“Petri Net Diagram Compiler”

Andrea Quattri[1], Antonio Cosseddu[2] & Giovanni Battista Nava[3]
Computer Engineering, University of Bergamo, Italy

[1] and.quattri@gmail.com
[2] ant.cosseddu@gmail.com
[3] giovanni.b.nava@gmail.com

Contents

List of Figures

Chapter 1

Petri Net Diagram Compiler

1.1 Introduction

This project has been developed as completion of the Language and Compilers course by professor Giuseppe Psaila, Computer Engineering, University of Bergamo, Italy. The subjects of this project fall within the course's field, in particular they focuses on the specific aspects of the arguments treated. Through the development of this project, important matters have been treated such as predefined grammar structures recognition, syntactic analysis (parsing), semantic structure definition, different grammar reconstruction and management of exceptions, most and more of the programme of the course of Language and Compilers.

The purpose of the project is to create a simple language that allows users to generate a diagram of a given Petri net represented in a text format. In particular, we have chosen this type of modeling because Petri nets are among the most use nets in software engineering.

The source code is available in the "Files" section of project's website:

`https://sourceforge.net/projects/petri-net-diagram-compiler/`

1.2 Aims

The goal of our project is to generate the diagram requested by the user and to export it in a convenient format like PDF. To facilitate the user we created instructions in

a pseudo-natural language, very intuitive and simple. Furthermore, the program is able to perform syntactic and semantic checks so as to realize the diagram only if the entered code is completely correct.

To make easier for the user to interact with the compiler we have a user-friendly interface. Furthermore the output, both in the case of syntactic or semantic errors both in the case of compilation performed successfully, has the same format of Eclipse, to make clear to the user the row and column in which there is an error and which token or tokens must review.

The end result is the image of the network inserted. To achieve it we relied on a LaTeX engine that has presented itself as the most effective tool to achieve our goal. The library used makes possible to export the network to pdf image having as input a string that describes the typical components of this diagram. This string is not, however, in natural language and then our program, after compiling, takes care of translating the instructions written in our language into the instructions accepted by LaTeX.

1.3 Background knowledge for the development

To implement the project some preliminary knowledge is required like BNF grammars, lexical, syntactic and semantic structures.

A compiler can be written in many programming languages. We chose to use Java and to develop the project in Eclipse adding two plugs-in:

- **ANTLR**: for the generation of LEXER and PARSER components in a semi-automatic manner. In fact, from the compilation of a file ".g", where it is established the grammar and syntax of the whole language, two classes are automatically generated that make possible the realization of Lexer and Parser typical functions.
- **WindowBuilder**: for the generation of graphical interactive windows in order to improve the user experience. The interface of our program is meant to be visual and not on the command line using an Eclipse console.

Its also required a good knowledge of the language LaTeX in order to achieve a consistent and standardized output format. In particular the knowledge of the TikZ

library was necessary to allow the realization of visual elements in LaTeX. This project intends to generate a valid file .tex that represents the Petri Net. This .tex will be passed later to the latex engine.

Chapter 2

Software requirements

2.1 System settings

For the smooth running of the program flow a preparatory step is necessary, in which the LaTeX engine must be installed, in order to allow the automatic start of the program that generates the PDF. This software is available at the following website: `http://miktex.org/download`. If you are working on a 64-bit machine enter the section "Other Downloads" and download all the proposed packages. Check for the pdflatex.exe program within the computer, if installed with the default settings it will be in the folder specified by the path:

C:\Program Files\MiKTeX 2.9\miktex\bin\x64\pdflatex.exe

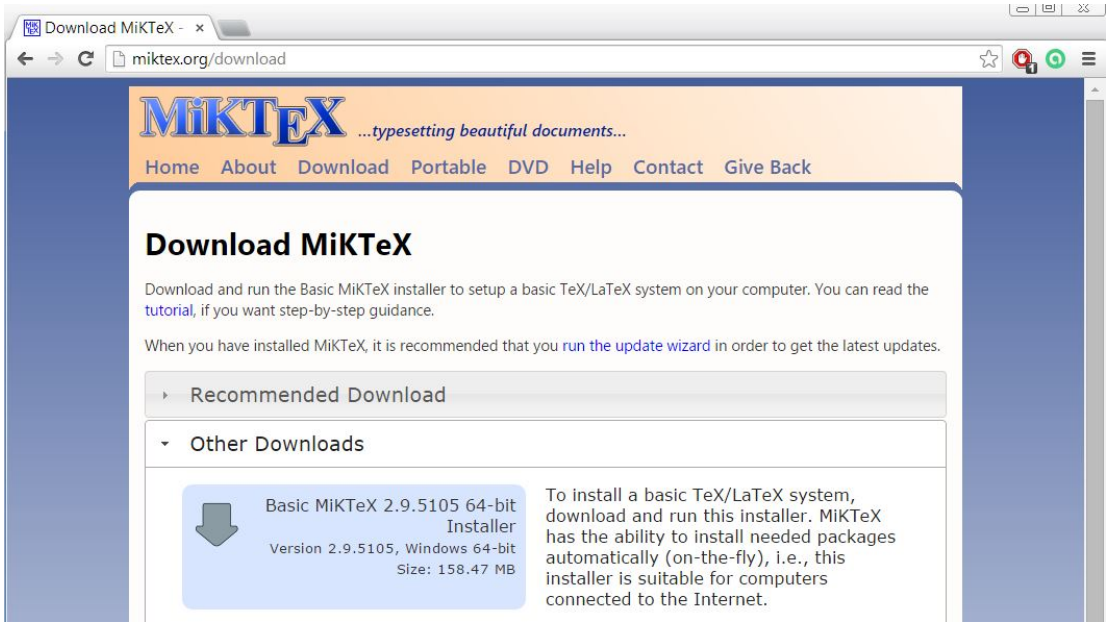

Figure 2.1: URL of the MiKTEX engine.

This will have to be the default opening application for files .tex.

2.2 ANTLR installation in Eclipse

To install and configure ANTLR in Eclipse you need to follow these steps:

- Within the editor, click Support/Install new software by selecting the tool to the URL `http://download.eclipse.org/technology/dltk/updates-dev/3.0-stable/`; check the Dltk package box and continue with the installation;
- Within the editor, click Support/Install new software selecting the tool to the URL`http://antlrv3ide.sourceforge.net/updates`; check the ANTLR IDE box and execute the installation;
- Download the ANTLR library from the Web (recommended and tested the **version 3.4**);
- Save the Jar file in the Eclipse folder (recommended);
- Add the library you just downloaded from the menu:

  Window/Preferences/ANTLR/Builder by clicking on the add button and selecting the folder in which you downloaded ANTLR 3.4;

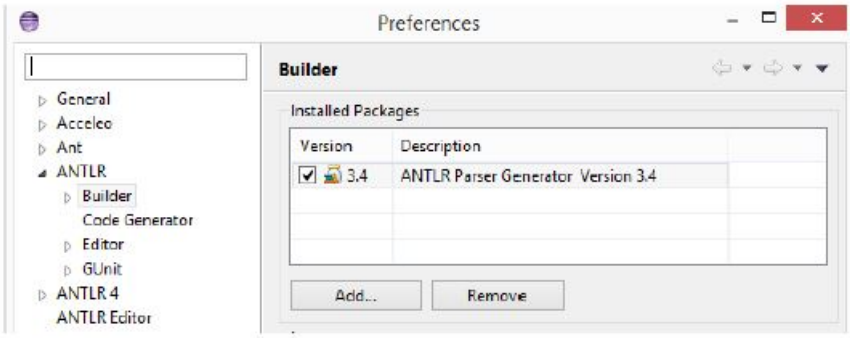

Figure 2.2: Navigation menu window of Eclipse.

