PAMOJA: A Component Framework for Grammar-Aware Engineering

PAMOJA User's Manual

By Jackline Ssanyu

August 20, 2020

Contents

List of Figures

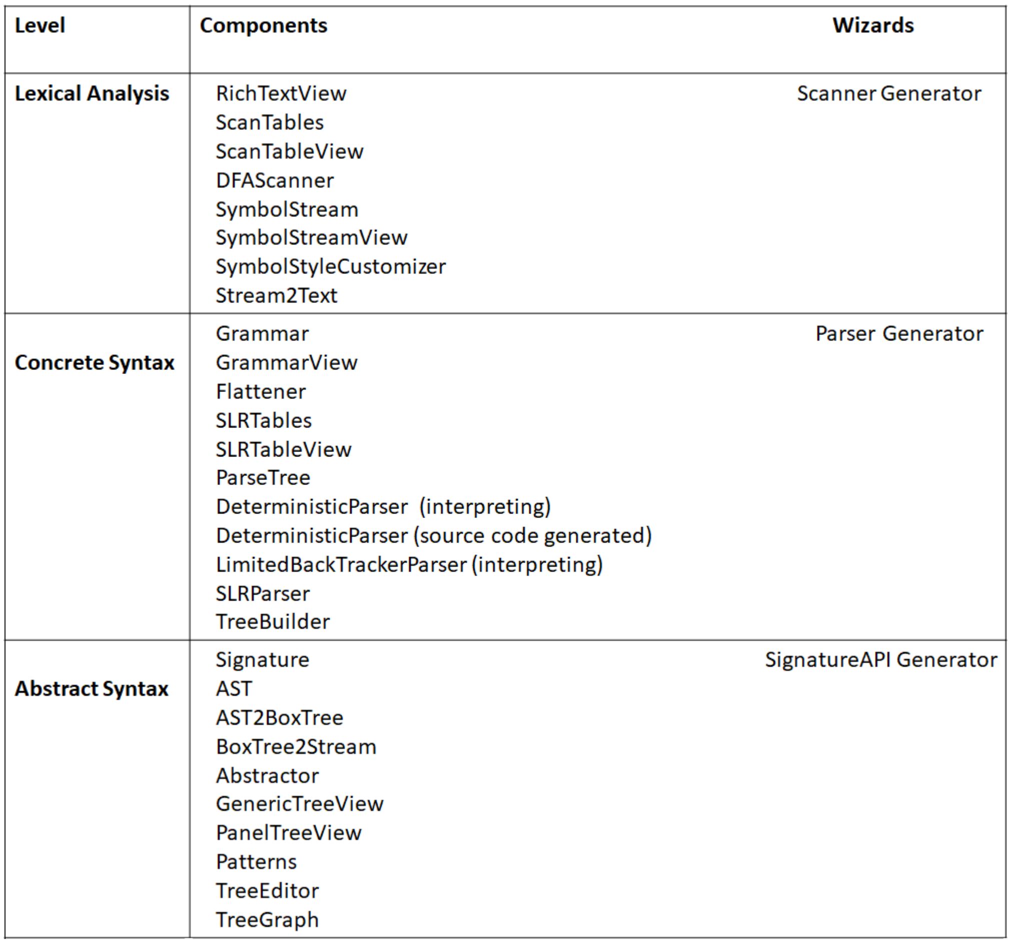
List of Tables

1. Introduction

1.1. The PAMOJA Framework

PAMOJA is a component framework for Grammar-Aware Engineering (GAE) consisting of components and wizards, written in Java, to be used in Integrated Development Environments (IDEs), such as NetBeans [?]. PAMOJA framework is being developed by Jackline Ssanyu as part of her PhD research work[1].

PAMOJA provides 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 several wizards for generating Java source-code from formal language specifications. Moving from concrete text to ASTs and back is divided into three levels, that is to say, lexical analysis, concrete syntax and abstract syntax. Therefore, to deal with each level, PAMOJA provides a collection of components for holding language specifications such as grammars and signatures, for holding data and mappings, 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. All source- code in PAMOJA is generated in Java programming language. In addition, PAMOJA provides composite components assembled from other components of the framework. Table 1 gives an overview of PAMOJA's basic components and wizards. In Section 3 is a brief description about PAMOJA components and wizards.



Level	Components	Wizards
Lexical Analysis	RichTextView	Scanner Generator
	ScanTables	
	ScanTableView	
	DFAScanner	
	SymbolStream	
	SymbolStreamView	
	SymbolStyleCustomizer	
	Stream2Text	
Concrete Syntax	Grammar	Parser Generator
	GrammarView	
	Flattener	
	SLRTables	
	SLRTableView	
	ParseTree	
	DeterministicParser (interpreting)	
	DeterministicParser (source code generated)	
	LimitedBackTrackerParser (interpreting)	
	SLRParser	
	TreeBuilder	
Abstract Syntax	Signature	SignatureAPI Generator
	AST	
	AST2BoxTree	
	BoxTree2Stream	
	Abstractor	
	GenericTreeView	
	PanelTreeView	
	Patterns	
	TreeEditor	
	TreeGraph	

Figure 1: PAMOJA Components and Wizards at lexical, concrete and abstract syntax.

