UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
DEPARTAMENTO DE CIÊNCIAS EXATAS E TECNOLÓGICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

JÂNITOR MÁRCIO SILVA PRATES

GTSLR: FERRAMENTA PARA GERAÇÃO DE TABELA SLR

Vitória da Conquista – BA
2013

JÂNITOR MÁRCIO SILVA PRATES

GTSLR: FERRAMENTA PARA GERAÇÃO DE TABELA SLR

Monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, da Universidade Estadual do Sudoeste da Bahia.

Orientador: Prof. Me. Stênio Longo Araújo
Coorientador: Prof. Me. José Carlos Martins Oliveira

Vitória da Conquista - BA
2013

JÂNITOR MÁRCIO SILVA PRATES

GTSLR: FERRAMENTA PARA GERAÇÃO DE TABELA SLR

Monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação, da Universidade Estadual do Sudoeste da Bahia.

BANCA EXAMINADORA

_____

Prof. Me. Stênio Longo Araújo (Orientador) – UESB

_____

Prof. Me. José Carlos Martins Oliveira (Coorientador) – UESB

_____

Prof. Dr. Roque Mendes Prado Trindade – UESB

À minha família pelo apoio e carinho incondicional e irrestrito na minha trajetória.

AGRADECIMENTOS

A elaboração de um trabalho de conclusão de curso de graduação é um momento muito importante, principalmente tratando-se da primeira formação superior na vida acadêmica do estudante, em vista disso gostaria de agradecer a todos que contribuíram para construir este momento.

Ao Prof. José Carlos e ao Prof. Stênio Longo por acreditarem na proposta deste trabalho e pela competente orientação.

À minha namorada Palloma pelo apoio, incentivo e ajuda numa parte importante deste projeto.

Aos meus amigos Caio, Matias, Rafael e, principalmente, Iran por compartilhar seus conhecimentos e experiência na área de programação.

Aos professores que um dia dividiram o seu conhecimento comigo e todos que de alguma forma contribuíram pra conclusão deste trabalho.

Ao povo brasileiro por ter financiado minha graduação.

“A tarefa não é tanto ver aquilo que ninguém viu, mas pensar o que ninguém ainda pensou sobre aquilo que todo mundo vê.” (Arthur Schopenhauer)

RESUMO

O principal objetivo deste trabalho foi desenvolver um aplicativo em Java para construção de tabelas SLR e, consequentemente, facilitar a compreensão dos estudantes de uma parte importante da criação de um compilador que é a análise sintática. Para o desenvolvimento deste trabalho foi realizada uma revisão bibliográfica e uma pesquisa por ferramentas relacionadas. Como resultado dessa busca, foram encontradas algumas ferramentas que vieram a servir de referência. As ferramentas pesquisadas não apresentam a tabela SLR estendida para recuperação de erros da análise LR e este é o diferencial do aplicativo desenvolvido.

Palavras-chave**:** Compilador. Análise Sintática. Análise Sintática Ascendente.

ABSTRACT

The main objective of this work was to develop an application in Java for building tables SLR and hence facilitate students' understanding of an important part of creating a compiler that is parsing. For the development of this work was carried out an extensive literature review and a search for related tools. As a result of this search, we found some tools that have come to serve as a reference. The tools surveyed do not have a table SLR extended error recovery for LR analysis and this is the differential of the application developed.

Keywords: Compiler. Syntactic analysis. Syntactic Analysis Ascending.

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

Segundo Price et al. (2008), na programação de computadores, uma linguagem de programação serve como meio de comunicação entre o ser humano que deseja resolver um determinado problema e o computador escolhido para ajudá-lo na solução. Para fazer essa intermediação necessita-se de um *Tradutor* que é, no contexto da programação, um sistema que recebe como entrada um programa escrito numa dada linguagem (linguagem fonte) e produz um programa equivalente em outra linguagem (linguagem alvo). Dentre a classificação dos tradutores de linguagem de programação existe o compilador. Definido por Aho et al. (1977) como:

> Um compilador é um programa que lê um programa escrito em uma linguagem – a linguagem de origem – e o traduz em um programa equivalente em outra linguagem – a linguagem destino. Como uma importante parte no processo de tradução, o compilador reporta ao seu usuário a presença de erros no programa origem. (AHO et al. 1977)

O processo de tradução normalmente é estruturado em fases e cada uma das fases tem um objetivo principal:

- **Análise Léxica:** faz a leitura do programa fonte, caractere a caractere, e o traduz para uma sequência de símbolos léxicos, também chamados *tokens* (PRICE *et al.*, 2008);
- **Análise Sintática** é a parte essencial do compilador. Nesta fase que há a construção da tabela SLR, responsável por verificar se os símbolos contidos no programa fonte formam um programa válido, ou não (DELAMARO, 2004);
- **Análise Semântica** tem a função de prover métodos pelos quais verifica as estruturas construídas pelo analisador sintático testando se irão fazer sentido durante a execução ou não (WILHELM *et al.*, 1995);
- **Geração de Código Intermediário** ocorre a transformação da árvore sintática em uma linguagem intermediária mais próxima da linguagem objeto do que o código fonte (PITTMAN *et al., 1992*);
- **Geração de Código Objeto** tem como objetivos: reserva de memória para constantes e variáveis, seleção de registradores entre outros;

- **Gerência de tabelas:** este módulo compreende um conjunto de tabelas e rotinas associadas que são utilizadas por quase todas as fases do compilador;
- **Tratamento de Erros** tem por objetivo tratar os erros que são detectados em todas as fases de análise do programa fonte.

A disciplina compiladores é de extrema importância no contexto acadêmico, pois congrega conhecimentos adquiridos das disciplinas do curso, como: algoritmos, linguagens programação, estruturas de dados, sistemas operacionais, arquitetura de computadores, teoria da computação, entre outras. Por esses motivos, e pela complexidade dos algoritmos envolvidos, o aprendizado de compiladores representa um grande desafio junto aos alunos. Esta monografia disserta sobre a implementação de uma ferramenta que automatiza a construção da tabela SLR através das funções *closure, goto, first* e *follow* auxiliando os graduandos no entendimento de uma etapa importante do processo de compilação que é a análise sintática.

1.1. Justificativa

Para escrever um compilador é indispensável o conhecimento teórico e prático de temas, tais como: projeto de linguagens de programação, analisadores léxicos, analisadores sintáticos, interpretadores de código fonte, geradores de código, entre outros. Diante da complexidade do processo de construção de compiladores, o uso de ferramentas auxiliares é imprescindível.

Visto que o processo manual de construção da tabela SLR é trabalhoso e cansativo, o desenvolvimento de uma ferramenta que proporcione ao estudante um meio ágil para elaboração da referida tabela, se torna extremamente interessante.

Esta ferramenta diminui as dificuldades de se arquitetar uma tabela SLR, além de minimizar os erros causados pela falha humana em processos com grandes quantidades de cálculos.

1.2. Objetivos

1.2.1 Geral

O objetivo geral deste trabalho foi desenvolver uma ferramenta para gerar tabelas SLR a fim de auxiliar à aprendizagem de compiladores.

1.2.2 Específicos

- Agilizar o processo de criação de compilador no meio acadêmico;
- Produzir um *software* que ofereça facilidade e versatilidade para o usuário;
- Implementar os algoritmos para obtenção da tabela SLR.

1.4 Metodologia

Antes de iniciar o desenvolvimento de uma ferramenta para automatização da construção da tabela SLR, apurou-se a necessidade de se realizar uma revisão bibliográfica sobre os algoritmos utilizados na construção da tabela SLR. O resultado dessa revisão é exposto no capítulo seguinte.

Concorrentemente a isso, buscas por trabalhos já realizados na área foram feitas, usando-se como principal fonte de procura a Internet, já que a rede mundial de computadores representa um canal rápido e de fácil acesso a inúmeras universidades e desenvolvedores de todo o mundo (MELO, 2003). Como resultado dessa pesquisa, foram encontradas algumas ferramentas, vide capitulo 3, que vieram a servir de referência. Após a análise das ferramentas pesquisadas notou-se que nenhuma delas apresentava a tabela SLR estendida para recuperação de erros da análise LR e a partir deste fato que surgiu a ideia deste projeto.

1.5 Estrutura do trabalho

Além deste capítulo, o trabalho consta de mais 4 capítulos. O capítulo 2 trata do referencial teórico, abrangendo os principais conceitos de Análise Sintática. O capítulo 3 descreve sobre as principais ferramentas relacionadas ao trabalho desenvolvido. O capítulo 4 apresenta e descreve a implementação da ferramenta GTSLR. Por fim, o capítulo 5 mostra as considerações finais e algumas sugestões para trabalhos futuros.

2 ANÁLISE SINTÁTICA

A análise sintática tem por função verificar se a estrutura gramatical do programa desenvolvido está certa ou seja se a estrutura foi formada usando as regras gramaticais da linguagem (PRICE et al. 2008).

Analisador sintático é o responsável por verificar se as construções utilizados no programa estão gramaticalmente corretas. Conforme Louden (2004), a principal função de um analisador sintático é determinar a estrutura ou a sintaxe de um programa, descrita através de uma linguagem de programação. Esta estrutura é geralmente especificada por regras gramaticais definidas em uma Gramática Livre de Contexto.

Os métodos de análise sintática mais comuns usados nos compiladores são classificados como *top-down* ou *bottom-up*. Os analisadores sintáticos *top-down* constroem árvores gramaticais da raiz para as folhas, enquanto que os *bottom-up* começam pelas folhas e trabalham a árvore até a raiz (AHO et al. 1977). Nos dois casos, a cadeia de entrada é percorrida da esquerda para a direita, um símbolo de cada vez.

Neste trabalho, são considerados dois algoritmos *bottom-up* da família LR: LR(0) e SLR(1). Os algoritmos pertencentes a essa família são formados por uma entrada, uma saída, uma pilha, um programa diretor e uma tabela sintática. Todos os tipos de analisadores sintáticos LR possuem o mesmo programa diretor, sendo que a diferença entre eles está na construção da tabela sintática (AHO et al. 1986).

A estrutura genérica de um analisador LR é mostrada na Figura 1. A cadeia de entrada $(x_1, x_2, \ldots x_n \$)$ mostra a sequência de símbolos a ser analisada, e a pilha armazena símbolos da gramática (T_j) intercalados com estados (E_j) do analisador. O símbolo da base da pilha é E_0 , estado inicial do analisador. O qual é dirigido pela Tabela de Análise, cuja estrutura é mostrada na Figura 2.

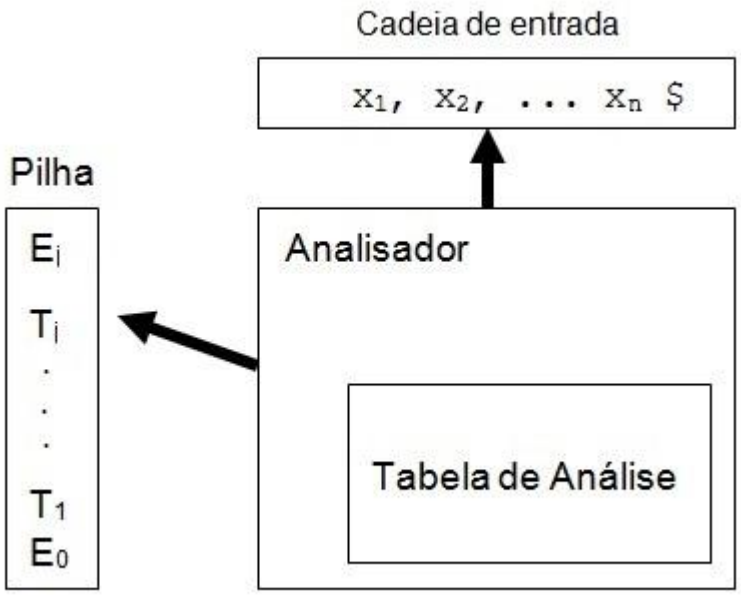


Figura 1 – Estrutura dos analisadores LR

O núcleo deste capítulo está voltado a apresentação dos algoritmos necessários para construção da tabela de ação e transição/desvio, do tipo SLR, que é uma representação eficiente do autômato de pilha que reconhece a linguagem.

Figura 2 – Estrutura da Tabela de Análise

Segundo Price et al. (2008), a tabela de análise é formada por duas partes: a parte AÇÃO, vinculados aos terminais da cadeia de entrada; e a parte TRANSIÇÃO que contém transições de estado com relação aos símbolos não terminais.

De acordo com Louden (2004), as duas ações possíveis em um analisador sintático ascendente (*bottom-up*), que ocorrem à medida que os elementos são lidos da cadeia de entrada, são as ações de empilhar e reduzir, detalhadas na sequência:

• Empilhar: esta ação ocorre a partir de uma transição gerada por um item não completo. O número do novo estado é empilhado juntamente com o elemento lido. Um erro ocorre se houver um elemento não esperado na cadeia de entrada;

• Reduzir: esta ação ocorre a partir de um item completo presente no estado. Todos os elementos presentes na parte direita da regra de produção são retirados da pilha e é empilhado o não terminal do item, juntamente com o novo estado.

As outras duas opções da tabela AÇÃO são “aceita” que o analisador reconhece a sentença de entrada como válida e a condição de “erro” que a execução do analisador é interrompida identificando um erro sintático.

2.1 Analisadores LR

Para o desenvolvimento de um compilador é necessário fazer análise léxica e a análise sintática de uma linguagem de programação. A análise sintática pode ser feita através de analisadores LR (*Left to right with Rightmost derivation*) que são redutores eficientes que leem a sentença em análise da esquerda para direita e produzem uma derivação mais à direita ao reverso. De acordo com Price et al. (2008), existem, basicamente, três tipos de analisadores LR:

> a) SLR (*Simple* LR), fáceis de implementar, porém aplicáveis a uma classe restrita de gramáticas;
> b) LR Canônicos, mais poderosos, podendo ser aplicados a um grande número de linguagens livres do contexto;
> c) LALR (*Look Ahead* LR) de nível intermediário e implementação eficiente, que funciona para a maioria das linguagens de programação. O programa YACC gera esse tipo de analisador. (PRINCE et al., 2008, p. 66 e 67)

O trabalho aqui proposto concentrou-se no analisador SLR que se baseia em uma tabela SLR. A construção da tabela de controle para analisadores SLR gera um Conjunto Canônico de Itens LR que juntamente com a função *follow* são a base para o algoritmo de construção da tabela SLR.

