PAMOJA: A Component Framework for Grammar-Aware Engineering

PAMOJA User's Manual

By Jackline Ssanyu

March 25, 2023

Contents

1. Introduction **3**
1.1. The PAMOJA Framework . . . 3
1.2. Design goals . . . 4
1.3. About this manual . . . 4

2. Installing PAMOJA **4**
2.1. Installing PAMOJA Components . . . 5
2.2. Installing the Wizards . . . 8
2.3. Uninstalling PAMOJA . . . 8
2.3.1. Components . . . 8
2.3.2. Wizards . . . 9

3. Building Software Tools with PAMOJA **9**

4. Building a Language Front-end using PAMOJA **9**
4.1. Main Steps . . . 10
4.2. Example:Developing a Front-end for GCLSharp . . . 10

5. Building a Tool for Visualizing Parsers **17**
5.1. The Parser Visualizer Tool . . . 17
5.2. Required Components . . . 17
5.3. Designing the GUI . . . 19
5.4. Adding functionality . . . 21
5.5. Replacing a Parser . . . 22

6. Bugs and Deficiencies **24**

7. Copying and Licence **24**

Appendix A. PAMOJA Components and Wizards **25**

Appendix B. Simple Expression Grammar **26**

Appendix C. Signature for GCLSharp **26**

Appendix D. AOLO grammar specifications for GCLsharp **27**

List of Figures

1. Window displaying PAMOJA components that can be added to the component palette. 6
2. NetBeans IDE showing a component palette on the right-hand side. 7
3. A view showing grammar definitions and analysis information for a `STAT` symbol. 11
4. A form showing program text and corresponding symbolstream. 13
5. A form showing a program for Simple GCL and the corresponding Parse Tree. . 14
6. A form showing signature definitions, a list of names for the generated Java files and sample code for a sort `Stat`. 15
7. 16
8. Data flow among the subcomponents of the parser visualization tool 18
9. LL(1) Parser visualizer GUI 21

List of Tables

1. List of components currently available in the PAMOJA component toolkit at lexical, concrete and abstract syntax. 25

1. Introduction

1.1. The PAMOJA Framework

PAMOJA is a component framework that provides support for Grammar-Aware Engineering (GAE) [5] in the NetBeans [2] Integrated Development Environment (IDE). PAMOJA framework is being developed by Jackline Ssanyu as part of her PhD research work[1].

PAMOJA offers a framework and, a collection of components that deal with transformation of language terms from a concrete textual form to Abstract Syntax Trees (ASTs) and the converse transformation from abstract to concrete, and wizards for generating source-code from formal language specifications. Moving from concrete text to ASTs and back is divided into three levels:

- lexical syntax analysis - deals with the transformation of text to symbolstream;
- concrete syntax analysis - transforms the transformation of a symbolstream to an ECFG parse tree;
- abstract syntax analysis - deals with the transformation of an ECFG parse tree to AST.

To deal with each level, PAMOJA provides a collection of components for:

- holding the language specifications such as grammars (e.g., Context Free Grammar (CFG)) and signatures;
- holding data (e.g., text, symbolstream, parse tree, AST, and boxtree) and mappings (e.g., scanner, parser, abstractor, flattener and unscanner);
- views and editors (for manipulating data); and
- generators which take a language specification and produce language-specific support data or language-specific source code. Generators for language-specific support data are hidden within the data components whereas generators which generate source-code are implemented as wizards.

The components are implemented as JavaBeans [7] and all source-code is generated in Java programming language.

In addition, PAMOJA provides composite components assembled from other components of the framework. For a detailed description of PAMOJA, we refer the reader to [**?**]. Table 1 gives an overview of PAMOJA's basic components and wizards along with a brief description of their tasks. For a detailed description of the tasks, features and information about the implementation (API documentation) of these components and wizards, we refer the user to the Javadoc [**?**] of PAMOJA.

Typical applications of PAMOJA include:

- language prototyping in a language laboratory,
- building tools for teaching and experimenting with compiler front-end algorithms (such as scanning and parsing), in Section 5 we return to this subject and explain how to construct teaching tool for visualizing some parsing algorithms.
- building front-ends for programming languages. See Section 4.
- integration of language processing in all kinds of applications which are not necessarily compilers.

[1] At the school of computing and informatics technology, Makerere University

1.2. Design goals

The main design goals of PAMOJA framework are:

- It is oriented towards interactive development (there is less emphasis on text inputs). The main mode of operation is drag-and drop approach and use of wizards.
- Convenient way of writing language specifications by using component's simple editable properties. Users do not have to learn special-purpose languages to define the specifications. Of course users have to define the lexical, concrete and abstract syntax, but the well known specification formalisms of regular expressions, Context Free Grammars (CFGs) and signatures are used and no other special-purpose languages are required. To get an impression of what the language specifications look like, see the specifications for GCLSharp (Guarded Command Language) presented in Appendices **??**.
- Faster and easier generation of front-end tools (such as scanners and parsers) because components can be easily integrated.
- Convenient way of generating source-code files (e.g., source-code generated scanners and parsers, and AST classes) by using wizards inside NetBeans IDE.
- Automatic cascading of changes among cooperating components using observer/observable mechanism without having to write special scripts.

1.3. About this manual

This manual gives a description of the PAMOJA component framework. It assumes that users are familiar with the issues of front-end language processing (i.e., lexical analysis, syntax analysis and abstract syntax). The references [1], [3], and [6] provide a good introduction to these topics. The rest of this manual is structured as follows. Section 2 describes how to install PAMOJA in the NetBeans IDE. Section 4 contains an example (developing a language front-end) demonstrating the use of PAMOJA components and wizards. Another example is given in Section 5 demonstrating how to use PAMOJA components to construct an educational tool for visualizing parsing algorithms. Sections 6 and **??** provide information on currently known bugs/deficiencies and licenses, respectively.

2. Installing PAMOJA

PAMOJA has been tested with NetBeans 7.1 and later versions, including Apache NetBeans 17.

Note: PAMOJA components are packaged as JavaBeans [7] in a JAR (Java Archive) file and could be used in IDEs that support JavaBeans as well as the drag and drop style of developing applications, however the wizards (i.e., scanner, parser and signature API generator) are packaged as NBM (NetBeans module) files and are therefore NetBeans specific. This manual is tied to NetBeans.

PAMOJA Installation files can be downloaded from https://github.com/ssanyu/PAMOJA/tree/master/ins

The NetBeans IDE [2] needs to be installed before installing PAMOJA. Note that it is not mandatory to install the PAMOJA component's JAR and the NBM modules for the PAMOJA framework to operate correctly. The packages may be installed on the need basis. For instance, the wizards may not be installed until source-code generation for either a scanner, parser or signature API (Application Programming Interface) is needed.

2.1. Installing PAMOJA Components

PAMOJA components can be integrated into the NetBeans IDE, on the NetBeans Component Palette in a similar way you would add any other JavaBean components stored in a JAR file. To add PAMOJA components to the NetBeans palette follow these steps:

1. Download "PAMOJAComponents.jar" file from https://github.com/ssanyu/PAMOJA/tree/master/
   and save on a local drive.

2. Start NetBeans IDE.

3. Choose **Tools** > **Palette** > **Swing/AWT Components** from the NetBeans menu to display the Palette Manager window.

4. NetBeans needs to know which section of the palette will receive PAMOJA components. We could easily add PAMOJA components to an existing category of the palette, but let's keep it separate from the NetBeans components. Let us create our own category to place PAMOJA components within. In the Palette Manager window, click on the **New Category...** button to create a new category and name it, say "PAMOJAComponents".

5. Click on the **Add from JAR...** button (still in the Palette Manager window). NetBeans asks you to locate the JAR file that contains PAMOJA components you wish to add to the palette. Locate the file (PAMOJAComponents.jar) you downloaded to your local drive and click **Next**. A window similar to the one in Figure 1 is displayed showing a list of PAMOJA components in the JAR file.

6. Make sure **Show marked JavaBeans** option is selected. Then you can choose PAMOJA components you wish to add to the palette. In this case, select all the available components and click **Next**..

7. Choose **PAMOJAComponents** (i.e., the category you created in Step 4 to put the Components) and click **Finish**.

8. Click **Close** to make the Palette Manager window go away. Now take a look in the palette. PAMOJA components are installed in the **PAMOJAComponents** section. The palette looks something similar to Figure 2.

