USP - INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

GrView

Um Gerador Gráfico de Analisadores Sintáticos

MAC 0499 – Trabalho de Formatura Supervisionado

Aluno: Gustavo Henrique Braga
Orientador: Valdemar W. Setzer

Dezembro de 2009

Resumo

Este trabalho fundamentou-se na reformulação de uma ferramenta gráfica para a criação e depuração de compiladores, desenvolvida em 2004 [1] por Luiz Fernando dos Santos Pereira na forma de um *plugin* para a plataforma *Eclipse* [2]. Havia na ferramenta algumas limitações e dependências indesejáveis, as quais ocasionariam problemas constantes de manutenção em função da incompatibilidade do *plugin* com versões atualizadas do *Eclipse*. Esses fatores motivaram a criação de um aplicativo gráfico, totalmente autônomo, que contasse com uma ferramenta similar, além de incorporar, na medida do possível, uma série de refinamentos e recursos adicionais.

Essa ferramenta viabiliza a criação, manipulação e visualização de grafos sintáticos que representam graficamente uma gramática livre de contexto restrita. A partir do grafo é gerada uma tabela que, por sua vez, juntamente com um interpretador genérico, constitui um analisador sintático a ser usado, tanto na programação profissional de compiladores, como para testes e, educacionalmente, no aprendizado de gramáticas. Esse analisador trata de uma subclasse das gramáticas *LL(1)*, característica que, além de favorecer a eficiência do analisador, permite uma notável simplicidade estrutural e facilidade na depuração de erros na gramática.

Ao aplicativo, desenvolvido em Java, foi dado o nome de ***grView.*** Neste trabalho foram executados os seguintes objetivos:

- Implementação de rotinas gráficas, para independização em relação ao *Eclipse.*
- Uso de linguagem Java, de modo a ser compatível com qualquer sistema operacional, com jvm 1.5, ou superior); e utilização de licença de código aberto.
- Criação de ambiente de programação para criação e edição de:
    - analisadores léxicos (ou, modificação de um analisador léxico fornecido)
    - rotinas semânticas, com possibilidade de modificação em tempo de execução (por meio de linguagem de script Groovy).
- Criação de um editor para arquivos com código em Java, e para arquivos

contendo definições do analisador léxico, e de rotinas semânticas.

- Novos métodos de depuração dos grafos sintáticos.

- Introdução de uma estrutura de projetos, agrupando todos os arquivos concernentes a cada grafo sintáticos, dando facilidade para seu gerenciamento.

- Exportação automática de relatórios em HTML, contendo diversas informações sobre um grafo sintático, conteúdo de pilhas utilizadas durante o processo de análise sintática do grafo, etc.

- Geração de código em Java contendo um analisador sintático completo.

1. Introdução

Este trabalho, que se contextualiza nas áreas de criação de compiladores, conceitos de linguagens de programação e engenharia de software, teve por objetivo o desenvolvimento do *grView*, um aplicativo baseado em interface gráfica de usuário. Sua principal funcionalidade consiste na criação e manipulação de grafos, denominados Grafos Sintáticos Simples (GSSs), capazes de representar uma subclasse das gramáticas LL(1), que foi denominada por V.W.Setzer de GSLL(1), *Generalized Simple* LL(1), generalizando as gramáticas ESLL(1) introduzidas por ele [4]. A partir desses grafos, o sistema gera analisadores sintáticos em Java, independentes da plataforma de execução desses analisadores.

O *grView*, por meio da utilização dos GSSs e da geração de analisadores GSLL(1), propõe tornar ágil e relativamente fácil o processo de especificação de gramáticas, e de geração dos analisadores sintáticos correspondentes. Dentre outras características dos analisadores GSLL(1) inclui-se a emissão automática de mensagens de erro durante a compilação usando o analisador, sem referências a nós não terminais das produções, e a recuperação de erros sintáticos, sem a necessidade de introduzir produções especiais para a detecção de erros sintáticos e sua "correção" (como é o caso de analisadores *bottom-up*, como os baseados em gramáticas LR(1), por exemplo Yacc e Bison). Esta abordagem, que o diferencia de qualquer sistema semelhante, faz do *grView* um sistema particularmente atraente enquanto alternativa para o desenvolvimento rápido de analisadores sintáticos eficientes e práticos. Além disso, pode ser usado para fins didáticos em cursos de análise sintática e de compilação. Foram esses os principais fatores que motivaram sua criação.

Um *plugin* para a plataforma Eclipse [2], chamado *ASIN* [1], desenvolvido por Luiz Fernando dos Santos Pereira, já permitia a criação de analisadores como os do *grView*, ainda que de forma incompleta: nem todos os elementos constituintes dos GSSs estavam disponíveis e o *ASIN* não dispunha de métodos diretos para configuração do analisador léxico, das rotinas semânticas e nem para exportação de código final do analisador. Sua maior limitação, contudo, estava em sua dependência do sistema Eclipse, exigindo sua disponibilidade. Além disso, seria necessária uma manutenção constante para manter o *ASIN* totalmente compatível com versões posteriores desse sistema. Não teria sido, portanto, uma boa opção simplesmente retomar o desenvolvimento do *ASIN*. Um requisito primordial para o *grView* seria o de ser pensado, desde o início, como um aplicativo totalmente autônomo. Este requerimento levantou alguns problemas: além de implementar funcionalidades semelhantes às já presentes no *plugin,* seria desejável criar um ambiente de desenvolvimento que replicasse os recursos existentes no Eclipse dos quais a usabilidade do *plugin* se beneficiava. Em particular, um layout baseado em abas acopláveis (*docking*) que oferecesse flexibilidade para a visualização simultânea de diversos componentes, a possibilidade de editar arquivos localmente e um sistema de gerenciamento dos diversos arquivos de um projeto.

2. Conceitos

Nesta seção serão esclarecidos alguns conceitos fundamentais para o desenvolvimento e utilização do programa, incluindo análises léxicas, sintáticas e semânticas, analisadores descentes (*top*-down), gramáticas LL(1) e GSLL(1), GSSs e técnicas de programação orientada a objeto relevantes.

Para fins de esclarecimento, a geração de analisadores sintáticos do *grView* deverá ser entendida, daqui em diante, como o processo de criação de componentes para a análise sintática de um código fonte na linguagem especificada pelo grafo sintático (ver adiante). O *grView*, dá ao programador a possibilidade de gerar um analisador léxico e de inserir rotinas semânticas para análise de contexto e geração de código.

No restante desta seção será adotada a seguinte nomenclatura: o não terminal inicial de uma gramática será denominado "símbolo inicial da gramática", ao passo que o lado esquerdo de uma produção será chamado de "não terminal à esquerda".

2.1. Análise e Síntese

O processo de geração de analisadores sintáticos pode ser mais bem compreendido se separado em duas fases: *análise* e *síntese* (ver [3] e figura 1) que, para efeitos didáticos serão inicialmente considerados independentes. Esse tipo de compilador é denominado de "compilador de dois passos".

Neste tipo de compilador o primeiro passo é a *análise.* A partir os caracteres de um texto (código fonte de entrada) a análise agrupa-os em estruturas, detecta e atribui contextos aos elementos dessas estruturas e gera um código intermediário. Gera como resultados tabelas auxiliares tanto para a própria análise quanto para a fase de síntese (tabela intermediárias), como ilustrado na figura 1. Além disso, detecta erros sintáticos e inconsistências semânticas. Caso seja bem sucedida, a fase de análise produzirá automaticamente uma tabela de símbolos com os identificadores da linguagem sendo analisada. Caso não seja bem sucedida, nesta fase erros e informações para depuração devem ser emitidos.

Durante a *síntese*, o código intermediário e as tabelas auxiliares são usados para se fazer uma geração de código objeto.

O *grView* gera um analisador para um compilador de um só passo. Isto é, a síntese é feita à medida que a análise a permite, e não necessariamente em dois momentos distintos e consecutivos. Contudo, pode-se notar que o *grView* também permite, por meio de rotinas semânticas adequadas, a realização da primeiro passo somente.

Embora no *grView*, essas duas fases não sejam independentes, para utilizá-lo e, eventualmente modificá-lo, é especialmente útil compreender detalhadamente a fase de análise. Essa fase será, por sua vez, dividida em três partes, conforme ilustrado na figura 1: *Análise Léxica, Análise Sintática* e *Análise de Contexto.* As seções a seguir descrevem com mais detalhes estas partes da análise.

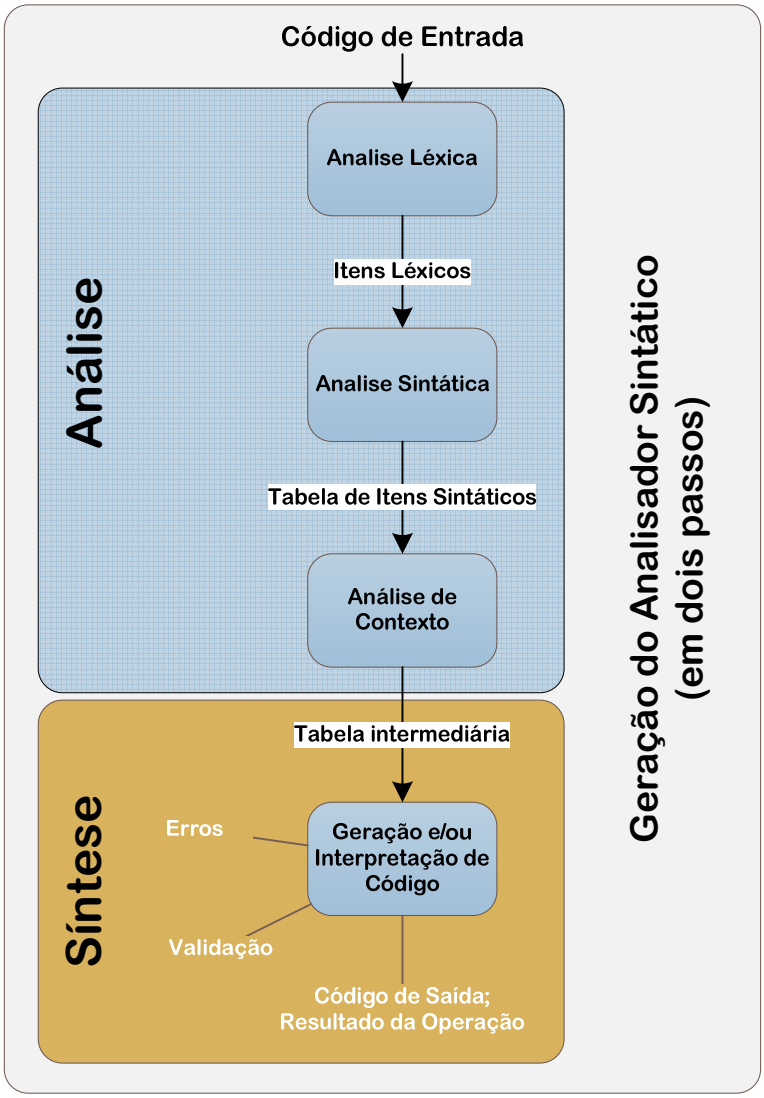


Figura 1 - Geração do analisador sintático

2.2. Análise Léxica

A tarefa de um analisador léxico é ler, individualmente, os caracteres do código de entrada (ou código-fonte), e agrupá-los em seqüências associadas a símbolos reservados (símbolos especiais e palavras reservadas), a identificadores e a números. Essas seqüências são chamadas *itens léxicos*, e constituem os símbolos gramaticais mais elementares.