2.2 Gramáticas livres de contexto

Gramáticas Livres de Contexto (GLC) constituem o formalismo essencial para descrever a estrutura de programas em uma linguagem de programação (GRUNE et al. 2001).

De acordo com Greenlaw (1998), uma GLC é uma quádrupla, definida como $< N, T, P, S >$, onde N é um conjunto finito de símbolos auxiliares, chamados de não terminais, que denotam cadeias de caracteres; T é o conjunto finito sobre o qual a linguagem é definida, cujos elementos são os símbolos terminais a partir dos quais as cadeias são formadas, com $T \cap N = \{ \}$; P é o conjunto de regras ou de produções da forma $X \rightarrow z$, onde $X \in N$ e $z \in (N \cup T)^*$, que especifica de que maneira os símbolos não terminais e os terminais podem ser combinados para formar as cadeias; e S é um símbolo não terminal distinto, chamado símbolo inicial, a partir do qual toda derivação é iniciada, sendo que $S \in N$.

Segue abaixo exemplos simples de gramáticas que servirão de testes para as ferramentas citadas no capítulo 3 e para a ferramenta desenvolvido por este trabalho.

Exemplo 1:

$G_1 = < \{A\}, \{a,(,)\}, P_1, A >$ onde $P_1 = \{ A \rightarrow (A), A \rightarrow a \}$.

Exemplo 2:

$G_2 = < \{ E \},\{+, n\},P_2,E >$ onde $P_2 = \{ E \rightarrow E + n , E \rightarrow n \}$.

Nos exemplos acima a primeira gramática é LR(0) e a segunda é SLR(1).

2.3 Construção da tabela SLR

Um item LR, para uma gramática *G,* é uma produção com um ponto em alguma posição do lado direito (PRICE et al., 2008). O ponto é uma indicação de até onde uma produção já foi analisada no processo de reconhecimento. Por exemplo, a produção $A \rightarrow X Y Z$ origina quatro itens:

$A \rightarrow \bullet XYZ$

$A \rightarrow X \bullet YZ$

A → XY●Z

A → XYZ●

Um item pode ser escrito por um par de inteiros, onde o primeiro número representa a produção e o segundo a posição do ponto (PRICE et al., 2008). De modo intuitivo, um item indica quanto de uma produção já foi examinado a certa altura do processo de análise sintática. Por exemplo, o primeiro item acima indica que em seguida irá examinar uma cadeia na entrada, derivável a partir de XYZ. O segundo item indica que já foi examinado na entrada uma cadeia derivável a partir de X e que em seguida virá a cadeia derivável a partir de YZ.

A coleção de conjuntos de itens LR(0), ou seja, Conjunto Canônico de Itens LR(0), providencia a base para a construção de analisadores sintáticos SLR. A construção desse conjunto requer duas operações:

1) acrescentar à gramática a produção S' → S (onde S é o símbolo inicial da gramática) definindo assim uma gramática aumentada;

2) computar a função *closure* e *goto* para a nova gramática.

Sendo G uma gramática com símbolo de partida S, então G′ (a *gramática aumentada* para G) é G com um novo símbolo de partida, S′ mais a produção S'→S (PRICE et al. 2008). O objetivo desta nova produção de partida é o de indicar ao analisador sintático quando o mesmo deve parar de analisar e aceitar a entrada. A aceitação ocorre somente quando o analisador sintático estiver para reduzir através de S' → S.

2.3.1 Função closure

Se I é um conjunto de itens LR para uma gramática *G,* então o conjunto de itens *closure* é construído a partir de I por essas duas regras (PRICE et al., 2008):

1) Cada item em I é adicionado ao conjunto *closure(I)*;

2) Se A → a ●X β está no conjunto *closure(I)* e X → y é uma produção, então adicione X → y ao conjunto se já não estiver lá. Esta regra se aplica até que não possam ser adicionados novos itens ao conjunto *closure(I)*.

A → a ●X β em *closure(I)* indica que, em algum ponto do processo de análise gramatical, virá em seguida na entrada uma subcadeia derivável a partir de X β. Se X → y for uma produção, também virá uma subcadeia derivável de y àquele ponto. Por esta razão inclui-se X →● y no conjunto *closure(I).*

Utilizando a gramática de expressões aumentada abaixo como exemplo:

S′ → S
S → S + H | H
H → H * P | P
P → (S) | num

Se *I* for o conjunto {[S' → ●S]}, então o *closure(I)* tem os itens

S′ → ●S
S → ●S + H
S → ●H
H → ●H * P
H → ●P
P → ● (S)
P → ●num

Primeiro S′ → ●S é adicionado ao *closure(I)* pela regra (1). Depois como tem um S à direita do ponto, coloca-se, pela regra (2), as produções de S com pontos nas extremidades à esquerda, ou seja, S → ●S + H e S → ●H. Agora tem um H imediatamente à direita do ponto e consequentemente adiciona-se H → ●H * P e H→●P. Logo depois, o P à direita do ponto inclui P → ● (S) e de P → ●num.

A função *closure(I)* pode ser computada conforme o algoritmo abaixo (AHO et al. 1977, p. 101):

```
função closure(I);
início
 conjunto := I;
     repetir
         para cada item A → a●Xβ em conjunto e cada produção X → y de G
                 tal que X →●y não esteja em conjunto
         faça incluir X →●y a conjunto
     até que não possam ser adicionados mais itens a conjunto;
   retornar conjunto
 fim
```

Algoritmo 1 – função *closure*

2.3.2 Função goto(I,X)

Informalmente, goto(I,X), "avanço do ponto sobre X em I", consiste em coletar as produções com ponto no lado esquerdo de X, passar o ponto para a direita de X, e obter a função *closure* desse conjunto.

Formalmente, a função goto(I,X), onde *I* é um conjunto de itens e *X* um símbolo gramatical, goto(I,X) é definido como o *closure* do conjunto de todos os itens A→aX● β tais que A → a●X β esteja em *I*.

Por exemplo, sendo *I* o conjunto de dois itens {[S' → S●], [S → S● + H]}, então goto(I,+) será

S → S + ●H

H → ●H * P

H → ●P

P → ● (S)

P → ●num

Através do exame dos itens com + imediatamente à direita do ponto calcula-se goto(I,+). S' → S● não é um desses itens, porém S → S● + H é. Move-se o

ponto por sobre o + para obter { [S → S + ●H] } e em seguida efetua-se o calculo *closure* deste conjunto.

A função *goto(I,X)* pode ser computada conforme o algoritmo abaixo (AHO et al. 1977, p. 101):

```
função goto( I, X);
inicio
    seja J o conjunto de itens [ A → a X●β, a] tais que [A → a●Xβ, a]
    esteja em I;
  retornar closure(J)
fim;
```

Algoritmo 2 – função goto

2.3.3 Construção da coleção canônica de conjuntos de itens LR(0)

Para construir C, coleção canônica de conjuntos de itens LR(0), para uma gramática aumentada G' usa-se o seguinte algoritmo (AHO et al. 1977, p. 101):

```
procedimento itens ( G' );
 inicio
  coleçãoCanônica := { closure ( { [ S' → ●S] }) };
  repetir
   para cada conjunto de itens I em coleçãoCanônica e cada símbolo
        gramatical X tal que goto( I, X ) não seja vazio e não esteja em
        coleçãoCanônica
    faça incluir goto( I, X) a coleçãoCanônica
  até que não haja mais conjuntos de itens a serem incluídos a
  coleçãoCanônica
 fim
```

Algoritmo 3 – conjunto canônico de itens LR

Segue abaixo a coleção canônica de conjuntos de itens LR(0) para a gramática usada como exemplo por este trabalho:

I_0 = {S′ → ●S
S → ●S + H
S → ●H
H → ●H * P
H → ●P
P → ● (S)
P → ●num }

goto(I, S) = I_1 = { S′ → S●
S → S● + H }

goto(I, H) = I_2 = { S → H●
H → H● * P }

goto(I, P) = I_3 = { H → P● }

goto(I, "(") = I_4 = { P → (● S)
S → ●S + H
S → ●H
H → ●H * P
H → ●P
P → ● (S)
P → ●num }

goto(I, num) = I_5 = { P → num ● }

goto(I, +) = I_6 = {S → S + ●H
H → ●H * P
H → ●P
P → ● (S)
P → ●num }

goto(I, *) = I_7 = {H → H * ●P
P → ● (S)
P → ●num }

goto(I, S) = I_8 = {P → (S●)
S → S● + H }

goto(I, H) = I_9 = {S → S + H●
H → H ●* P }

goto(I, P) = I_{10} = {H → H * P● }

goto(I, ")") = I_{11} = { P → (S) ●}

2.3.4 Função *first*

Se α for uma forma sentencial ou seja uma cadeia de símbolos gramaticais qualquer, então *first*(α) é o conjunto dos símbolos terminais que começam as cadeias derivadas a partir de α. Se α → λ então λ (representa vazio) também pertence ao conjunto *first*(α).

Para se calcular o *first(*X*)* para todos os símbolos gramaticais X, aplique-se as seguintes regras até que nenhum terminal ou λ possa ser adicionado a qualquer conjunto *first*:

1. Se X for um terminal, então *first(X)* é { X };
2. Se X → λ for uma produção , adicionar λ a *first(X)*.
3. Se X for um não terminal e X → Y_1Y_2 ...Y_k uma produção, colocar ***a*** em *first(X)* se, para algum *i*, ***a*** estiver em *first*(Y_i) e λ estiver em todos *first*(Y_1),..., *first*(Y_{i-1}). Se λ estiver em *first*(Y_i) para todos os j = 1, 2,..,k, adiciona-se, então, λ a *first(X).* (AHO et al., 1977, p. 84)

Exemplo:

S → A C E
A → a
A → b
A → λ
C → c
C → d
C → λ
E → e

O conjunto *first* da gramática acima será:

First(E) = { e }; *First*(C) = {c, d, λ}; *First*(A) = {a, b, λ}; *First*(S) = {a, b, c, d, e}

2.3.5 Função *follow*

Para um não terminal X, calcula-se *follow*(X) como sendo o conjunto de terminais α que estiver imediatamente à direita de X em alguma forma sentencial, ou seja, o conjunto de terminais α tais que exista uma derivação de forma S→αXaβ, para algum α e β. Pode acontecer de ter existido, em algum tempo durante a derivação símbolos entre X e a porém, se isso acontecer e os mesmos derivarem λ (vazio) eles desaparecem. Se X puder ser o símbolo mais à direita em alguma forma sentencial, então $ (símbolo de fim de cadeia) está em *follow*(X).

Para calcular *follow*(X) para todos os não terminais X, aplique as regras abaixo até que nada mais possa ser adicionado a qualquer conjunto *follow*:

1. Colocar $ em *follow*(S), onde S é o símbolo inicial e $ o marcador de fim de entrada à direita;
2. Se existir uma produção X → αBβ, então tudo em *first*(β), exceto λ, é inserido no conjunto *follow*(B);
3. Se existir uma produção A→αB ou uma produção A→αBβ onde *first*(β) contém λ (ou seja, β→λ), então, tudo em *follow*(A) está em *follow(*B*).* (AHO et al. 1977, p. 84)

Exemplo:

S → A B

A → c

A → λ

B → c b B

C → c a

O conjunto *follow* da gramática acima será:

follow (B) = { *follow(*S*)* = $ }; *follow* (A) = { *first(*B*)* = c }; *follow* (S) = {$};

2.4 Construção da Tabela de Análise SLR

Dada uma gramática G, obtém-se G', aumentando G com a produção S' → S, onde S é o símbolo inicial de G. A partir de G', determina-se o conjunto canônico

C. Finalmente, constroem-se as tabelas AÇÃO e DESVIO, ou seja a tabela SLR (PRICE et al. 2008).

O algoritmo para construção da tabela SLR segue o método abaixo (PRICE et al. 2008, p. 72) :

Entrada: O conjunto C para G'.

Resultado: A tabela de análise SLR para G'.

Método: Seja $C = \{I_0, I_1, ..., I_n\}$. Os estados do analisador são 0, 1, ..., n sendo o 0 (zero) o estado inicial. 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 *goto*(Ii, a) = Ij, então faça AÇÃO[i, a] = empilha j;

2) se A → α● está em Ii, então para todo a em FOLLOW(A), faça AÇÃO[i, a] = reduz n, sendo n o número da produção A → α●.

3) se S' → S● está em Ii, então faça AÇÃO[i, $] = aceita.

Se ações conflitantes surgirem, como por exemplo num mesmo estado ter uma ação de reduz e uma de empilhar, seguindo as regras acima então a gramática não é SLR(1) (PRICE et al. 2008).

As ***transições*** para o estado i são construídas, usando a regra:

4) se *goto*(Ii, A) = Ij, então TRANSIÇÃO(i , A) = j;

Utilizando o conjunto canônico de itens calculado no tópico 2.3.3 a tabela SLR resultante é apresentada na Figura 3:

OPÇÕES	AÇÃO						TRANSIÇÃO		
Estado	+	*	(	)	num	$	S	H	P
0			E4		E5		1	2	3
1	E6					OK			
2	R2	E7		R2		R2			
3	R4	R4		R4		R4			
4			E4		E5		8	2	3
5	R6	R6		R6		R6			
6			E4		E5			9	3
7			E4		E5				10
8	E6			E11					
9	R1	E7		R1		R1			
10	R3	R3		R3		R3			
11	R5	R5		R5		R5			

Figura 3 - Tabela SLR

É importante comentar que a tabela SLR é uma representação eficiente do autômato de pilha que reconhece a linguagem. O topo da pilha sempre terá o estado atual do autômato. Com o par estado atual e símbolo de entrada, a tabela indica a ação a ser executada ou seja empilhar ou reduzir. As entradas em branco são situações de erro (PRICE et al. 2008).