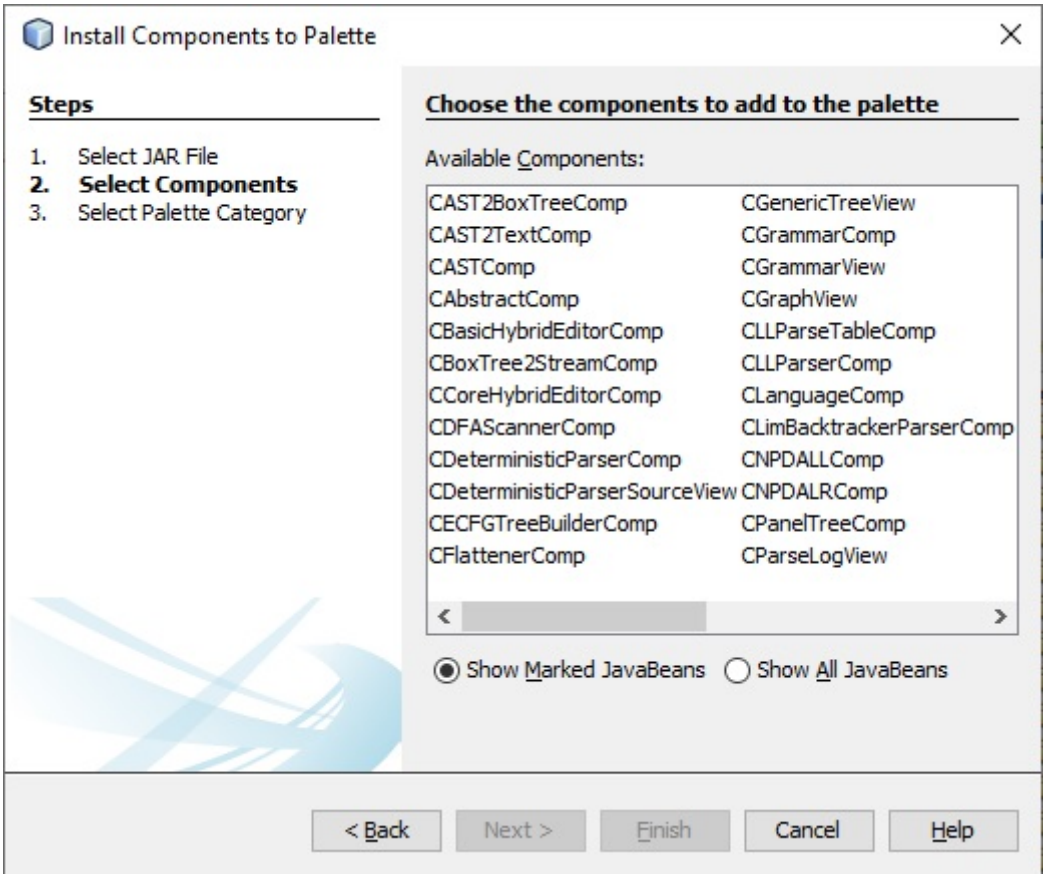

Figure 1: Window displaying PAMOJA components that can be added to the component palette.

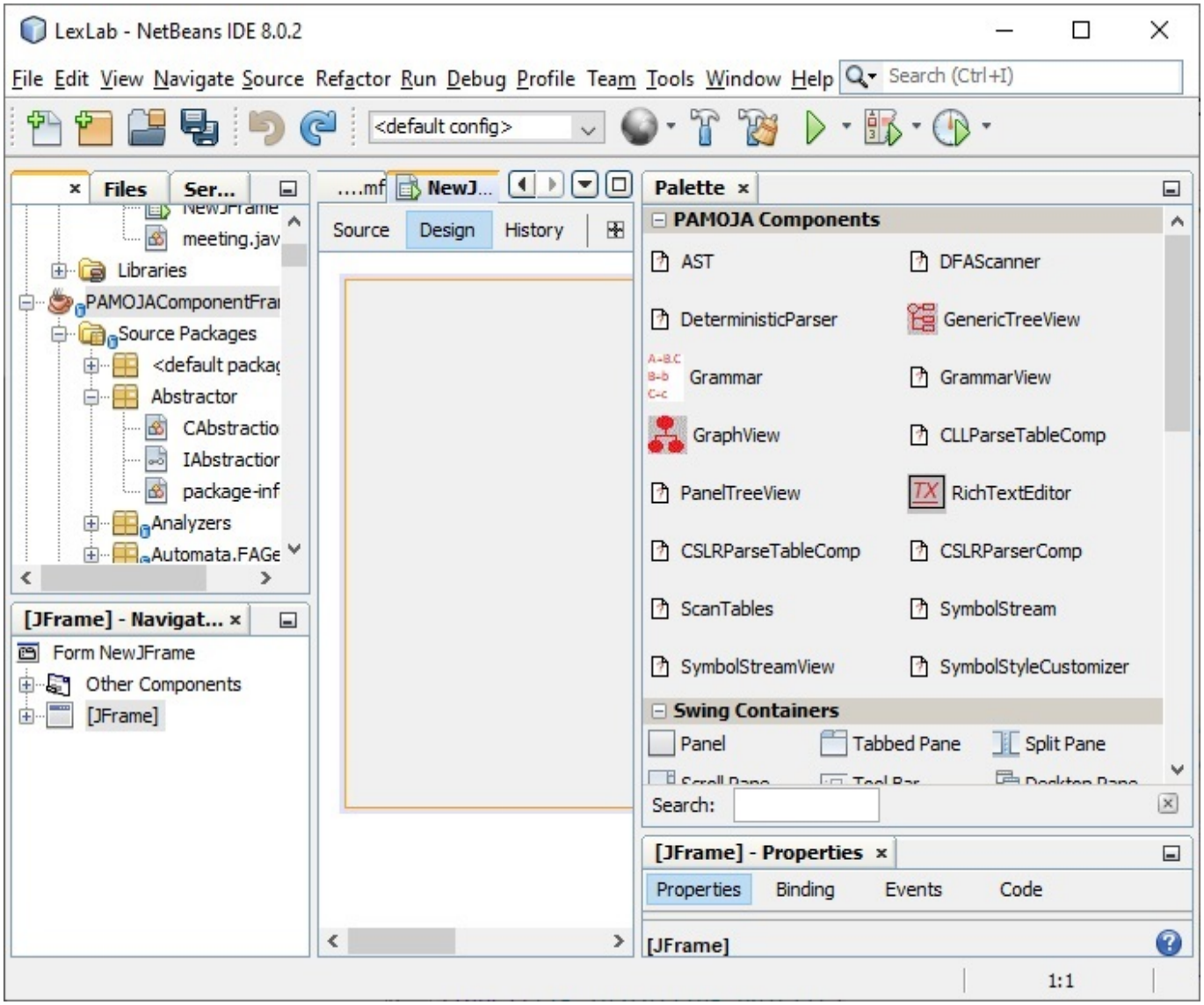

Figure 2: NetBeans IDE showing a component palette on the right-hand side.

2.2. Installing the Wizards

PAMOJA wizards (i.e., the files org-parser-generatorWizard.nbm, org-scanner-generatorWizard.nbm and org-signatureAPI-generatorwizard.nbm) are installed in NetBeans by following similar steps which are used to install other NBM modules (commonly known as 3^{rd} party plugins). To install PAMOJA wizards follow these steps:

1. Download the installers - org-parser-generatorWizard.nbm, org-scanner-generatorWizard.nbm and org-signatureAPI-generatorwizard.nbm from https://github.com/ssanyu/PAMOJA/tree/master and save on local disk.
2. Open the Netbeans IDE. (if not already open)
3. Choose **Tools** > **Plugins** from the NetBeans menu.
4. A Plugins dialog box opens up the **Available Plugins** tab by default; change the tab to **Downloaded**.
5. Click the **Add Plugins...** button; A window pops up; navigate to the local disk where you saved the files for the PAMOJA wizard(s) and add them. The files show up as visible in the dialog box.
6. Click the **install** button on the Bottom-left corner of the dialog box.
7. A confirmation window pops up; click **Next** to continue the installation.
8. Next you need to read the license agreement and accept it. Note that PAMOJA wizards are not licensed yet.
9. Since PAMOJA wizards' files are not digitally signed, you would need to ignore the Validation Warning about unsigned and not trusted plugins and continue passed this warning.
10. The PAMOJA wizards will install automatically.
11. Once the installation is finished, click **Finish** and restart the IDE. If the installation is completed successfully, the PAMOJA wizards will appear as sub-items in the File menu of NetBeans IDE and they can now be used with the IDE.

2.3. Uninstalling PAMOJA

Like installation, the steps for uninstalling PAMOJA components and wizards from NetBeans are similar to the steps for uninstalling other JavaBeans and NBM files. We present these steps in the proceeding sections.

2.3.1. Components

Remove PAMOJA components from NetBeans Palette as follows:

1. On the NetBeans palette, right-click on the section which contains PAMOJA components and click **Delete Category** from the pop-up menu which shows up. Or use the NetBeans Palette Manager[2].
2. If you clicked **Delete Category**, click the **Yes** button to confirm removal of PAMOJA components.

[2]A dialog which allows you to fully customize the component palette.

Note: To remove individual PAMOJA components from the NetBeans palette, right-click on the component itself and click **Remove** from the pop-up menu which shows up; click the **Yes** button to confirm. Alternatively, you can use the NetBeans Palette Manager.

2.3.2. Wizards

To un-install the PAMOJA wizards from NetBeans follow these steps:

1. Choose **Tools** > **Plugins** from the NetBeans menu.
2. A Plugins dialog box opens up the **Available Plugins** tab by default; change the tab to **Installed**.
3. A list of plugins installed in NetBeans shows up on the left corner of the dialog box; select the PAMOJA wizards you want to uninstall and click the **Uninstall** button below the list.
4. A confirmation window pops up; click **Uninstall** to continue uninstalling.
5. To complete the uninstallation process you will be required to restart the IDE (the default option is **Restart IDE Now**). Click **Finish** button; the IDE will be closed.