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

PAMOJA is used for language front-end processing in a general-purpose framework. Typical applications include:

- language prototyping in a language laboratory,
- experimenting with various compiler front-end algorithms in an educational environment,
- implementation of front-ends for programming languages,
- integration of language processing in all kinds of applications which are not necessarily compilers.

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. Therefore, 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 and signatures are used and no other special-purpose languages. To get an impression of what the language specifications look like, see the specifications for GCLSharp (Guarded Command Language) presented in Appendix 6.
- Faster and easier generation of front-end applications (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 a hierarchy of 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 [?], [?], and [?] provide a good introduction to these topics. Section 2 of this manual describes installation procedures for PAMOJA in the NetBeans IDE. Section 3 briefly provides some highlights on PAMOJA components and wizards. An example (developing a language front-end) demonstrating the use of several PAMOJA components and wizards is presented in Section 4. Sections 5 and **??** provides the currently known bugs/deficiencies and licence information respectively.

2. Installing PAMOJA

PAMOJA has been tested with NetBeans 7.1 and earlier versions including NetBeans 12.0.

Note: PAMOJA components are packaged as JavaBeans [?] in a JAR (Java Archive) file and could be used in IDEs where drag and drop is used but the wizards (i.e. the scanner, parser and signature API generator) are each packaged in a NBM (NetBeans module) file hence netbeans specific. This manual is tied to NetBeans.

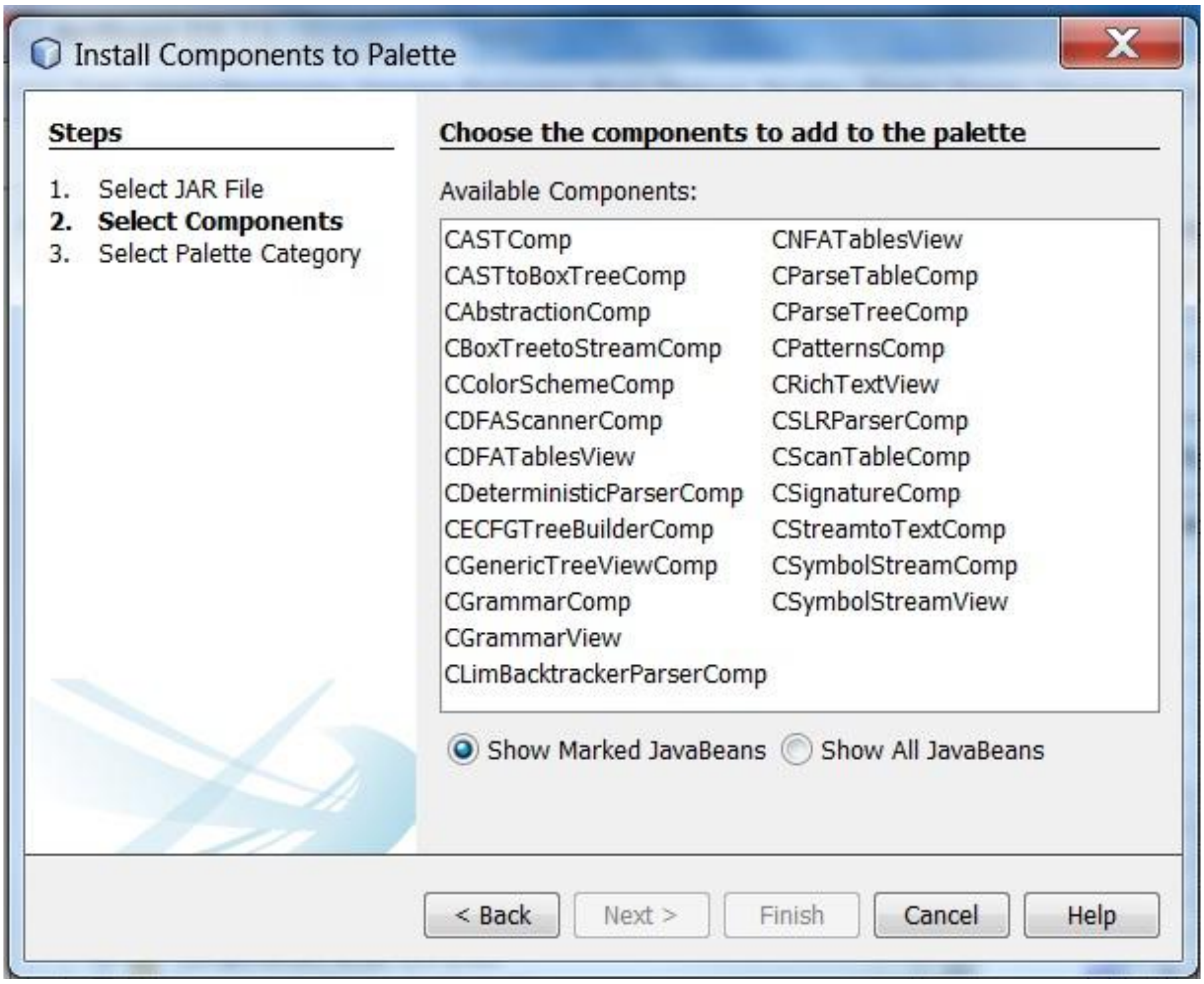
PAMOJA Installation files can be downloaded from https://github.com/ssanyu/PAMOJA/tree/master/ins