2.5 Métodos de recuperação de erros

Quando um compilador detecta um erro de sintaxe, é desejável que ele tente continuar o processo de análise de modo a detectar outros erros que possam existir no código ainda não analisado. Isso envolve realizar a recuperação de erros. Existem algumas estratégias de reparação de erros, algumas deles são explicados adiante.

2.5.1 Recuperação na modalidade do desespero

Este é o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática. Ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de *tokens* de sincronização (AHO et al. 1977). Os *tokens* de sincronização são usualmente delimitadores, tais como o ponto-e-vírgula ou o **fim**, cujo papel no programa-fonte seja claro. Naturalmente, o projetista do compilador precisa selecionar os *tokens* de sincronização apropriados à linguagem fonte. A correção na modalidade do desespero, que frequentemente pula

uma parte considerável da entrada sem verificá-la, procurando por erros adicionais, possui a vantagem da simplicidade e tem a garantia de não entrar num laço infinito (PRICE et al., 2008).

2.5.2 Recuperação local

Ao descobrir um erro, o analisador sintático pode realizar uma correção local na entrada restante. Isto é, pode substituir um prefixo da entrada remanescente por alguma cadeia que permita ao analisador seguir em frente. Correções locais seriam substituir uma *token* por outro, remover um *token* estranho ou inserir um ausente (PRICE et al., 2008). A escolha da correção local é deixada para o projetista do compilador. Deve-se escolher substituições que não levem a laços infinitos, como seria o caso, por exemplo, se inserisse para sempre na cadeia de entrada algo à frente do seu símbolo corrente. Sua maior desvantagem está na dificuldade que tem ao lidar com situações nas quais o erro efetivo ocorreu antes do ponto de detecção.

2.6 Recuperação de erros na Análise LR

Na análise LR, os erros são identificados durante a leitura dos símbolos na cadeia de entrada do analisador. Na tabela SLR, as lacunas em branco representam situações de erro e deve acionar rotinas de recuperação (PRICE et al., 2008). A Figura 4 é a tabela de análise vista na Figura 3, porém preenchida para o tratamento de erros.

Segundo Price et al. (2008), nas linhas em que houverem apenas reduções, pode repetir a mesma ação de redução para as outras lacunas da mesma linha, pois, de qualquer forma, os erros serão detectados nos passos seguintes pois o símbolo errado permanece na cadeia de entrada ou seja a ação de reduzir irá se repetir até encontrar o símbolo que deveria está no lugar deste *token* errado.

Nas linhas em que existem apenas ações de empilhar, as lacunas em branco são preenchidas com chamadas a rotinas que identificam qual foi o erro e apresenta qual o símbolo que deveria está naquela posição da cadeia de entrada.

	AÇÃO						TRANSIÇÃO		
Estado	+	*	(	)	num	$	S	H	P
0	Erro3	Erro3	E4	Erro3	E5	Erro3	1	2	3
1	E6	Erro2	Erro2	Erro2	Erro2	OK			
2	R2	E7	Erro1	R2	Erro1	R2			
3	R4	R4	R4	R4	R4	R4			
4	Erro3	Erro3	E4	Erro3	E5	Erro3	8	2	3
5	R6	R6	R6	R6	R6	R6			
6	Erro3	Erro3	E4	Erro3	E5	Erro3		9	3
7	Erro3	Erro3	E4	Erro3	E5	Erro3			10
8	E6	Erro4	Erro4	E11	Erro4	Erro4			
9	R1	E7	Erro1	R1	Erro1	R1			
10	R3	R3	R3	R3	R3	R3			
11	R5	R5	R5	R5	R5	R5			

Figura 4 - Tabela SLR estendida para tratamento de erros

As rotinas de erro da tabela são as seguintes:

Erro1: “ * ” esperado.

Erro2: “ + ” esperado.

Erro3: “ (“ ou “ num “ esperado.

Erro4: “) “ ou “ + “ esperado.

3 TRABALHOS RELACIONADOS

O analisador sintático ascendente é formado basicamente pelo algoritmo de análise sintática e por um conjunto de tabelas. O algoritmo depende unicamente da variante do método de análise sintática utilizado. As tabelas dependem não só do método como também da gramática.

Sendo assim, uma vez escolhido o método de análise sintática, a implementação do algoritmo pode ser a mesma para todas as gramáticas que atendam às restrições impostas pelo método.

A construção manual dessas tabelas é trabalhosa e sujeita a erros. A construção dessas tabelas através de ferramentas dedicadas é possível assim como também torna possível a “construção automática” do analisador sintática. Este capítulo pretende oferecer uma introdução das principais ferramentas que têm auxiliado a construção de compiladores.

3.1 LEX/FLEX: Gerador de Analisador Léxico

A ferramenta Lex ou Flex (PAXSON, 1998), permite especificar um analisador léxico definindo expressões regulares para descrever padrões para os tokens.

Existem muitas versões diferentes de Lex. A versão mais popular é denominada Flex (Fast Lex). Ela é distribuída como parte do pacote de compilação Gnu, produzido pela Free Software Foundation, e está disponível gratuitamente em diversos endereços na internet.

Lex é um programa que recebe como entrada um arquivo de texto contendo expressões regulares, juntamente com as ações associadas a cada expressão. A notação de entrada para a ferramenta Lex é chamada de linguagem Lex, e a ferramenta em si é o compilador Lex.

O compilador Lex transforma os padrões de entrada em um diagrama de transição e gera código em um arquivo chamado lex.yy.c, que simula esse diagrama de transição.

A Figura 5 ilustra como criar um analisador léxico com o Lex.

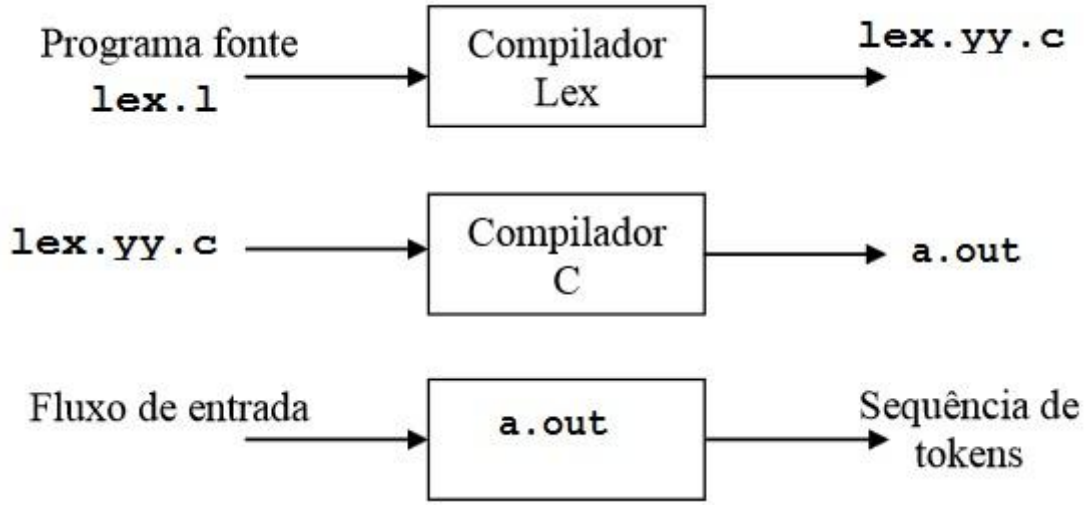


Figura 5 - Criando analisador léxico

Um arquivo de entrada, lex.l, é escrito na linguagem Lex e descreve o analisador léxico a ser gerado. O compilador Lex transforma lex.l em um programa C, e o armazena em um arquivo que sempre se chama lex.yy.c. Esse último arquivo é compilado pelo compilador C em um arquivo sempre chamado a.out. A saída do compilador C é o analisador léxico gerado, que pode receber como entrada um fluxo de caracteres e produzir como saída um fluxo de tokens.

Na tabela abaixo temos algumas das convenções de metacaracteres em Lex.

Padrão	Significado
`x`	o caractere `x`
“`x`”	caractere `x`, mesmo se `x` for um metacaractere
`\x`	caractere `x` se `x` for um metacaractere
`x*`	zero ou mais repetições de `x`
`x+`	uma ou mais repetições de `x`
`x?`	um `x` opcional
`a | b`	`a` ou `b`
`( x )`	`x` propriamente dito
`[abc]`	qualquer caractere entre `a, b` e `c`
`[a-d]`	qualquer caractere entre `a, b, c` e `d`
[^ab]	qualquer caractere, exceto `a` ou `b`

Há muitas outras convenções de metacaractes Lex.

Um arquivo de entrada Lex é composto por três partes: coleção de definições (ou declarações), coleção de regras de tradução e coleção de rotinas auxiliares (ou rotinas de usuário). As três seções são separadas por dois sinais de porcentagem que aparecem em linhas separadas iniciando na primeira coluna.

```
{definições}
%%
{regras}
%%
{rotinas auxiliares}
```

A seção de definições inclui declarações de variáveis, constantes e definições regulares. Cada uma das regras de tradução possui o formato:

```
padrão {ação}
```

Cada padrão é uma expressão regular, que pode usar as definições regulares da seção de declaração. As ações são fragmentos de código, normalmente escritos em C.

A terceira seção contém quaisquer funções adicionais usadas nas ações. Essa seção pode também conter um programa principal, se quisermos compilar a saída Lex como um programa independente.

3.2 YACC: Gerador de analisadores sintáticos

O YACC, iniciais de *Yet Another Compiler-Compiler* (BROWN et al., 1992), 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 embutia, 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.

3.3 JFLEX

JFlex (KLEIN, 2013) é um gerador de analisador léxico (ou gerador de scanner) em código Java, e é escrito em Java. É uma reescrita do gerador de analisador léxico JLex, que foi desenvolvido por Elliot Berk na Princeton University. Ambos são baseados no Lex, que foi feito originalmente por Mike Lesk e Eric Schmidt, porém com alguns adicionais como: geração mais eficiente; código gerado é mais rápido; foi feito de modo que o código gerado pudesse ser usado com o gerador de analisador sintático CUP, com suporte a uma lista de características adicionais que podem ser vistas em (JFLEXFEATURES, 2013).

O JFlex recebe como entrada um arquivo .flex, que segue o modelo descrito na tabela 1.

```
Código do Usuário
%%
Opções e declarações
%%
Regras léxicas
```

Tabela 1 – Corpo de um arquivo FLEX

De acordo com a tabela 1, o arquivo de entrada é dividido em três seções. Na seção ‘Código do Usuário’ contém código que será copiado diretamente no início do arquivo gerado pelo JFlex, usualmente importações, e pacotes. Na segunda seção, se encontra a definição de opções para customização do código gerado, como por exemplo, a opção `%cup` que gera um analisador para ser usado em conjunto com o CUP, nesta seção é onde também são definidas as definições regulares, as definições regulares são definidas no formato: `id = expressão regular`,

também pode inserir código que estará no corpo da classe, bastando colocar código Java entre `%{ %}`.

Na terceira seção é onde se encontram as regras léxicas, bem como as ações associadas a elas, uma regra é definida da seguinte forma: `ExpressãoRegular Ação`. As ações são definidas colocando código Java entre chaves.

3.4 CUP

CUP é uma abreviação para: *Java Based Constructor of Useful Parsers* (HUDSON, 2013). CUP é um sistema para gerar analisadores LALR a partir de especificações simples. Tem o mesmo papel do programa amplamente utilizado YACC e na verdade possui a maioria das características do YACC. No entanto, CUP é escrito em Java, utiliza especificações incluindo código Java embutido, e produz analisadores que são implementadas em Java.

O uso do CUP envolve a criação de uma especificação simples baseada na gramática para qual será feito o reconhecimento, juntamente com a construção de um scanner capaz de ler tokens a partir da entrada.

```
Especificações de pacote e imports.
Código do usuário
Lista de símbolos (terminais e não terminais)
Declarações de precedência
Gramática
```

Tabela 2 – corpo de arquivo CUP

Uma especificação começa com as declarações opcionais de *package* e *import*. As quais têm a mesma sintaxe, e o mesmo papel das declarações de pacote e importações encontrados num programa Java normal.

Seguindo as declarações de pacote e importações podem ocorrer uma série de códigos que são definidos pelo usuário e que serão usados pelo código gerado, existem quatro tipos de definições podem ser feitas, códigos que serão usados pelas ações semânticas: `action code {:código java :};` , códigos que serão utilizados pelo `parser: parser code {:código java :};` , códigos que serão executados antes da execução do `parser: init with code {:código`

`java :};` , e códigos que especifica como o *parser* irá obter o próximo token: `scan with code {:código java :};`.

Após isto tem-se a definição dos símbolos, os símbolos terminais são declarados após a palavra chave ‘terminal’ e os não terminais após a palavra chave ‘non terminal’. Os símbolos podem ter um tipo, onde um tipo é qualquer classe Java, bastando informar o tipo de um símbolo antes de sua declaração.

Na quarta seção temos as declarações de precedências, que é opcional, e especifica a precedência e associatividade dos terminais. Existem três tipos de declarações de precedência, conforme a tabela 3.

```
precedence left terminal[, terminal...];
precedence right terminal[, terminal...];
precedence nonassoc terminal[, terminal...];
```

Tabela 3 – Declaração de precedência

Tem-se então a definição da gramática, esta seção opcionalmente começa com a declaração `start with non-terminal;` que especifica o símbolo não terminal inicial, se um símbolo inicial não for declarado, o símbolo a esquerda da primeira regra declarada será o inicial. Uma regra é definida por um não terminal seguido pelo símbolo “::=” seguido por uma série de símbolos, terminais ou não, zero ou mais ações, e encerrado com um ponto-e-vírgula, os símbolos podem receber um identificador, definindo-o utilizando o símbolo “:” seguido por uma palavra. Uma ação é definida em código Java entre `{:` e `:}`. Se existe mais de uma produção para o mesmo não terminal, elas podem ser agrupadas em uma só, separando-as usando o símbolo “|”.

