(Production p, int index)
        {
            IntList rhsP = new IntList();
            for (int k=0; k<productions.size(); k++)
            {
                   if ( (productions.getProd(k).get_lhs() == p.get_rhs().get(index)) &&
                               (isEpsilon(productions.getProd(k).get_rhs())) )
                   {
                           rhsP = productions.getProd(k).get_rhs();
                           break;
                   }
            }
            IntList rhs = new IntList();
            //int[] rhs = new int[p.get_rhs().size()-1];
            for (int k=0; k < index; k++)
                rhs.add(p.get_rhs().get(k));
            for (int k=0; k < rhsP.size(); k++)
                   rhs.add(rhsP.get(k));
```

```
    for (int k=index+1; k < p.get_rhs().size(); k++)
        rhs.add(p.get_rhs().get(k));

    return createProduction(p.get_lhs(), rhs.toArray());
}

private String addTail(String s)
{
   s = s.substring(0,s.length()-1) + "_T>";

    for (int i = 0; i < symbols.length; i++)
    {
        if (symbols[i] != null && symbols[i].equals(s))
        {
            s = s.substring(0,s.length()-1) + "_T>";
            i = 0;
        }
    }
    return s;
}

/**
 * Reordena os símbolos e as produções
 */
public void sort()
{
   for (int i=FIRST_NON_TERMINAL; i < FIRST_SEMANTIC_ACTION(); i++)
   {
          String s = symbols[i].substring(0, symbols[i].length()-1) + "_T>";
          int j=i+1;
          for ( ; j < FIRST_SEMANTIC_ACTION(); j++)
                if (symbols[j].equals( s ))
                      break;
          if (j < FIRST_SEMANTIC_ACTION()) //achou
          {
                int to = i+1,
                    from = j;

                if (to != from)
                {
                      moveSymbol(from, to);
                }
          }
   }
   moveSymbol(startSymbol, FIRST_NON_TERMINAL);

   Collections.sort(productions);
}

   private void moveSymbol(int from, int to)
   {
          String s = symbols[from];
          for (int k=from; k > to; k--)
                symbols[k] = symbols[k-1];
          symbols[to] = s;

          if (startSymbol == from)
                startSymbol = to;
          else if (startSymbol >= to && startSymbol < from)
                startSymbol++;

          for (Iterator iter = productions.iterator(); iter.hasNext(); )
          {
                Production p = (Production) iter.next();

                if (p.get_lhs() == from)
                      p.set_lhs(to);
                else if (p.get_lhs() >= to && p.get_lhs() < from)
                      p.set_lhs(p.get_lhs() + 1);
                IntList rhs = p.get_rhs();
```

```
                for (int k=0; k < rhs.size(); k++)
                {
                        if (rhs.get(k) == from)
                                rhs.set(k, to);
                        else if (rhs.get(k) >= to && rhs.get(k) < from)
                                rhs.set(k, rhs.get(k) + 1);
                }
        }
    }


/**
 * Verifica as condições para esta gramática ser LL
 */
public boolean isLL()
{
    return
           isFactored() &&
           !hasLeftRecursion() &&
           passThirdCondition();
}

/**
 * Verifica se esta gramática possui recursão à esquerda
 */
public boolean hasLeftRecursion()
{
    ProductionList prods = transformToFindRecursion(productions);

           for (int i = 0; i < prods.size(); i++)
    {
        if (prods.getProd(i).get_lhs() == prods.getProd(i).firstSymbol())
        {
           return true;
        }

    }
    return false;
}

/**
 * Verifica se esta gramática está fatorada
 */
public boolean isFactored()
{
    for (int i=0; i< productions.size(); i++)
    {
        Production P1 = productions.getProd(i);
        for (int j=i+1; j< productions.size(); j++)
        {
            Production P2 = productions.getProd(j);

            if (P1.get_lhs() == P2.get_lhs())
            {
                BitSet first = first(P1.get_rhs());
                first.and(first(P2.get_rhs()));
                if (! first.isEmpty())
                    return false;
            }
        }
    }
    return true;
}

/**
 * Verifica a terceira condição LL
 */
public boolean passThirdCondition()
{
    BitSet derivesEpsilon = markEpsilon();
    for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++)
```