Mais detalhadamente, os caracteres de entrada são agrupados para formar palavras, que são verificadas contra um conjunto de regras para determinar se são válidas, ou reconhecíveis, e nesse caso utilizá-las para compor um item léxico correspondente. Essas regras são simples e definíveis por gramáticas regulares (o tipo mais simples de gramática formal), permitindo que os analisadores léxicos sejam, tipicamente, muito eficientes; podem incluir também instruções para converter identificadores especiais em determinados símbolos, consultando para tanto uma tabela pré-configurada.

Além de gerar unicamente os itens léxicos, um analisador léxico normalmente pode realizar outras tarefas, como eliminar comentários e compactar espaços em branco (ou tabulações, caracteres de nova linha, etc.) consecutivos em um só. Pode-se dizer que ao realizar essas tarefas preliminares, a análise se encontra na fase de varredura (ou, em inglês, *scanning*) do código-fonte.

Alguns exemplos de itens léxicos são dados na tabela 1. Nesta tabela, a primeira coluna se refere tanto a símbolos sintáticos, estabelecidos por uma gramática, quanto aos identificadores especiais supracitados.

Símbolo, ou identificador	Descrição	Exemplos de palavras lidas	Item léxico reconhecido, <símbolo, palavra>
IF	"se" condicional	IF	<if,if>
while	loop while	While	<while, while>
Numb	números inteiros ou decimais	9, -8, 325.65	<Numb, 9> <Numb, -8> <Numb, 325.65>
Comp	operador de comparação	:=. >=, <	<Comp, :=> <Comp, >=> <Comp, <>

Tabela 1 - Exemplo de criação de léxicos.

Vale ressaltar que a importância de tratar essa parte da análise como um componente separado está no fato de que um analisador léxico dedicado é tipicamente muito eficiente e de maior simplicidade de implementação do que o analisador sintático, que não precisa, por exemplo, tratar comentários e caracteres redundantes.

O *grView* utiliza um componente externo para realizar a análise léxica, o JFlex (ver seção 3.1), embora ofereça diversas funcionalidades para facilitar e tornar mais célere a configuração do JFlex.

2.3. Análise Sintática e Análise de Contexto

Em termos gerais, a tarefa de um analisador sintático é, dada uma seqüência de itens léxicos de um código fonte, definir de que forma nessa seqüencia podem ser

reconhecidas estruturas sintáticas estabelecidas pelas regras definidas pelas produções de uma gramática. Adicionalmente, espera-se que no caso de erros um analisador sintático emita mensagens de depuração de modo inteligível ao usuário e siga estratégias de correção para os erros, podendo assim continuar com a análise. Alternativamente, pode ser construída uma árvore sintática decorada, na qual os nós são os elementos não-terminais da gramática e as folhas são os terminais; a ambos podem ser agregados dados semânticos.

O analisador de contexto é responsável pela atribuição um significado semântico a cada item sintático. Essa representação permitirá tarefas relacionadas à interpretação e geração de código, constituindo assim a fase de síntese.

Os analisadores sintáticos, no contexto relevante a esse trabalho, geram representações sem ambigüidades e podem ser separados em dois tipos que identificam o modo pelo qual geram a árvore sintática: descendente (*top-down*), caso a gerem a partir da raiz da gramática, ou ascendente (*bottom-up*), caso partam das folhas. Analisadores descendentes são usados para subclasses de gramáticas LL, ao passo que os ascendentes são mais usados para gramáticas LR; eles serão denominados *Analisadores LL* e *Analisadores LR* nas seções a seguir.

2.3.1. Analisadores LR

Estes são os analisadores mais populares atualmente, implementados em sistemas como o Yacc [21] e Bison [22]. O L refere-se a *Left-to-right scanning*, um indicativo de que o varredura da entrada é feita da esquerda para a direita. O R refere-se à geração *Rightmost*, ou seja, no processo de geração da cadeia de entrada, (atenção, é o contrário da análise) os símbolos não terminais mais a direita são reconhecidos primeiro.

Por exemplo, dada a gramática a seguir:

```
S -> aMNb
M->c
N->d
```

A análise ascendente se dá da na seguinte ordem (para a entrada `acdb`):

```
acdb -> aMdb -> aMNb -> S
```

Ao passo que a geração da mesma seqüência, se dá na seguinte ordem (observe que o não terminal mais a direita, N, é atribuído primeiro):

```
S -> aMNb -> aMdb -> acdb
```

Diz-se que um analisador LR é *LR(**k**)* para informar o número ***k*** símbolos, chamados símbolos de *look-ahead*, que são os símbolos de entrada à direita

verificados antecipadamente pelo analisador a fim de que este possa escolher deterministicamente a próxima ação a tomar (reconhecer um grupo de símbolos, incluindo não terminais, ou ler o próximo item léxico). A mesma convenção de *look-ahead* é aplicável a outros tipos de analisadores sintáticos. Por questão de eficiência *k* é sempre tomado como 1.

Variações comuns e mais eficientes de analisadores LR são os analisadores SLR (*Simple* LR) e LALR (*look-ahead* LR). A última variação trata de uma subclasse de gramáticas LR, e é amplamente utilizada na forma de LALR(1) (tipo de análise usado no Bison e no Yacc), que parte de um autômato LR(0) e adiciona transições LR(1) somente no caso de conflitos na análise.

2.3.2. Analisadores LL

Analisadores sintáticos descendentes, que não precisam realizar *backtracking*, são chamados de analisadores LL(***k***), onde LL se refere a *Left-to-Right scanning* (como no caso de analisadores LR) e geração *Leftmost*, ou seja, no processo de geração da cadeia de entrada, os símbolos não terminais mais a esquerda são reconhecidos primeiro. **k** se refere ao número de símbolos que o analisador precisa ler antecipadamente para tomar decisões, ou *look-ahead.*

Por exemplo, dada a gramática a seguir (dada na seção 2.3.1):

```
S -> aMNb
M -> c
N -> d
```

A análise descendente se dá da na seguinte ordem (para a entrada `acdb`):

```
S -> aMNb -> acNb -> acdb
```

Ao passo que a geração da mesma seqüência, se dá na mesma ordem (observe que o não terminal mais a esquerda, N, é atribuído primeiro):

```
S -> aMNb -> acNb -> acdb
```

As gramáticas aceitas por esses analisadores, em particular gramáticas LL(1), são expressivas o suficiente para cobrir a maior para das linguagens de programação modernas, ainda que se façam necessários alguns cuidados para a especificação: um não terminal não pode gerar esse mesmo não terminal mais à esquerda na produção (como, por exemplo, a produção `M->Ma`). Essa limitação, contudo, é facilmente contornada por meio de uma fatoração das produções que não altera a linguagem gerada pela gramática.

As principais vantagens dos analisadores LL(1) estão na simplicidade de sua implementação, na possibilidade da correção automática de erros sintáticos e na

possibilidade análise de contexto em cada símbolo do lado direito de uma produção. Visando, principalmente, essas últimas vantagens, o ASIN [1] foi desenvolvido sobre um analisador derivado do LL(1), que será apresentado na seção 2.3.3.

2.3.3. O Analisador GSLL(1)

A subclasse das gramáticas LL(1) tratadas pelo analisador implementado originalmente pelo ASIN, e herdado pelo *grView,* será aqui chamada de GSLL(1), ou *Generic Simple* LL(1). Estas gramáticas são inferidas a partir das restrições impostas pelos Grafos Sintáticos Simples (ver a seção 2.4), com exceção de uma restrição a menos: as gramáticas GSLL(1) aceitam alternativas para nós não terminais.

2.4. Grafos Sintáticos Simples

As definições que seguem foram extraídas do Relatório Técnico RT-MAC-9005: "*Simple syntax graphs, their parse with automatic error recovery and an ANSI C simple syntax graph"*, por Valdemar W. Setzer e Roberto C. Mayer [4].

Grafos Sintáticos (GSs), dos quais os GSSs são derivados são definidos como uma quádrupla ordenada $(S,\ Vn,\ V_t,\ P)$, onde Vn e V_t são conjuntos disjuntos, finitos e não-vazios de símbolos *não-terminais* e *terminais*, respectivamente; $S \in V_n$ é o símbolo não-terminal inicial, e P é um conjunto não-vazio de pares ordenados (N, G_n), onde $N \in V_n$ e G_n é o grafo correspondente ao não terminal N, tal que para cada N existe exatamente um par (N, G_n).

Um grafo G_n para o não-terminal N é um grafo conexo, composto por uma quádrupla (k_0, K, E_a, E_s), onde K, E_a e E_s são conjuntos finitos disjuntos, chamados, respectivamente *conjunto de nós*, *conjunto de alternativas* e *conjunto de sucessores*; K é não-vazio e k_0 é chamado de *nó inicial* do grafo para um não-terminal. Cada um dos nós de K representa um símbolo não-terminal, um símbolo terminal, ou um símbolo vazio λ, e podem ser ligados entre si por arestas de dois tipos: *alternativas* e *sucessores*. Alternativas no conjunto E_a são definidas por um par ordenado (n_1, n_2), onde n_1 e n_2 são nós em K e n_2 é a alternativa para n_1. Equivalentemente, sucessores no conjunto E_a são definidos por par ordenado (n_1, n_2) onde n_1 e n_2 são nós em K e n_2 é o nó que sucede n_1.

Desta forma, P define, efetivamente, as produções de uma gramática.