3. Building Software Tools with PAMOJA

Generally speaking, building a software tool with PAMOJA with the drag and drop approach of the NetBeans environment commences in the following steps:

1. **Creating a project** - create a project for the tool that you are going to develop.
2. **Designing the GUI** - using the drag-and-drop development style the required PAMOJA subcomponents are placed on a form. Components are selected from a palette window, dropped on a `JFrame` form. Native Swing components are used in the design of the GUI, such as: (1) panels, split panes and tabbed panes — for organizing the PAMOJA subcomponents on the `JFrame`; (2) text components — for receiving user input (e.g., regular expressions and CFGs); and (3) Menu items and buttons — for controlling GUI functionality.
3. **Connecting and setting the properties of PAMOJA components** - observable components are connected to their observers via reference properties. In addition, the desired properties of the components are customized either using dedicated property editors or through their setter methods in the source view.
4. **Adding source-code** - add source-code, for instance, for the menu items and buttons that activate and control the functionality of the tool.

In Sections 4 and 5, we apply these general steps to the construction of a front-end for a small programming language, as well as an educational tool for visualizing parsers.

4. Building a Language Front-end using PAMOJA

In this section we build a front-end for a small language, GCLSharp (a simplified version of the Guarded Commands Language (GCL) [4]) using PAMOJA component framework. First, we list the main steps required to develop the front-end. Then, we illustrate how to develop a front-end for GCLSharp.

4.1. Main Steps

Generally speaking, a front-end for GCLSharp consists of a mapping from a GCL program to an abstract syntax tree. Constructing a front-end using PAMOJA in the NetBeans IDE requires the following general steps:

1. Creating language specifications which include: signature specifications, grammar specifications (i.e., lexical and CFG) corresponding to the signature, and formatting specifications.
2. Creating a new `JFrame` form in an application in a NetBeans project.
3. Adding a grammar component on the form and loading the grammar specifications.
4. Adding scanning-related components on form, for scanning input plain text and generating a stream of symbols.
5. Adding parsing-related components on the form which input a stream of symbols as input and generate a parse tree.
6. Launching the signature API Generator wizard to load the signature specifications and generate a package of Java source-code files for the node factory.
7. Adding an abstractor component on the form, which loads a parse tree and uses the generated node factory to construct an AST.
8. Adding formatter-related components on the form which transform an AST to formatted text.

Note that all components are stored on the component palette of NetBeans IDE.

4.2. Example:Developing a Front-end for GCLSharp

To develop a language front-end using PAMOJA components we follow the following steps:

1. Create a signature and a grammar for GCLSharp. See Appendix C and AOLOgrammarfor GCLSharp language specifications.
2. Create a new `JFrame` form in an application in a NetBeans project.
3. Loading grammar specifications.
   a) Put a `Grammar` component on the form, name it (say Grammar1) and load it with the GCLSharp grammar.
   b) Put a `GrammarView` on the form and connect it to Grammar1. `GrammarView` displays the grammar definitions in tabular form and in tree-like form. `GrammarView` can be used to view the various grammar properties:$First$, $Last$, $Follow$ and $Lookahead$ sets, and predicates such as $Nullable$, $Empty$ and $Reachable$. To view analysis information switch to **Preview Design** or **Run Mode**. A sample of a form in run mode is as shown in Figure 3.
   c) Now, switch back to design view to proceed with the construction of the GCLSharp front-end. **Note:**In case you do not want to inspect the grammar, step b) and step c) are optional.

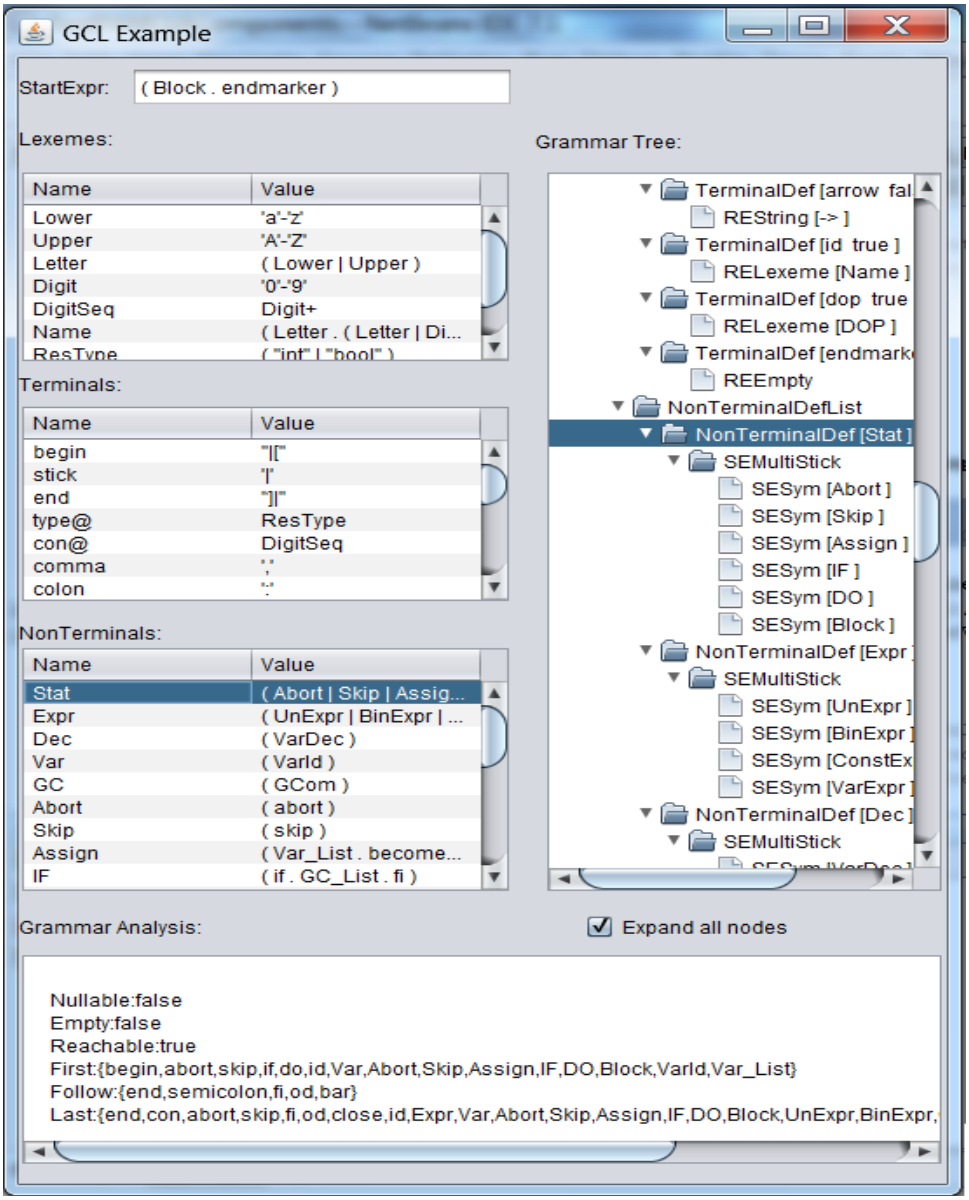

Figure 3: A view showing grammar definitions and analysis information for a `STAT` symbol.

4. Adding components for scanning.
   Note that we can use any type of scanner that is available in PAMOJA framework but for this example let us use a DFA-based scanner. To add scan-related components on the form follow the following steps:

   a) Add `DFAScanner` component on the form and name it (say DFAScannerComp1). DFAScannerComp1 needs access to scan-tables of GCLSharp. Provide the scan-tables by following these three steps:

      i. Add `ScanTables` on the form, name it (say ScanTablesComp1).

      ii. Connect ScanTablesComp1 to GrammarComp1 by setting the `Grammar` property of ScanTablesComp1 to GrammarComp1. The scan-tables are now generated.

      iii. Connect DFAScannerComp1 to ScanTablesComp1, by setting `ScanTables` property of DFAScannerComp1 to ScanTablesComp1.

   b) Add `RichTextView`, `SymbolStream` and `SymbolStreamview` components to the form. Name them RichTextView1, SymbolStreamComp1 and SymbolStreamView1 respectively.

c) Connect SymbolStreamView1 to SymbolStreamComp1.

d) Open the `text` property of `RichTextView1` and add some program text.

e) Before scanning can start, add a `JButton` on the form and name it say Scan1.

f) Set the `text` property of Scan1 to Scan.

g) Double-click Scan1. This will take you to source view where the GUI builder has created an empty event handler i.e. a method declaration of the following form:

```
private void Scan1ActionPerformed(java.awt.event.ActionEvent evt){
    // TODO add your handling code here:
}
```

h) Add the following handling code:

```
RichTextView1.getText();
RichTextView1.setScanner(DFAScannerComp1);
SymbolStreamComp1.setScanner(DFAScannerComp1);
```

i) Run the application.

j) Click **Scan**. If all the settings are successful, the scanner scans the text in RichTextView1 and generates a symbol stream whose contents are shown in SymbolStreamView1. A sample of a portion of a form showing text in RichTextView1 and symbol information in SymbolStreamView1 is as shown in Figure 4.