NetBeans [?] 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 JavaBeans 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 2 is displayed showing a list of the classes for 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 3.



Install Components to Palette

Steps

1. Select JAR File
2. **Select Components**
3. Select Palette Category

Choose the components to add to the palette

Available Components:

CASTComp	CNFATablesView
CASTtoBoxTreeComp	CParseTableComp
CAbstractionComp	CParseTreeComp
CBoxTreetoStreamComp	CPatternsComp
CColorSchemeComp	CRichTextView
CDFAScannerComp	CSLRParserComp
CDFATablesView	CScanTableComp
CDeterministicParserComp	CSignatureComp
CECFGTreeBuilderComp	CStreamtoTextComp
CGenericTreeViewComp	CSymbolStreamComp
CGrammarComp	CSymbolStreamView
CGrammarView	
CLimBacktrackerParserComp	

[✓ Show Marked JavaBeans] Show All JavaBeans

Figure 2: Window showing classes representing available PAMOJA components.

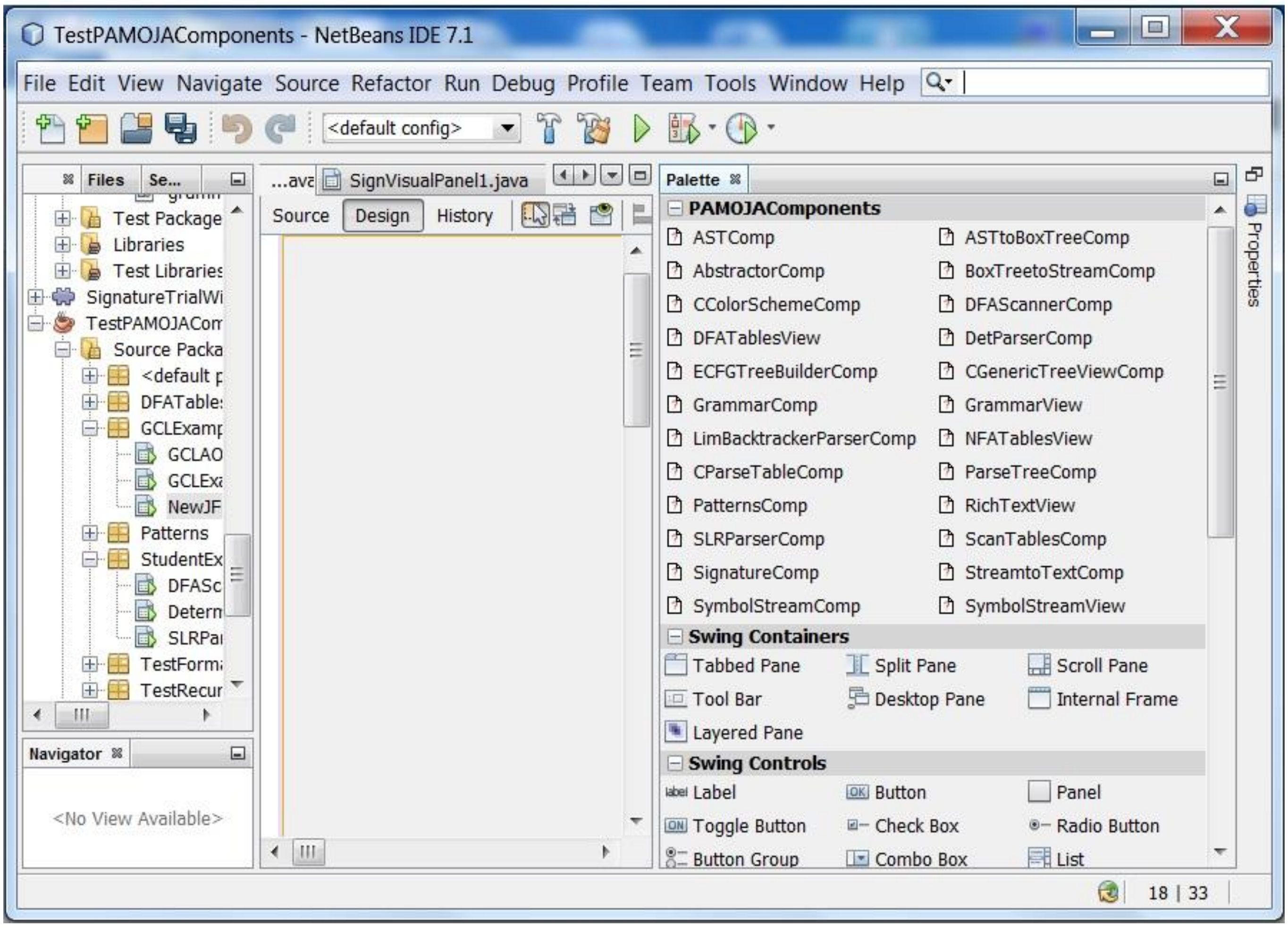


Figure 3: Component palette on the right-hand side of NetBeans IDE.

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. PAMOJA Components and Wizards

As mentioned already in Section 1.1, PAMOJA offers components and wizards (see Figure 1) for front-end processing.

For the description of the tasks, features and information about the implementation (API documentation) of PAMOJA components, refer to the Javadoc [?] of PAMOJA.