Um breve exemplo de um GS, Gr, que será utilizado adiante:

$$Gr = (A, \{B, C\}, \{a, b, c, d\}, P)$$

$$P = \{(A, G_1), (B, G_2), (C, G_3)\}$$

$$G_1 = (B, \{B, a\}, \{\emptyset\}, \{(B, a)\})$$

$$G_2 = (b, \{C, b, d\}, \{d, C\}, \{b, d\})$$

$$G_3 = (c, \{c, \lambda\}, \{c, \lambda\}, \{\emptyset\})$$

Exemplo 1 – A gramática Gr, representado por um GS

Em notação usual de gramáticas, a gramática Gr poderia ser escrita da seguinte forma:

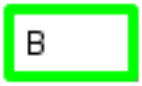
```
A -> Ba
B -> b(d|C)
C -> c| λ
```

2.4.1. *Representação Visual*

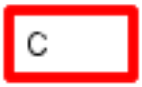
O simbolismo adotado neste trabalho e na implementação que foi feita, para representações visuais de grafos sintáticos é exemplificada a seguir:

Nó do símbolo inicial não terminal da gramática S

Nó não terminal à esquerda A



Nó não terminal B



Nó terminal C

Nó vazio, ou nó λ

Seta para nó sucessor

Seta para nó alternativo

Tabela 2 – Representação visual dos elementos de um GS, ou de um GSS

A representação visual do grafo dado no exemplo 1 seria, portanto:

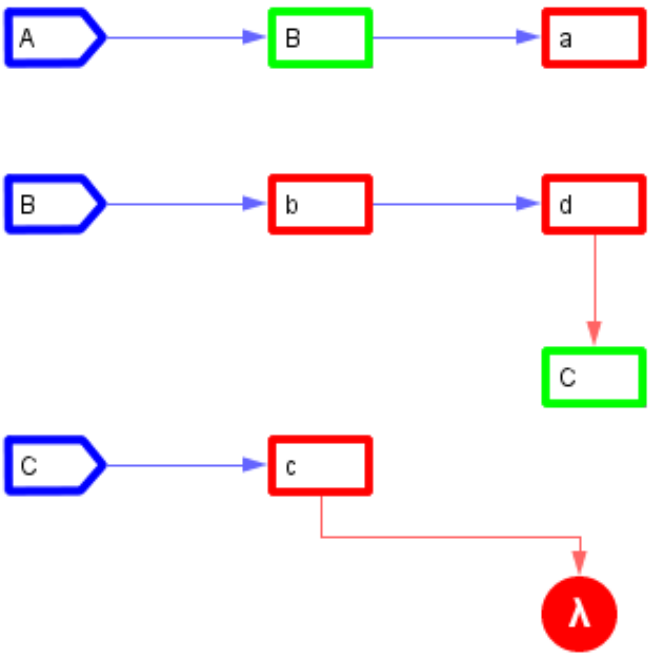


Figura 2 - Representação visual da gramática GR

GSSs aplicam restrições bem definidas aos GSs, de forma a corresponder a gramáticas GSLL(1). A corretude dos GSSs, quando aplicados a gramáticas GSLL(1), depende de apenas duas regras:

1. *Um nó c em um seqüência de alternativas não pode gerar mais à esquerda um símbolo terminal gerado mais à esquerda por qualquer outro nó dessa seqüência de alternativas.*

   São exemplos de GSSs inválidos, de acordo com essa regra, as figuras 3, 4 e 5, a seguir:

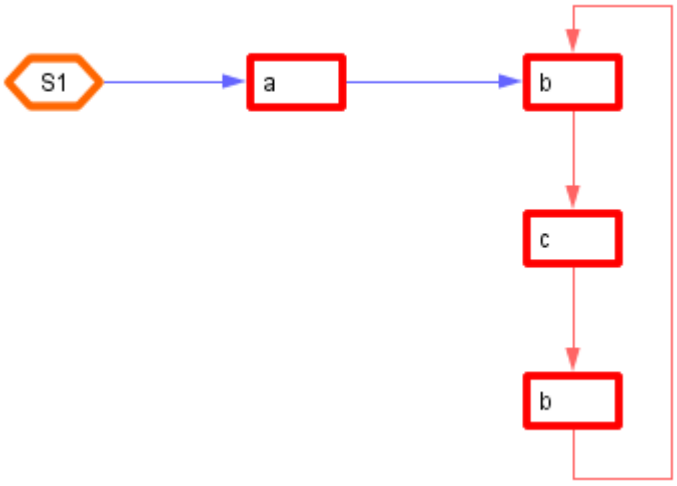


Figura 3 – GSS inválido: o nó terminal b repete-se na seq. de alternativas.

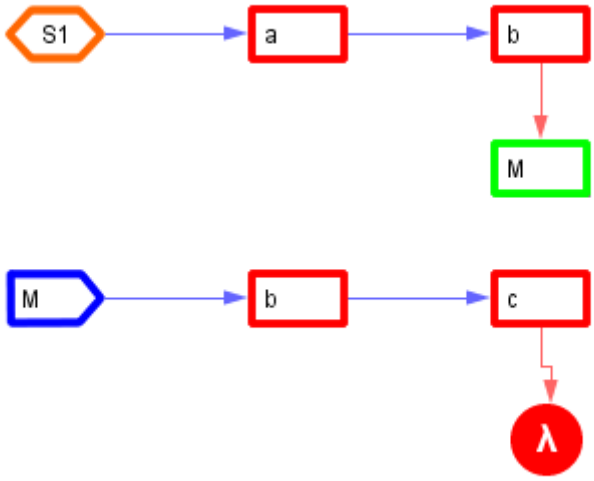


Figura 4 – GSS inválido: O primeiro terminal gerado pelo não-terminal M é um terminal presente em sua seqüência de alternativas

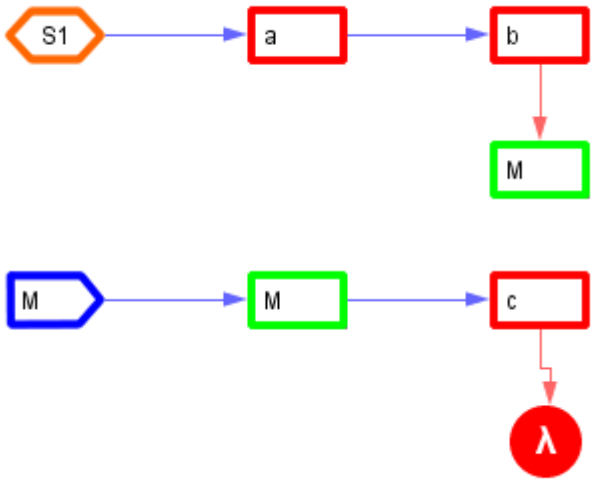


Figura 5 – GSS inválido: o não terminal M produz uma recursão à esquerda que não poderá ser resolvida pelo analisador GSLL(1).

2. *Nós λ não devem possuir alternativas.*

Ao usar gramáticas definidas por GSSs para gerar analisadores sintáticos, temos que as restrições dadas acima são suficientes para garantir que no máximo um símbolo de *look-ahead* é suficiente para o analisador eliminar quaisquer ambigüidades (ou, em outras palavras, nós inatingíveis pelo algoritmo de análise). Estas restrições também permitem que a análise sintática (um algoritmo para GSSs definido formalmente pode ser visto em [4]) seja muito mais simples do que se

gramáticas puramente LL(1) Outras características e vantagens de usar os GSSs incluem:

- As restrições exigidas podem ser verificadas facilmente por verificação visual ou por um programa

- É muito fácil prever o funcionamento do analisador *top-down* a partir do gráfico de um GSS

- Rotinas semânticas podem ser associadas a qualquer nó da produção, diferentemente do que ocorre em analisadores *bottom-up*.

- GSSs são mais fáceis de entender do que grafos com poucas restrições, e portanto são bastante adequadas para se aprender a sintaxe de uma linguagem de programação.

- A expressividade dos GSSs é suficiente para cobrir a definição de linguagens de programação, como exemplificado pela definição de ANSI C em [4].

3. Tecnologias

Parte importante do desafio que foi desenvolver o *grView* estava na seleção de bibliotecas e arcabouços (ou *frameworks*, do inglês) que não apenas fossem poderosos o suficiente para atender aos requisitos de software do programa, mas que também fosse simples para permitir um ciclo de aprendizado curto, e que estivessem disponíveis sob licenças de software livre, preferencialmente GPL [5]. As mais relevantes tecnologias utilizadas são explicadas nas seções seguintes.

3.1. JFlex

JFlex [6] é um gerador de analisadores léxicos, escrito em Java, baseado, em parte, no JLex [7]. Todo processo de análise léxica no *grView* é feito por meio de chamadas ao JFlex. Ele foi adotado por compatibilidade com o código herdado do ASIN, e mostrou-se notavelmente robusto e configurável. Além disso, os arquivos de definições do JFlex, chamados *scanners* mostraram-se muito fáceis de criar e adaptar a diferentes requerimentos, permitindo que os analisadores gerados implementassem qualquer *interface* e portanto pudessem ser usados transparentemente pelo *grView*, que utiliza um esquema de geração dinâmica de analisadores léxicos e precisa que os analisadores gerados tenham alguns métodos de uso interno do programa.

Alguns dos recursos mais interessantes do JFlex, como suas capacidades de depuração e de contagem de colunas (usada para indicar visualmente no código fonte o lugar em que erros foram encontrados) não são utilizados, mas há planos de incorporá-los ao programa, que se beneficiaria de novos auxílios visuais para a depuração de códigos.

A versão do JFlex utilizada é a mais recente na data presente, 1.4.2 e está disponível sob GPL [5].

3.2. Infonode Docking Windows

O *Infonode Docking Windows* [8] (IDW) é um arcabouço para a implementação de *layouts* com abas acopláveis escrito em Java com a biblioteca Swing. O IDW permite a exibição simultânea de múltiplas abas, que podem ser agrupadas em painéis redimensionáveis, selecionadas, movidas de um painel a outro, maximizadas, minimizadas para um painel especial, fechadas ou desacopladas em janelas independentes. Essas características motivaram a escolha desse abas para implementar a funcionalidade de abas acopláveis existente no Eclipse sem ter de extrair e reescrever qualquer código dessa plataforma.

A utilização do IDW transcorreu sem complicações, pois além de haver uma documentação excelente disponível, muito pouco código teve de ser implementado para inserir todos os componentes Swing do *grView* em uma janela de abas criada com o IDW. De fato, a janela de abas em si pode ser implementada em apenas uma ou duas classes e, além do *layout* (*View*), não exigiu a implementação de modelos (*Model*) e controladores (*Controller*).

A versão utilizada no *grView* é a 1.6.1, que está disponível sob GPL [5].

3.3. Netbeans Visual Library

O maior problema nos momentos iniciais do desenvolvimento do *grView* foi definir que API (*Application Programming Inteface*, do inglês) de criação de gráficos bidimensionais poderia ser usada no lugar do GEF [9] (*Graphical Editing Framework*, do inglês), API utilizada no ASIN, uma vez que o GEF também é um *plugin* para o Eclipse e não pareceu ser muito prático para a criação de um aplicativo autônomo. A solução encontrada consistiu em usar a *Netbeans Visual Library* [10] (NVL), uma API integrada à plataforma *Netbeans 6.0* [11], capaz de gerar e visualizar gráficos, mas fortemente orientada à modelagem de grafos. Embora pareça, em princípio, um contra-senso utilizar um componente de outra plataforma, semelhante ao Eclipse, o NVL foi inicialmente projetado como uma API autônoma e é muito simples extraí-lo do Netbeans e usá-lo independentemente; além disso, o NVL, por ser orientado a grafos, mostrou ser a opção perfeita.