```
    {
        if (derivesEpsilon.get(i))
        {
            BitSet first = (BitSet)firstSet[i].clone();
            first.and(followSet[i]);
            if (! first.isEmpty())
                return false;
        }
    }
    return true;
}

/**
 * Calcula os estados produtivos
 * @return conjunto dos estados produtivos
 */
private BitSet getProductiveSymbols()
{
   BitSet SP = new BitSet();
    for (int i=FIRST_TERMINAL; i< FIRST_NON_TERMINAL; i++)
        SP.set(i);

    for (int i=FIRST_SEMANTIC_ACTION(); i<= LAST_SEMANTIC_ACTION(); i++)
        SP.set(i);

    SP.set(EPSILON);
    boolean change;

    do
    {
        change = false;
        BitSet Q = new BitSet();
        for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++)
        {
            if (! SP.get(i))
            {
                for (int j=0; j< productions.size(); j++)
                {
                    Production P = productions.getProd(j);
                    if (P.get_lhs() == i)
                    {
                        boolean pass = true;
                        for (int k=0; k<P.get_rhs().size(); k++)
                            pass = pass && SP.get(P.get_rhs().get(k));
                        if (pass)
                        {
                            Q.set(i);
                            change = true;
                        }
                    }
                }
            }
        }
        SP.or(Q);
    }
    while (change);
    return SP;
}

/**
 * Remove os símbolos inalcançáveis da gramática
 * @throws EmptyGrammarException se o símbolo inicial for removido
 */
protected void removeUnreachableSymbols() throws EmptyGrammarException
{
    BitSet SA = getReachableSymbols();

    updateSymbols(SA);
}

/**
```

```
 * Calcula os símbolos que são alcansáveis
 *
 * @return BitSet indicando os symbolos alcansáveis
 */
private BitSet getReachableSymbols()
{
    BitSet SA = new BitSet();
     SA.set(startSymbol);
     boolean change;
     do
     {
         change = false;
         BitSet M = new BitSet();
         for (int i=0; i<symbols.length; i++)
         {
             if (! SA.get(i))
             {
                 for (int j=0; j< productions.size(); j++)
                 {
                     Production P = productions.getProd(j);
                     if (SA.get(P.get_lhs()))
                     {
                         for (int k=0; k<P.get_rhs().size(); k++)
                         {
                             if (P.get_rhs().get(k) == i)
                             {
                                 M.set(i);
                                 change = true;
                                 break;
                             }
                         }
                     }
                 }
             }
         }
         SA.or(M);
     }
     while (change);
     return SA;
}

    public String uselessSymbolsHTML()
    {
            Grammar clone = (Grammar) clone();

            try
            {
                    clone.removeUselessSymbols();
            }
            catch (EmptyGrammarException e)
            {
            }

            String[] cs = clone.symbols;

            BitSet s = new BitSet();


            for (int i=2; i<symbols.length; i++)
            {
                    for (int j=0; j<cs.length; j++)
                    {
                            if (cs[j].equals(symbols[i]))
                            {
                                    s.set(i);
                                    break;
                            }
                    }
            }

            StringBuffer result = new StringBuffer();
```

```
                result.append(
                                        "<HTML>"+
                                        "<HEAD>"+
                                        "<TITLE>Símbolos inúteis</TITLE>"+
                                        "</HEAD>"+
                                        "<BODY><FONT face=\"Verdana, Arial, Helvetica, sans-
serif\">");

                int count = 0;
                for (int i=2; i<symbols.length; i++)
                {
                        if (!s.get(i))
                        {

        result.append(HTMLDialog.translateString(symbols[i])+"<br>");
                                count++;
                        }
                }
                if (count == 0)
                        result.append("Não há símbolos inúteis");

                result.append(
                                        "</TABLE>"+
                                        "</FONT></BODY>"+
                                        "</HTML>");

                return result.toString();
        }

    /**
     * Gera uma representação de um BitSet utilizando os símbolos da Gramática
     *
     * @param b BitSet a ser convertido
     *
     * @return representação do BitSet
     */
    public String setToStr(BitSet b)
    {
        StringBuffer bfr = new StringBuffer("{ ");
        for (int j = 0; j < b.size(); j++)
        {
            if (b.get(j))
                bfr.append("\"").append(symbols[j]).append("\" ");
        }
        bfr.append("}");
        return bfr.toString();
    }