PAMOJA currently provides three wizards: the *scanner generator wizard* which generates source-code for an ELL(1)-based scanner, the *parser generator wizard* which generates source-code for a deterministic recursive descent parser and *signature API generator wizard* which generates a hierarchy of Java AST classes for a given language.

4. Developing a Language Front-end using PAMOJA

In this section we list the general steps required to develop a language front-end using PAMOJA. Then, we illustrate the use of PAMOJA with an example for constructing a front-end for GCLSharp, a simplified version of the GCL [?] language. A front-end for GCLSharp consists of a mapping from a GCL program to an abstract syntax tree.

4.1. General Steps

Constructing a front-end using PAMOJA in the NetBeans IDE requires the following general steps:

1. Creating language specifications which include: signature specifications and grammar specifications (i.e., lexical and context free grammar) corresponding to the signature.
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 abstract syntax tree.

8. Constructing a formatter which involves:

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

4.2. Example:Developing a Front-end for GCLSharp

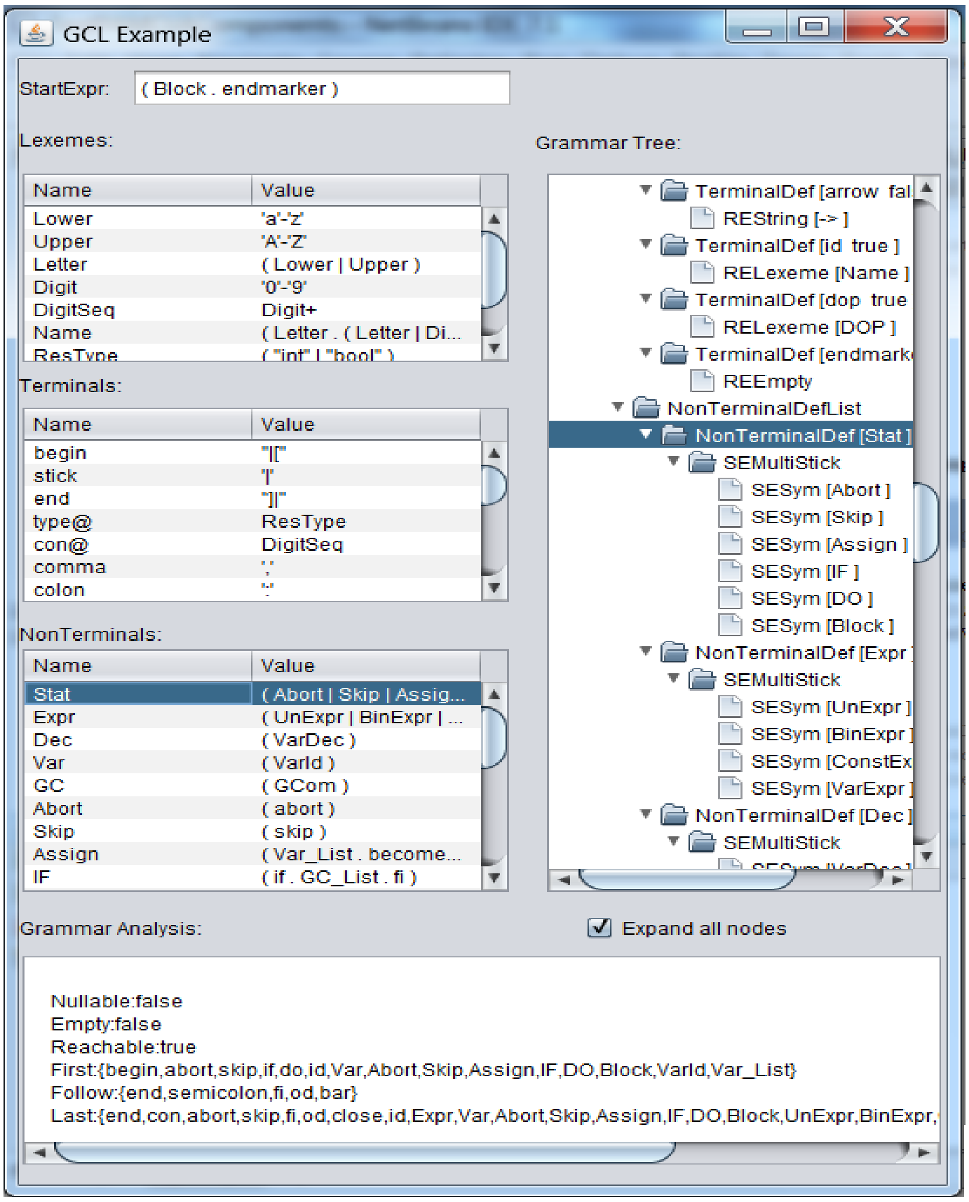
To develop a language front-end using PAMOJA components we follow the following steps:

1. Create a signature and a grammar for GCLSharp. See Appendix A 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 4.
   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.

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.



StartExpr: (Block . endmarker)

Lexemes:

Name	Value
Lower	'a'-'z'
Upper	'A'-'Z'
Letter	(Lower | Upper)
Digit	'0'-'9'
DigitSeq	Digit+
Name	(Letter . (Letter | Di...
ResType	("int" | "bool")

Terminals:

Name	Value
begin	"|["
stick	'|'
end	"]|"
type@	ResType
con@	DigitSeq
comma	','
colon	':'

NonTerminals:

Name	Value
Stat	(Abort | Skip | Assig...
Expr	(UnExpr | BinExpr | ...
Dec	(VarDec)
Var	(VarId)
GC	(GCom)
Abort	(abort)
Skip	(skip)
Assign	(Var_List . become...
IF	(if . GC_List . fi)

Grammar Tree:

```
    TerminalDef [arrow  fal
      REString [-> ]
    TerminalDef [id  true ]
      RELexeme [Name ]
    TerminalDef [dop  true
      RELexeme [DOP ]
    TerminalDef [endmark
      REEmpty
  NonTerminalDefList
    NonTerminalDef [Stat ]
      SEMultiStick
        SESym [Abort ]
        SESym [Skip ]
        SESym [Assign ]
        SESym [IF ]
        SESym [DO ]
        SESym [Block ]
    NonTerminalDef [Expr ]
      SEMultiStick
        SESym [UnExpr ]
        SESym [BinExpr ]
        SESym [ConstEx
        SESym [VarExpr ]
    NonTerminalDef [Dec ]
      SEMultiStick
        SESym [VarDec ]
```

Grammar Analysis: ✓ Expand all nodes

```
Nullable:false
Empty:false
Reachable:true
First:{begin,abort,skip,if,do,id,Var,Abort,Skip,Assign,IF,DO,Block,VarId,Var_List}
Follow:{end,semicolon,fi,od,bar}
Last:{end,con,abort,skip,fi,od,close,id,Expr,Var,Abort,Skip,Assign,IF,DO,Block,UnExpr,BinExpr,
```

Figure 4: A view showing grammar definitions and analysis information for a `STAT` symbol.

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:

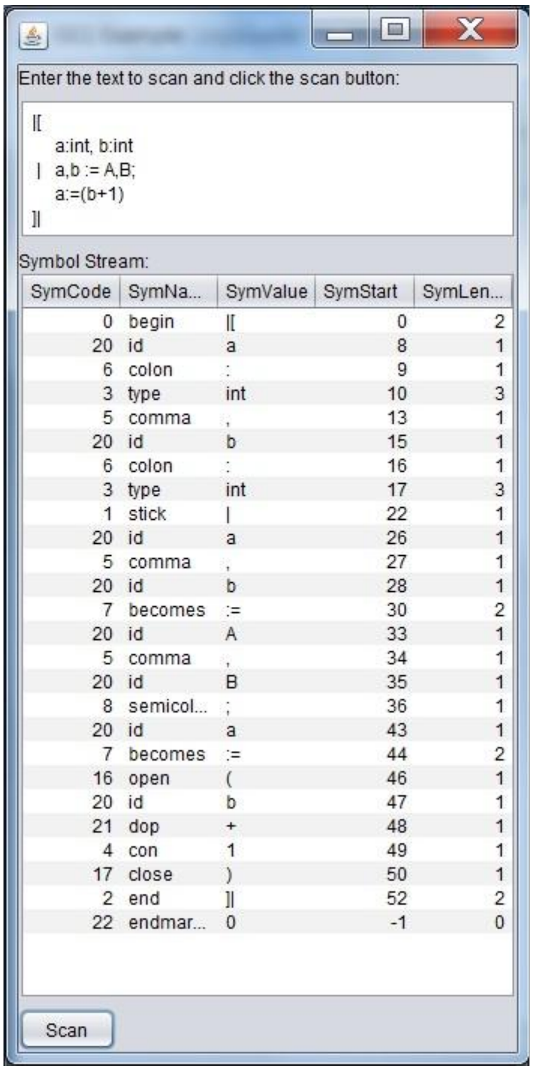
```
private void Scan1ActionPerformed(java.awt.event.ActionEvent evt){
    // TODO add your handling code here:
}
```

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 5.



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

Figure 5: A form showing program text and corresponding symbolstream.

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()){
        ParseResult.setText("Success");
     }else{
        ParseResult.setText("Parse␣failed␣at:␣"+v.toText()+Determini
              }
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 6.

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.

      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.

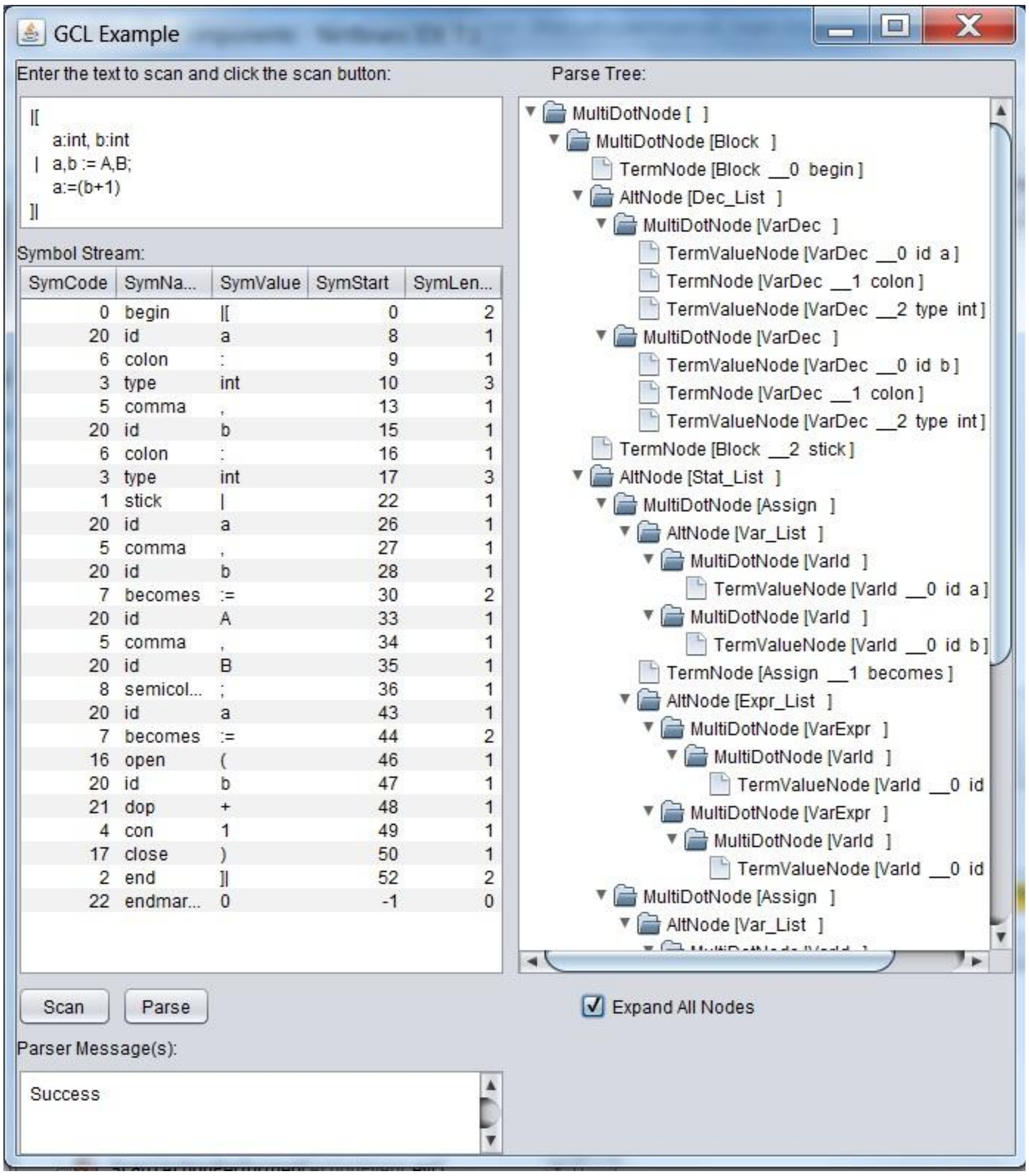
      vi. Enter the name of the package in which to store the generated files. If every thing is done correctly, the wizard window will look something similar to Figure 7.