3.5 GALS: Gerador de Analisadores Léxicos e Sintáticos

GALS (GESSER, 2003) 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.

É através do ambiente gráfico, mostrado na Figura 6, que o usuário entra com sua especificação para a geração do analisador.

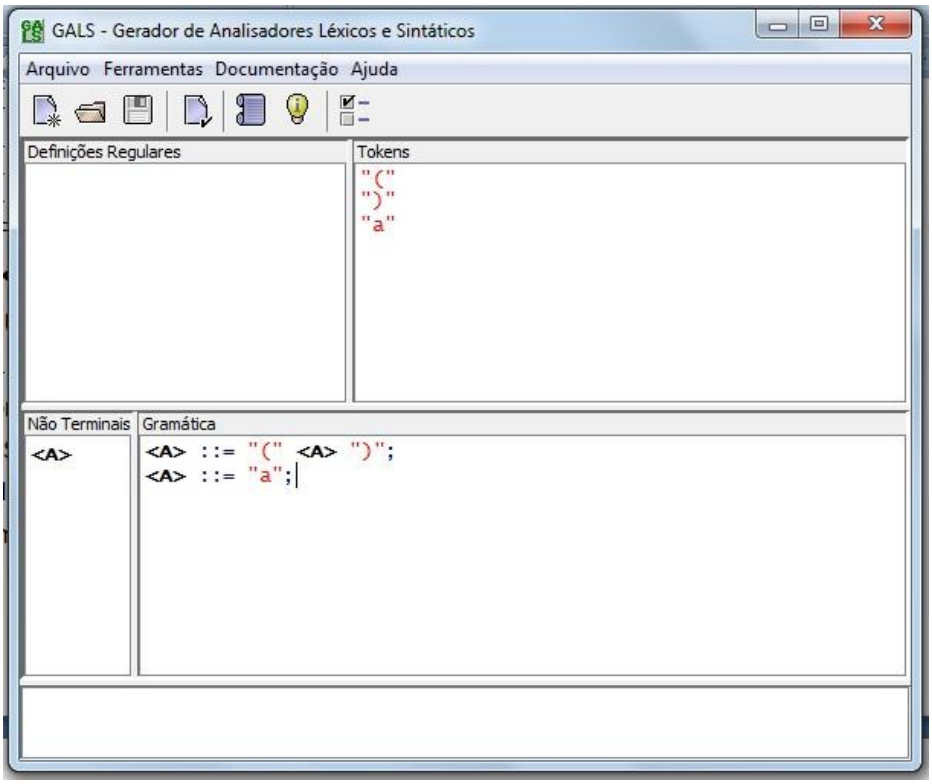


Figura 6 - Tela inicial do GALS

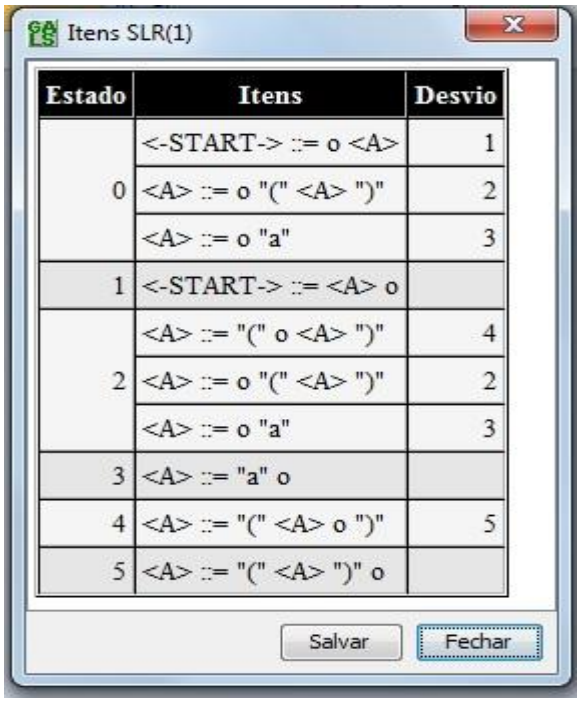
Nesta interface são definidos aspectos léxicos e sintáticos de uma forma conjunta. Porém, dependendo da escolha que se faça, para gerar apenas o analisador léxico ou o sintático, o ambiente é rearranjado de modo adequado.

Efetuando o teste para a gramática G_1 e G_2, visto no capítulo 2, a ferramenta GALS produziu como resultado:

Figura 7 - (GALS) Conjunto first e follow G_1

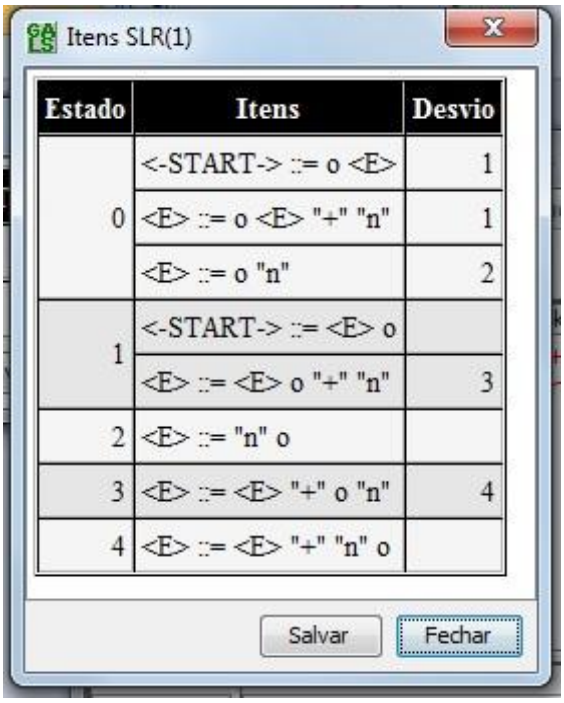
Figura 8 (GALS) Conjunto *first* e *follow* - G_2

Na Figura 7 e na Figura 8 são apresentados uma tabela que mostra o resultado dos algoritmos *first* e *follow* composta por três colunas: a coluna SÍMBOLO, a coluna FIRST e a coluna FOLLOW e a quantidade de linhas é a mesma quantidade de símbolos não terminais da gramática analisada.



Estado	Itens	Desvio
0	<-START-> ::= o <A>	1
	<A> ::= o "(" <A> ")"	2
	<A> ::= o "a"	3
1	<-START-> ::= <A> o	
2	<A> ::= "(" o <A> ")"	4
	<A> ::= o "(" <A> ")"	2
	<A> ::= o "a"	3
3	<A> ::= "a" o	
4	<A> ::= "(" <A> o ")"	5
5	<A> ::= "(" <A> ")" o	

Figura 9 (GALS) Conjunto canônico de itens LR - G_1

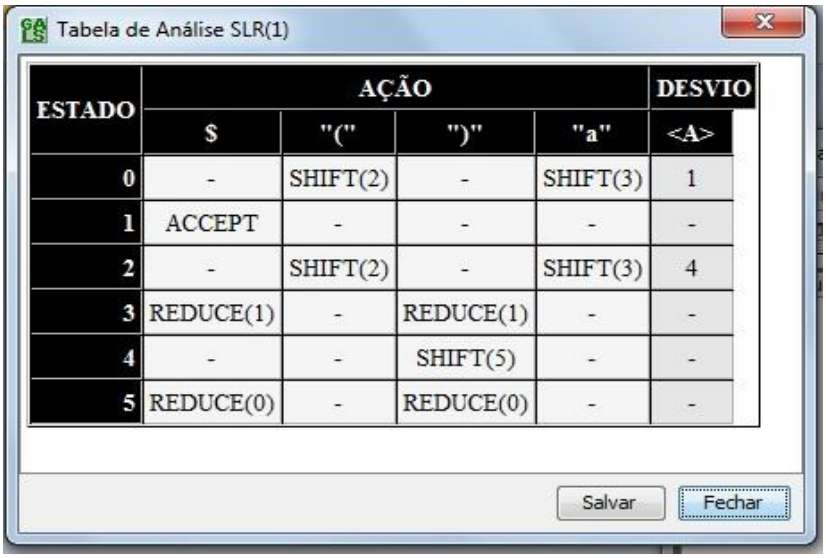


Estado	Itens	Desvio
0	<-START-> ::= o <E>	1
	<E> ::= o <E> "+" "n"	1
	<E> ::= o "n"	2
1	<-START-> ::= <E> o	
	<E> ::= <E> o "+" "n"	3
2	<E> ::= "n" o	
3	<E> ::= <E> "+" o "n"	4
4	<E> ::= <E> "+" "n" o	

Figura 10 - (GALS) Conjunto canônico de itens - G_2

As Figura 9 e 10 mostram a tabela representativa do conjunto canônico de itens LR. Esta tabela tem três colunas e a quantidade de linhas é igual a quantidade de estados. A primeira coluna é “Estado” que indica o estado atual analisado, a segunda coluna é “Itens” que apresenta as regras de produção da gramática com a indicação do ponto que representa até onde a produção foi lida e a terceira coluna é o “Desvio” que indica o número do estado que o analisador finalizou a leitura da produção daquela linha.

As Figuras 11 e 12 apresenta a tabela SLR completa das gramáticas G_1 e G_2.



ESTADO	AÇÃO				DESVIO
	$	"("	")"	"a"	<A>
0	-	SHIFT(2)	-	SHIFT(3)	1
1	ACCEPT	-	-	-	-
2	-	SHIFT(2)	-	SHIFT(3)	4
3	REDUCE(1)	-	REDUCE(1)	-	-
4	-	-	SHIFT(5)	-	-
5	REDUCE(0)	-	REDUCE(0)	-	-

Figura 11 (GALS) Tabela SLR - G_1

Figura 12 - (GALS) Tabela SLR - G_2

3.6 JFLAP

O JFLAP (RODGER, 2009) (Java Formal Language and Automata Package) é uma ferramenta visual usada para criar e simular diversos tipos de autômatos, e converter diferentes representações de linguagens. O seu principal objetivo é facilitar o aprendizado de teoria de linguagens formais, através de uma interface simples e intuitiva. Ele pode ser usado tanto como aparato de auxílio às aulas, como ferramenta de estudo e pesquisa, facilitando tanto a criação de autômatos, quanto a verificação se estão corretos.

O passo inicial para a utilização do JFLAP é a criação de autômatos. O usuário utiliza a interface visual para criar um grafo representando os estados, o diagrama de transições e os seus *labels*. Em seguida, pode definir a palavra de entrada e então visualiza a execução de cada passo do autômato, verificando se o seu projeto é coerente. Isso pode ser feito com três escolhas de execução, um modo rápido, que indica a resposta imediata, um modo passo a passo, que mostra os estados percorridos, e o modo múltiplo, que mostra o teste de diversas palavras, da mesma maneira que o modo rápido.

No modo de conversão do JFLAP, é possível converter a representação de uma linguagem em outro tipo de representação. Dentro das linguagens regulares é possível converter um AFN para um AFD, um AFD para um AFD mínimo, um AFD

para uma expressão regular, um AFN para uma gramática regular e uma gramática regular para um AFN. As transformações suportadas para linguagens livres de contexto são a conversão de um autômato a pilha não determinístico para uma gramática livre de contexto, e três algoritmos para a conversão de uma gramática livre de contexto para um autômato a pilha não determinístico.

O modo inicial de trabalho do JFLAP pode ser escolhido através do menu de entrada:

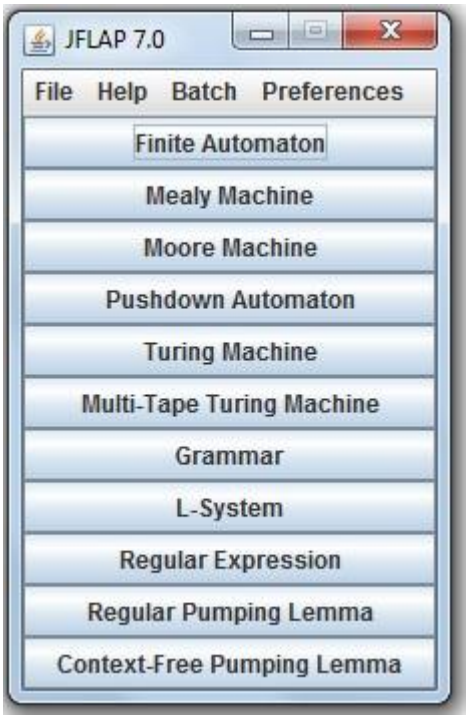


Figura 13 - Tela inicial do JFLAP

A escolha do modo de trabalho abre uma tela especifica para cada tipo de autômato ou estrutura de linguagem (RODGER, 2011). Como visto na Figura 13 no JFLAP existe um modo de trabalho chamado “Grammar”. O modo de criação de uma gramática no JFLAP abrange todas as gramáticas previstas (regulares, livres de contexto...). A entrada dos dados consiste em colocar os não terminais na coluna da esquerda e na direita, os não terminais, o *token* cadeia vazia e os terminais.