- Edit an entry in the section Code Generator: in *Window/Preferences/ANTLR/-CodeGenerator*, check the "append java package to folder output";

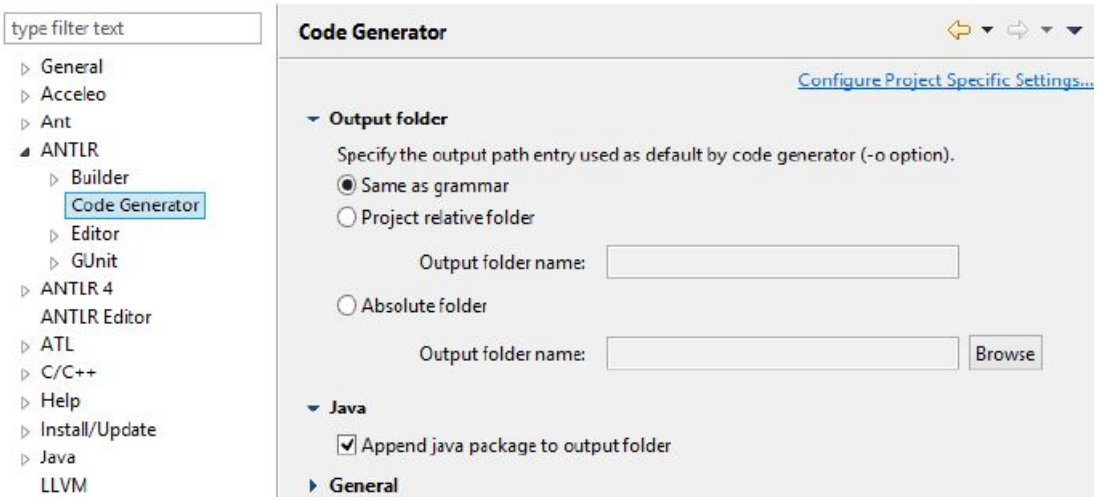

Figure 2.3: Java language settings.

- Create a new Java Project by giving it a name (exactly as if we were operating with a normal Java project);

- Right-click on the project by selecting *Configure/Convert to ANTLR Project*;

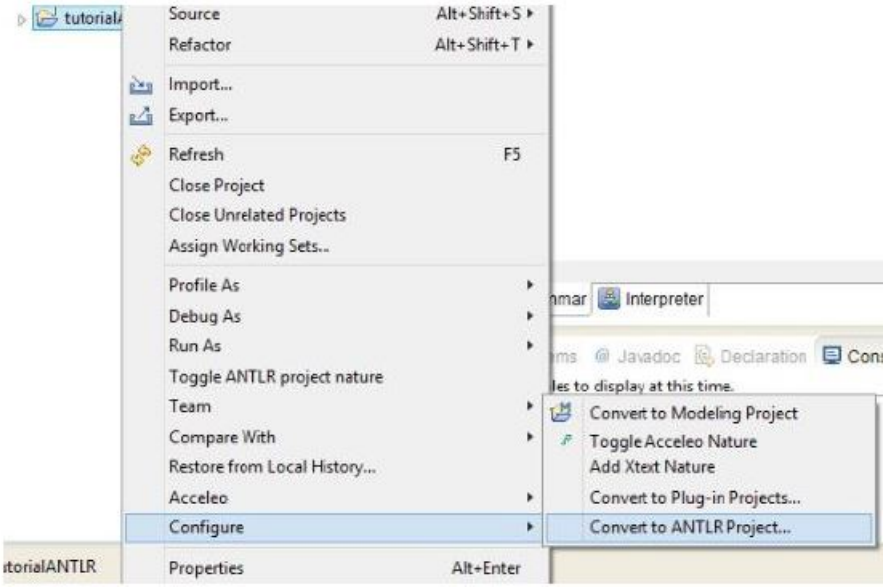

Figure 2.4: Configuring Java project -> ANTLR/Java project.

- Add the library ANTLR 3.4 to the project: Right click on the project Build *path/Add external Archievs*, and select the library antlr-3.4-complete;

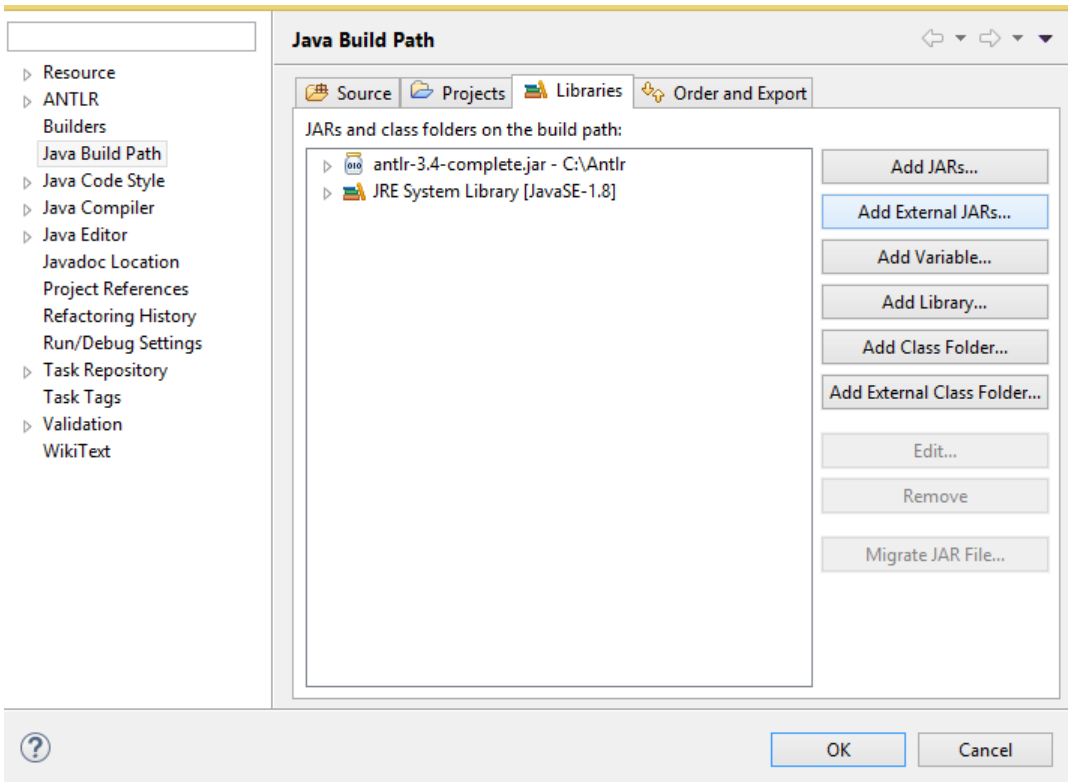

Figure 2.5: Library antlr-3.4-complete.