      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.


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 ]
```

Scan Parse

[✓] Expand All Nodes

Parser Message(s):

Success

Figure 6: A form showing a program for Simple GCL and the corresponding Parse Tree.

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

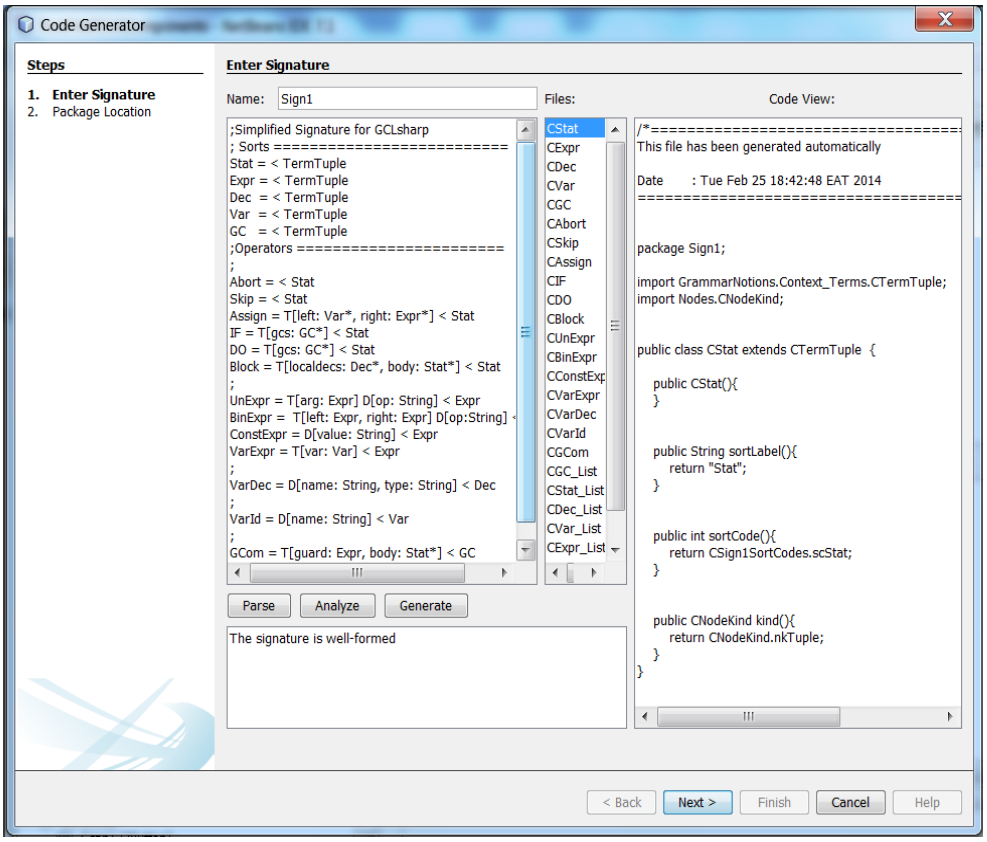
f) Double click **Generate AST** button and add the following lines of code to its corresponding event handler method declaration:

```
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 8



Name: Sign1

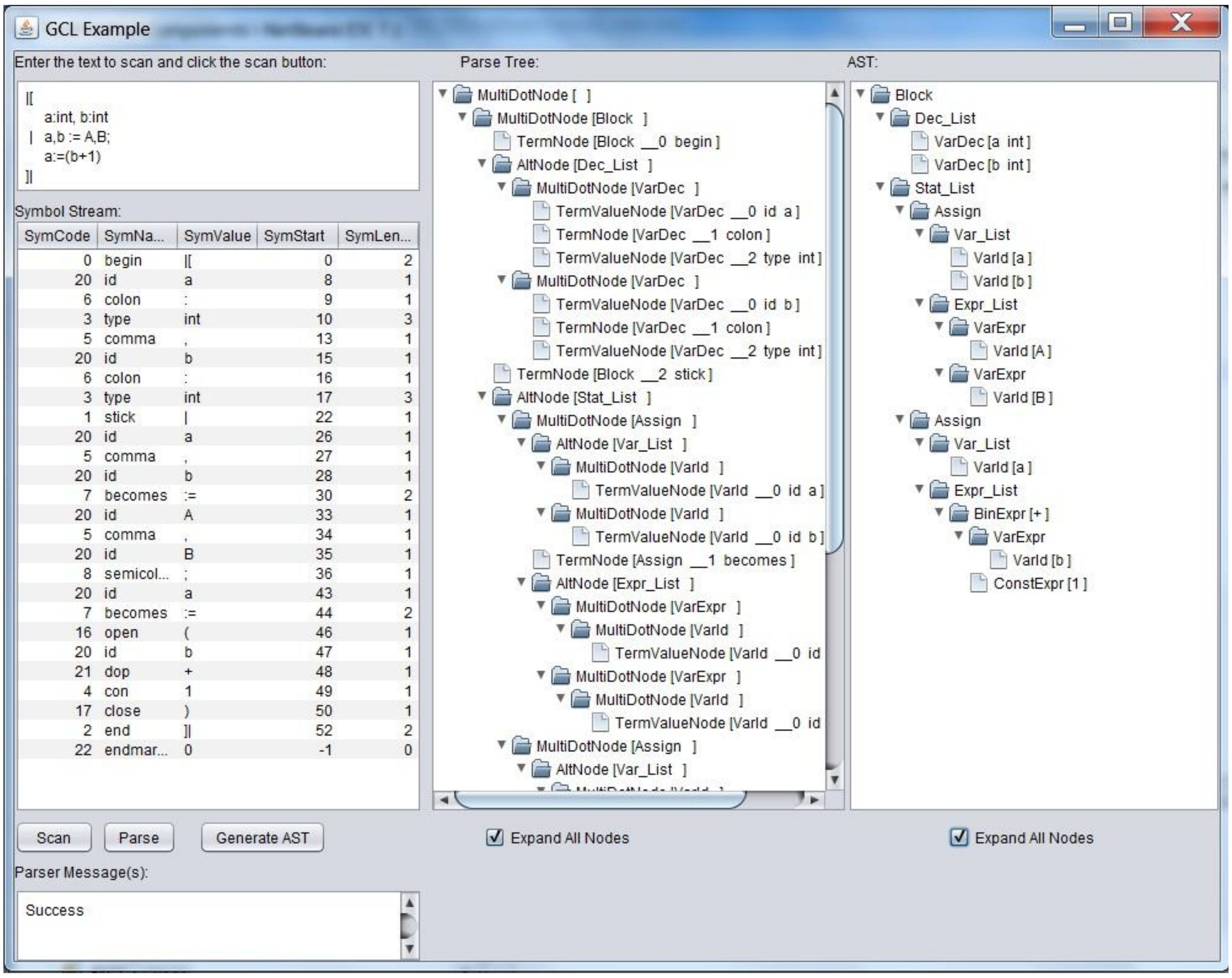
```
;Simplified Signature for GCLsharp
; Sorts ==========================
Stat = < TermTuple
Expr = < TermTuple
Dec  = < TermTuple
Var  = < TermTuple
GC   = < TermTuple
;Operators =======================
;
Abort = < Stat
Skip = < Stat
Assign = T[left: Var*, right: Expr*] < Stat
IF = T[gcs: GC*] < Stat
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] <
ConstExpr = D[value: String] < Expr
VarExpr = T[var: Var] < Expr
;
VarDec = D[name: String, type: String] < Dec
;
VarId = D[name: String] < Var
;
GCom = T[guard: Expr, body: Stat*] < GC
```

```
CStat
CExpr
CDec
CVar
CGC
CAbort
CSkip
CAssign
CIF
CDO
CBlock
CUnExpr
CBinExpr
CConstExp
CVarExpr
CVarDec
CVarId
CGCom
CGC_List
CStat_List
CDec_List
CVar_List
CExpr_List
```

```
/*===================================
This file has been generated automatically

Date     : Tue Feb 25 18:42:48 EAT 2014
====================================


package Sign1;

import GrammarNotions.Context_Terms.CTermTuple;
import Nodes.CNodeKind;


public class CStat extends CTermTuple  {

   public CStat(){
   }


   public String sortLabel(){
      return "Stat";
   }


   public int sortCode(){
      return CSign1SortCodes.scStat;
   }


   public CNodeKind kind(){
      return CNodeKind.nkTuple;
   }
}
```

The signature is well-formed

Figure 7: A form showing signature definitions, a list of names for the generated Java files and sample code for a sort `Stat`.


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 8:

5. Bugs and Deficiencies

[To be completed]

6. Copying and Licence

[To be completed]

Appendix A 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 B 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
```