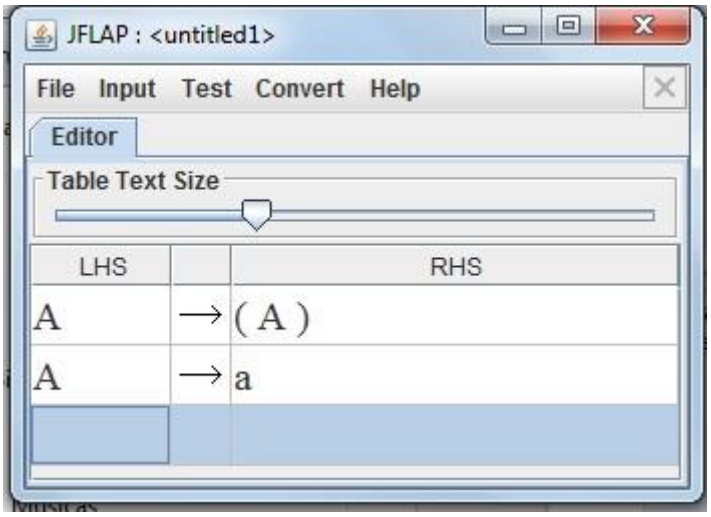


Figura 14 - Entrada de dados de uma gramática

Efetuando o teste para a gramática G_1, visto no capítulo 2, a ferramenta JFLAP produziu como resultado:

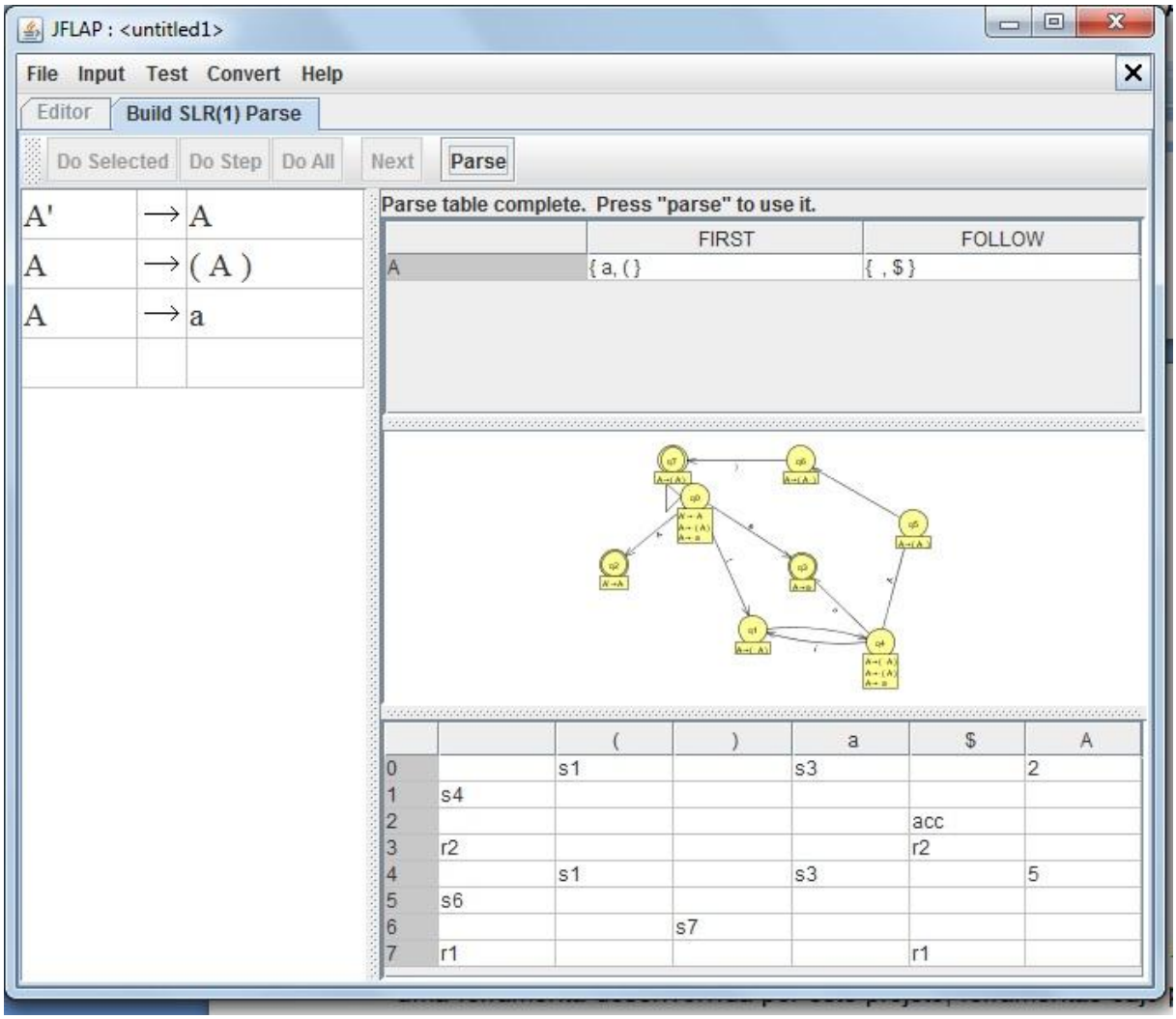


Figura 15 – (JFLAP) Tabela SLR, conjunto *first* e *follow* e conjunto de itens - G_1

Como visto na Figura 15 e 16 a ferramenta JFLAP gera a tabela SLR, o conjunto *first* e *follow* e o autômato representativo do conjunto canônico de itens numa mesma janela.

Efetuando o teste para a gramática G_2, visto no capítulo 2, a ferramenta JFLAP produziu como resultado:

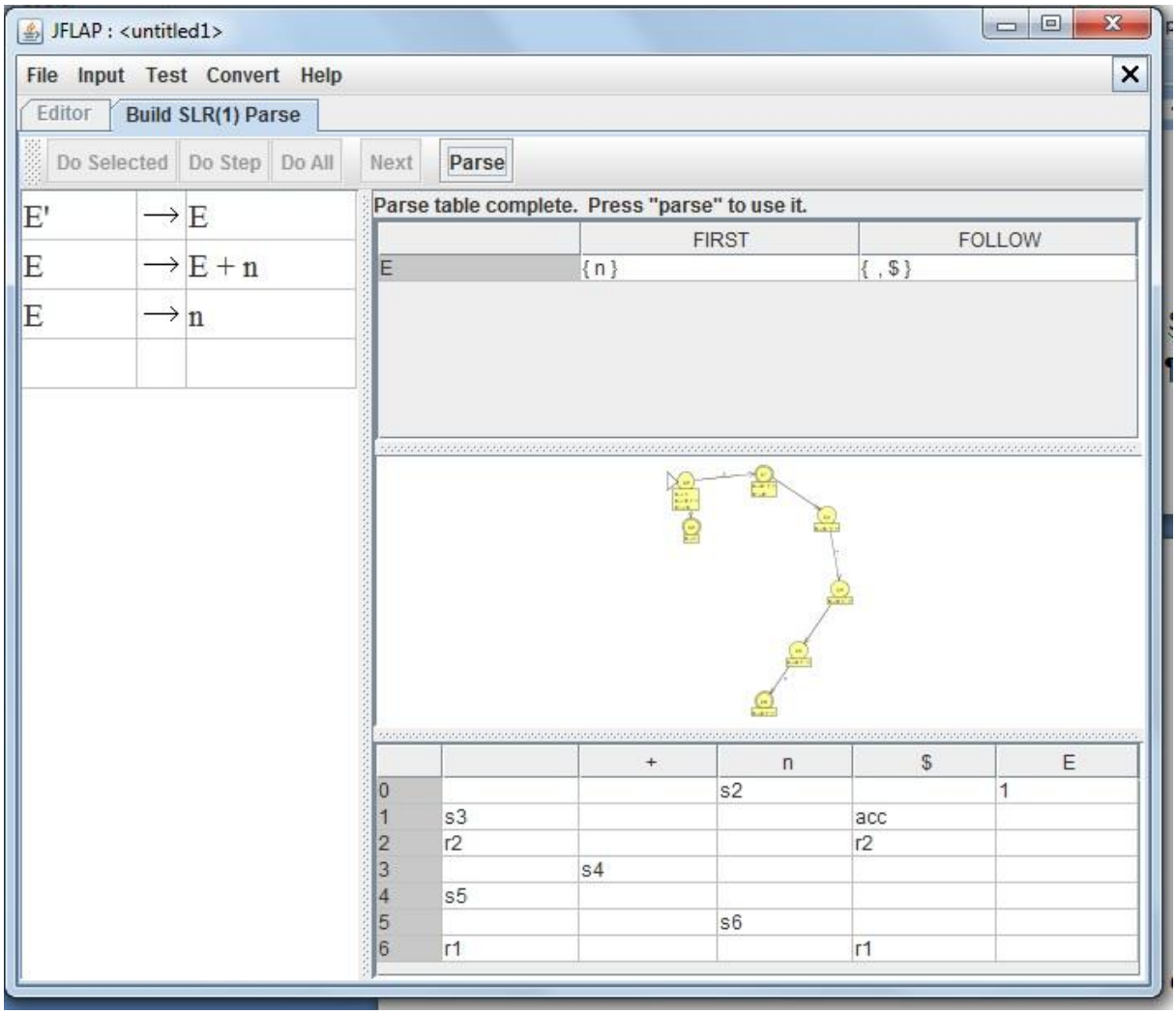


Figura 16 – (JFLAP) Tabela SLR, conjunto first e follow e conjunto de itens - G_2

3.7 Estudo comparativo

Com o objetivo de revelar quais recursos deveriam ser implementados na ferramenta desenvolvida por este projeto, ferramentas cujo propósito é auxiliar a confecção de compiladores foram analisadas. Desta forma foram descobertos os pontos fracos e fortes de cada uma, considerando seu uso em um ambiente educacional.

No quesito interface com o usuário aos sistemas FLEX, YACC, JFLEX e CUP apresentados, a interação entre o usuário e o programa é através da linha de comando. Isto obriga o usuário a especificar um compilador em um arquivo texto, que é enviado para a entrada do programa, gerando como resultado o código fonte de uma parte do *front end* de um compilador ou mensagens de erro. Esta característica comum constitui uma barreira que dificulta o uso das ferramentas apresentadas por parte de usuários iniciantes, pois o aluno está sujeito a cometer muitos erros, e estes somente são vislumbrados no momento em que o arquivo produzido é utilizado para tentativa de geração de código.

A entrada dos programas diminui a facilidade de uso e requer mais tempo de aprendizado da ferramenta. Tal processo deveria ser reduzido para que o aluno se atenha exclusivamente na aplicação da teoria absorvida em sala de aula. O ideal seria que a utilização se desse por meio de uma interface gráfica intuitiva e objetiva sem tornar o usuário dependente de leitura extensiva de manuais.

A ferramenta GALS possui interface gráfica no entanto para inserir os dados da gramática a ser analisada é necessário um estudo prévio da linguagem que o programa se utilizar para entrada dos dados, o que acarreta nas mesmas dificuldades expostas acima.

O programa JFLAP tem uma interface gráfica e não tem a necessidade de aprender uma linguagem. No entanto o programa JFLAP é um projeto que abrange vários tópicos da teoria da computação e que torna está ferramenta completa.

Apesar de já existirem essas ferramentas que auxiliam na construção como todo de compiladores o trabalho aqui proposto foi projetado para uma parte específica que é a construção da tabela SLR e com um diferencial muito importante que é a extensão da tabela para tratamentos e recuperação de erros na análise LR.

4 FERRAMENTA GTSLR

Os possíveis erros que podem surgir ao se implementar os algoritmos de construção da tabela SLR são difíceis de serem detectados caso seja feita sem a ajuda de ferramentas que auxiliem no cálculo das funções, ver capitulo 2, que servem de base para a construção da tabela. A proposta deste trabalho é desenvolver uma ferramenta que irá servir de apoio no desenvolvimento de um compilador.

A ideia é criar um aplicativo para produzir tabelas SLR. O algoritmo SLR(1) foi escolhido não devido a grande complexidade de sua aplicação, mas a grande quantidade de informação produzido pelo mesmo. A ferramenta gera a tabela SLR para uma gramática que o usuário tenha inserido no programa. Concentrando-se em dois aspectos fundamentais: exibir com detalhes o resultado do algoritmo e detectar e relatar todos os erros que ocorrem quando da definição da gramática pelo usuário.

Como visto no capítulo 2 desta monografia, existem vários algoritmos para a construção da tabela SLR a partir da gramática que descreve a estrutura sintática de uma certa linguagem. Neste trabalho implementou-se os algoritmos *closure*, *goto* e o algoritmo para obter o conjunto canônico de itens LR que são a base para construção da tabela SLR e foi implementado, como diferencial de outras ferramentas semelhantes existente, uma extensão para a tabela SLR para recuperação de erros .

A ferramenta foi especificada usando *Unified Modeling Language* (UML) e desenvolvida em Java. O diagrama de caso de uso da Figura 17 apresenta as principais funcionalidades da ferramenta.