- Create a file .g which will contain the grammar: right click on the src project

and select *new/other/ANTLR/combined grammar*;

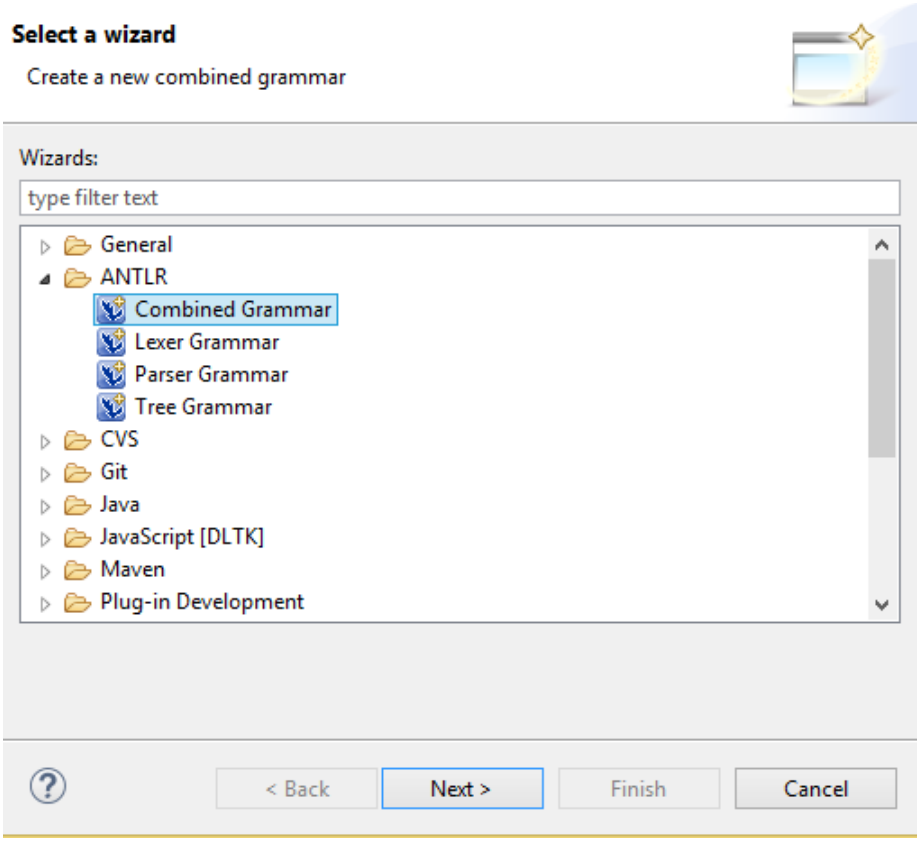

Figure 2.6: Creation file .g.

- Give a name to the file just created which, when saved, will create two classes that depend on it: *nameLexer.java* and *nameParser.java*;
- Click on finish;

Lastly, right-click on the project, select *Properties/ANTLR/Code Generator* and select "*append java package to folder output*".

From this moment on, all the preparations can be considered completed, so you can start working within the specific and instantiate all the classes and packages needed to implement the information flow that constitutes the compiler.

Chapter 3

Language

This section will analyze the structure of the grammar that we decided to implement localized within the file "**language.g**". It's important to explain in a simple and effective manner both the input grammatical structure, both the output grammatical structure so as to facilitate the user who is going to utilize this compiler. We chose to make a Petri net composed of states, transitions and arrows linking the former to the latter.

3.1 Input

The input language must be similar to the natural language, with well defined fields and a well-defined grammar.

In particular, we use well-defined constructs in a rigid grammar, based on the recognition of strings.

Noteworthy, the input language is declarative, in fact the aim is to declare a number of elements and connectors which are going to be checked syntactically and semantically, and only after this operation the LaTeX code output will be generated , to be used as input of the LaTeX engine.

We decided to make the input not case sensitive because it can happen that the user mistypes or simply does not care how to insert the code strings.

We have also established a field (the label) where the user can enter characters of his choice, since these are simple text strings that need to be completely customizable by the user.

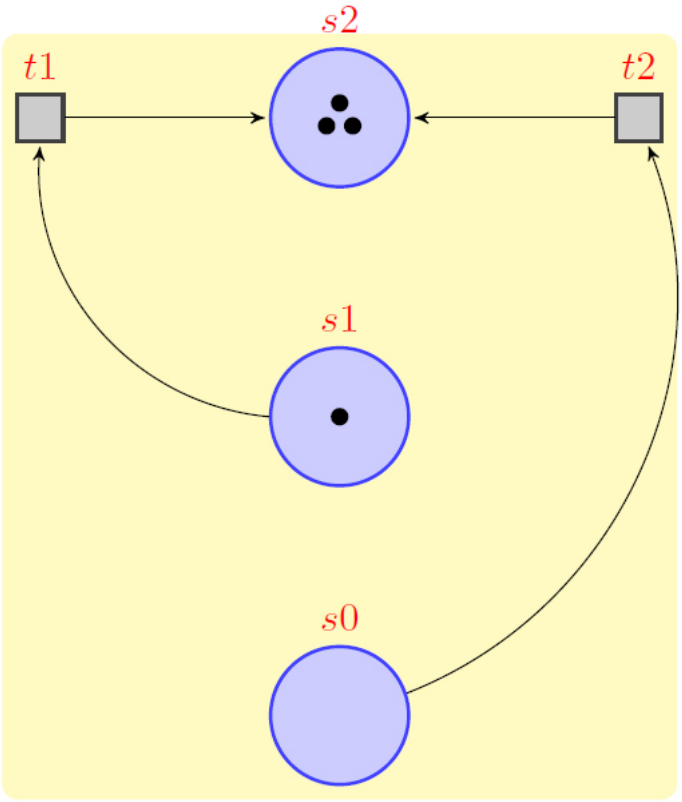

Figure 3.1: Simple example of the output diagram.

3.1.1 Input Structure

The input, is characterized by a series of commands that the interpreter will evaluate and execute sequentially.

```
draw_start Start "s0";
DRAW_STATE BELOW_OF Start State1 "s1" (1);
DRAW_STATE ABOVE_OF Start State2 "s2" (3);
DRAW_STATE BELOW_OF State1 State3 "s3" (0);
DRAW_STATE BELOW_OF State3 State4 "s4" (2);
DRAW_STATE ABOVE_OF State2 State5 "s5" (1);

DRAW_TRANSITION LEFT_OF Start Transition1 "t1";
DRAW_TRANSITION RIGHT_OF Start Transition2 "t2";
DRAW_TRANSITION LEFT_OF State3 Transition3 "t3";
DRAW_TRANSITION RIGHT_OF State3 Transition4 "t4";
DRAW_TRANSITION RIGHT_OF State5 Transition5 "t5";
```

```
DRAW_ARROW Transition1 Start;
DRAW_ARROW Start Transition2;
DRAW_ARROW Transition3 State3;
DRAW_ARROW State3 Transition4;
DRAW_ARROW State5 Transition5;
DRAW_CURVED_ARROW Transition1 State5 (left);
DRAW_CURVED_ARROW Transition5 State2 (left);
DRAW_CURVED_ARROW State2 Transition1 (right);
DRAW_CURVED_ARROW Transition2 State2 (right);
DRAW_CURVED_ARROW Transition1 State1 (right);
DRAW_CURVED_ARROW State1 Transition2 (right);
DRAW_CURVED_ARROW Transition3 State1 (left);
DRAW_CURVED_ARROW State1 Transition4 (left);
DRAW_CURVED_ARROW Transition4 State4 (left);
DRAW_CURVED_ARROW State4 Transition3 (left);
```

Is therefore essential to declare the objects you intend to use before you refer to them, which means that if I create a *<State>* component referring to another component *<State>* that I declare after it, the semantic analyzer will detect an error Missing Reference.

Each instruction must start with a *<draw_component>* and end with the terminator *<;>*. The inside part is constituted by a series of field optional / mandatory according to the structural specifications that will be revealed in the next section.

The example was presented written on multiple lines, but this is not necessary. You could write down all the instructions on the same line if they respect the rules set out above.