5. Adding Components for parsing and tree construction.

a) Go back to design view to proceed with the construction of the language front-end.

b) Add a Parser component to the form, for example a `DeterministicParser` and name it DetParserComp1. Note that you can also use any other type of parser component (such as `LimBacktrackerParser`, `SLRParser`, and Deterministic Recursive Descent parser generated as source-code) available in PAMOJA.

c) Add `ECFGTreeBuilder` component to the form and name it ECFGTreeBuilderComp1.

d) Add a `GenericTreeView` to the form and name it GenericTreeView1.

e) Connect DetParserComp1 to GrammarComp1 and to ECFGTreeBuilderComp1.

f) Set `TreeBuilding` property of DetParserComp1 to allow tree building or not.

g) Set `MultiStick`, `NoDataTerminal`, and `NonTerminal` properties of ECFGTreeBuilderComp1 to allow construction of multistick, terminals without data and nonterminal nodes or not.

h) Before parsing can start:

i. To control the parsing process, add a `JButton` to the form and name it say Parse1.

ii. Set the `text` property of Parse1 to Parse.

iii. Add a `JTextArea` to the form and name it say ParseResult. It will be used to display the result of the parsing process.

iv. Double click Parse1 to go to source view and add the following code to its event handler method declaration:

```
DeterministicParserComp1.setSymbolStream(SymbolStreamComp1);
CParserResult r=DeterministicParserComp1.getParser().parse();
CPosition v=DeterministicParserComp1.getParser().getSymPos();
    if(r.isSuccess()){
```

Figure 4: A form showing program text and corresponding symbolstream.

```
            ParseResult.setText("Success");
        }else{
            ParseResult.setText("Parse␣failed␣at:␣"+v.toText()+DeterministicPar
                 }
GenericTreeView1.updateTreeView(DeterministicParserComp1.getNode());
```

i) Run the application.

j) Click **Scan** to generate a symbolstream which the parser will use to parse the text.

k) Click **Parse**. If all the settings are successful, the parser parses the text in RichTextView1 and generates a parse tree which is shown in GenericTreeView1. A portion of a form showing text in RichTextView1 and GenericTreeView1 is as shown in Figure 5.

6. Building the abstract syntax tree.

   a) Use the GCL signature defined in Step 1 and generate a node factory by using the signature wizard. Start the signature wizard as follows:

      i. From the NetBeans IDE menu, choose **Tools** → **Open Signature Wizard**.

      ii. Enter the signature-name (say sign1) and the signature definitions in the wizard.

      iii. Click **Parse** button followed by **Analyze** button to check whether the signature is well-formed.

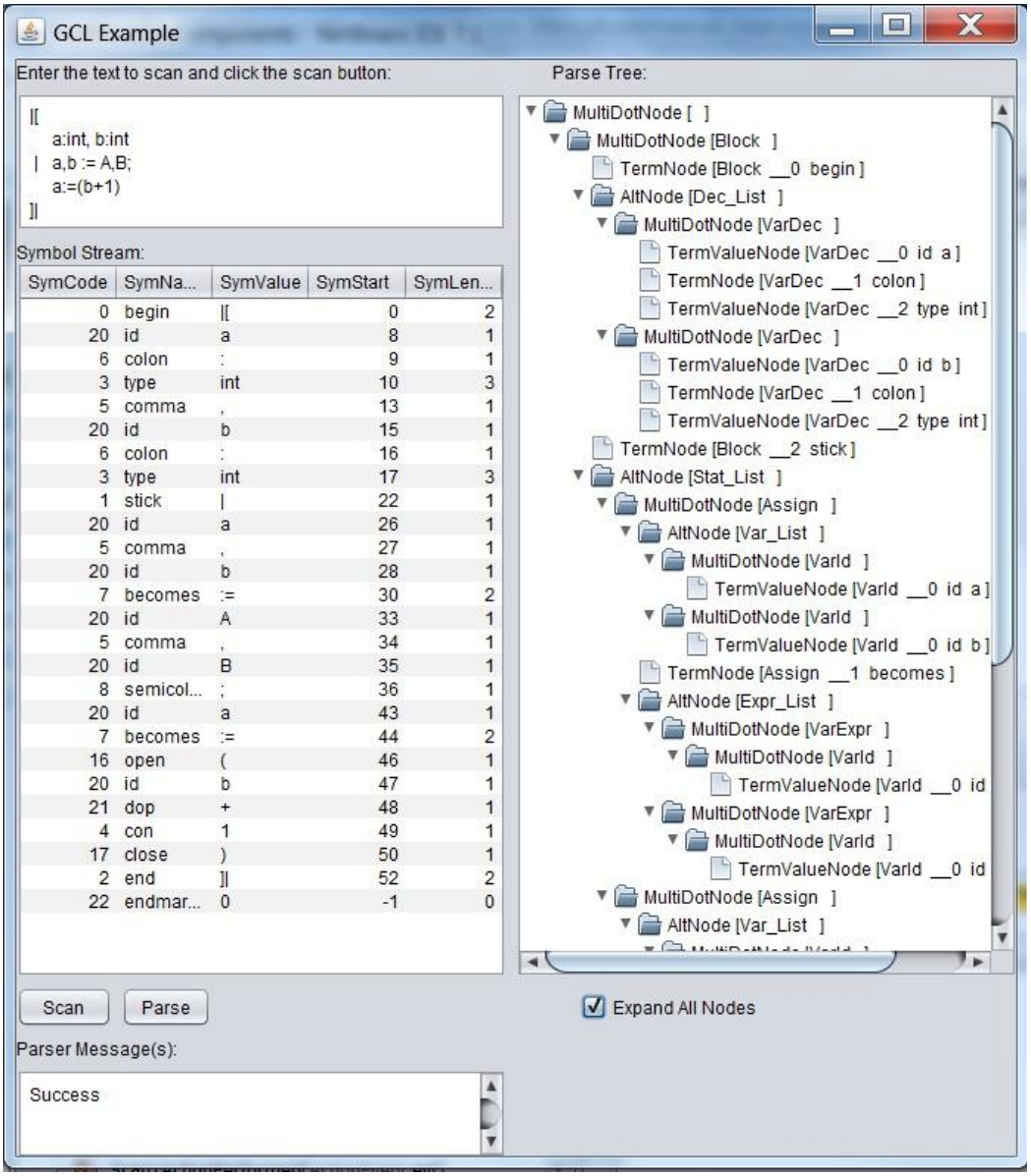

Figure 5: A form showing a program for Simple GCL and the corresponding Parse Tree.

iv. If the signature is well-formed, click **Generate** button. A list of Java source files is generated and the source-file's names are displayed in the Files List. To preview the source code for a particular source file click on its name in the list and the corresponding code will be displayed in **Code View**.

v. Click **Next** button. A second window in the wizard is displayed.

vi. Enter the name of the package in which to store the generated files. If every thing is done correctly, the wizard window will look something similar to Figure 6.

vii. Click **Finish** button. A package of Java source files for the node factory is created in the current project and compiled.

b) Add `Abstractor` component on the form and name it say AbstractorComp1 and connect it to DetParserComp1.

c) Add a second `GenericTreeView` on the form and name it say GenericTreeView2 and connect it to AbstractorComp1.

d) To control the abstraction process, add a `JButton` to the form and name it say ASTgen.

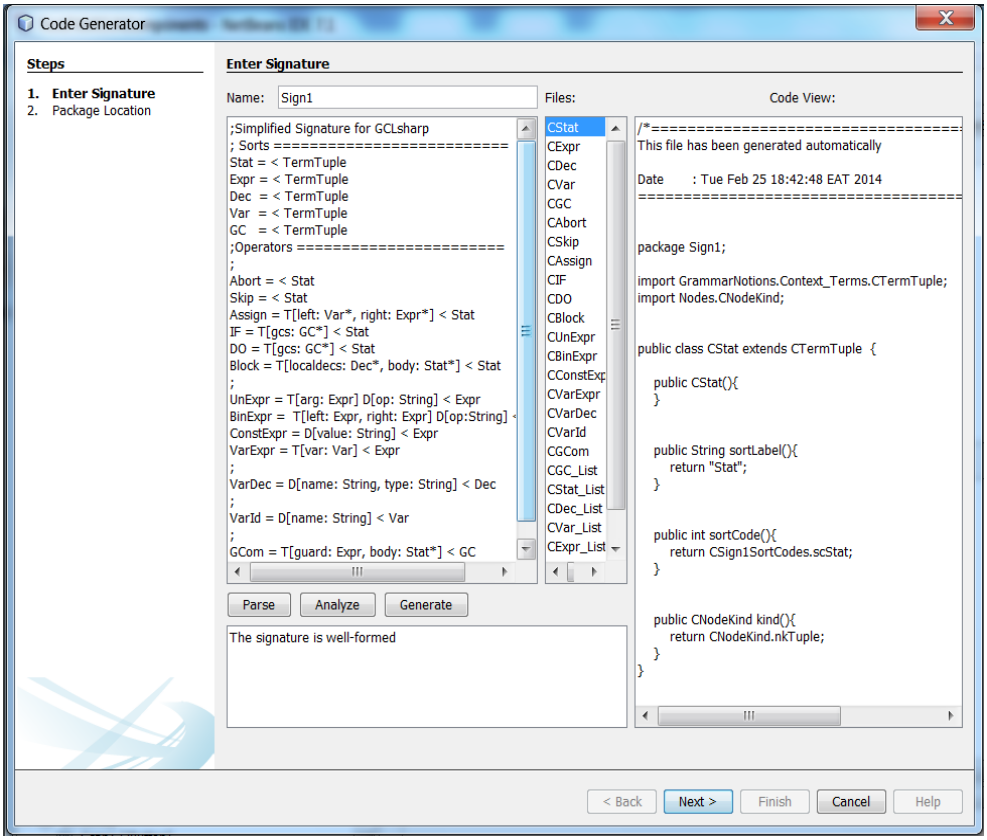

Figure 6: A form showing signature definitions, a list of names for the generated Java files and sample code for a sort `Stat`.