O *grView* utiliza extensivamente muitos recursos do NVL, e foi, portanto, necessário entender a fundo essa API.

A versão utilizado no *grView* é o Netbeans Visual Library 2.0, disponível sob licença dual CDDL/GPL [12].

3.4. JEdit

Ao longo do desenvolvimento do *grView* ficou claro que o editor de texto padrão do Swing não oferecia uma experiência satisfatória para edição de códigos para programação do sistema. Como uma parte importante do *grView* -- diferentemente do ASIN -- envolve edição desses códigos, tipicamente em linguagem Java ou similares, foi tomada a decisão de incrementar essa funcionalidade. Obviamente esse editor não faz parte dos analisadores gerados.

A melhor alternativa encontrada foi adaptar ao programa partes do código do *JEdit* [13], um editor de texto voltado para programação, escrito em Java e distribuído sob GPLv.2 [14]. Desta forma foi possível incrementar a edição de textos para incluir não apenas coloração automática de sintaxe, mas também alinhamento vertical automática do texto, casamento de parênteses/colchetes, histórico de ações e muitos outros recursos. O JEdit tem a capacidade de realizar essas funções em

dezenas de linguagem diferentes e, para adicionar novas linguagens, basta editar um arquivo em XML com as definições necessárias.

Um ponto interessante sobre a arquitetura do *grView* é que para facilitar a criação de *plugins* e de macros, todas as ações que se podem realizar no programa são definidas em arquivos de texto. Durante a inicialização do programa esses arquivos são carregados por um interpretador simplificado para uma linguagem de scripts semelhante a Java [15], o *BeanShell* [16]. Este método de definir ações mostrou-se notavelmente flexível e simples, de modo que além de incorporá-lo no *grView* para controlar as ações do novo editor de textos decidiu-se expandir o mesmo método para controlar as ações relativas a diversos outros componentes, mais notadamente as barras de ferramentas e o carregador de rotinas semânticas.

A versão do JEdit utilizada é a 4.3.

3.5. BeanShell

O *BeanShell* [16], é um interpretador para uma linguagem de scripts [15] utilizado extensivamente no *grView* para dois propósitos: definir rotinas curtas de forma desacoplada do código, em arquivos de script, principalmente para definir o efeito de variadas ações; e implementar um console, que não apenas intercepta a saída padrão da máquina virtual, mas também oferece uma ferramenta de depuração e controle fino para desenvolvedores e usuários avançados.

A versão utilizada foi a 2.0b4, distribuída sob licença dual SPL [17] e LGPL [16].

3.6. Outros Componentes e Tecnologias

Outros componentes externos usados incluem o *HTML parser* [17], utilizado para fazer a análise sintática de arquivos HTML, e é usado tanto para implementar a visualização das pilhas, quanto das tabelas intermediárias e da saída padrão.

Dentre as demais tecnologias estudas, o que mais se sobressai são as técnicas de *Serialization* e *Reflection*, implementados pelo próprio *kit* de desenvolvimento para programas escritos em Java (*Sun Java Development Kit*).

Serialization, em termos básicos, consiste em converter instâncias de objetos em representações simples, em texto ou binária, que podem ser salvas ou transferidas para serem posteriormente decodificadas e transformadas em cópias exatas dos objetos instanciados originais. Esta técnica é usada no *grView* principalmente para salvar informações de estado; isso foi necessário para permitir o mecanismo de fazer/desfazer implementado para a modelagem de GSSs.

Reflection é o termo usado para designar a técnica, disponível em Java, para o rastreamento de todos os campos, métodos e elementos constituintes de qualquer

classe ou objeto. É usada de duas maneiras no *grView*: para monitorar, em tempo de execução, qualquer alteração no arquivo em Java do analisador e, caso ele tenha sido modificado, para instanciar a classe definida no arquivo e invocar os métodos necessários do objeto instanciado; também é usado para rastrear os métodos em programas externos e assim gerar um arquivo de modelo para a criação de adaptadores de entrada para o interpretador, recurso que será exemplificado mais à frente.

4. Desenvolvimento

O *grView*, inicialmente pensado como uma simples tradução do ASIN em um aplicativo autônomo em relação ao eclipse, superou muito as expectativas em torno da dimensão do projeto. As primeiras versões, focadas exclusivamente na principal funcionalidade do *grView* (ou seja, a criação visual de GSSs, e a geração de analisadores sintáticos para esses GSSs) foram desenvolvidas com MVC. Dada a natureza fortemente orientada à interface gráfica, existente no projeto, essa decisão pareceu a mais acertada. Entretanto, conforme novos componentes foram adicionados, a utilização de MVC terminou por tornar-se um empecilho. Conforme o desenvolvimento do sistema prosseguiu, dentro do paradigma de MVC, evidenciou-se uma forte tendência a concentrar código demais nos controladores. Esse fator deve-se ao fato de a interface gráfica de usuário possuir, efetivamente, poucos componentes, e ao fato de a utilização de bancos de dados e outras formas de persistência ser muito rara no programa. Ademais, ao passo que novos componentes foram incluídos (em particular a área de edição de código do JEdit), o acoplamento de código aumentou muito.

Por acoplamento alto, entenda-se que os diferentes componentes do programa, cujas funcionalidades não estão objetivamente relacionadas, podiam apresentar uma dependência excessiva entre si, ou seja, trata-se de um problema de modularização. A junção desse acoplamento alto com a utilização indevida de MVC constitui um grande problema de projeto.

A solução desse problema se deu por meio de duas decisões que levaram a uma alteração significava da organização do código e do processo de desenvolvimento.
Isso envolveu a reestruturação de todos os pacotes do código, visando uma divisão lógica mais significativa, e a utilização de técnicas diferentes de desenvolvimento, com injeção de dependência.

O principal objetivo da reestruturação aplicada aos pacotes do projeto foi o de evitar a concentração excessiva de classes que antes existia em pacotes que sugeriam a existência de controladores, ou de tipos específicos de classes aplicáveis a esses controladores (como classes que implementam ações aplicáveis sobre um modelo, e classes que definem estratégias aplicáveis a essas ações). Esta abordagem misturava, por exemplo, classes relativas ao funcionamento da área de desenho e ao funcionamento dos menus dentro do mesmo pacote. Isso, embora não aumentasse, necessariamente, o acoplamento do código dificultava bastante o trabalho de manutenção do mesmo. Portanto, embora tenha mantido válidos alguns preceitos de MVC (como a separação do código que cria os componentes de visualização, do código que controla a lógica de funcionamento desses componentes), optou-se por evitar ao máximo a união de classes de funcionalidade pouco relacionadas dentro de um mesmo pacote. Isto é, em termos da organização dos pacotes a utilização do paradigma de MVC deixou de ser uma preocupação.

A seguir é exemplificada a divisão em pacotes do projeto, (vários pacotes são excluídos, para efeito de brevidade):

Pacote	Descrição
org.grview.actions	Modela ações abstratas utilizadas internamente
org.grview.bsh.*	Pacotes que contém código para interpretação de BeanShell
org.grview.canvas.*	Contém o Canvas, que implementa a área de desenho, e classes relacionadas
org.grview.editor.*	Contém as classes relacionadas à implementação do editor de texto herdado do JEdit.
org.grview.lexical	Contém as classes para o analisador léxico.
org.grview.project.*	Contém os pacotes para as classes que implementam a criação de projetos.
org.grview.semantics	Contém as classes permitir a programação de rotinas semânticas.
org.grview.syntax	Contém as classes com o analisador sintático e diversas classes para suporte e para geração de código.
org.grview.ui.*	Contém as Classes responsáveis pelos elementos de interface gráfica.

5. Resultados

O *grView* já é utilizável em seu estado atual. Uma versão em estado *alpha*, portanto, está atualmente disponibilizada no site da disciplina. Esta versão já dispõe de um bom conjunto de funcionalidades, incluindo todas as fundamentais, como a modelagem de GSSs, geração de analisadores GSLL(1), impressão de informações para rastreamento da análise sintática, acompanhamento visual do processo de análise sintática, geração de código Java para exportação do analisador, impressão de relatórios e gerenciamento de projeto.

A seguir será exemplificado, de forma resumida, o uso do *grView*. Uma documentação mais completa estará disponível em:

```
https://sourceforge.net/projects/grview
```

5.1. Instalação

Para executar o *grView* é preciso ter disponível uma máquina virtual Java, versão 1.6.0, ou superior. Não é necessário instalação, o programa é disponibilizado na forma de um arquivo no formato *jar*, de forma que para executá-lo basta utilizar o comando:

```
java –jar grview.jar
```

5.2. Início

Uma vez executado o comando anterior, um selecionador de área de trabalho será exibido (figura 6). É possível indicar uma área de trabalho pré-existente, para continuar com um projeto salvo, ou indicar um diretório inexistente para criar um novo projeto.

Figura 6 - Tela para seleção de área de trabalho.

O *grView* utiliza, em várias tarefas, um compilador de Java e, portanto, uma caixa de diálogo pode ser exibida para que o caminho para o arquivo *tools.jar* seja informado. É preciso que o arquivo pertença a um *JDK* (*Java Development Kit*) versão 1.5.0, ou superior.

Em seguida é aberta a janela principal do programa, com o projeto selecionado (novo ou pré-existente) carregado. No caso de um novo projeto é exibida uma janela como na figura 7.