The compiler is in fact set to ignore “falls” and “carriage”; regarding the management of spaces within each instruction, they are parsed and considered the separator of two tokens.

```
// Blank spaces and indentation
WS  : (
        ' '
        | '\t'
        | '\r'
        | '\n'
      )
```

```
    {
      $channel=HIDDEN;
    };
```

The **minimum structures** that the compiler can accept consist of:

- A particular state called **START**;
- One **STATE**;
- One **TRANSITION**;
- One **CONNECTOR**, made by two **ARROW**;

```
DRAW_START Start "s0";
DRAW_STATE BELOW_OF Start State1 "s1" (1);

DRAW_TRANSITION LEFT_OF Start Transition1 "t1";

DRAW_ARROW Start Transition1;
DRAW_ARROW Transition1 State1;
```

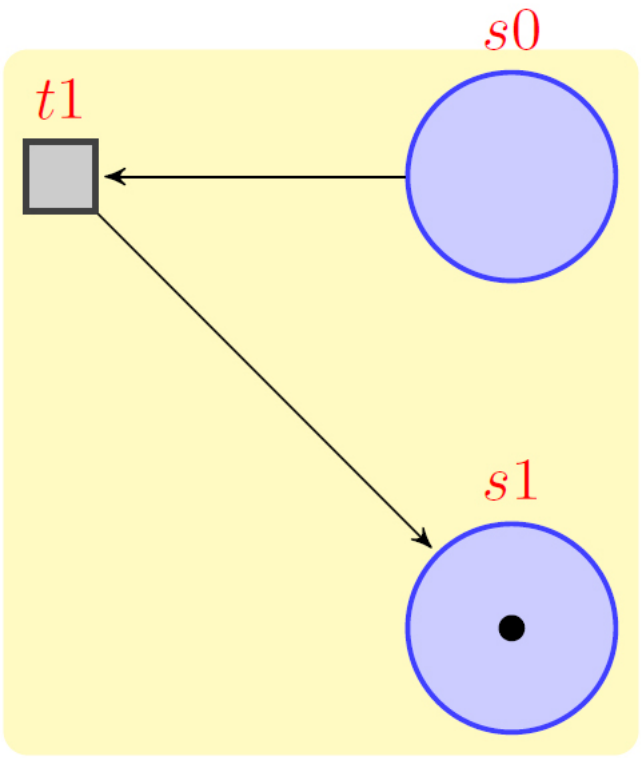

Figure 3.2: Example of minimum structure of the Petri net.

3.1.2 Input Syntax

We come now to detail the syntax that the compiler is able to accept, considering all the constituent elements of every possible instruction which is defined and recognized by the grammar. To achieve our purpose we will make use of "*RailRoad View*" provided by the environment ANTLR; a sort of flow chart of all the non-terminals and terminals of the grammar, starting with the start-rule down along the whole tree.

3.1.2.1 Petri net

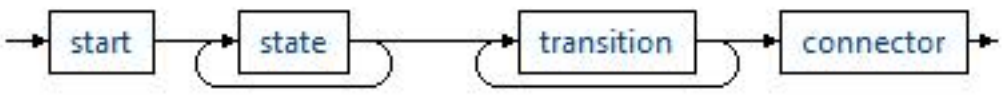

Figure 3.3: RailRoad View of Petri net rule.

```
// start_rule
petri_net
@init
{
  // Init the environment
  init();
}
:
  start
  state+
  transition+
  connector
;
```

This is the start-rule of the whole tree.

It indicates that there must be present a Start, one or more states, one or more transitions and a connector. The latter consists of at least two arrows.

3.1.2.2 Start

Figure 3.4: RailRoad View of Start rule.

```
start: DRAW_START
       na=NAME
       lb=LABEL?
       tok=TOKENS?
       SEMICOL
       {
          // Create a start component and insert into
             petriNet
          String name = na.getText();
          String label = "";
```

```
        String tokens = "";

        if (lb != null)
          label = Model.cutDoubleQuote(lb.getText());

        if(tok != null)
          tokens = Model.cutParenthesis(tok.getText());

        Start start = new Start(name, label, tokens);

        env.petriNet.add(start);
        env.stateNames.add(name);
    }
    ;
```

Start is the rule that identifies the opening object of the block diagram, and it's a particular state of the Petri net because it is the first around which the rest of the network will be developed; it is constituted by:

- **Draw_start**: indicates the command that expresses the intention to draw an object of type start;
- **Name**: indicates the name of the object;
- **Label**: indicates an identifying label of the object that can be displayed on screen. It must begin with the character " and end with the character ";
- **Token**: indicates the number of tokens within the state;
- **Semicol**: indicates the instruction terminator <;>;

Now in detail the productions that make the start rule.

Name

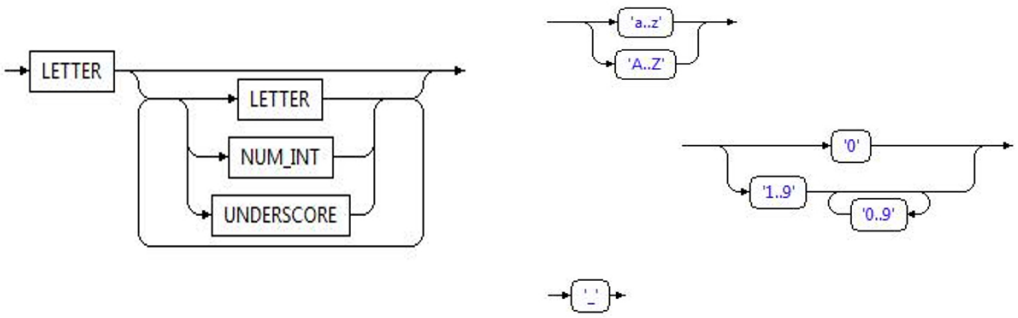

Figure 3.5: RailRoad View of NAME rule.

```
NAME:
  LETTER (LETTER | NUM_INT | UNDERSCORE)*;
```

The name of a component must start with a *LETTER*. Following it there can be a series of *LETTER*, *NUM_INT* and *UNDERSCORE*.

Clearly the *NAME* can consist of just a *LETTER*. *LETTER* is a character of the english alphabet both lowercase and uppercase; *NUM_INT* is an integer consisting of one or more digits while *UNDERSCORE* is used to split the words.

Label

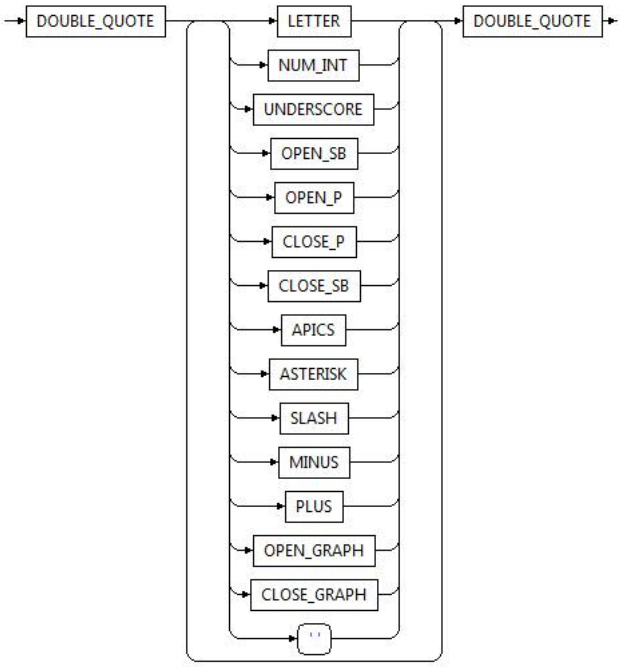

Figure 3.6: RailRoad View of LABEL rule.

```
LABEL:
      ( DOUBLE_QUOTE
      ( LETTER
      | NUM_INT
      | UNDERSCORE
      | OPEN_SB
      | OPEN_P
      | CLOSE_P
      | CLOSE_SB
      | APICS
      | ASTERISK
      | SLASH
      | MINUS
      | PLUS
      | OPEN_GRAPH
      | CLOSE_GRAPH
      | ' '
      )+
      DOUBLE_QUOTE )
```

```
    ;
```

A LABEL, which will be an attribute displayed in the net, is an optional attribute. If it's present, it must be written in quotation marks (*DOUBLE_QUOTE*). The inside part can contain one or more characters like *LETTER*, *NUM_INT*, *UNDERSCORE* and other typical characters.