    /**
     * Fatora a gramática
     */

    public void factorate() throws LeftRecursionException
    {
        if (hasLeftRecursion())
            throw new LeftRecursionException();

                boolean change = true;
                while (change)
                {
                        change = false;
                for (int i=FIRST_NON_TERMINAL; i<FIRST_SEMANTIC_ACTION(); i++)
                {
                change = change || factorate(i);
                }
                }
    }

    /**
     * Efetua a fatoração das produções que possuam <code>symb</code> como lado esquerdo
```

```
     *
     * @param symb lado esquerdo das produções a serem fatoradas
     * @return <code>true</code> se hove alguma mudança, <code>fals</code>e em caso
contrário
     */
    private boolean factorate(int symb)
    {
       boolean result = false;
        BitSet prods = productionsFor(symb);

        BitSet conflict = new BitSet();

        int confictSymbol = conflict(prods, conflict);

        if (! conflict.isEmpty())
        {
            result = true;

                      //transforma as producoes para revelar os conflito indiretos
            for (int i=0; i< productions.size(); i++)
            {
                Production p = productions.getProd(i);
                if (p.get_lhs() == symb && first(p.get_rhs()).get(confictSymbol) &&
p.firstSymbol() != confictSymbol)
                {
                    ProductionList np = leftMostDerive(p);
                    productions.remove(i);
                    productions.addAll(np);
                    i--;
                    fillFirstSet();
                       fillFollowSet();
                }
            }

            conflict = new BitSet();
            for (int i=0; i< productions.size(); i++)
            {
                Production p = productions.getProd(i);
                if (p.get_lhs() == symb && p.firstSymbol() == confictSymbol)
                {
                    conflict.set(i);
                }
            }

            int newIndex = createSymbol(addTail(symbols[symb]));

            IntList prefix = extractPrefix(conflict);

            for (BitSetIterator it = new BitSetIterator(conflict); it.hasNext(); )
            {
                Production p = productions.getProd(it.nextInt());
                p.set_lhs(newIndex);
                if (p.get_rhs().size() > prefix.size())
                    p.get_rhs().removeRange(0, prefix.size());
                else // p.rhs.length == prefix.length
                    p.get_rhs().clear();
            }
            IntList rhs = new IntList();
            rhs.addAll(prefix);

            rhs.add(newIndex);
            productions.add(createProduction(symb, rhs));

            fillFirstSet();
            fillFollowSet();
            sort();
        }
        return result;
    }

    /**
```

```
     * Executa uma derivação mais a esquerda na produção passada como parametro
     *
     * @param p produção a sofrer a derivação
     */
    public ProductionList leftMostDerive(Production p)
    {
        if (isTerminal(p.firstSymbol()))
            return new ProductionList();
        else
        {
            ProductionList newProds = new ProductionList();
            int symb = p.firstSymbol();
            IntList actions = new IntList();
            for (int i=0; i<p.get_rhs().size() && isSemanticAction(p.get_rhs().get(i));
i++)
               actions.add(p.get_rhs().get(i));

            for (BitSetIterator it = new BitSetIterator(productionsFor(symb));
it.hasNext(); )
            {
                Production p1 = productions.getProd(it.nextInt());
                IntList rhs = new IntList();
                for (int i=0; i<actions.size(); i++)
                      rhs.add(actions.get(i));
                for (int i=0; i<p1.get_rhs().size(); i++)
                      rhs.add(p1.get_rhs().get(i));
                for (int i=actions.size()+1; i<p.get_rhs().size(); i++)
                      rhs.add(p.get_rhs().get(i));

                Production n = createProduction(p.get_lhs(), rhs);
                if (n != null && !newProds.contains(n))
                      newProds.add(n);
            }
            return newProds;
        }
    }

        /**
         * Calcula o prefixo comum de um conjunto de produções.
         *
         * @param prods conjunto de produções com prefixo comum.
         *
         * @return prefixo comum entre as produções.
         *
         */

    private IntList extractPrefix(BitSet prods)
    {
        IntList prefix = new IntList();
        boolean repeat;
        int index = 0;
        do
        {
            repeat = true;
            BitSetIterator it = new BitSetIterator(prods);
            Production pro = productions.getProd(it.nextInt());
            if (pro.get_rhs().size() > index)
            {
                int s = pro.get_rhs().get(index);
                for ( ; it.hasNext(); )
                {
                    Production p = productions.getProd(it.nextInt());
                    if (p.get_rhs().size() <= index || p.get_rhs().get(index) != s)
                        repeat = false;
                }
                if (repeat)
                {
                    prefix.add(pro.get_rhs().get(index));
                    index++;
                }
            }
```