e) Set the `text` property of ASTgen to **Generate AST**.

f) Double click **Generate AST** button and add the following lines of code to its corresponding event handler method declaration:

```
CSign1ASTNodeFactory fNodeFactory=new CSign1ASTNodeFactory();
AbstractionComp1.setNodeFactory(fNodeFactory);
GenericTreeView2.updateTreeView(AbstractionComp1.getAST());
```

g) Run the application.

h) Scan and parse the text to generate a symbolstream and a parse tree.

i) Click **GenerateAST**. If all the settings are successful, an abstract syntax tree will be generated and shown in GenericTreeView2. A sample portion of a form showing the abstract syntax tree generated is as shown in Figure 7

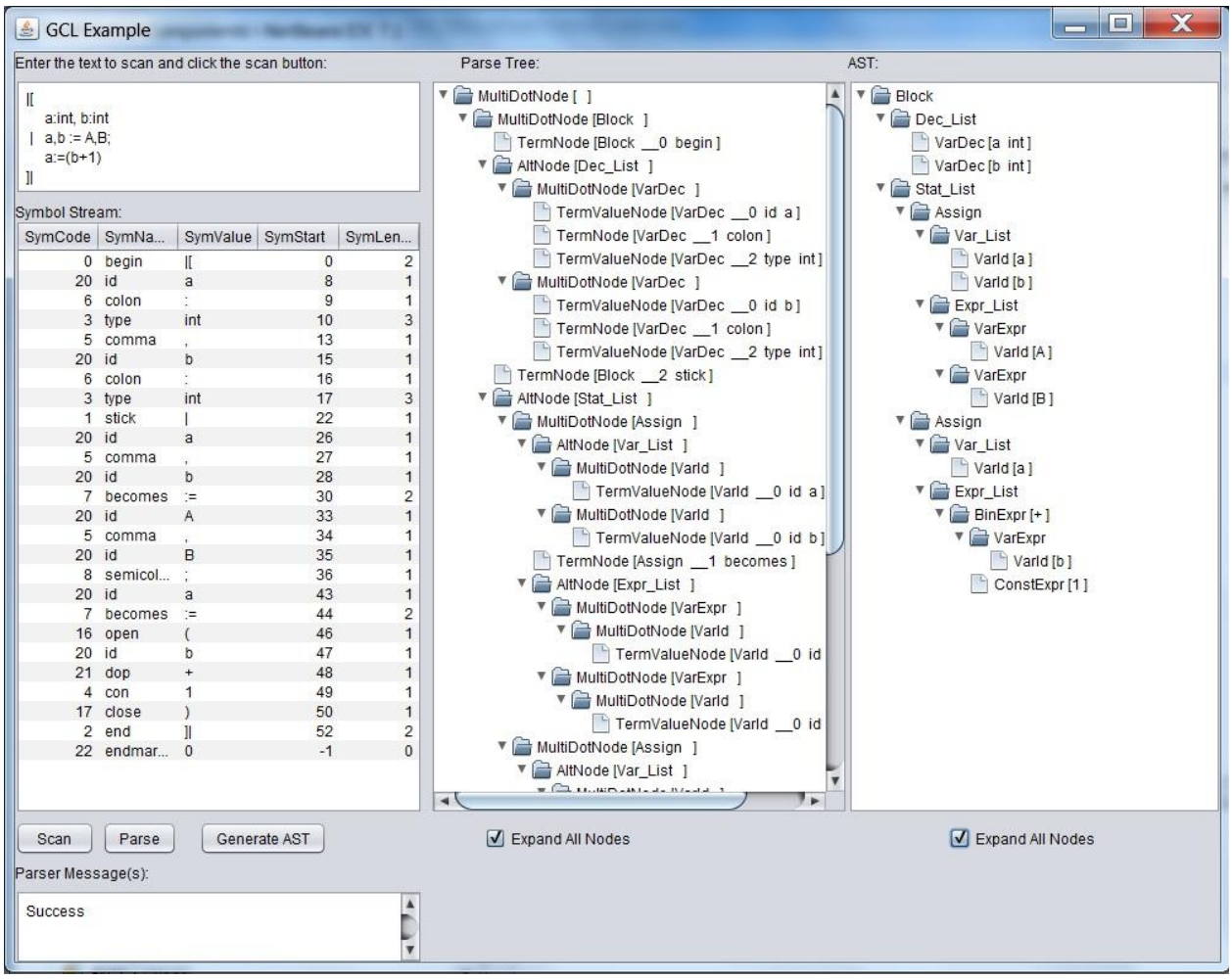
GCL Example
Enter the text to scan and click the scan button:
|[
a:int, b:int
| a,b := A,B;
a:=(b+1)
]|
Symbol Stream:
SymCode SymNa... SymValue SymStart SymLen...
0 begin |[0 2
20 id a 8 1
6 colon : 9 1
3 type int 10 3
5 comma , 13 1
20 id b 15 1
6 colon : 16 1
3 type int 17 3
1 stick | 22 1
20 id a 26 1
5 comma , 27 1
20 id b 28 1
7 becomes := 30 2
20 id A 33 1
5 comma , 34 1
20 id B 35 1
8 semicol... ; 36 1
20 id a 43 1
7 becomes := 44 2
16 open (46 1
20 id b 47 1
21 dop + 48 1
4 con 1 49 1
17 close) 50 1
2 end]| 52 2
22 endmar... 0 -1 0
Parse Tree:
MultiDotNode [ ]
MultiDotNode [Block]
TermNode [Block __0 begin]
AltNode [Dec_List]
MultiDotNode [VarDec]
TermValueNode [VarDec __0 id a]
TermNode [VarDec __1 colon]
TermValueNode [VarDec __2 type int]
MultiDotNode [VarDec]
TermValueNode [VarDec __0 id b]
TermNode [VarDec __1 colon]
TermValueNode [VarDec __2 type int]
TermNode [Block __2 stick]
AltNode [Stat_List]
MultiDotNode [Assign]
AltNode [Var_List]
MultiDotNode [VarId]
TermValueNode [VarId __0 id a]
MultiDotNode [VarId]
TermValueNode [VarId __0 id b]
TermNode [Assign __1 becomes]
AltNode [Expr_List]
MultiDotNode [VarExpr]
MultiDotNode [VarId]
TermValueNode [VarId __0 id
MultiDotNode [VarExpr]
MultiDotNode [VarId]
TermValueNode [VarId __0 id
MultiDotNode [Assign]
AltNode [Var_List]
AST:
Block
Dec_List
VarDec [a int]
VarDec [b int]
Stat_List
Assign
Var_List
VarId [a]
VarId [b]
Expr_List
VarExpr
VarId [A]
VarExpr
VarId [B]
Assign
Var_List
VarId [a]
Expr_List
BinExpr [+]
VarExpr
VarId [b]
ConstExpr [1]
Scan
Parse
Generate AST
Expand All Nodes
Expand All Nodes
Parser Message(s):
Success

Figure 7:

5. Building a Tool for Visualizing Parsers

In this section we will build a simple educational tool for visualizing parsers (e.g., LL(1) and SLR(1)). First, we provide an overview of the parser visualizer tool that we want to construct in Section 5.1, followed by an inventory of components which will be needed to construct the tool in Section 5.2. Second, we provide a step by step guide of how to design the GUI of our parser visualizer in Section 5.3. Then, in Section 5.4 we describe how to add back-end functionality to the GUI just designed. At this point the visualizer will only be able to visualize LL(1) parsers. Finally in Section 5.5 we explain how to replace an LL(1) parser component of our parser visualizer with an SLR(1) component without changing the rest of PAMOJA components in the visualizer and with very minimal source-code.

5.1. The Parser Visualizer Tool

The parser visualizer tool has an interface consisting of a set of interacting views that allow users to explore the various data structures used in parsing, like grammar specifications, input to be processed, parse tables, derivation trees, pushdown stack, and parser log (or parser trace). A user can direct parsing of an input string step by step forward, backward, or parse the whole string in a single step. Each parsing step shows: (1) the current parse trace contents in the parse-log view; and (2) the parser's visual execution in the graph view, which includes: current stack contents, input symbolstream with a marker (red vertical line) indicating that symbols to the left of the marker have already been read and symbols to the right of the marker must still be read, and a partially constructed parse tree. E.g see Figure 9.