Tokens

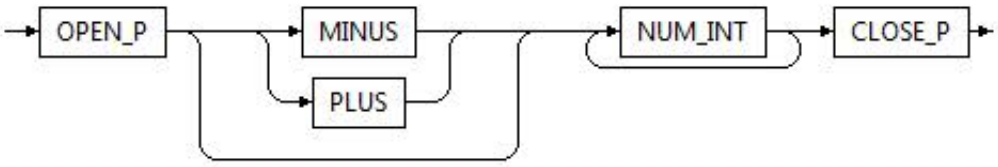

Figure 3.7: RailRoad View of TOKENS rule.

```
TOKENS:
      ( OPEN_P

      (MINUS | PLUS)?

      ( NUM_INT )+
      CLOSE_P )
      ;
```

A *TOKENS* is a production that shows the tokens contained in a component. It is a number enclosed in parentheses.

3.1.2.3 State

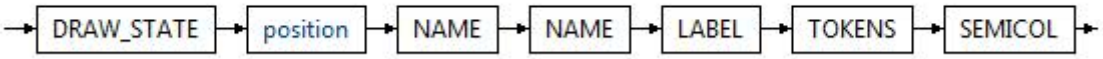

Figure 3.8: RailRoad View of STATE rule.

```
state:
      DRAW_STATE
      po=position
      naref=NAME
      nam=NAME
      lab=LABEL?
      tok=TOKENS?
      SEMICOL
      {
        // Create a state component and insert into petriNet
        String position = po;
        String nameRef = naref.getText();

        String name = nam.getText();
        String label = "";
        String tokens = "";

        if (lab != null)
          label = Model.cutDoubleQuote(lab.getText());

        if(tok != null)
          tokens = Model.cutParenthesis(tok.getText());

        Component state = new State(position, nameRef, name,
           label, tokens);

        env.petriNet.add(state);
        env.stateNames.add(name);
      }
      ;
```

State is the rule that identifies the object state that is a circle with a name and a

label that may contain tokens.

It is constituted by:

- **Draw_state**: indicates the command that expresses the intention to draw an object of state type;
- **Position**: indicates the command that expresses the relative position of the object in reference to another object (already drawn);
- **Name**: the first name indicates the object identifier to which the state is positioned relatively;
- **Name**: the second name indicates the identifier of the state that we are drawing;
- **Label**: indicates an identifying label of the object that can be displayed on screen. It must begin with the character " and end with the character ";
- **Semicol**: indicates the instruction terminator <;>;

Now you can see in detail the productions which make up the rule state.

Position

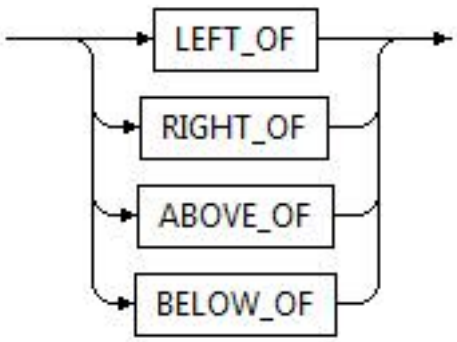

Figure 3.9: RailRoad View of POSITION rule.

```
position returns [String s]
        :
```

```
        (
          l=LEFT_OF {s = l.getText();}
          |
          r=RIGHT_OF {s = r.getText();}
          |
          a=ABOVE_OF {s = a.getText();}
          |
          b=BELOW_OF {s = b.getText();}
        )
        ;
```

A POSITION is a production that shows the relative position of a component in reference to another one. It is a string that can be one of these values:

- LEFT_OF;
- RIGHT_OF;
- ABOVE_OF;
- BELOW_OF;

Name

See the previous paragraph NAME of Start rule.

Label

See the previous paragraph LABEL of Start rule.

Tokens

See the previous paragraph TOKENS of Start rule.

3.1.2.4 Transition

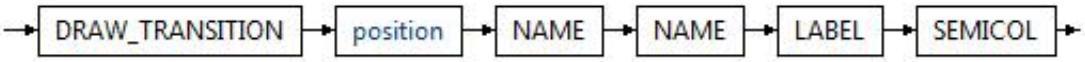

Figure 3.10: RailRoad View of TRANSITION rule.

```
transition:
    DRAW_TRANSITION
    pos=position
    narefe=NAME
    names=NAME
    lbl=LABEL?
    SEMICOL
    {
      // Create a transition and insert it into petriNet
      String position = pos;
      String nameRef = narefe.getText();

      String name = names.getText();
      String label = "";

        if (lbl != null)
          label = Model.cutDoubleQuote(lbl.getText());

      Component transition = new Transition(position, nameRef
         , name, label);

      env.petriNet.add(transition);
      env.transitionNames.add(name);

    }
    ;
```

Transition is the rule that identifies the transition object that is a rectangle with a name and a label. It will be reached by one or more arrows and / or one or more arrows will start from it.

It is constituted by:

- **Draw_transition**: indicates the command that expresses the intention to draw a transiting object type;
- **Position**: indicates the command that expresses the relative position of the object in reference to another object (already drawn);
- **Name**: the first name indicates the object identifier to which the transition is positioned relatively;
- **Name**: the second name indicates the identifier of the transition that we are drawing;
- **Label**: indicates an identifying label of the object that can be displayed on screen. It must begin with the character " and end with the character ";
- **Semicol**: indicates the instruction terminator <;>;

Now you can see in detail the productions which make up the rule transition.

Position

See the previous paragraph POSITION of State rule.

Name

See the previous paragraph NAME of Start rule.

Label

See the previous paragraph LABEL of Start rule.

3.1.2.5 Connector

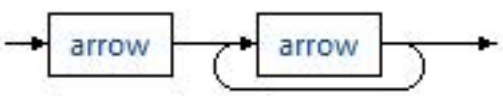

Figure 3.11: RailRoad View of CONNECTOR rule.

```
connector:
            arrow
            (arrow)+
            ;
```

The Connector is a series of arrows. In particular, the arrows must be at least 2.

3.1.2.6 Arrow

Figure 3.12: RailRoad View of ARROW rule.

```
arrow:
      (
        straight_arrow | curved_arrow
      )
      ;
```

The Arrow component can be of two types, straight and curved. This is to allow the user to better represent the network being able to choose whether to link the components described above with a straight or curved arrow.