```
        else
            repeat = false;
    }
    while (repeat);
    return prefix;
}

/**
 * Seleciona em um conjunto de produções, aquelas que possuem
 * o mesmo simbolo iniciando o lado direito.
 * Caso existam dois grupos de produções conflitantes, o grupo maior
 * é selecionado
 *
 * @param prods produçoes a seram pesquizadas
 *
 * @return produções conflitantes
 */
private int conflict(BitSet prods, BitSet result)
{
   int[] symbs = new int[symbols.length];
    //BitSet epsilon = markEpsilon();

    for (int i = 0; i < symbs.length; i++)
    {
        symbs[i] = 0;
    }

    for (BitSetIterator it = new BitSetIterator(prods); it.hasNext(); )
    {
        Production p = productions.getProd(it.nextInt());
        for (BitSetIterator i=new BitSetIterator(first(p.get_rhs())); i.hasNext(); )
            symbs[i.nextInt()]++;
    }

    symbs[EPSILON] = 0;
    symbs[DOLLAR] = 0;

    int max = 0;
    int indexMax = 0;
    for (int i = 0; i < symbs.length; i++)
    {
        if (symbs[i] > max)
        {
            max = symbs[i];
            indexMax = i;
        }
    }

   // BitSet result = new BitSet();
    if (max > 1)
    {
        for (BitSetIterator it = new BitSetIterator(prods); it.hasNext(); )
        {
            int pos = it.nextInt();
            if (first(productions.getProd(pos).get_rhs()).get(indexMax))
                result.set(pos);
        }
    }

    return indexMax;
}

/**
 * @return a representação de gramática em String
 */
public String toString()
{
    StringBuffer bfr = new StringBuffer();
    String lhs = "";
    boolean first = true;
    for (int i = 0; i < productions.size(); i++)
```

```
    {
        Production P = productions.getProd(i);
        if (! symbols[P.get_lhs()].equals(lhs))
        {
           if (! first)
           {
                  bfr.append(";\n\n");
           }
           first = false;
           lhs = symbols[P.get_lhs()];
           bfr.append(lhs).append(" ::=");
        }
        else
        {
           bfr.append("\n");
           for (int j=0; j<lhs.length(); j++)
                  bfr.append(" ");
                         bfr.append("   |");
        }
        if (P.get_rhs().size() == 0)
        {
                         bfr.append(" "+EPSILON_STR);
        }
        else
        {
               for (int j = 0; j < P.get_rhs().size(); j++)
               {
                   bfr.append(" ");
                   if (isSemanticAction(P.get_rhs().get(j)))
                   {
                          int action = P.get_rhs().get(j) - FIRST_SEMANTIC_ACTION();
                          bfr.append("#"+action);
                   }
                   else
                   {
                          String s = symbols[P.get_rhs().get(j)];
                          bfr.append(s);
                   }
               }
        }
    }
    bfr.append(";\n");
    return bfr.toString();
}

/**
 * Cria uma cópia da Gramática
 */
public Object clone()
{
    String[] T = new String[FIRST_NON_TERMINAL-2];
    String[] N = new String[FIRST_SEMANTIC_ACTION() - FIRST_NON_TERMINAL];
    for (int i = 0; i < T.length; i++)
         T[i] = new String(symbols[i+2]);
    for (int i = 0; i < N.length; i++)
         N[i] = new String(symbols[i+FIRST_NON_TERMINAL]);
    ProductionList P = new ProductionList();
    for (int i = 0; i < productions.size(); i++)
    {
        int[] rhs = new int[productions.getProd(i).get_rhs().size()];
        for (int j=0; j<rhs.length; j++)
            rhs[j] = productions.getProd(i).get_rhs().get(j);
        P.add( new Production(null, productions.getProd(i).get_lhs(),rhs));
    }
    return new Grammar(T, N, P, startSymbol);
}

private void removeSymbol(int s)
{
   String[] newSymbols = new String[symbols.length-1];
   System.arraycopy(symbols, 0, newSymbols, 0, s);
```