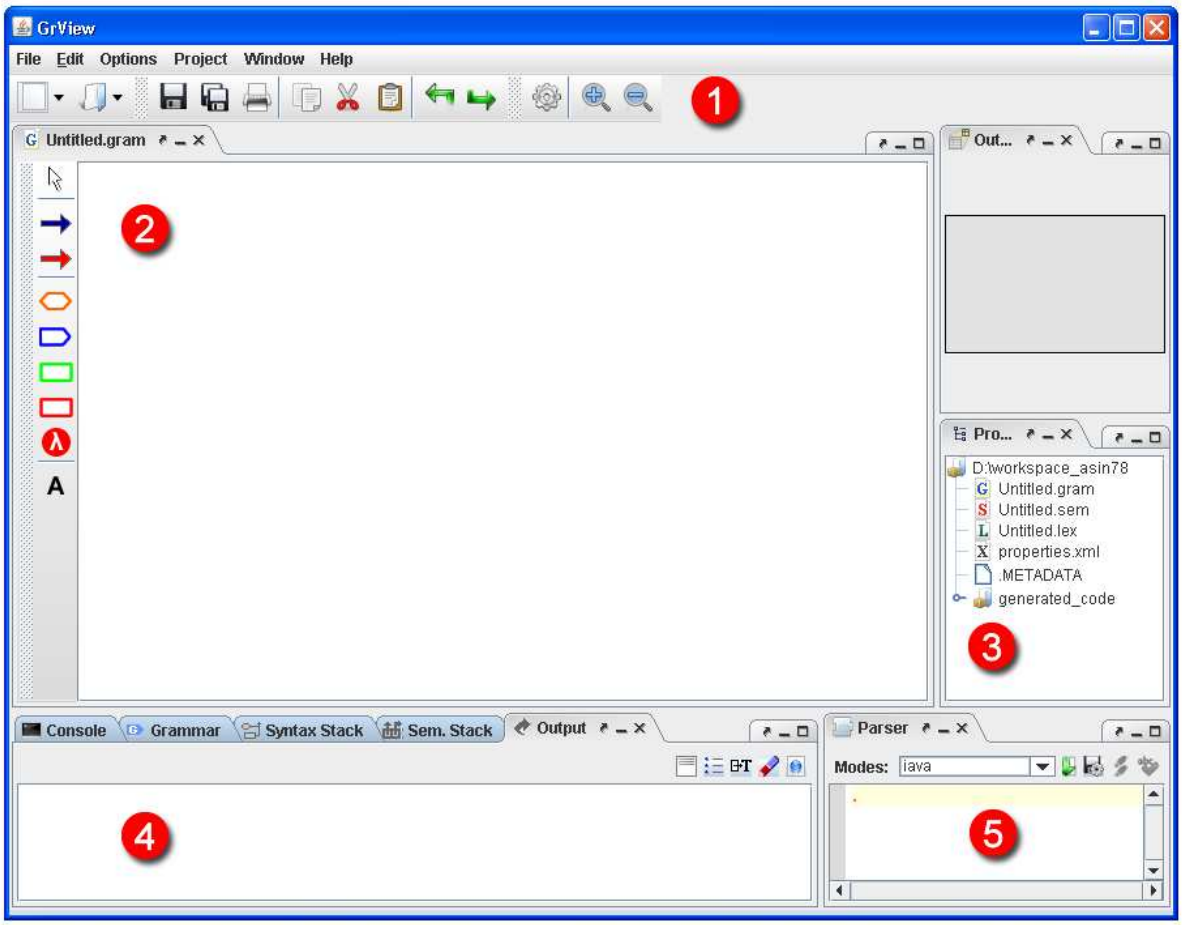


Figura 7 - Janela inicial de um novo projeto.

1. *Barra de ferramentas principal; a função das ferramentas nesta barra pode mudar de acordo com a aba selecionada.*

2. *Área de modelagem de GSSs, também referida como Área de trabalho, ou Área de desenho.*

3. *Navegador de arquivos do projeto, ou simplesmente Gerenciador de Projeto. É possível abrir e alternar o nome dos arquivos aqui presentes, embora apenas arquivos supérfluos possam ser removidos.*

4. *Conjunto de abas de saída. Possui a saída principal e abas para exibição das pilhas (semântica e sintática) e das tabelas intermediárias. Um console também está presente neste conjunto.*

5. *Interpretador. Nesta área é possível escrever e carregar arquivos contendo expressões. Controles para controlar o processo de análise estão presentes nesta área também.*

5.3. Modelagem do GSS

Para iniciar a modelagem do GSS seleciona-se a aba com o arquivo de extensão *.gram* aberto, ou é possível abri-lo a partir do gerenciador de projeto; em um novo projeto este será o arquivo "Untitled.gram". As ferramentas de modelagem (figura 8) são exibidas à esquerda da área de desenho, e separadas em quatro áreas: seleção, criação de arestas de conexão entre nós, inserção de nós e ferramentas auxiliares.

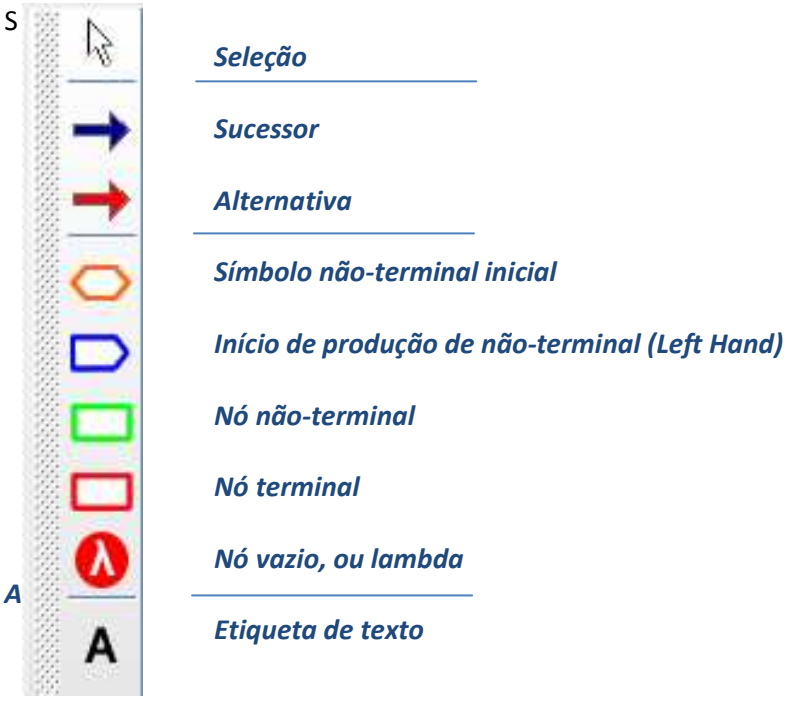


Figura 8 – Ferramentas de modelagem.

Ao selecionar uma ferramenta de criação de nós, o formato do cursor irá mudar quando estiver sobre a área de desenho; basta acionar o botão esquerdo do mouse para inserir um nó to tipo selecionado. Para criar as arestas de conexão, seleciona-se uma das ferramentas de criação de arestas disponível e arresta-se o mouse, com o botão esquerdo pressionado do nó de origem até o nó de destino da ligação.

A ferramenta para criação de etiquetas de texto permite que pequenos textos sejam inseridos em qualquer parte da área de desenho. Eles em nada interferem na definição do GSS e são ignorados na geração do analisador.

Uma vez criado um nó, o símbolo a ele associado pode ser editado a partir de um clique duplo do botão esquerdo sobre o nó. Além disso, ao pressionar o botão direito sobre um nó selecionado, surge um menu contextual com opções para copiar, colar, cortar e apagar o nó (figura 9). Também é possível usar a funcionalidade de fazer e refazer ações. (Estas opções também estão presentes na barra de ferramentas principal, como exibido na figura 10).

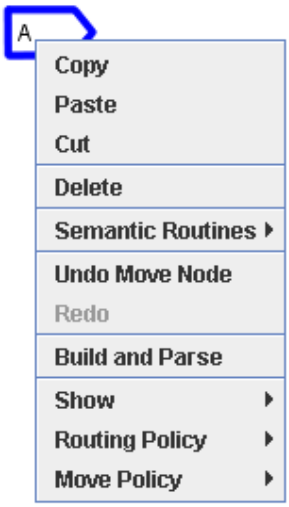


Figura 9 – Menu contextual para um nó selecionado.

Figura 10 – Ferramentas para copiar, recortar e colar nós; e para desfazer e refazer ações.

5.4. Visualização do GSS

O *grView* dispõe de algumas funcionalidades, ferramentas e opções de modos de exibição para facilitar a visualização dos GSSs criados.

Uma visualização panorâmica pode ser encontrada na aba *outline* (figura 11). Toda a área de trabalho é exibida simultaneamente, em uma escala menor, e todos os elementos selecionados podem ser vistos como tal. O quadrado cinza representa o espaço visível da área de desenho e arrastá-lo também altera o espaço visível.

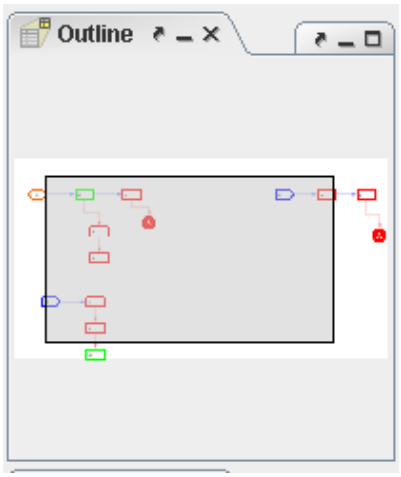


Figura 11 – Aba outline, observada em detalhe, exibindo um GSS.

A área de desenho também pode ser exibida em diferentes escalas por meio das ferramentas de Zoom presentes na barra de ferramentas principal (figura 12), ou mantendo a tecla *Ctrl* pressionada e girando a roda do mouse sobre a área.

Figura 12 – Ferramenta para aumentar e diminuir Zoom.

Os nós e arestas na área de desenho podem ser selecionados individualmente pressionando-se o botão esquerdo do mouse sobre qualquer um deles, ou coletivamente, arrastando-se o mouse, também com o botão esquerdo pressionado, sobre um grupo de nós e arestas. O resultado dessa ação é mostrado na figura 13.

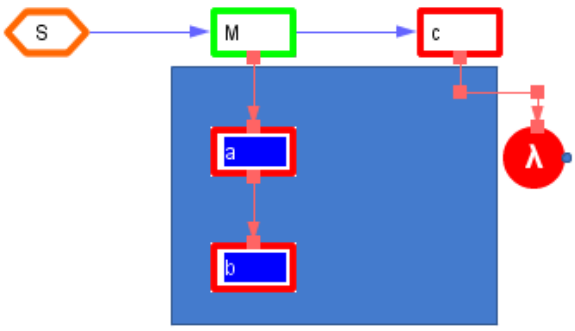


Figura 13 – Seleção simultânea de vários nós.

Uma vez selecionado qualquer número de nós é possível arrastá-los com o botão esquerdo do mouse pressionado; as arestas ligadas a eles os seguirão. Quando selecionadas, as arestas exibem dois ou mais pontos de controle, representados por

pequenos quadrados, como pode ser visto na figura acima. Estes pontos de controle podem ser arrastados para modificar manualmente o caminho da aresta. Para criar novos pontos de controle, basta dar um clique duplo sobre a aresta no local desejado. Os pontos de controle podem ser vistos na figura 14.

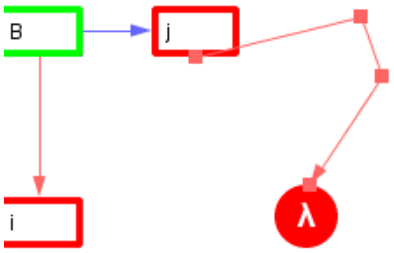


Figura 14 – Aresta com caminho editado. Os pontos de controle são os quadrados vermelhos.