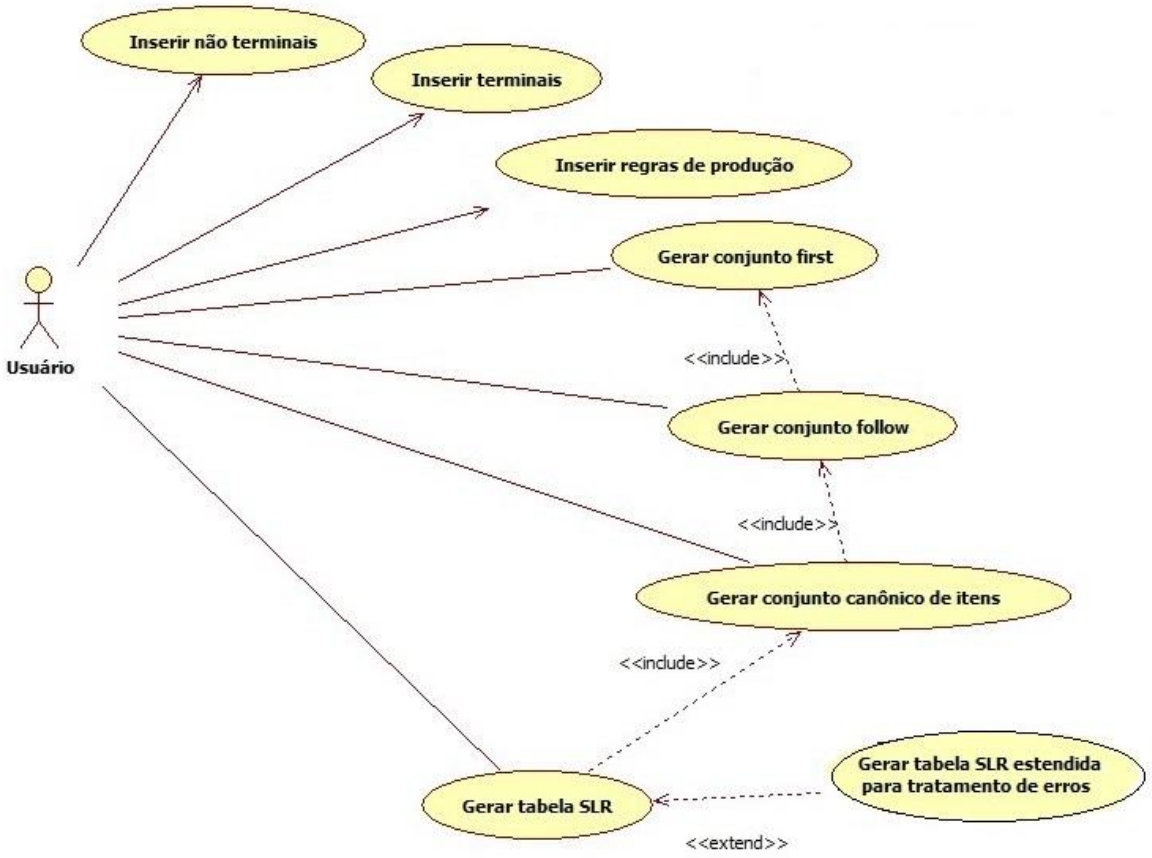


Figura 17 - Diagrama de Caso de Uso

4.1 Interface da ferramenta GTSLR

Ao executar o programa a janela de “Boas-Vindas” é exibida com duas opções: clicar no botão “Entrar” para acessar a janela principal do GTSLR ou clicar no botão “Sair” para encerrar a aplicação.

Figura 18 - Janela de Boas-Vindas

Ao clicar no botão “Entrar” o usuário acessa a tela principal da aplicação mostrada na Figura 19:

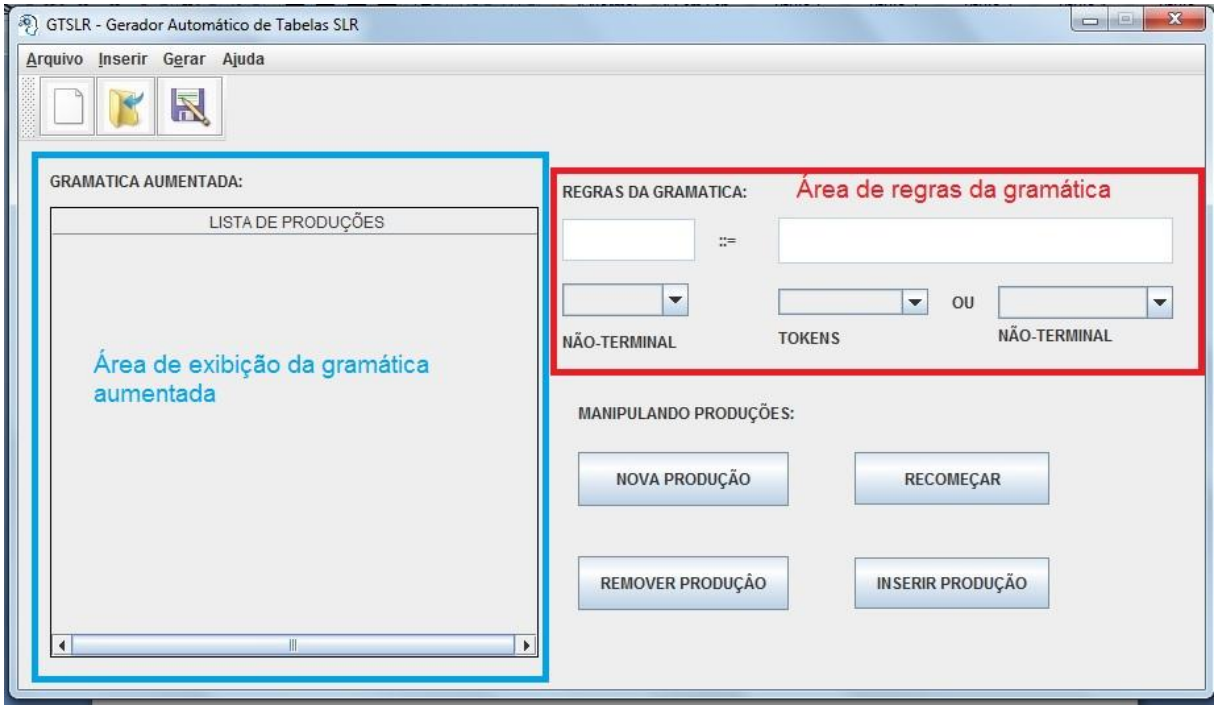


Figura 19 - Tela principal

Na tela principal o usuário deverá utilizar o menu “Inserir” para informar os símbolos terminais e não terminais da gramática livre de contexto.

Figura 20 - Menu Inserir

Clicando no item de menu “Símbolos Terminais” o usuário acessa a tela mostrada na Figura 21:

Figura 21 - Inserir símbolos terminais

Clicando no item de menu “Símbolos Não Terminais” o usuário acessa a tela mostrada na Figura 22:

Figura 22 - Inserir símbolos não terminais

O usuário pode clicar no botão “novo” na barra de ferramentas ou no menu arquivo para ter outra opção de entrada de dados. Ao acessar a funcionalidade de novo arquivo o usuário acessará a janela “entrada de dados” e verá um aviso para definir a lista de Símbolos Terminais e Símbolos não Terminais (Figura 23).

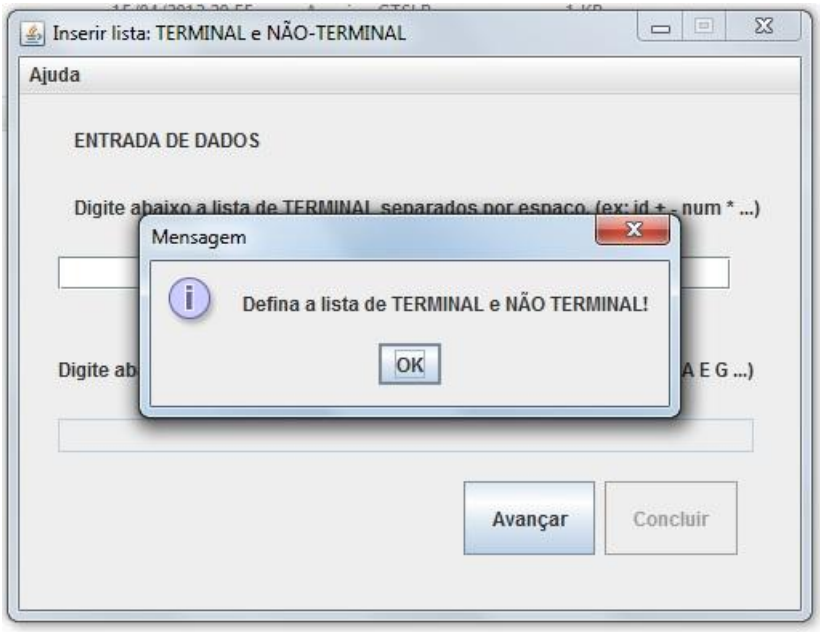


Figura 23 - Aviso para definir a lista de terminal e não terminal

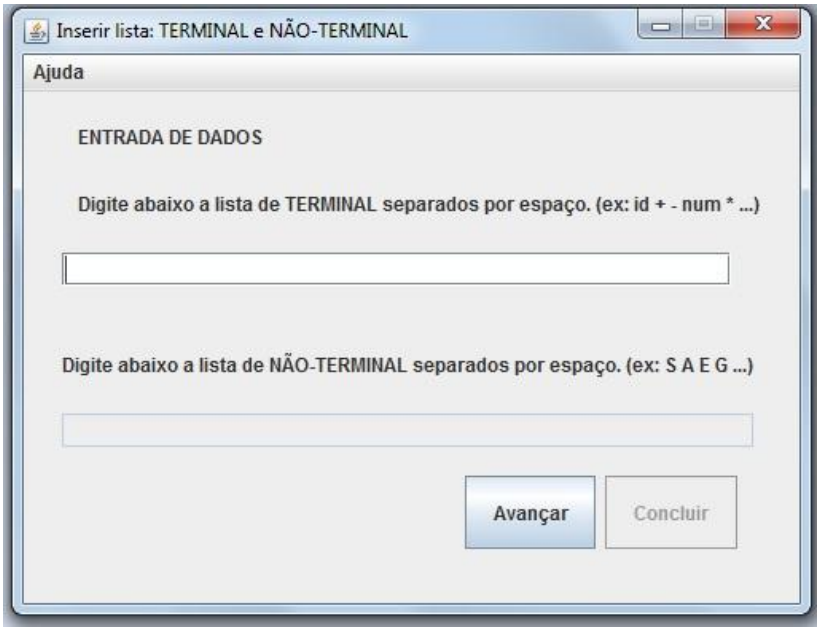


Figura 24 - Janela “Entrada de dados”

A entrada de dados contém os seguintes campos:

a) Campo de texto de entrada de terminais: nesta área devem ser inseridos todos os terminais da gramática, o programa irá identificá-los como tal. O modo de

inclusão é a introdução de uma lista de não terminais separados por pelo menos um espaço. Em seguida, a ferramenta irá descartar e gerar um aviso caso detecte que o usuário entrou por engano o mesmo terminal várias vezes.

Exemplo: `id - + ( ) num`

b) Campo de texto de entrada de não terminais: nesta área serão introduzidos, os símbolos não terminais da gramática. Os símbolos que forem inseridos várias vezes serão descartados.

Exemplo: `S E F`

4.2 Inserir a gramática

Definida a lista de terminais e não terminais ao clicar no “botão concluir” irá ativar novamente a janela principal do programa. É nesta janela que se encontra a área de “Regras da gramática” e a área de “Exibição da gramática aumentada”. conforme mostrado na Figura 19.

a) Área de regras da gramática: Esta é sem dúvida a mais crítica de entrada de dados. O usuário irá inserir cada regra gramatical em uma linha separada. Para evitar equívocos do usuário, a lista de terminais e não terminais previamente definidos, foram inseridos na aplicação. Sendo assim, o usuário simplesmente irá montar a regra de produção da gramática. Por exemplo, caso o usuário queira inserir a produção “`A -> ( A )`”, primeiramente seleciona o símbolo não terminal “`A`” na caixa de combinação “NÃO TERMINAL” do lado esquerdo, depois clica na caixa de combinação “TOKENS” para selecionar o símbolo terminal “`(`”, em seguida clica no símbolo não terminal “`A`” na caixa de combinação do lado direito e finalmente seleciona o símbolo terminal “`)`” na caixa de combinação “TOKENS” conforme mostra a Figura 25.

Figura 25 - Área de regras da gramática

Feito isso, clica-se no “botão inserir produção” e a regra gramatical será exibida na área de “Exibição da gramática aumentada”. No exemplo, como a produção “`A -> ( A )`” foi a primeira a ser inserida, o GTSLR atribuiu o símbolo não terminal “`A`” como sendo o símbolo inicial da gramática acrescentando a produção “`AInicial -> A`”, gerando assim a gramática aumentada, conforme mostra a Figura 26.

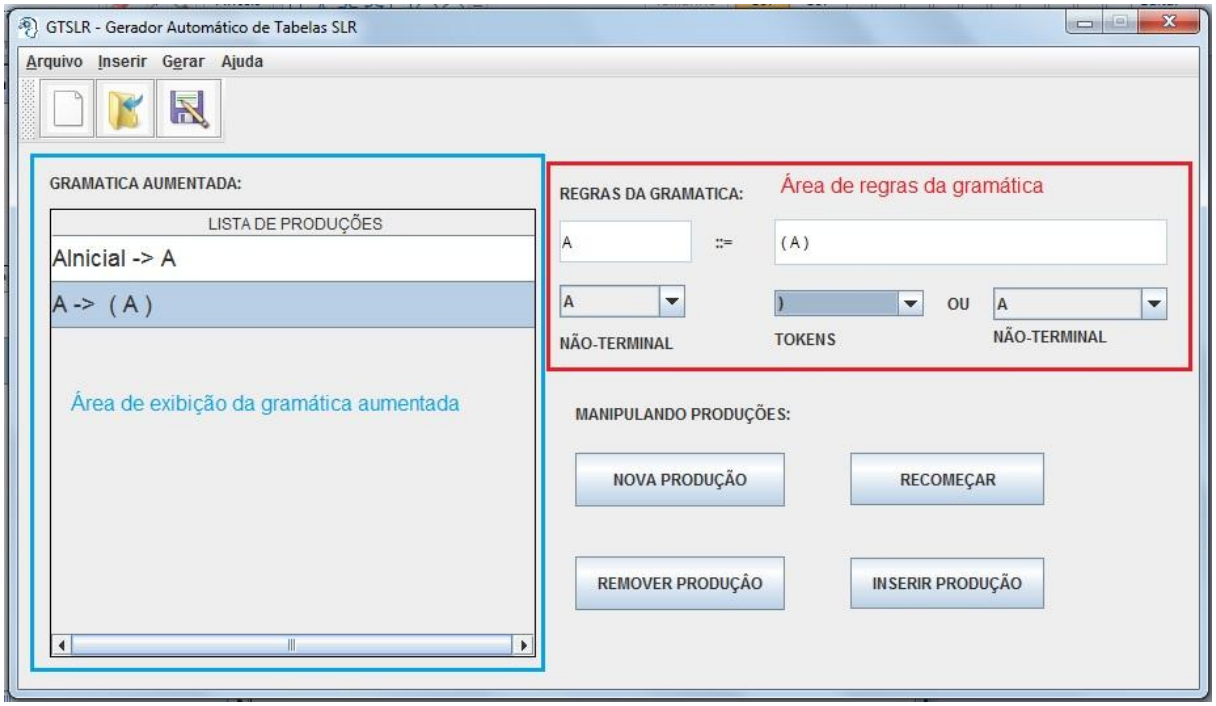


Figura 26 – Inserindo produção

Caso o usuário tenha errado uma regra, basta clicar no “botão nova produção” e começar a montar a regra gramatical novamente.