```
        System.arraycopy(symbols, s+1, newSymbols, s, symbols.length - s - 1);
        symbols = newSymbols;

        if (startSymbol > s)
                startSymbol--;
        if (FIRST_NON_TERMINAL > s)
                FIRST_NON_TERMINAL--;
        for (Iterator i = productions.iterator(); i.hasNext();)
                {
                        Production p = (Production) i.next();

                        if (p.get_lhs() == s)
                        {
                                i.remove();
                                continue;
                        }
                        else if (p.get_lhs() > s)
                                p.set_lhs(p.get_lhs()-1);

                        for (int j=0; j<p.get_rhs().size(); j++)
                        {
                                if (p.get_rhs().get(j) == s)
                                {
                                        i.remove();
                                        break;
                                }
                                if (p.get_rhs().get(j) > s)
                                        p.get_rhs().set(j, p.get_rhs().get(j) - 1);
                        }
                }
    }

    /**
     * Remove todos os symbolos, exceto os que devem ser mantidos;
     * @paramam keep conjunto dos símbolos a serem mantidos
     * @throws EmptyGrammarException se o símbolo inicial for removido
     */
    private void updateSymbols(BitSet keep) throws EmptyGrammarException
    {
        keep.set(EPSILON);
        keep.set(DOLLAR);

                /*
        if (checkEmpty && ! keep.get(startSymbol))
            throw new EmptyGrammarException();
        */
        int removed = 0;
        for (int i=0; i<symbols.length; i++)
                if (! keep.get(i) )
                {
                        removeSymbol(i - removed);
                        removed++;
                }

        fillFirstSet();
        fillFollowSet();
    }
}

package gesser.gals.scannerparser;

import gesser.gals.analyser.SemanticError;
import gesser.gals.generator.OptionsDialog;
import gesser.gals.generator.scanner.*;
import gesser.gals.util.BitSetIterator;

import java.util.*;
import java.util.BitSet;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
import java.util.Map.Entry;

/**
 * @author Gesser
 */

public class FiniteAutomataGenerator implements Constants
{
	private Map definitions = new HashMap();
	private Map expressions = new HashMap();
	private Map specialCases = new HashMap();
	private Node root = null;
	private BitSet alphabet = new BitSet();
	private int lastPosition = -1;
	private List tokenList = new ArrayList();
	private boolean sensitive = true;

	private int contextCount = 0;

	public FiniteAutomataGenerator()
	{
		sensitive = OptionsDialog.getInstance().getOptions().scannerSensitive;
	}

	private BitSet[] next;
	private Node[] nodes;

	public void addDefinition(String id, Node root) throws SemanticError
	{
		if (definitions.containsKey(id))
			throw new SemanticError("Definição repetida: "+id);

		definitions.put(id, root);

		alphabet.or(root.getAlphabet());
	}

	public Node getDefinition(String id)
	{
		return (Node) definitions.get(id);
	}

	public void addExpression(String id, Node root, boolean backtrack) throws
SemanticError
	{
		/*
		if (tokenList.contains(id))
			throw new SemanticError("Token '"+id+"' já definido");
		*/
		alphabet.or(root.getAlphabet());

		if (!tokenList.contains(id))
			tokenList.add(id);

		int pos = tokenList.indexOf(id);

		Node end = Node.createEndNode(pos+2, backtrack);
		root = Node.createConcatNode(root, end);

		Node ctx = root.getLeft().getRight();
		if (ctx != null)
		{
			ctx = ctx.deepestLeft();
			if (ctx != null && ctx.getContext() >= 0)
			{
				contextCount++;
				ctx.setContext(contextCount);
				end.setContext(contextCount);
			}
		}
```

```
                expressions.put(id, root);


                if (this.root == null)
                        this.root = root;
                else
                {
                        this.root = Node.createUnionNode(this.root, root);
                }
        }

        public void addIgnore(Node root, boolean backtrack)
        {
                alphabet.or(root.getAlphabet());