Por meio do menu contextual da área de trabalho (*Acionar botão direito -> Move Policy*) é possível alterar a o comportamento dos nós ao serem arrastados. Para tanto se escolhe uma das opções abaixo:

- *Snap to grid*: Os nós se moverão em incrementos determinados pelo tamanho de uma grade exibida ao fundo da área de desenho.

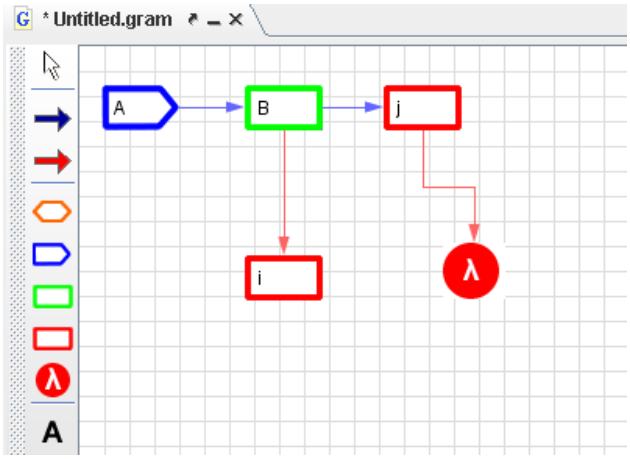


Figura 15 – Área de desenho no modo *Snap to grid*.

- *Auto align:* Os nós se alinharam automaticamente, verticalmente e horizontalmente, com os nós mais próximos.

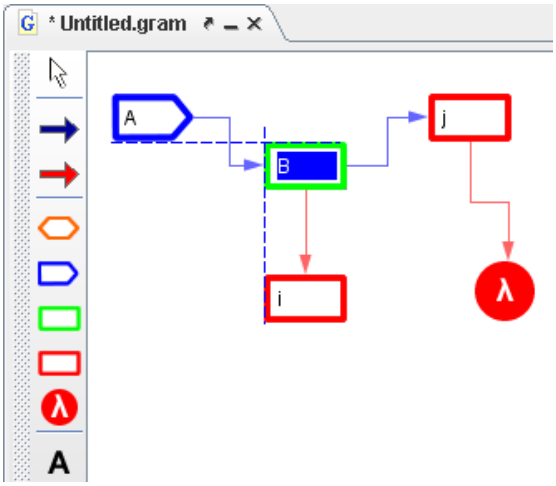


Figura 16 – Área de desenho no modo *Auto Align*. As semi-retas pontilhadas surgem quando dois nós estão alinhados.

- *Snap to lines:* Uma grade de linhas numeradas é exibida ao fundo da área de trabalho. Os nós só podem ser movidos de linha em linha.

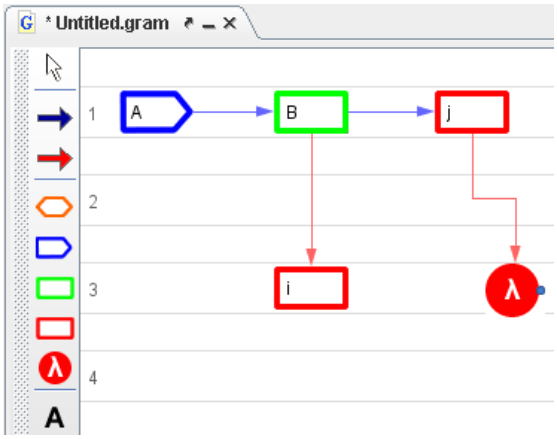


Figura 17 – Área de desenho no modo *Snap to Lines*.

- *Free Move:* Não são feitas restrições ao movimento.

O modo de exibição das arestas também pode ser modificado por meio do menu contextual na área de trabalho (*Acionar botão direito -> Routing Policy*). As opções possíveis são listadas abaixo:

- *Direct:* Os nós são ligados diretamente. Cada aresta possui apenas dois pontos de controle: um no nó de origem, e outro no nó de destino.

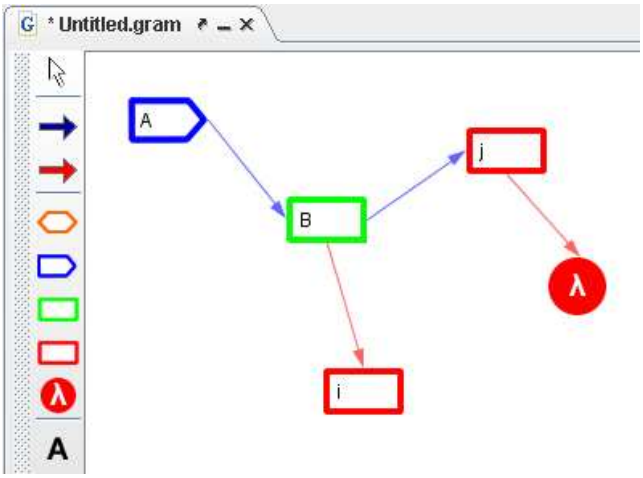


Figura 18 – Área de trabalho com modo de exibição de arestas *Direct*.

- *Free:* Similar ao *Direct*, mas quando mais pontos de controle são adicionados eles permanecem fixos nos pontos em que foram criados, a menos que os próprios pontos sejam movidos.

- *Orthogonal:* As arestas seguem apenas por direções ortogonais e o caminho escolhido é sempre o que possui o menor número possível de cantos sem sobrepor nenhum outro nó. (*As ilustrações dos GSSs neste trabalho utilizam sempre este tipo de modo de exibição, a menos quando indicado*).

5.5. Configuração do Analisador Léxico

O arquivo de configuração do analisador léxico pode ser configurado a qualquer momento. A sintaxe desse arquivo está documentada em [6], mas é importante ressaltar que os trechos de código em linguagem Java são de uso interno do *grView* e não podem ser modificados de forma alguma.

O arquivo de definições do analisador pode ser aberto por meio do gerenciador de projeto, e deve sempre possuir uma extensão *.lex*; quando um novo projeto é criado esse arquivo é chamado “Untitled.lex”. Será exibido então um editor de textos com o código do analisador léxico. Para salvar as modificações pressione *Ctrl-S* ou utilize a ferramenta *Save* na barra de ferramentas principal. Uma vez salvo, o arquivo é repassado para o JFlex que cria imediatamente o código em Java para o analisador léxico. O arquivo com esse código é chamado “Yylex.java”, e é colocado dentro da pasta “generated_code”, que pode ser aberta no gerenciador de projeto.

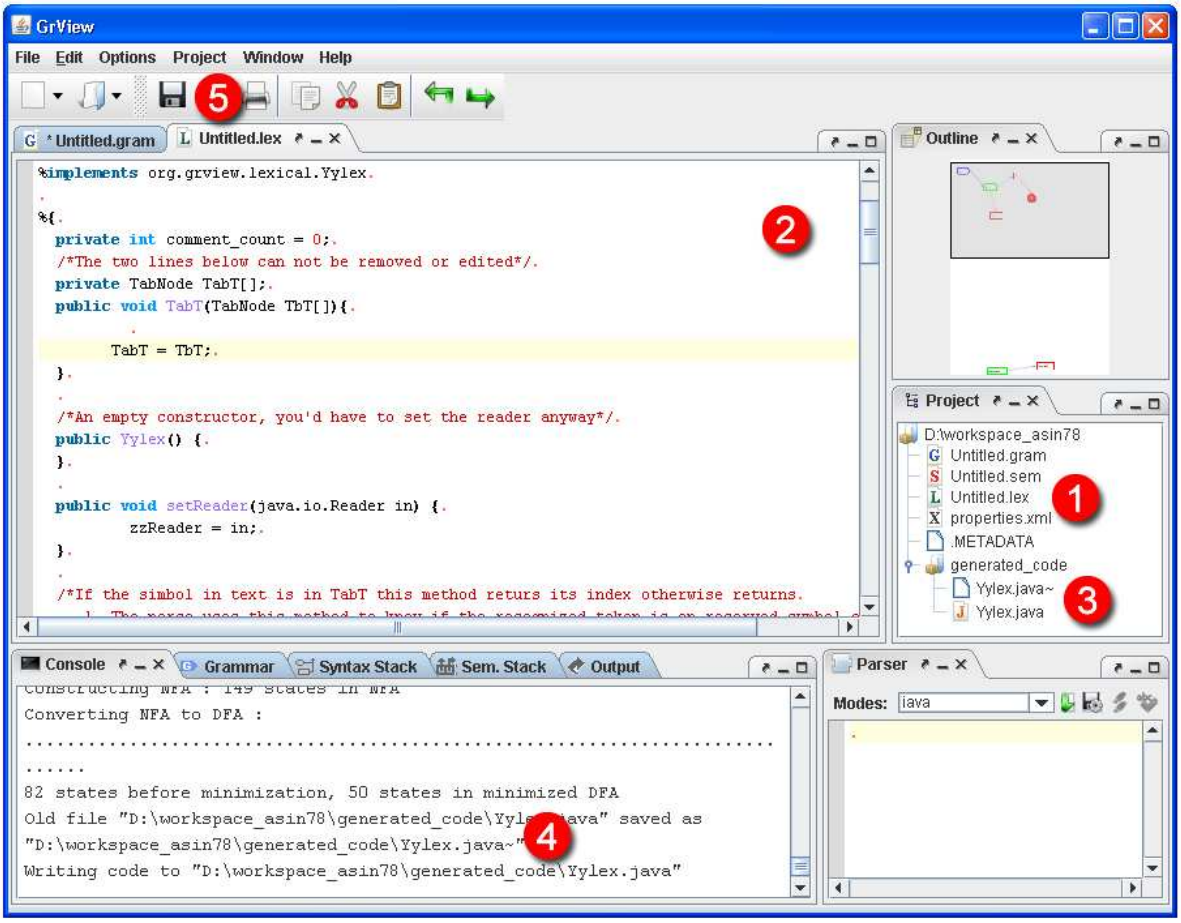


Figura 19 – Janela principal do GrView, com o arquivo de definições do analisador léxico aberto.