3.1.2.7 Straight Arrow

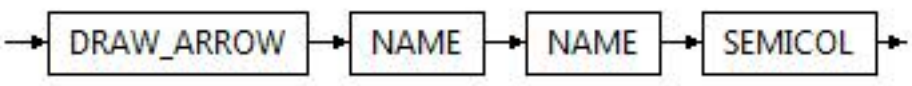

Figure 3.13: RailRoad View of STRAIGHT_ARROW rule.

```
straight_arrow:
               DRAW_ARROW
               or=NAME
               des=NAME
               SEMICOL
               {
                 // Create a straightArrow component and
                    insert into petriNet
                 Connector straightArrow = new StraightArrow(
                    or.getText(),des.getText());
                 env.petriNet.add(straightArrow);
                 env.referenceNames.add(or.getText());
                 env.referenceNames.add(des.getText());
               }
               ;
```

Straight Arrow is the rule that identifies the object Straight_arrow which is a straight arrow characterized by the two components that it will connect.

- **Draw_arrow**: indicates the command that expresses the intention to draw an object of type straight_arrow;
- **Name**: the first name indicates the identifier of the object from which the arrow will start;
- **Name**: the second name indicates the identifier of the object to which the arrow arrives;
- **Semicol**: indicates the instruction terminator <;>;

Now you can see in detail the productions which make up the rule straight_arrow.

Name

See the previous paragraph NAME of Start rule.

3.1.2.8 Curved Arrow

Figure 3.14: RailRoad View of CURVED_ARROW rule.

```
curved_arrow:
    DRAW_CURVED_ARROW
    ori=NAME
    dest=NAME
    be = bend
    SEMICOL
    {
        // Create a curvedArrow component and insert into
           petriNet
        String bend = "";

    if (be != null)
      bend = Model.cutParenthesis(be);
    Connector curvedArrow = new CurvedArrow(ori.getText(),
       dest.getText(), bend);
    env.petriNet.add(curvedArrow);
    env.referenceNames.add(ori.getText());
        env.referenceNames.add(dest.getText());
    }
    ;
```

Curved Arrow is the rule that identifies the object Curved_arrow which is a curved arrow that is characterized by the two components that it will connect and by the orientation of its curvature.

- **Draw_curved_arrow**: indicates the command that expresses the intention to draw an object of type curved_arrow;
- **Name**: the first name indicates the identifier of the object from which the arrow will start;

- **Name**: the second name indicates the identifier of the object to which the arrow arrives;
- **Bend**: is the orientation of the arrow curvature that can be left or right;
- **Semicol**: indicates the instruction terminator <;>;

Now you can see in detail the productions which make up the rule curved_arrow.

Name

See the previous paragraph NAME of Start rule.

Bend

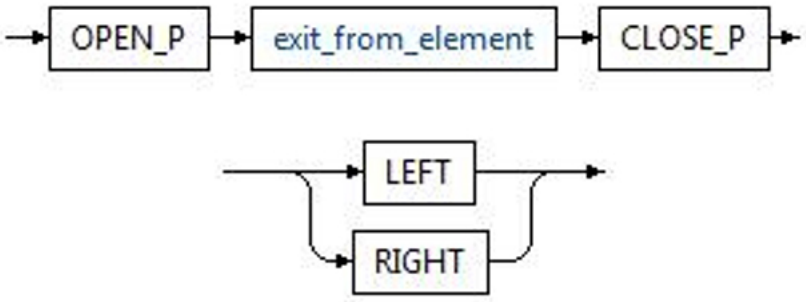

Figure 3.15: RailRoad View of BEND rule.

```
bend returns [String bend] :
(
    op = OPEN_P
    ex=exit_from_element
    cp=CLOSE_P
    {
    bend = "";
        bend += op.getText() + ex + cp.getText();
    }
)
;
```

A BEND is a production that shows the orientation of the arrow curvature. It is a string that can be one of these values:

- LEFT;
- RIGHT;

3.2 Output

For the output generation we use the LaTeX language that works with file whose extension is "*.tex*". Given that it is a real programming language, that we will run through the pdflatex application, our grammar will have to be converted to correspond to the commands recognized by that application.

Then the output to be processed and displayed in pdf also needs a method that opens and closes the file *.tex* necessary to the output visual formatting. This mechanism is build by means of a special class containing the constants and conversion methods, appropriately called and scheduled when necessary.

3.2.1 Output Structure

The output, as said, will be a *.tex* file. From the example of the previous input we get the following output to which follows a detailed explanation of the individual pieces of code that make up the file.

```
1  \documentclass{article}
2  \usepackage{tikz}
3  \usetikzlibrary{arrows,backgrounds,positioning,fit,petri}
4  \begin{document}
5  \begin{tikzpicture} [
6  node distance=2.6cm,on grid,>=stealth',
7  bend angle=45,
8  auto,
9  every place/.style= {minimum size=12mm,thick,draw=blue!75,fill=blue!20},
10 every transition/.style={thick,draw=black!75,fill=black!20},
11 every label/.style= {red}
12 ]
13 \node [place, tokens=0] (Start) [label=above:$s0$] {};
14 \node [place, tokens=1] (State1) [below=of Start,label=above:$s1$] {};
15 \node [place, tokens=3] (State2) [above=of Start,label=above:$s2$] {};
16 \node [place, tokens=0] (State3) [below=of State1,label=above:$s3$] {};
17 \node [place, tokens=2] (State4) [below=of State3,label=above:$s4$] {};
18 \node [place, tokens=1] (State5) [above=of State2,label=above:$s5$] {};
19 \node [transition] (Transition1) [left=of Start,label=above:$t1$] {};
20 \node [transition] (Transition2) [right=of Start,label=above:$t2$] {};
21 \node [transition] (Transition3) [left=of State3,label=above:$t3$] {};
22 \node [transition] (Transition4) [right=of State3,label=above:$t4$] {};
23 \node [transition] (Transition5) [right=of State5,label=above:$t5$] {};
```

Figure 3.16: Example of LaTeX code of a Petri net (1).

```
\draw [post] (Transition1) to (Start);
\draw [post] (Start) to (Transition2);
\draw [post] (Transition3) to (State3);
\draw [post] (State3) to (Transition4);
\draw [post] (State5) to (Transition5);
\draw [post] (Transition1) to [bend left] (State5);
\draw [post] (Transition5) to [bend left] (State2);
\draw [post] (State2) to [bend right] (Transition1);
\draw [post] (Transition2) to [bend right] (State2);
\draw [post] (Transition1) to [bend right] (State1);
\draw [post] (State1) to [bend right] (Transition2);
\draw [post] (Transition3) to [bend left] (State1);
\draw [post] (State1) to [bend left] (Transition4);
\draw [post] (Transition4) to [bend left] (State4);
\draw [post] (State4) to [bend left] (Transition3);
\begin{scope}[on background layer]
\node [fill=yellow!30,rounded corners,fit=(Start)(State1)(State2)(State3)(State4)
(State5)(Transition1)(Transition2)(Transition3)(Transition4)(Transition5)] {};
\end{scope}
\end{tikzpicture}
\end{document}
```

3

4

Figure 3.17: Example of LaTeX code of a Petri net (2).

The blue blocks (1 and 4) indicate standard code portions necessary for start-up and termination of the LaTeX engine. The blocks 2 and 4 instead, indicate code automatically generated by our compiler on the basis of input into the natural pseudo language we designed.

1. Provides information needed to LaTeX engine to compile properly. In particular runs the import of *tikz* package and related libraries that make the drawing commands available. Also it defines the structure of the elements State and Transition, which will be defined by the user.