                Node end = Node.createEndNode(0, backtrack);
                root = Node.createConcatNode(root, end);

                if (this.root == null)
                        this.root = root;
                else
                {
                        this.root = Node.createUnionNode(this.root, root);
                }
        }

        public void addSpecialCase(String id, String base, String value) throws
SemanticError
        {
                if (! sensitive)
                        value = value.toUpperCase();

                if (!expressions.containsKey(base))
                        throw new SemanticError("Token '"+base+"' não definido");

                int b = tokenList.indexOf(base)+2;

                if (tokenList.contains(id))
                        throw new SemanticError("Token '"+id+"' já definido");

                Integer i = new Integer(tokenList.size()+2);

                Map s = (Map) specialCases.get(new Integer(b));

                if (s == null)
                {
                        s = new TreeMap();
                        specialCases.put(new Integer(b), s);
                }
                else if (s.get(value) != null)
                        throw new SemanticError("Já houve a definição de um caso especial
de '"+base+"' com o valor\""+value+"\"");

                s.put(value, i);

                tokenList.add(id);
        }

        public FiniteAutomata generateAutomata() throws SemanticError
        {
                List states = new ArrayList();
                Map context = new TreeMap();
                Map ctxMap = new TreeMap();
                Map trans = new TreeMap();
                Map finals = new TreeMap();
                Map back = new TreeMap();

                if (root == null)
                        throw new SemanticError("A Especificação Léxica deve conter a
definição de pelo menos um Token");
```

```
		computeNext();

		states.add(root.metaData.first);
		for (int i=0; i< states.size(); i++)
		{
			BitSet T = (BitSet) states.get(i);
			for (BitSetIterator it = new BitSetIterator(alphabet);
it.hasNext(); )
			{
				char c = (char)it.nextInt();

				BitSet U = new BitSet();

				for (BitSetIterator it2 = new BitSetIterator(T);
it2.hasNext(); )
				{
					int p = it2.nextInt();
					Node n = nodes[p];
					if (n.getEnd() >= 0)
					{
						Integer in = new Integer(i);
						if (!finals.containsKey(in))
						{
							finals.put(in, new
Integer(n.getEnd()));
							back.put(in, new
Boolean(n.doBackTrack()));

							if (n.getContext() > 0)
							{
								if (!
context.containsKey(in))
									context.put(in,
ctxMap.get(new Integer(n.getContext())));
							}
						}
					}
					if (n.getContext() >= 0)
					{
						if (! ctxMap.containsKey(new
Integer(n.getContext()))
							ctxMap.put(new
Integer(n.getContext()), new Integer(i));
					}

					if (n.getAlphabet().get(c))
						U.or(next[p]);

				}

				int pos = -1;
				if (! U.isEmpty() )
				{
					pos = states.indexOf(U);
					if (pos == -1)
					{
						states.add(U);
						pos = states.size()-1;
					}
				}
				Integer I = new Integer(i);
				if (! trans.containsKey(I))
					trans.put(I, new TreeMap());
				if (pos != -1)
					((Map)trans.get ( I )).put(new Character(c), new
Integer(pos));
			}
		}

		return makeAtomata(states, trans, finals, back, context);
```

```
	}

	public FiniteAutomata makeAtomata(List states, Map trans, Map finals, Map back,
Map context)
		throws SemanticError
	{
		Map[] transitions = new Map[states.size()];
		int count = 0;
		for (Iterator it = trans.values().iterator(); it.hasNext(); )
		{
			transitions[count] = (Map) it.next();

			count++;
		}

		int[] fin = new int[states.size()];
		for (int i=0; i<fin.length; i++)
		{
			Integer expr = (Integer) finals.get(new Integer(i));
			if (expr != null)
			{
				fin[i] = expr.intValue();
				Boolean b = (Boolean) back.get(new Integer(i));
				if (b != null && b.booleanValue() == false)
				{
					for (int k = i-1; k>0; k--)
					{
						if (fin[k] >= 0)
							break;
						Integer state = new Integer(k+1);
						for (Iterator iter =
transitions[k].entrySet().iterator(); iter.hasNext(); )
						{
							Integer next = (Integer) ((Map.Entry)
iter.next()).getValue();
							if (next.equals(state))
							{
								fin[k] = -2;
								break;
							}
						}
					}
				}
			}
			else
				fin[i] = -1;