1. *Arquivo de definições do analisador léxico.*

2. *Editor de texto, e arquivo do analisador aberto. O esquema de coloração de sintaxe e alinhamento vertical do editor adapta-se melhor aos trechos de código em Java.*

3. *Arquivos do analisador léxico, em Java, na pasta "generated_code".*

4. *Console com mensagens de diagnóstico produzidas pelo JFlex.*

5. *Ferramenta para salvar arquivos.*

5.6. Programação das Rotinas Semânticas

Uma referência a uma rotina semântica (para análise de contexto e geração do código objeto, completando a compilação produzida com o analisador GSSL(1)) podem ser inseridas em qualquer nó do GSS por meio do *grView*, embora o analisador GSLL(1) ignore as rotinas presentes em cabeçalhos de produção (*nó do*

símbolo inicial da gramática e nós inicial do não-terminais à esquerda): o fato de ser possível inseri-las em cabeçalhos deve-se à possibilidade de algum analisador diferente levar em consideração as rotinas semânticas nesses nós. Para inserir uma rotina semântica basta abrir o menu contextual sobre um nó e selecionar "*Semantic Routine -> Create New...*" (figura 20) para criar uma nova rotina, ou selecionar na lista mostrada uma das rotinas semânticas previamente criadas. Após esse procedimento o nome da rotina será indicado ao lado do símbolo do nó. Alternativamente, também é possível indicar a rotina semântica pelo próprio símbolo do nó. Para isso é necessário alterar seu símbolo anexando, logo em seguida ao símbolo existente do nó, o caractere # e o nome da rotina semântica desejada diretamente após o nome do nó. Assim, por exemplo, se um nó tem o símbolo "B", e deseja-se utilizar a rotina semântica "*semantic_routine1*", basta alterá-lo para "B#semantic_routine1" (figura 20, em verde).

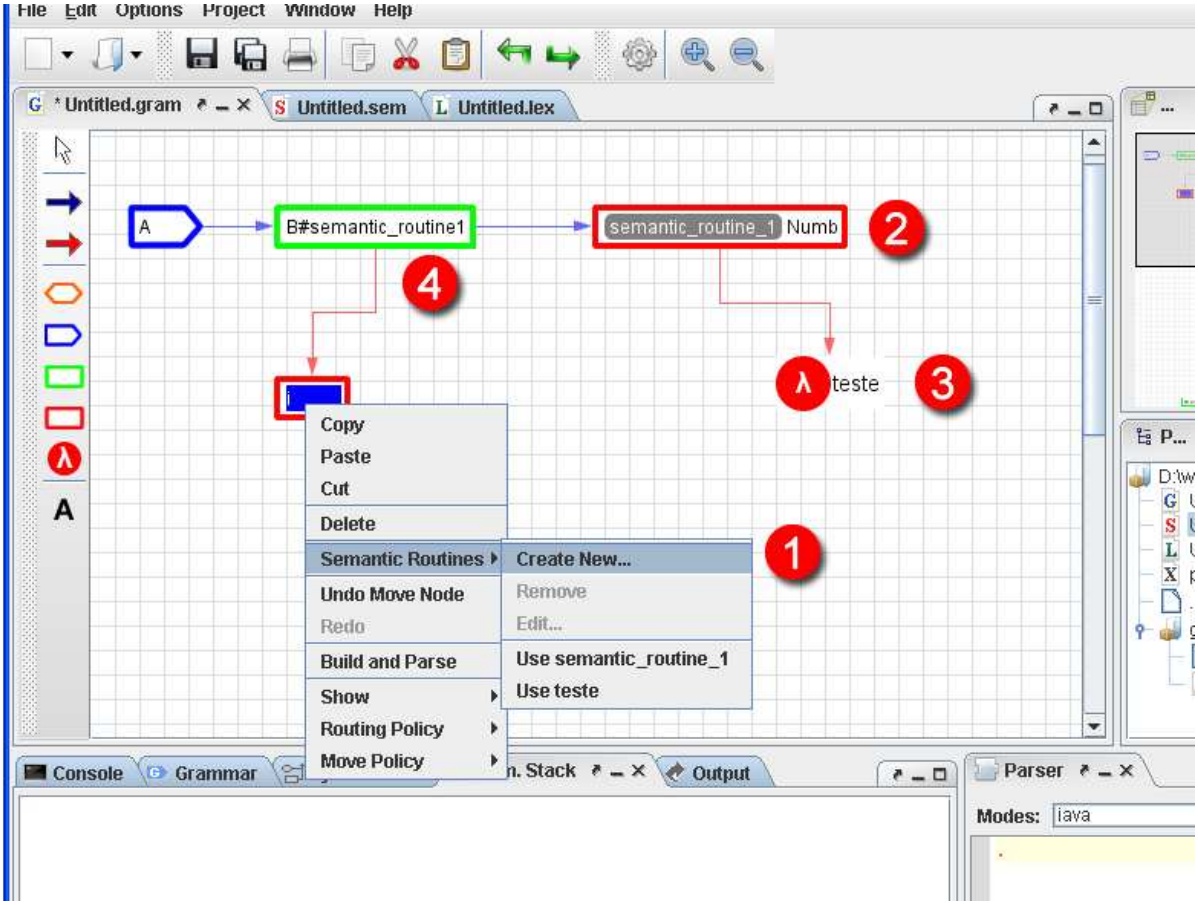


Figura 20 – Nós com rotinas semânticas.

1. *Criação de uma nova rotina semântica para o nó selecionado (na figura 20, em azul).*

2. *Um nó, rotulado* "Numb" *com a rotina semântica* "semantic_routine_1".

3. *Um nó com a mesma rotina, inserida da forma alternativa.*

4. *Um nó lambda com a rotina* "teste"*.*

Ao criar uma nova rotina semântica pelo método descrito acima será exibida uma caixa de diálogo onde se deverá informar o nome da nova rotina e inserir o código correspondente. O código para rotinas semânticas deve ser escrito em Groovy [23]. Algumas variáveis especiais são disponibilizadas para imprimir resultados, bem como ler e modificar a tabela sintática:

- *parseStack* – Cada elemento dessa pilha contém dois campos: um símbolo sintático (um terminal ou não terminal), ou melhor, um ponteiro para o nó do grafo, onde está o símbolo, e dados semânticos (ponteiro para a tabela de símbolos do compilador, um rótulo ou ponteiro para uma tabela de rótulos para desvios, etc.). Este objeto é uma instância de *java.util.Stack,* onde cada elemento é uma instância de *org.grView.syntax.model.ParseStackNode.*

- *tabT* – Um vetor com os símbolos de todos os terminais. Neste vetor os índices dos nós são correspondentes aos índices em *parseStack.*

- *output* – Esta variável aponta para a instância de um objeto utilizado para imprimir resultados. Os principais métodos associados a este objeto são: *println(String)* e *DisplayTextExt(String),* usados para imprimir resultados no interpretador e no campo de saída padrão, respectivamente.

Mais detalhes sobre a criação de rotinas semânticas podem ser encontrados em [20].

Uma vez criadas as rotinas semânticas, elas são armazenadas no diretório do projeto em um arquivo de extensão *.sem*; "Untitled.sem" para novos projetos. Este é escrito na linguagem Groovy. O código de cada rotina é inserido no campo indicado pelos comentários no arquivos. O exemplo abaixo mostra a função teste escrita nesse arquivo.

```
import org.grview.syntax.model.ParseStackNode

/*------ SEMANTIC ROUTINES ESPECIFICATION -----*/
/* you can modify the lines bellow */
  void teste() {
       println "hello world"
  }

/* do not modify the lines bellow */
/*------ SELF GENERATED METHODS ---------------*/
this
```

Novas rotinas podem ser inseridas diretamente neste arquivo, desde que respeitem todas as restrições da sintaxe. Para tanto basta abrir normalmente o arquivo a partir do gerenciador de arquivos e salvá-lo após a edição.

5.7. Geração do Analisador Sintático

O analisador sintático pode ser gerado, a partir do menu contextual obtido ao pressionar-se o botão direito do mouse em qualquer ponto da área de trabalho, ou por meio da ferramenta *Build* na barra de ferramentas principal. Deve-se então selecionar a opção *Build and Parse*, ou *Build and Export*. A diferença entre as duas opções que esta última, além de gerar o analisador, também exporta o código em Java correspondente ao analisador gerado. Esse código gerado é colocado na pasta *export_code,* e pode ser integrado em outros dispositivos ou extraído e utilizado como um programa de linha de comando autônomo.

Uma vez terminada a criação do analisador, o interpretador presente na aba *Parser*, pode ser usado para se interagir com o analisador. Este interpretador é disponibilizado escolhendo-se qualquer uma das opções disponíveis no menu contextual supracitado.

Caso tenha-se optado por exportar o código do analisador e utilizá-lo fora do *grView*, é possível utilizar um interpretador em modo interativo a partir de um console.

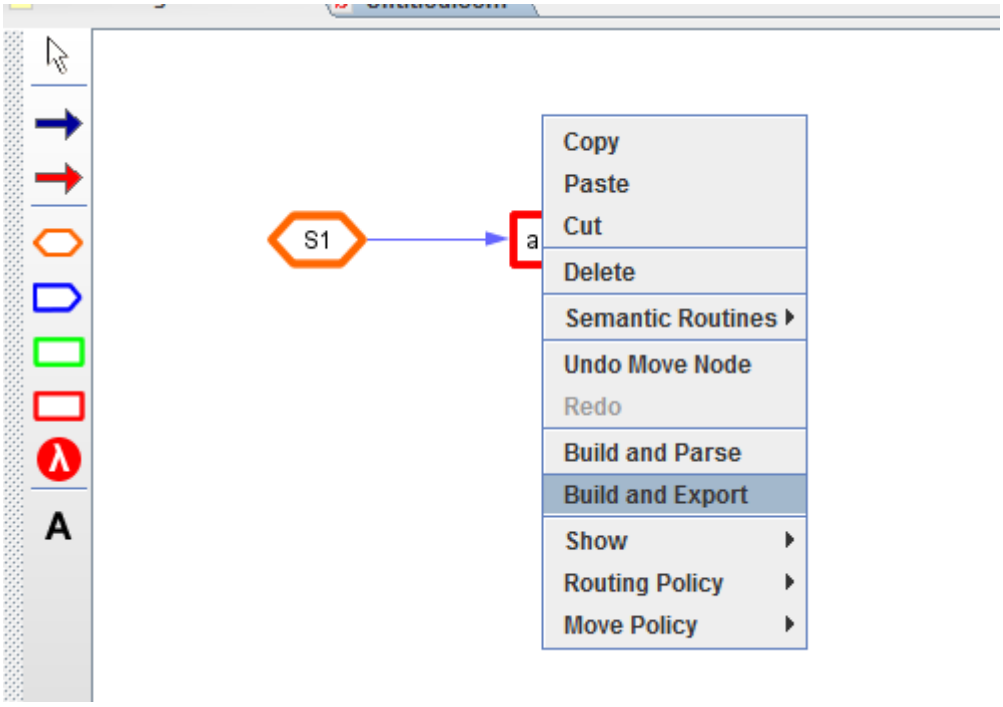


Figura 21 – Opção *Build and Export* em destaque no menu contextual.