Clicando no botão “remover produção” a ferramenta irá excluir da lista de produções, que se encontra na área de “Exibição da gramática aumentada”, a produção selecionada.

O analisador testará se existem regras de gramática repetidas, descartando e emitindo um aviso de erro e percorrerá a gramática na mesma ordem que foram inseridas as regras gramaticais.

b) Área de exibição da gramática aumentada: nesta área conforme o usuário for inserindo as regras da gramática, o mesmo poderá ir visualizando a gramática por completo. O símbolo inicial de uma gramática, normalmente é a letra S', no entanto o programa usa como símbolo inicial o não terminal da primeira regra gramatical definida acrescido da palavra "Inicial", possibilitando ao usuário esta flexibilidade de escolha.

Caso o usuário tenha errado na composição da gramática basta clicar no "botão recomeçar" que irá remover todas as regras da gramática permitindo assim que o usuário posso começar a inserir as produções novamente.

4.3 Testes

No menu gerar o usuário tem a opção de visualizar o conjunto *first* e *follow*, o conjunto canônico de itens e a tabela SLR.

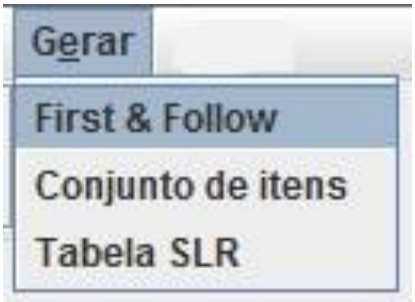


Figura 27 - Menu Gerar

4.3.1 Gramática LR(0)

Efetuando o teste para a gramática G_1 do capítulo 2, a ferramenta produziu como resultado:

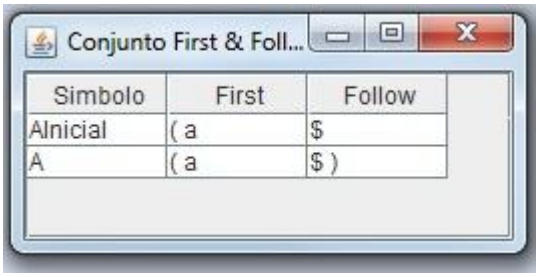


Figura 28 – (GTSLR) Conjunto first e follow - G_1

A Figura 28 apresenta o resultado do cálculo das funções *first* e *follow* na forma de tabela. A tabela tem três colunas: uma coluna “Símbolo” que representa os não terminais analisados, a coluna “First” onde aparece os terminais que resultaram do cálculo da função *first* do respectivo não terminal e a coluna “Follow” com o resultado do cálculo da função *follow*.

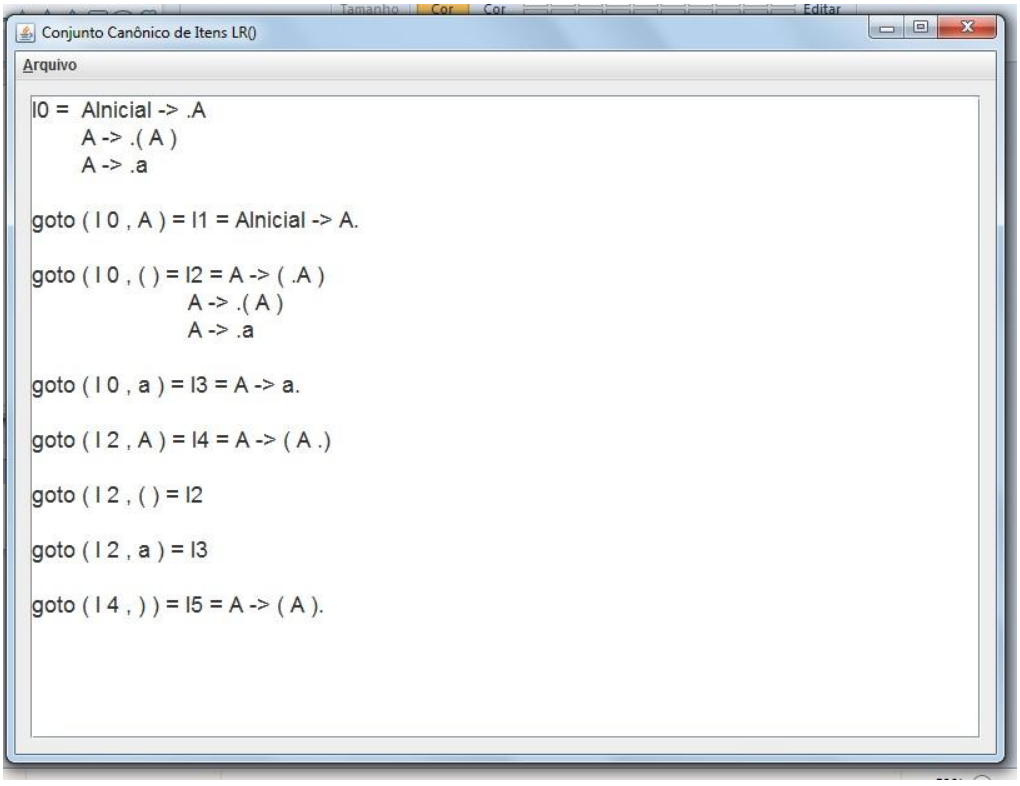
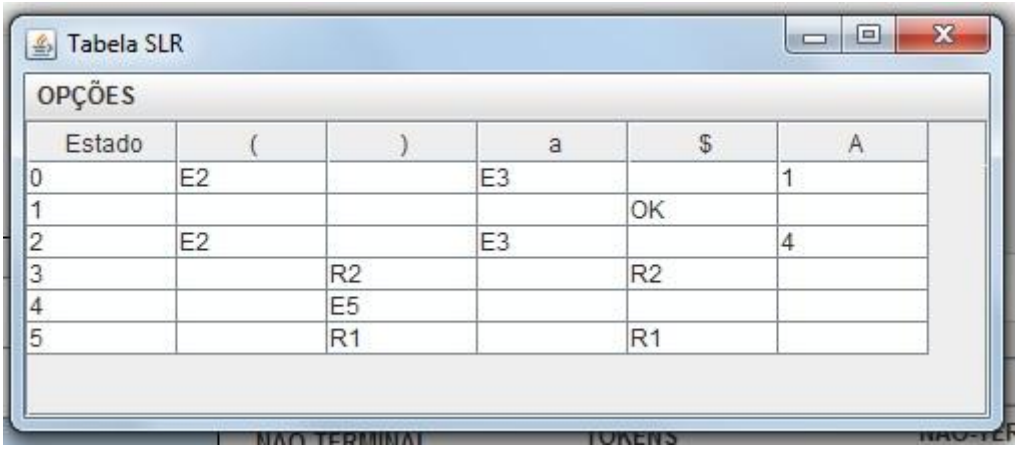


Figura 29 – (GTSLR) Conjunto canônico de itens LR - G_1

A Figura 29 mostra o resultado do cálculo do conjunto canônico de itens LR. Começa com o “I0” que representa o estado inicial do autômato, o próximo passo é fazer a leitura dos símbolos que estão depois do ponto: no estado “I0” quando o analisador ler o símbolo não terminal “A”, avança-se para o estado “I1”; “I0” lendo o símbolo “(” passa para o estado “I2”; no estado “I0” lendo “a” muda-se

para o estado "I3"; feito a análise de todos os símbolos depois do ponto no estado "I0" começa a fazer a análise do estado "I1" porém neste estado o ponto já chegou ao fim da produção ou seja já foi feita a leitura até o final, por isso passa –se a ser feita a análise no estado "I2". Estando no estado "I2" lendo o símbolo "A" passa para o estado "I4"; "I2" lendo "(" permanece no estado "I2" pois ao avançar o ponto no símbolo "(" têm-se a mesma situação que determinou o estado "I2"; no estado "I2" lendo o símbolo "a" volta muda-se para o estado "I3". Feita a leitura de todas as produções do estado "I2" passa a ser analisado o próximo estado porém o estado "I3" a analise já chegou ao fim da produção portanto avança a analise para o estado "I4", no estado "I4" lendo o símbolo ")" gera o estado "I5" e finaliza a análise.



Estado	(	)	a	$	A
0	E2		E3		1
1				OK	
2	E2		E3		4
3		R2		R2	
4		E5			
5		R1		R1	

Figura 30 – (GTSLR) Tabela SLR - G_1

A Figura 30 mostra a tabela SLR pronta, formada por uma coluna "Estado", uma coluna com o símbolo "$" que representa o fim de cadeia, e entre estas colunas tem-se um coluna pra cada símbolo terminal da gramática e por fim tem-se uma coluna para cada símbolo não terminal da gramática.

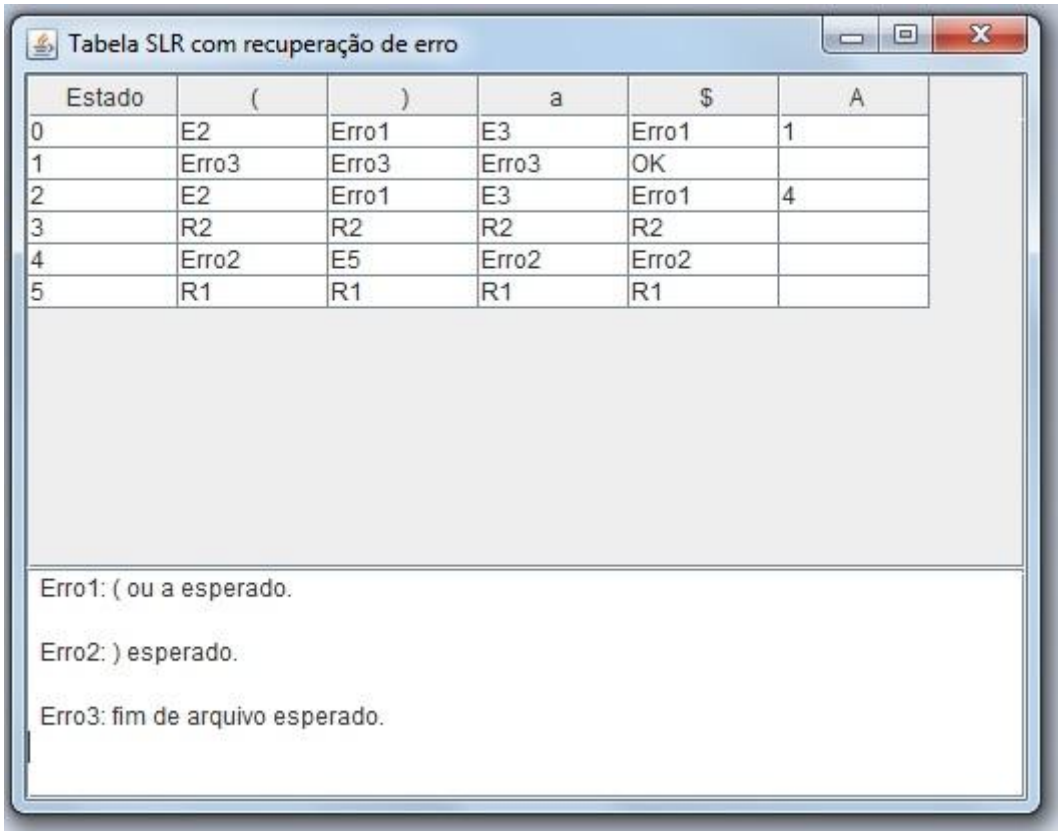


Figura 31 – (GTSLR) Tabela SLR estendia para tratamento de erro - G_1

:A Figura 31 mostra a mesma tabela SLR da Figura 30 porém as células em branco foram preenchidas com rotinas de erro e abaixo tem-se uma legenda traduzindo o significado de cada tipo de erro.

4.3.2 Gramática SLR(1)

Efetuando o teste para a gramática G_2 do capitulo 2, a ferramenta produziu como resultado:

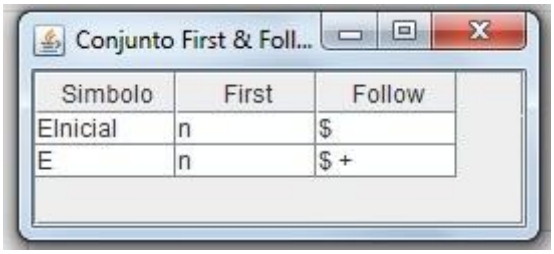


Figura 32 – (GTSLR) Conjunto First e Follow - G_2

A Figura 32 apresenta o resultado do cálculo das funções *first* e *follow* na forma de tabela. A tabela tem três colunas: uma coluna “Símbolo” que representa os não terminais analisados, a coluna “First” onde aparece os terminais que resultaram do cálculo da função *first* do respectivo não terminal e a coluna “Follow” com o resultado do cálculo da função *follow*.