		}
		List scList = new ArrayList();
		int[][] scIndexes = new int[tokenList.size()+2][];
		for (int i=0; i<scIndexes.length; i++)
		{
			Map m = (Map) specialCases.get(new Integer(i));
			int start = scList.size();
			if (m != null)
			{
				for (Iterator it = m.keySet().iterator(); it.hasNext(); )
				{
					String k = (String) it.next();
					Integer v = (Integer) m.get(k);

					scList.add(new FiniteAutomata.KeyValuePar(k,
v.intValue()));
				}
			}
			int end = scList.size();
			scIndexes[i] = new int[]{start, end};
		}
		FiniteAutomata.KeyValuePar[] sc = new
FiniteAutomata.KeyValuePar[scList.size()];
		System.arraycopy(scList.toArray(), 0, sc, 0, sc.length);
```

```
                int[][] cont = new int[states.size()][2];
                for (int i=0; i<cont.length; i++)
                {
                        cont[i][0] = 0;
                        cont[i][1] = -1;
                }
                for (Iterator i=context.entrySet().iterator(); i.hasNext(); )
                {
                        Map.Entry entry = (Entry) i.next();
                        Integer key = (Integer) entry.getKey();
                        Integer value = (Integer) entry.getValue();

                        cont[value.intValue()][0] = 1;
                        cont[key.intValue()][1] = value.intValue();

                }

                return  new FiniteAutomata(alphabet, transitions, fin, scIndexes, sc,
cont, tokenList);
        }

        public void computeNext()
        {
                computeMetaData(root);

                next = new BitSet[lastPosition+1];
                nodes = new Node[lastPosition+1];

                for (int i=0; i<next.length; i++)
                {
                        next[i] = new BitSet();
                }

                computeNext(root);
        }

        private void computeMetaData(Node root)
        {
                if (root.getLeft() != null)
                        computeMetaData(root.getLeft());

                if (root.getRight() != null)
                        computeMetaData(root.getRight());

                Node.MetaData n = root.metaData;
                Node l = root.getLeft();
                Node r = root.getRight();

                switch (root.getId())
                {
                        case CHAR:
                                lastPosition++;

                                n.position = lastPosition;
                                n.nullable = false;
                                n.first.set(lastPosition);
                                n.last.set(lastPosition);
                                break;

                        case OPTIONAL:
                        case CLOSURE:
                                n.nullable = true;
                                n.first.or(l.metaData.first);
                                n.last.or(l.metaData.last);
                                break;

                        case CLOSURE_OB:
                                n.nullable = false;
                                n.first.or(l.metaData.first);
                                n.last.or(l.metaData.last);
                                break;
```

```
                        case UNION:
                                n.nullable = l.metaData.nullable || r.metaData.nullable;

                                n.first.or(l.metaData.first);
                                n.first.or(r.metaData.first);

                                n.last.or(l.metaData.last);
                                n.last.or(r.metaData.last);
                                break;

                        case -1://concat
                                n.nullable = l.metaData.nullable && r.metaData.nullable;

                                n.first.or(l.metaData.first);
                                if (l.metaData.nullable)
                                        n.first.or(r.metaData.first);

                                n.last.or(r.metaData.last);
                                if (r.metaData.nullable)
                                        n.last.or(l.metaData.last);
                                break;
                }
        }

        private void computeNext(Node root)
        {
                switch (root.getId())
                {
                        case -1: //concat
                                for (BitSetIterator it = new
BitSetIterator(root.getLeft().metaData.last); it.hasNext(); )
                                {
                                        int i = it.nextInt();
                                        next[i].or(root.getRight().metaData.first);
                                }
                                break;
                        case CLOSURE:
                        case CLOSURE_OB:
                                for (BitSetIterator it = new
BitSetIterator(root.getLeft().metaData.last); it.hasNext(); )
                                {
                                        int i = it.nextInt();
                                        next[i].or(root.getLeft().metaData.first);
                                }
                                break;
                        case CHAR:
                                nodes[root.metaData.position] = root;
                                break;
                }

                if (root.getLeft() != null)
                        computeNext(root.getLeft());

                if (root.getRight() != null)
                        computeNext(root.getRight());
        }


        public String toString()
        {
                StringBuffer result = new StringBuffer();
                result.append(root);

                return result.toString();
        }
}
```