5.8. Exemplo completo - Calculadora

Nesta seção será mostrada brevemente a implementação de uma calculadora simples. A implementação no ASIN dessa calculadora pode ser vista em [1]. O grafo sintático criado no *grView* é exibido na figura 22:

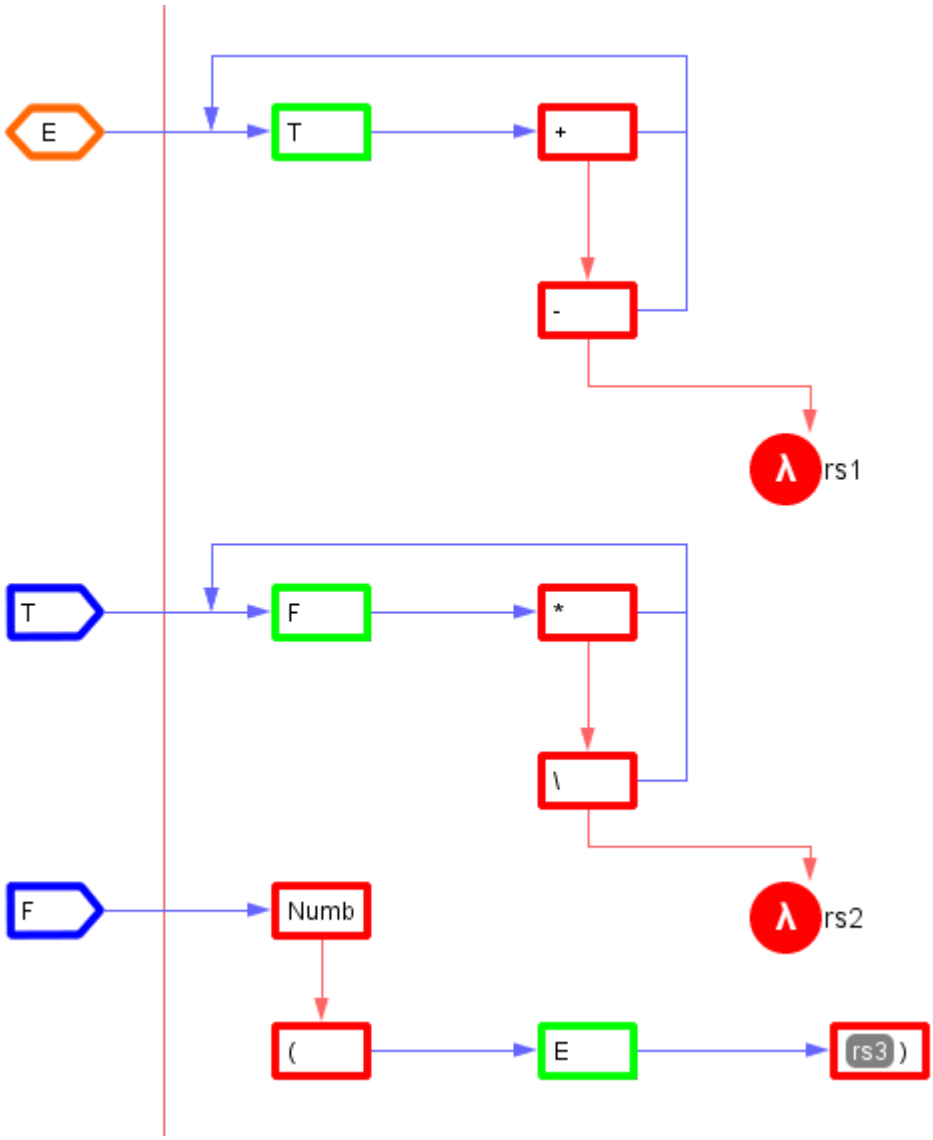


Figura 22 – Grafo para a gramática de uma calculadora simples.

As rotinas semânticas `rs1` e `rs2`, são utilizadas para implementar as operações de soma, subtração, divisão e multiplicação. A rotina `rs1` é mostrada a seguir:

```
void rs1() {
      ParseStackNode aux
      int index = parseStack.size() - 1 //top
      int acumulator
      aux = (ParseStackNode)parseStack.elementAt(index)
      acumulator = aux.intSem()
      index--
      while (index > 0){
            aux = (ParseStackNode)parseStack.elementAt(index)
            if (aux.getSyn().equals("+")){
```

```
                        index--
                        aux = (ParseStackNode)parseStack.elementAt(index)
                        acumulator = acumulator + aux.intSem()
                        index--
                  }
                  else if (aux.getSyn().equals("-")){
                        index--
                        aux = (ParseStackNode)parseStack.elementAt(index)
                        acumulator = aux.intSem() - acumulator
                        index--
                  } else {
                        break
                  }
            }
            /* Stores the final result assigned to the non-terminal T into
the first parse and semantic stack cell of T;
             * this way, the semantic stack will show the partial and final
results. */
            ((ParseStackNode)
parseStack.elementAt(++index)).setSem(acumulator+"")
            if (index <= 0){
                  output.println("Result: "+acumulator)

            }
      }
```

Note que variáveis auxiliares do tipo `ParseStackNode` podem ser usadas para guardar valores da pilha de símbolos terminais.

6. Referências

1. Pereira, L. F. dos Santos. ***ASIN User Manual***, 2004.
   URL: http://Eclipse.ime.usp.br/projetos/grad/DocAsin/index.html

2. Eclipse. Ambiente de desenvolvimento para a plataforma Java.
   URL: http://www.Eclipse.org

3. Aho, Alfred V. ***Compilers, principles, techniques, and tools***, 2ª ed., Boston, MA, USA; Pearson/Addison Wesley, 2006

4. Setzer, V. W. e Mayer R. C. Simple Syntax Graphs, their parse with automatic error recovery and an ANSI C simple sintax graph. ***Technical Report RT-MAC-9005***, Dept. of Computer Science, Institute of Mathematics and Statistics, University of São Paulo, Brazil, 1990.

5. GPL – *GNU General Public License*
   URL: http://www.gnu.org/licenses/gpl.html

6. JFlex – Gerador de analisadores léxicos para linguagem Java, escrito em Java.
   URL: http://jflex.de

7. JLex – Gerador de analisadores léxicos em Java.
   URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/

8. Infonode Docking Windows – Arcabouço para a criação de janelas com abas acopláveis em Swing.
   URL: http://sourceforge.net/projects/infonode/files/

9. Graphical Editing Framework (GEF). – *Plugin* do Eclipse para a modelagem de gráficos.
   URL: http://www.Eclipse.org/gef/

10. Netbeans Visual Library – Biblioteca para criação de gráfica, enfocada na modelagem de grafos.
    URL: http://platform.netbeans.org/graph

11. Netbeans 6.0 – Ambiente de desenvolvimento para a linguagem Java.
    URL: http://netbeans.org/

12. Licença CDDL/GPL
    URL: http://www.netbeans.info/downloads/licence/nb-6.8-beta-2009-10-22-license.txt

13. JEdit – Um editor de textos para programação, com muitos *plugins* e macros, escrito em Java.
URL: http://www.jedit.org/

14. GPLv.2 – *GNU General Public License version 2*
URL: http://www.gnu.org/licenses/old-licenses/gpl-2.0.html

15. BeanShell, linguagem de *script*s.
URL: http://jcp.org/en/jsr/detail?id=274

16. BeanShell – um interpretador escrito em java para linguagem *BeanShell* [14].
URL: http://www.beanshell.org/

17. SPL – *Sun Public License*
URL: http://www.opensource.org/licenses/sunpublic.php

18. LGPL – *Lesser GNU General Public License*
URL: http://www.gnu.org/copyleft/lesser.html

19. HTML parser – Biblioteca em java para a leitura e análise de arquivos HTML.
URL: http://htmlparser.sourceforge.net/

20. Manual de usuário do *GrView*
URL: http://www.linux.ime.usp.br/~henrick/

21. Yacc – *Yet another compiler compiler*
URL: http://dinosaur.compilertools.net/yacc/

22. Bison – *Gnu Parser Generator*
URL: http://www.gnu.com/software/bison

23. Groovy – Linguagem dinâmica de programação para máquina virtual de Java.
URL: http://groovy.codehaus.org/

7. Parte Subjetiva

Sem sombras de dúvida que a questão mais desafiadora durante o desenvolvimento deste trabalho foi a quantidade de tempo exigido pelo trabalho de programação, em si. O tamanho do programa não apenas se escalou além do previsto inicialmente, como também emergiram uma série de dificuldades imprevistas, provenientes, em sua maioria, de particularidades técnicas das muitas bibliotecas e arcabouços utilizados. Em especial, tive muitos problemas em utilizar a *Visual Library* do Netbeans, que apesar de bem documentada e programada (e de ser de longe a melhor opção que encontrei para a criação da área de desenho no *grView*) não era flexível o suficiente para suportar naturalmente todas características necessárias ao *grView.* Desta forma foi necessário adaptar boa parte da API, especialmente a parte de tratamento de ações e os modelos de componentes visuais (como nós e arestas).

Embora a estrutura do código do programa tenha sido bem pensada de antemão os requerimentos mudaram bastante ao longo do ciclo de desenvolvimento. Decisões de projeto posteriores, como a inclusão de partes do JEdit, e mudança do sistema de ações levantaram muitos problemas, que se mostraram de difícil resolução. Mas, apesar de o código do programa certamente beneficiar-se de algumas refatorações no futuro, ele é, na verdade, bem dividido em componentes, aplica padrões de projeto bem reconhecidos e é relativamente fácil de compreender e desenvolver.

Minha maior frustração foi a de não ter tido tempo suficiente de implementar todos os recursos não essenciais que antevia para o *grView*, e nem tão pouco ter feito um processo de depuração completo e detalhista, que ainda vai demandar, no mínimo, alguns meses de trabalho. Na verdade, por conta dessa minha frustração, eu pretendo continuar o desenvolvimento pelo menos até o lançamento de uma versão estável, que conte com todos esses recursos adicionais.

Dentre os conceitos estudados nas disciplinas do curso, os mais importantes certamente são: gerenciamento e desenvolvimento de projetos de pequeno e médio porte, estudo em engenharia de software; basicamente todos os conceitos estudados sobre teoria dos autômatos; padrões de orientação a objetos na disciplina de programação orientada a objetos; analisadores sintáticos e léxicos em laboratório de programação; metodologias de programação, e criação de interpretadores, na matéria de conceitos de programação; além de programação concorrente e estrutura de dados.

Uma lista das matérias mais relevantes:

- MAC0323 Estruturas de Dados.
- MAC0211 Laboratório de Programação I e MAC0212 Laboratório de programação 2.
- MAC0332 Engenharia de Software.
- MAC0438 Programação Concorrente.
- MAC0441 Programação Orientada a Objetos:
- MAC0414 Linguagens Formais e Autômatos