2. Contains elements that the user has entered as input, reworked to fit the LaTeX format.

3. Code responsible for drawing the connectors between the various elements.

4. Components with the task of closing the scope, the various drawing windows and the entire file.

The drawing of the resulting Petri net is the following:

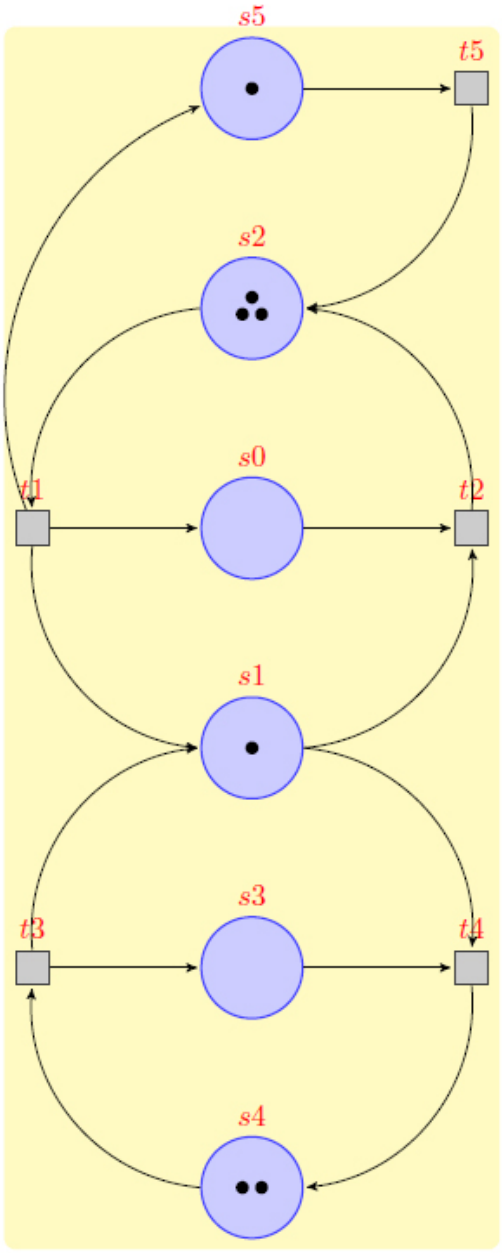

Figure 3.18: Example of a Petri net output diagram.

3.2.2 Output Syntax

This section will describe the syntax of the instructions needed to draw a Petri net in LaTeX.

3.2.2.1 Start

```
\node [place, tokens=0] (Start) [label=above:$s0$] {};
```

The word node indicates that it will be drawn an object, place indicates the type of that object, tokens indicates the number of token that will be inside the Place. (Start) is the identifying name of the Place and "label" its label (above indicates that you will find it above the object).

3.2.2.2 State

```
\node [place, tokens=3] (State2) [above=of Start,label=above:
   $s2$] {};
```

The word node indicates that it will be drawn an object, place indicates the type of that object, tokens indicates the number of token that will be inside the Place. (State2) is the identifying name of the Place. Above=of Start indicates the relative position of the element that we are drawing with respect to another existing object (in this case above Start) and "label" its label (above indicates that you will find it above the object);

3.2.2.3 Transition

```
\node [transition] (Transition1) [left=of Start,label=above:
   $t1$] {};
```

The word node indicates that it will be drawn an object, transition indicates the type of that object. (Transition1) is the identifying name of the Transition. Left=of Start indicates the relative position of the element that we are drawing with respect to another existing object (in this case to the left of Start) and "label" its label (above indicates that you will find it above the object);

3.2.2.4 Straight Arrow

```
\draw [post] (Transition3) to (State3);
```

The word draw indicates that it will be drawn a connector, post indicates that it will be drawn an arrow. (Transition3) is the identifying name of the starting object and (State3) is the identifying name of the ending object.

3.2.2.5 Curved Arrow

```
\draw [post] (State2) to [bend right] (Transition1);
```

The word draw indicates that it will be drawn a connector, post indicates that it will be drawn an arrow. (Transition3) is the identifying name of the starting object and (State3) is the identifying name of the ending object. Bend indicates the bend of the arrow, it can be left or right.

3.3 Error Handling

Error handling has the purpose to recognize and alert the user for the presence in the input text of syntactic or semantic errors regarding the grammar handled by our compiler. These error messages are then displayed in the console after compiling the input text.

3.3.1 Syntactic Errors

The presence of syntax errors in the input is handled directly by ANTLR that generates exceptions intercepted by the *displayRecognitionError()* method opportunely redefined by us:

```
// Override of ANTLR syntax error messages
  public void displayRecognitionError(String[] tokenNames,
     RecognitionException e)
  {
        String errorHeader = getErrorHeader(e);
```

```
        String errorMessage = getErrorMessage(e, tokenNames);
        String errorClass = e.getClass().toString();
        errorClass = errorClass.substring(errorClass.indexOf(
           " ")+1, errorClass.length());

        String syntaxError = "Exception " + errorClass + ":
           Unresolved compilation problem:" + "\n"
              + "Syntax error, " + Model.replaceSymbol(
                 errorMessage) + "\n"
              + "at " + errorHeader;

        env.syntaxError.add(syntaxError);
    }
```

This method takes as input a list of tokens detected by ANTLR and a RecognitionException exception that contains all the information about the type of error and their placement in the text.

The token list is generated by the Lexer. The Lexer's job is to pack the stream of characters (one at a time) from the input.

A token is any character or group of characters gathered from the Lexer. Tokens are components of the programming language considered as keywords, identifiers, symbols and operators.

Usually the Lexer removes comments and spacing from the input, like all other content that does not have a semantic value for the interpretation of the program.

All tokens generated by the scanner must be recognized. The Lexer generates errors related to character sequences that lexer fails to put in correspondence with the specific token types defined by its rules.

The exception contains all the error information detected that are used to generate the error string to the console. In particular we use:

- errorClass : ANTLR class that generated the error;
- errorMessage: error message associated to the specific detected error;
- errorHeader: information related to the error location;

All these informations, appropriately formatted, are inserted into a string that is added to the list syntaxError that contains all detected syntactic errors.

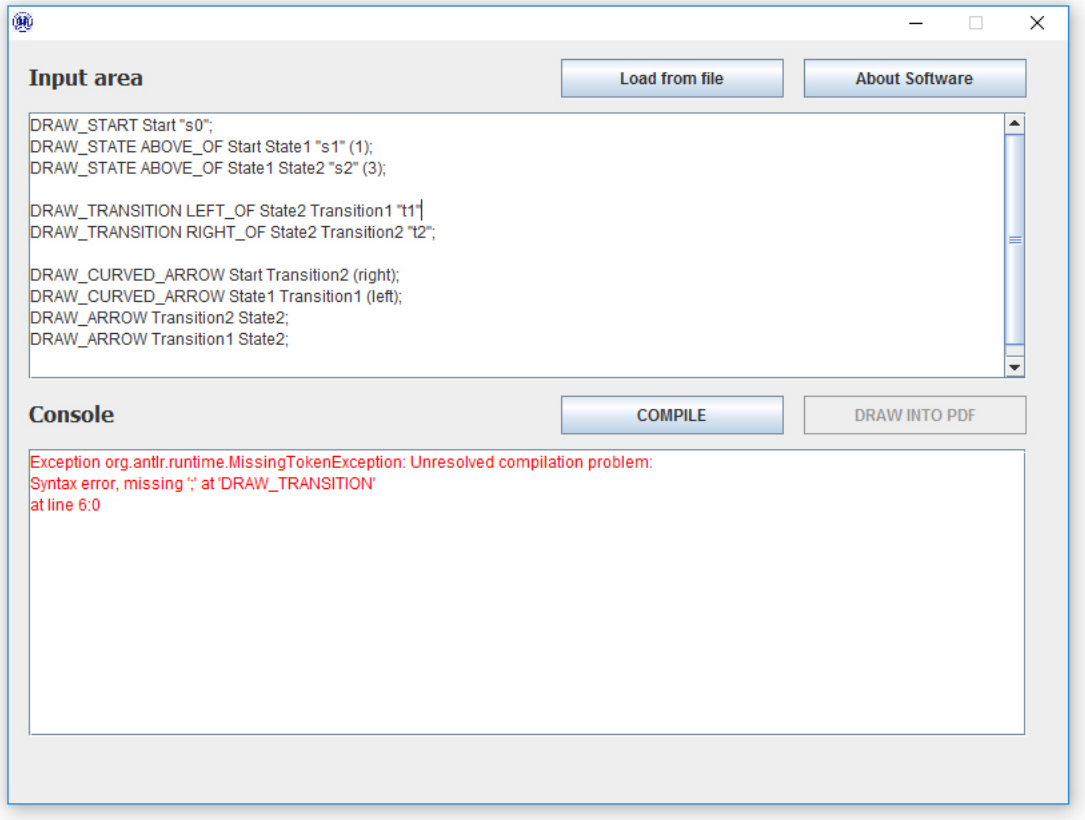

Figure 3.19: Example of syntax error.