5.2. Required Components

To construct our parser visualization tool, we need the following configuration of PAMOJA components:

- `Grammar` - An observable component that contains a lexical and CFG and ensures the grammar is well-formed. To access and modify the value of a grammar, there are two properties: `GrammarStructure` which represents the internal structure of a grammar, and `GrammarText`, which represents the grammar in text form. This component checks the well-formedness of the new grammar before accepting it whenever the grammar is changed. This component also analyzes the grammar and computes some general information, such as the sets `First`, `Last`, `Follow`, and the predicates `reachable`, `nullable`, `left recursive` and `ELL(1)`.

- `GrammarView` - A view with facilities for viewing grammar definitions and grammar analysis information and observes changes in the `Grammar`.

- `DFAScanner` - This a Deterministic Finite Automata-based lexical scanner (DFA). The component includes a hidden scanner generator based on Lex scanner algorithms, and it observes the `Grammar` component for changes in the lexical grammar.

- `RichTextEditor` - A text editor with text coloring and font style capabilities. It uses the `DFAScanner` and `SymbolStyleCustomizer` (not used in this construction) components to scan its text and to render the recognized symbols respectively, and it observes their changes.

- `SymbolStream` - An observer/observable component that holds a symbol stream generated by the `DFAScanner`. This component observes changes in the symbolstream of a `DFAScanner`.

- `SLRParser` - This is an SLR(1) parser component. The component includes a hidden parse-tables generator, based on the SLR(1) algorithms, and it observes the `Grammar` component for changes in the CFG as well as the `SymbolStream` for changes in the generated symbols.
- `LLParser` - This is an LL(1) parser component. The component includes a hidden parse-tables generator, based on the LL(1) algorithms, and it observes the `Grammar` component for changes in the CFG as well as the `SymbolStream` for changes in the generated symbols.
- `TreeBuilder` - This component has facilities for producing different kinds of parse tree nodes used for constructing the parse trees. It also contains options for deciding which kind of nodes should be used to create a parse tree.
- `ParseTableView` - A general tabular view with facilities for viewing parse tables (SLR and LL) and it observes parsers (`SLRParser`, `LLParser`) for changes in their parse tables.
- `ParseLogView` - A general tabular view with facilities for viewing parse logs (SLR and LL) and it observes parsers (`SLRParser`, `LLParser`) for changes in their parse logs.
- `GraphView` - A general view for displaying different kinds of graphs and trees, such as parse trees and ASTs anf it observes parsers (`SLRParser` or `LLParser`) for changes in their parse trees, stack and symbolstream.

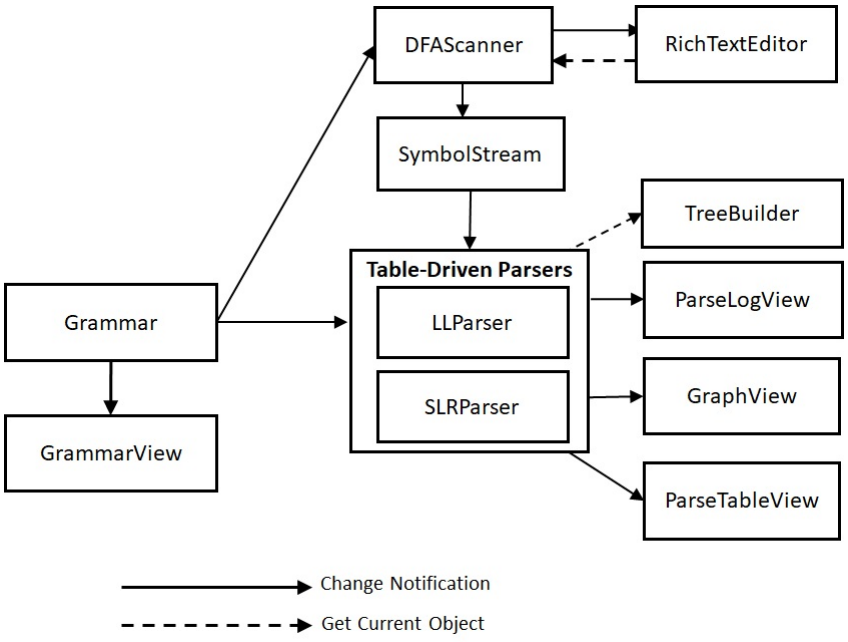

Figure 8: Data flow among the subcomponents of the parser visualization tool

Figure 8 depicts the architectural flow of information between the various cooperating components of our parser visualizer tool. The architecture depicted in Figure 8 has a close resemblance with the classical Model-View-Controller (MVC). Following the observer-observable design pattern, when grammar specifications of the `Grammar` component change, a property change is observed by the `GrammarView`, `DFAScanner` and the parser component (i.e., `LLParser` or `SLRParser`). The following three scenarios take place:

1. The `GrammarView` reacts to the observed property change of the `Grammar` by adjusting its view to that of the new grammar specifications.

2. The `DFAScanner` reacts to the change in the `Grammar` component by invoking its hidden scanner generator to regenerate its scan tables for the new language. The change in the `DFAScanner` is in turn observed by the `RichTextEditor`, which uses the adjusted `DFAScanner` to scan its text according to the new grammar, and produces a sequence of symbols. The scanner subsequently communicates the change in its symbol sequence to the `SymbolStream` component to update itself. In turn, the `SymbolStream` communicates the change in its symbol sequence to the parser component. In turn, the parser component uses the `Grammar`, `SymbolStream`, and `TreeBuilder` to construct a parse log and a parse tree. The change in the parser is in turn observed by the `ParseLogView` and `GraphView` components which update their views respectively.

3. The parser component reacts to the change in the `Grammar` by invoking a hidden generator to regenerate its parse tables for the new grammar. This change is in turn observed by the `ParseTableView` to adjust its view to that of the new parse tables. Additionally, the parser component uses the `Grammar`, `SymbolStream`, and `TreeBuilder` as stated in the second scenario.

In addition to the PAMOJA components explained above, we need the following native Swing components to assist with organizing the PAMOJA components, directing the parsing process and displaying messages produced by the parser visualizer tool.

5.3. Designing the GUI

In this section we describe how to create the parser visualizer GUI. In particular, we provide: (1) an overview on the use of Swing components to organize the layout of PAMOJA components in the GUI; and (2) a detailed step by step guide on how to connect PAMOJA components. For a more comprehensive guide to the use of Swing components in GUI building we refer the reader to NetBeans tutorial.

When NetBeans IDE is open, we see on the component palette various categories of components, such as PAMOJA Components, Swing Controls and Swing Containers. To construct the parser visualizer from these components, we follow these steps:

1. *Creating a JFrame container*
   a) create a form (`JFrame`) within which to place other required GUI components.
   b) using the form's property editor set the `title` property to "A visualizer for parsing algorithms".

2. *Adding Swing components on the form* - Using the palette we will populate our form with the following Swing components:
   a) `SplitPane` - add split panes which we will use to organize PAMOJA view and editor components, that is, `GrammarView`, `ParseTableView`, `ParseLogView`, `RichTextEditor`, and `GraphView`. See Figure 9.
   b) `Menu` - add a menu item, `Open`. Rename `jMenuItem1` to `mnuOpen`. This menu item will be used to load a grammar from a text file and store it in the `Grammar` component.
   c) `TextArea` - add a text area for displaying messages produced by the application. Rename the text area to `txtMsg`.
   d) `Button` - add four buttons to direct the parsing process.
      - Rename `jButton1` to `cmdStart` and the display text to `Start`.
      - Rename `jButton2` to `cmdStep` and the display text to `Step`.
      - Rename `jButton3` to `cmdBack` and the display text to `Back`.

- Rename `jButton4` to `cmdParseAll` and the display text to `ParseAll`.

3. *Adding PAMOJA components on the form* - Using the palette, add PAMOJA components using the following steps:

   a) Add a `Grammar` component on the form and use its property editor to set the grammar. The property editor shows the property `grammarText`, among others, used to set the grammar in text form. Use this property to set the grammar. When finished, the `Grammar` component checks that the grammar is well-formed, updates its contents and sends a change notification to its observers, if any. See Appendix B, for a sample grammar.

   b) Add a `GrammarView` on the form and connect it to the `Grammar` component by setting its `grammar` property to the instance of `Grammar` placed on the form in step 1. The following actions take place:

      - `GrammarView` is added to the list of observers of the `Grammar` component.
      - `GrammarView` updates its view according to the new grammar.

   c) Add a `DFAScanner` component on the form and connect it to the `Grammar` component by setting its `grammar` property. The following actions take place:

      - `DFAScanner` is added to the list of observers of the `Grammar` component.
      - `DFAScanner` invokes its hidden scanner generator to construct its scan tables and sends a property change to its observer components, if any.

   d) Add a `SymbolStream` component on the form and connect it to the `DFAScanner` component by setting its `scanner` property. The `SymbolStream` is added to the list of observers of the `DFAScanner` component.

   e) Add a `RichTextEditor` component on the form. Enter text in its `text` property. Connect it to the `DFAScanner` by setting its `scanner` property. `RichTextEditor` component reacts in the following way:

      - it starts to observe changes in the scan tables of the `DFAScanner`.
      - it uses the `DFAScanner` to scan the text according to the lexical syntax of the grammar. Additionally, the `DFAScanner` generates a sequence of symbols and notifies its observers, such as the `SymbolStream` component.

   f) Add `LLParser` component on the form and connect it to the `SymbolStream` and `Grammar` components by setting its `symbolstream` and `grammar` properties respectively. The `LLParser` reacts in the following ways:

      - it is added to the list of observers of the `SymbolStream` component as well as the `Grammar` component.
      - it invokes its hidden parser generator to construct its parse tables and sends a property change to its observer components, if any.

   g) Add `ParseTableView`, `ParseLogView` and `GraphView` components on the form. Connect them to the `LLParser` by setting their `parser` property. The view components are added to the list of observers of the `LLParser` component and they update their views according to the new parse tables, parse log, and parse tree visualization information respectively.