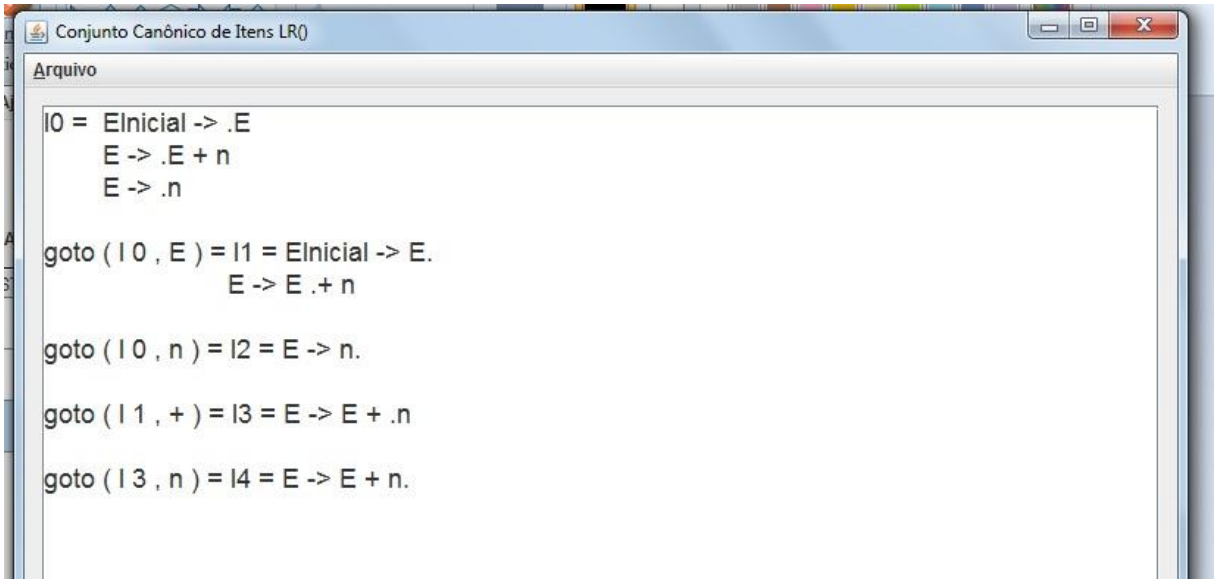
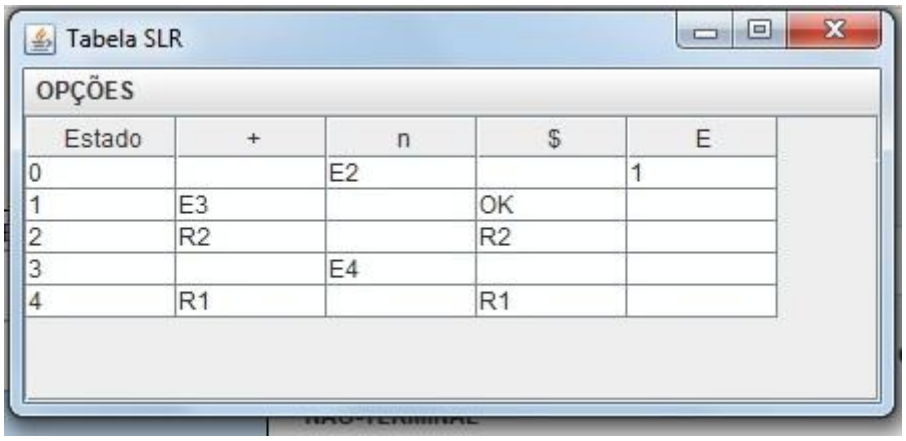


Figura 33 – (GTSLR) Conjunto canônico de itens LR - G_2

A Figura 33 mostra o resultado do cálculo do conjunto canônico de itens LR. Começa com o “I0” que representa o estado inicial do autômato, o próximo passo é fazer a leitura dos símbolos que estão depois do ponto: no estado “I0” quando o analisador ler o símbolo não terminal “E”, avança-se para o estado “I1”; “I0” lendo o símbolo “n” passa para o estado “I2”; feito a análise de todos os símbolos depois do ponto no estado “I0” começa a fazer a análise do estado “I1”. No estado “I1” a primeira produção já foi lida até o final logo a análise passa para o próximo símbolo que tem o ponto antes; “I0” lendo “+” muda-se para o estado “I3”. Feita a leitura de todas as produções do estado “I1” passa a ser analisado o estado “I2” no entanto este estado já foi lida a produção até o final portanto analisa-se o estado “I3” lendo o símbolo “n” que gera o estado final do autômato “I4”.



Estado	+	n	$	E
0		E2		1
1	E3		OK	
2	R2		R2	
3		E4		
4	R1		R1	

Figura 34 – (GTSLR) Tabela SLR - G_2

A Figura 34 mostra a tabela SLR pronta, formada por uma coluna “Estado”, uma coluna com o símbolo “$” que representa o fim de cadeia, e entre estas colunas tem-se um coluna pra cada símbolo terminal da gramática e por fim tem-se uma coluna para cada símbolo não terminal da gramática.

Estado	+	n	$	E
0	Erro1	E2	Erro1	1
1	E3	Erro2	OK	
2	R2	R2	R2	
3	Erro1	E4	Erro1	
4	R1	R1	R1	

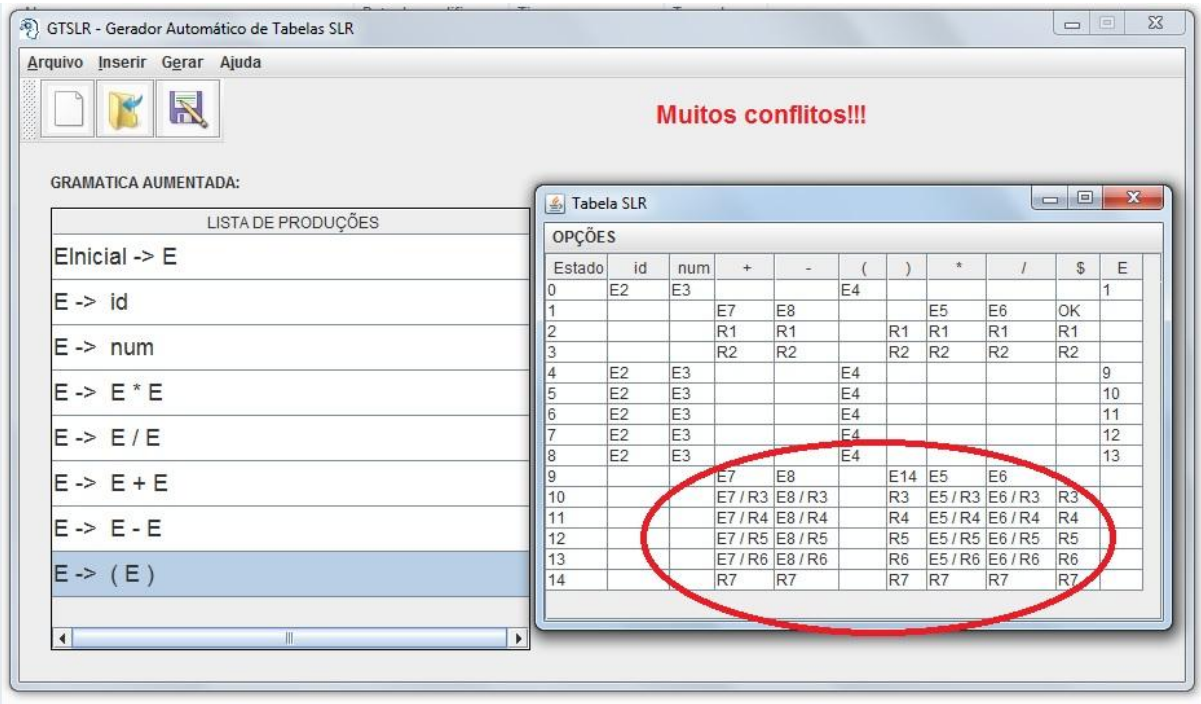
Figura 35 – (GTSLR) Tabela SLR estendida para tratamento de erro - G_2

:A Figura 35 mostra a mesma tabela SLR da Figura 34 porém as células em branco foram preenchidas com rotinas de erro e abaixo tem-se uma legenda traduzindo o significado de cada tipo de erro.

4.3.3 Gramática que não é SLR(1)

Segundo Price et al. (2008), se ações conflitantes são geradas na tabela de ação, onde numa mesma lacuna da tabela SLR existir mais de uma ação de empilhar ou reduzir, então a gramática não é SLR(1).

Na Figura 36 é apresentado um exemplo de gramática que não é SLR(1).



Estado	id	num	+	-	(	)	*	/	$	E
0	E2	E3			E4					1
1			E7	E8			E5	E6	OK	
2			R1	R1		R1	R1	R1	R1	
3			R2	R2		R2	R2	R2	R2	
4	E2	E3			E4					9
5	E2	E3			E4					10
6	E2	E3			E4					11
7	E2	E3			E4					12
8	E2	E3			E4					13
9			E7	E8		E14	E5	E6		
10			E7 / R3	E8 / R3		R3	E5 / R3	E6 / R3	R3	
11			E7 / R4	E8 / R4		R4	E5 / R4	E6 / R4	R4	
12			E7 / R5	E8 / R5		R5	E5 / R5	E6 / R5	R5	
13			E7 / R6	E8 / R6		R6	E5 / R6	E6 / R6	R6	
14			R7	R7		R7	R7	R7	R7	

Figura 36 - Gramática que não é SLR(1)

Numa gramática que não é SLR(1) a ferramenta GTSLR também gera a tabela SLR no entanto numa mesma célula aparece duas ações possíveis conforme mostra a Figura 36 na área com um círculo vermelho.

Por conseguinte a ferramenta GTSLR atingiu o objetivo principal de não apenas gerar a tabela SLR, como as outras ferramentas citadas neste trabalho fazem, como também gerar a tabela estendida pra tratamento de erros. Os testes apresentados neste capítulo provou a eficácia da ferramenta.

5 CONCLUSÕES

5.1 Considerações finais

O programa produzido elimina a necessidade do emprego de muito tempo e esforço para aprender a utilizar ferramentas cujo foco é voltado a aplicações profissionais, que exigem que as técnicas já estejam dominadas, ao invés de auxiliar o aluno a dominá-las.

A interface produzida permite a fácil aplicação e experimentação da teoria, com funcionalidades suficientes para servir de acompanhamento no decorrer da disciplina compiladores, auxiliando na eliminação de incertezas sobre uma parte do funcionamento de um compilador.

A ferramenta desenvolvida atende ao objetivo principal planejado que é o de servir como auxílio didático tanto para professores como alunos para a compreensão de uma parte importante da construção de compiladores.

5.2 Trabalhos futuros

O sistema apresentado é muito modesto em sua situação atual, logicamente não é um produto acabado, mas em desenvolvimento, é o princípio de um projeto muito mais ambicioso para trabalhos futuros que seria a extensão desta ferramenta para que além da construção da tabela SLR para gramáticas LR(0) e SLR(1) o programa poderia ir mais além, tratando outros tipos de gramáticas livres de contexto.

REFERÊNCIAS

AHO, Alfred V.; ULLMAN, Jeffrey D.; SETHI, Ravi. *Principles of Compiler Design*. Reading, Massachusetts, EUA: Addison-Wesley, 1977.

BROWN, D; LEVINE, J. R.; MASON, T. *Lex e yacc – second edtion*. O'Reilly Media, Inc., 1992.

DELAMARO, Marcio. *Como Construir um Compilador Utilizando Ferramentas Java.* São Paulo: Novatec, 2004.

GESSER, Carlos Eduardo. *Gals: gerador de analisadores léxicos e sintáticos.* Florianópolis, 2003.

GREENLAW, R. and HOOVER, H. J. *Fundamentals of the Theory of Computation - Principles and Practice*. Editora: Morgan Kaufmann Publishers, 1998.

GRUNE, D., Bal, H. E., Jacobs, C. J. H., and Langendoen, K. G. *Projeto Moderno de Compiladores: Implementacõess e Aplicacões*. Editora: Campus, 2001.

HUDSON, Scott. *CUP User's Manual*. Disponível em: <http://www2.cs.tum.edu/projects/cup/manual.html>. Acesso em: 25 ago. 2013.

JFLEXFEATURES. *JFLEX – Features*. Disponível em: http://www.jflex.de/features.html. Acesso em: 25 ago. 2013.

KLEIN, Gerwin. *JFlex User's Manual*. Disponível em: http://www.jflex.de/manual.html. Acesso em: 25 ago. 2013.

KNUTH, D. E. *Backus Normal Form vs. Backus Naur Form.* 1964.

LOUDEN, C. *Compiladores: Princípios e Práticas*. São Paulo: Pioneira Thomson Learning, 2004.

MELO, P. C. B. *Ferramenta de Especificação Gráfica de Máquinas de Estados Finitas para o Ambiente de Teste baseado em Injeção de Falhas por Software (ATIFS)*. Natal: Universidade Federal do Rio Grande do Norte, 2003.

PAXSON, Vern. *Flex – A scanner Generator*. 1998. Disponível em: <http://www.gnu.org/software/flex/manual>. Acesso em: 20 de ago. 2013.

PITTMAN, T; PETERS, J. *The Art of Compiler Design: Theory and Practice*. Englewood Cliffs, Nova Jersey, EUA: Prentice Hall, 1992.

PRICE, Ana M. A.; TOSCANI, Simão S. *Implementação de Linguagens de Programação – Compiladores*. Rio Grande do Sul: Bookman, 2008**.**

RODGER, S. H. et al. *Changes to JFLAP to Increase its Use in Courses*, 2011. Disponível em: <http://www.jflap.org/papers.html>. Acesso em: 20 ago. 2013.

RODGER, S. H. et al. *JFLAP 7.0 Tutorial,* 2009. Disponível em: < http://www.jflap.org/tutorial/>. Acesso em: 15 mar. 2013.

WILHELM, Reinhard; MAURER, Dieter. *Compiler Design. Harlow*. England: Addison-Wesley, 1995.