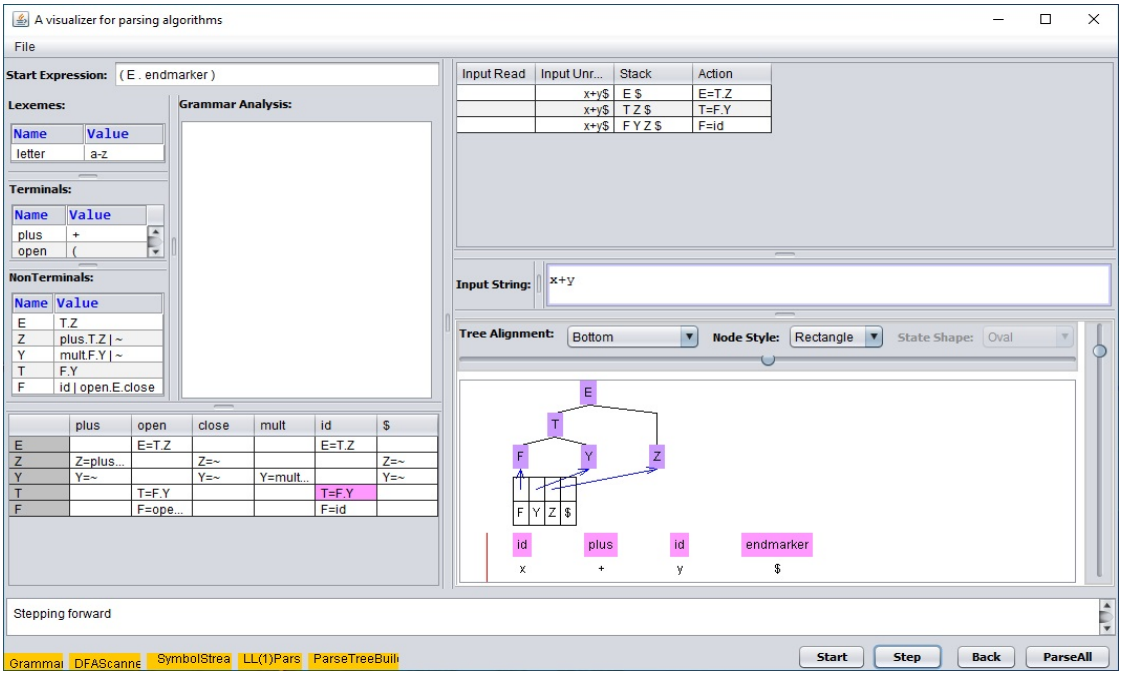

Figure 9: LL(1) Parser visualizer GUI

5.4. Adding functionality

In this section we describe how to add back-end functionality to the parser visualizer GUI created in the previous section. We are going to give functionality to the `Open` menu item, `Start`, `Step`, `Back`, and `ParseAll` buttons. The `txtMsg` box will be used to display messages from the parser visualizer tool.

1. *Making* `Open` *menu item work*
   - Double click the menu item `Open`
   - The IDE will open up the **Source Code** window and scroll to where you implement the action you want the menu item to do when it is pressed (either by mouse click or via keyboard). Your source code window should contain lines of the following kind:

     ```
     private void mnuOpenActionPerformed(java.awt.event.ActionEvent evt) {
      //TODO add your handling code here:
     }
     ```

   - Replace the **TODO** lines of the menu item using the code below

     ```
     // OpenFile needs the following PAMOJA package to be imported:
     // import General.OpenFile;
     private void mnuOpenActionPerformed(java.awt.event.ActionEvent evt) {
             OpenFile f=new OpenFile(); //creates and open a File dialogue box
             String vText=new String();
             vText=f.open("Open␣Grammar␣File", this);
             Grammar.setGrammarText(vText);
     }
     ```

2. *Making the buttons* `Start`, `Step`, `Back`, and `ParseAll` *work* - In order to give function to the buttons, we have to assign an event handler to each to respond to events.
   - Double click each button
   - The IDE will open up the **Source Code** window and scroll to where you implement the action you want the button to do when the button is pressed (either by mouse

click or via keyboard). For each button, your Source Code window should contain lines of the following kind:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
 //TODO add your handling code here:
}
```

- Replace the **TODO** lines for each button using the code below

```
private void cmdStartActionPerformed(java.awt.event.ActionEvent evt) {
    LLParser.start();
    txtMsg.setText(""); //clear the message box
    cmdStep.setEnabled(true); //enable Step button
    cmdBack.setEnabled(false); // disable Back button
}

private void cmdStepActionPerformed(java.awt.event.ActionEvent evt) {
    LLParser.parseStep();
    String action=LLParser.getParser().action();
    if(action.equals("Error")){
        cmdStep.setEnabled(false);
        txtMsg.setText("String␣is␣not␣accepted␣--␣there␣are␣errors");
    }else if(action.equals("Accept")){
        cmdStep.setEnabled(false);
        txtMsg.setText("String␣accepted");
    }
    ParseTable.setCellColor(LLParser.r, LLParser.c);
    cmdBack.setEnabled(true);
    txtMsg.setText("Stepping␣forward");
}

private void cmdBackActionPerformed(java.awt.event.ActionEvent evt) {
    LLParser.undo();
    cmdStep.setEnabled(true);
    txtMsg.setText("Stepping␣backward");
}

private void cmdParseAllActionPerformed(java.awt.event.ActionEvent evt) {
    LLParser.parseText();
    String action=LLParser.getParser().action();
    if(action.equals("Error")){
       txtMsg.setText("String␣is␣not␣accepted␣--␣there␣are␣errors");
    }else if(action.equals("Accept")){
       txtMsg.setText("String␣accepted");
    }
    cmdBack.setEnabled(false);
}
```

3. Test the parser visualizer using an LL(1) grammar of your choice. See Appendix B, for example.

5.5. Replacing a Parser

Our component-based development style facilitates replacement and extension of a constructed tool with more PAMOJA components without affecting the existing ones and with little programming code. As an example, we show how to replace an `LLParser` component of our parser visualization tool with an `SLRParser` component without changing the already existing PAMOJA components of the tool.

`SLRParser` is a table-driven kind of parser. As depicted in Figure 8, a table-driven parser, collaborates directly with six components to visualize parsing and tree construction of input string

: `Grammar`, `SymbolStream`, `TreeBuilder`, `ParseLogView`, `ParseTableView`, and `GraphView`. Apart from the `TreeBuilder`, all the other components are already part of the parser visualizer tool. To replace the `LLParser` with `SLRParser` without changing the collaborator components, we follow these steps:

1. Remove the `LLParser` component from the form.
2. Select `SLRParser` component from the palette, drop it onto the form, and:
   - Connect it to the `Symbolstream` component by setting its `symbolstream` property.
   - Add a `TreeBuilder` component on the form. Connect the `SLRParser` to the `TreeBuilder` component by setting its `treebuilder` property.
   - Set the `SLRParser` component's `MultiStick`, `NoDataTerminal`, `NonTerminal` and `treebuilding` properties to `true`.
3. Edit the source-code of the `Start`, `Step`, `Back`, and `ParseAll` buttons so that they can respond to the `actionPerformed` event. Note that most of the existing code does not need to be edited. See code below:

```
private void cmdStartActionPerformed(java.awt.event.ActionEvent evt) {
     SLRParser.start();
     txtMsg.setText("␣");
     cmdStep.setEnabled(true);
     cmdBack.setEnabled(false);
}
private void cmdStepActionPerformed(java.awt.event.ActionEvent evt) {
// Reject and Accept need the following PAMOJA packages to be imported:
// import TableGenerators.LR.Accept;
// import TableGenerators.LR.Reject;
     SLRParser.parseStep();
     if(SLRParser.getParser().getAction() instanceof Reject){
         cmdStep.setEnabled(false);
         txtMsg.setText("String␣is␣not␣accepted␣--␣there␣are␣errors");
     }else if(SLRParser.getParser().getAction() instanceof Accept){
         txtMsg.setText("String␣accepted");
     }
     TableView.setCellColor(SLRParser.r, SLRParser.c);
     cmdBack.setEnabled(true);
     txtMsg.setText("Stepping␣forward");
}

private void cmdBackActionPerformed(java.awt.event.ActionEvent evt) {
    SLRParser.undo();
    cmdStep.setEnabled(true);
    txtMsg.setText("Stepping␣backward");
}

private void cmdParseAllActionPerformed(java.awt.event.ActionEvent evt) {
// Move - needs the following PAMOJA package to be imported:
// import TableGenerators.LR.Move;
    SLRParser.parse();
    Move action=SLRParser.getParser().getAction();
    if(action instanceof Reject){
        txtMsg.setText("String␣is␣not␣accepted␣--␣there␣are␣errors");
    }else if(action instanceof Accept){
        txtMsg.setText("String␣accepted");
    }
    cmdBack.setEnabled(false);
    cmdStep.setEnabled(false);
}
```

4. Test the parser using an SLR(1) grammar of your choice. See Appendix B, for example.

It should be noted that instead of replacing an existing parser, as we have done in the previous steps, the parser visualizer tool can be extended with several parsers. This requires the following three main steps:

- Add the parsers on form and connect them to their collaborator components in a similar way as we have done above.
- Create menu items that allow a user to activate the parsers. Add source-code that implements `actionPerformed` event of each menu item.
- Edit the source-code of the `Start`, `Step`, `Back`, and `ParseAll` buttons so that, for a selected parser, they can respond to the `actionPerformed` event.

6. Bugs and Deficiencies

[To be completed]

7. Copying and Licence

[To be completed]

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques,and Tools*. Addison-Wesley, Boston, second edition, 2006.

[2] Apache Software Foundation. Apache Net Beans: Development Environment, Tooling Platform and Application Framework. `https://netbeans.apache.org//`, 2023.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[4] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[5] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(3):331–380, 2005.

[6] Wilhelm Reinhard and Maurer Dieter. *Compiler Design*. International Computer Science Series. Addison-Wesley, 1995.

[7] Sun Microsystems. JavaBeans API Specification. `https://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/`, 1997.

Appendix A PAMOJA Components and Wizards

A list of components and source-code generation wizards which PAMOJA currently provides is as shown in Table 1.

Table 1: List of components currently available in the PAMOJA component toolkit at lexical, concrete and abstract syntax.

Level	Component	Description
Lexical analysis	RichTextEditor	A text editor with facilities for color and font attributes.
	ScanTables	Generates NFA and DFA from a lexical grammar — based on Thompson and Subset construction algorithms.
	ScanTableView	A tabular view for NFA and DFA tables.
	DFAScanner	Maps an input string to a sequence of symbols.
	SymbolStream	Holds a sequence of tokens produced by a DFAScanner.
	SymbolStreamView	A tabular view for symbol properties (e.g., symbol name, length, start and end position in the input string).
	SymbolStyleCustomizer	Holds a mapping from lexical symbols to symbol categories and symbol categories to font and color attributes.
	Stream2Text	Maps 2D-symbolstream to 2D-text.
	ELL(1)ScannerGenerator	Generates ELL(1) scanner source-code from a lexical grammar.
Concrete syntax	Grammar	Holds an ECFG, maintains its consistency and computes grammar properties (e.g., First, Last, Follow and lookahead sets, and predicates such as Null, Empty and Reachable).
	GrammarView	Facilitates grammar specification and inspection of properties.
	SLRTables	Generates SLR(1) parse tables from a grammar.
	LLTables	Generates LL(1) parse tables from a grammar.
	ParseTableView	A generic view for parse tables.
	NPDA-TD	Holds a non-deterministic top-down parsing algorithm based on NPDA.
	NPDA-BU	Holds a non-deterministic bottom-up parsing algorithm based on NPDA.
	SLRParser	Holds SLR(1) algorithm which maps a symbolstream to a parse tree.
	LLParser	Holds LL(1) parsing algorithm which maps a symbolstream to a parse tree.
	DeterministicParser	A recursive descent parser which interprets ELL(1) grammars.
	LimitedBackTrackerParser	A recursive descent, limited backtracking parser which interprets a non left-recursive grammar.
	RecDecParser	Generates source-code for a deterministic recursive descent parser from an ELL(1) grammar.
	ParserSourceView	A view for inspecting recursive descent parser source per production rule.
	ParseLogView	A tabular view for inspecting parser log information.
	TreeBuilder	Facilitates the construction of different kinds of parse tree nodes.
	ParseTree	Holds a structural representation of a parse tree.
Abstract syntax	Signature	Holds abstract syntax and maintains its consistency.
	AST	Holds a structural representation of an AST.
	AST2BoxTree	Maps an AST to a box tree.
	BoxTree2Stream	Maps a box tree to a 2D-symbolstream.
	Abstractor	Maps a parse tree to an AST.
	GenericTreeView	A `JTree`-like view which displays PAMOJA-trees (e.g. parse trees, ASTs, and box trees) and allows inspection of their nodes.
	PanelTreeView	Displays an AST as a hierarchy of nested panels and allows its structural editing.
	Patterns	Holds box layout specifications for producing 2D-text and maintains their consistency.
	TreeEditor	Contains basic operations for editing different kinds of AST nodes.

Table 1 – *(continued)*

Component	Description	
	GraphView	Provides visual presentation of different structures (e.g., regular expressions, NFA, parse trees and ASTs).
	SignatureAPIGenerator	Generates AST classes for abstract syntax.

Appendix B Simple Expression Grammar

SLR Grammar==========

```
[Lexemes]
letter='a'-'z'

[Terminals]
plus='+'
times='*'
open='('
close=')'
id=letter+
endmarker='$'

[Nonterminals]
E=E.plus.T|T
T=T.times.F|F
F=id|open.E.close

[Start]
startexpr=E.endmarker
```

LL(1) Grammar ==========

```
[Lexemes]
letter='a'-'z'

[Terminals]
plus='+'
open='('
close=')'
mult='*'
id=letter+
endmarker=$

[Nonterminals]
E=T.Z
Z=plus.T.Z|~
Y=mult.F.Y|~
T=F.Y
F=id|open.E.close

[Start]
startexpr=E.endmarker
```

Appendix C Signature for GCLSharp

Signature for GCLSharp ==========

```
Sorts ==========================
Stat = < TermTuple
Expr = < TermTuple
Dec = < TermTuple
Var = < TermTuple
GC = < TermTuple

Operators ======================
Abort = < Stat
Skip = < Stat
Assign = T[left:  Var*, right:  Expr*] < Stat
IF = T[gcs:  GC*] < Stat
DO = T[gcs:  GC*] < Stat
Block = T[localdecs:  Dec*, body:  Stat*] < Stat

UnExpr = T[arg:  Expr] D[op:  String] < Expr
BinExpr = T[left:  Expr, right:  Expr] D[op:String] < Expr
ConstExpr = D[value:  String] < Expr
VarExpr = T[var:  Var] < Expr
BoolExpr = D[value:  String] < Expr

VarDec = D[name:  String, type:  String] < Dec

VarId = D[name:  String] < Var

GCom = T[guard:  Expr, body:  Stat*] < GC
```

Appendix D AOLO grammar specifications for GCLsharp

```
! AOLO grammar for GCLsharp
! Note: The input format is such that every stick expression of length 1
! starts with a '|'; similarly, every dot expression of length 1 starts with a '.'
! Grammar is derived from Signature for GCLsharp
! - Sorts of signature are used as Or-nonterminals
! - Operators of signature are used as And-nonterminals
! - For each S* in signature a List-nonterminal S_List and a rule S_List=S* are added

  [Lexemes]
Lower='a'-'z'
Upper='A'-'Z'
Letter=Lower|Upper
Digit='0'-'9'
DigitSeq=Digit+
Name=Letter.(Letter|Digit)*
ResType="int"|"bool"
DOP='='|"/="|'<'|"<="|'>'|">="|'+'|'-'|'*'|'/'
MOP='-'|"not"

  [Terminals]
begin="|["
stick='|'
```

```
end=`"]|"
type@=ResType
con@=DigitSeq
comma=',
colon=':
becomes=":="
semicolon=';
abort="abort"
skip="skip"
if="if"
fi="fi"
do="do"
od="od"
mop@=MOP
open='(
close=')
bar="[]"
arrow="->"
id@=Name
dop@=DOP
endmarker=0
```

[Nonterminals]

```
! Or-nonterminals ======================
Stat=Abort|Skip|Assign|IF|DO|Block
Expr=UnExpr|BinExpr|ConstExpr|VarExpr
Dec=|VarDec
Var=|VarId
GC=|GCom
  ! And-nonterminals ======================
! Stat
Abort=.abort
Skip=.skip
Assign=Var_List.becomes.Expr_List
IF=if.GC_List.fi
DO=do.GC_List.od
Block=begin.Dec_List.stick.Stat_List.end
  ! Expr
UnExpr=mop@.Expr
BinExpr=open.Expr.dop@.Expr.close
ConstExpr=.con@
VarExpr=.Var
  ! Dec
VarDec=id@.colon.type@
  ! Var
VarId=.id@

GCom=Expr.arrow.Stat_List
  ! List-nonterminals ======================
Dec_List=Dec%comma
```

```
Stat_List=Stat%semicolon
Var_List=Var%comma
Expr_List=Expr%comma
GC_List=GC%bar

  [Start]
startexpr=Block.endmarker
```