3.3.2 Semantic Errors

The presence of semantic errors in the input is handled in a personalized way by the programmer. In fact, the compiler can not detect their presence as they relate to the significance of the processed language that a machine is not able to understand and evaluate autonomously.

The search for semantic errors must occur after the search of syntactic errors and only after that any syntax errors have been corrected accordingly. A semantic error is found during the semantic analysis phase, when incorrect instructions or instructions sequences are detected: a typical error is the use of a variable not yet declared.

The control code of semantic errors is within the *SemanticHandler* class. Each type of semantic error is handled by a special method that generates an error string to be added to the list *semanticError* that contains all detected semantic errors.

Below all types of semantic errors managed are described .

3.3.2.1 Check for missing references

Semantic check for the existence of the object used as reference for the positioning of each object inserted.

```
// Check for missing references
  private void checkMissingRefence()
  {
    for (int i=1; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      // Check State component (nameRef) in stateNames AND
         transitionNames
      if (c instanceof State)
      {
        State state = (State) c;
        String nameRef = state.getNameRef();

        if (!(env.stateNames.contains(nameRef)) && !(env.
           transitionNames.contains(nameRef)))
          env.semanticError.add(getErrorGenerator(i,
             MISSING_REFERENCE));
      }
      // Check Transition component (nameRef) in stateNames
         AND transitionNames
      else if (c instanceof Transition)
      {

        Transition transition = (Transition) c;
        String nameRef = transition.getNameRef();

        if (!(env.stateNames.contains(nameRef)) && !(env.
           transitionNames.contains(nameRef)))
          env.semanticError.add(getErrorGenerator(i,
             MISSING_REFERENCE));
      }
```

```
      // Check CurvedArrow component (nameStart, nameEnd) in
         stateNames AND transitionNames
      else if (c instanceof CurvedArrow)
      {
        CurvedArrow curvedArrow = (CurvedArrow) c;
        String nameStart = curvedArrow.getNameStart();
        String nameEnd = curvedArrow.getNameEnd();

        if  ((!(env.stateNames.contains(nameStart))&&!(env.
           transitionNames.contains(nameStart)))
            || (!(env.stateNames.contains(nameEnd)) &&!(env.
               transitionNames.contains(nameEnd))))
          env.semanticError.add(getErrorGenerator(i,
             MISSING_REFERENCE));
      }
      // Check StraightArrow component (nameStart, nameEnd)
         in stateNames AND transitionNames
      else if (c instanceof StraightArrow)
      {
        StraightArrow straightArrow = (StraightArrow) c;
        String nameStart = straightArrow.getNameStart();
        String nameEnd = straightArrow.getNameEnd();

        if  ((!(env.stateNames.contains(nameStart))&&!(env.
           transitionNames.contains(nameStart)))
            || (!(env.stateNames.contains(nameEnd)) &&!(env.
               transitionNames.contains(nameEnd))))
          env.semanticError.add(getErrorGenerator(i,
             MISSING_REFERENCE));
      }
    }
  }
```

3.3.2.2 Check for duplicated names

Semantic check that verifies that two or more states or transitions with the same name don't exist.

```
// Check for duplicated names
  private void checkNameItemDuplicated()
  {
    for (int i=1; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);
      String name = "";

      if (c instanceof State)
      {
        State state = (State) c;
        name = state.getName();
      }
      else if (c instanceof Transition)
      {
        Transition transition = (Transition) c;
        name = transition.getName();
      }

      // Check all the previous components in the petriNet
      for (int j=0 ; j<i; j++){

        Component c2 = env.petriNet.get(j);
        if (c2 instanceof Start)
        {
          Start start2 = (Start) c2;
          String name2 = start2.getName();
          if (name.equals(name2)){
            env.semanticError.add(getErrorGenerator(i,
               NAME_ITEM_DUPLICATED));
            break;
          }

        }
        else if (c2 instanceof State){
          State state2 = (State) c2;
          String name2 = state2.getName();
          if(name.equals(name2)){
            env.semanticError.add(getErrorGenerator(i,
               NAME_ITEM_DUPLICATED));
```

```
            break;
          }
        }
        else if(c2 instanceof Transition){
          Transition transition2 = (Transition) c2;
          String name2 = transition2.getName();
          if (name.equals(name2)){
            env.semanticError.add(getErrorGenerator(i,
               NAME_ITEM_DUPLICATED));
            break;
          }

        }
      }
    }
  }
```

3.3.2.3 Check state unlinked

Semantic check that looks for the presence of states not connected with the rest of the net.

```
// Check for States not connected with the rest of the net
  private void checkStateUnlinked()
  {
    for (int i=0; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      if (c instanceof Start)
      {
        Start start = (Start) c;
        String name = start.getName();

        if (!(env.referenceNames.contains(name)))
          env.semanticError.add(getErrorGenerator(i,
             STATE_UNLINKED));
      }
```

```
      else if (c instanceof State)
      {
        State state = (State) c;
        String name = state.getName();

        if (!(env.referenceNames.contains(name)))
          env.semanticError.add(getErrorGenerator(i,
             STATE_UNLINKED));
      }

    }
  }
```

3.3.2.4 Check transition unlinked

Semantic check that looks for the presence of transitions not connected with the rest of the net.

```
// Check for Transitions not connected with the rest of the
   net
  private void checkTransitionUnlinked()
  {
    for (int i=1; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      if (c instanceof Transition)
      {
        Transition state = (Transition) c;
        String name = state.getName();

        if (!(env.referenceNames.contains(name)))
          env.semanticError.add(getErrorGenerator(i,
             TRANSITION_UNLINKED));
      }
    }
  }
```

3.3.2.5 Check for invalid links

Semantic check that verifies that there aren't any states connected to other states or transitions connected to other transitions, according with the Petri net theory.

```
// Check if a State is connected to an other State or a
   Transition is connected to an other Transition
  private void checkInvalidLink()
  {
    for (int i=1; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      if (c instanceof StraightArrow || c instanceof
         CurvedArrow)
      {
        Connector arrow = (Connector) c;
        String origin = arrow.getNameStart();
        String destination = arrow.getNameEnd();

        if (((env.stateNames.contains(origin)) && (env.
           stateNames.contains(destination))) ||
            ((env.transitionNames.contains(origin)) && (env.
               transitionNames.contains(destination))))
          env.semanticError.add(getErrorGenerator(i,
             INVALID_LINK));
      }
    }

  }
```

3.3.2.6 Check for autoreference

Semantic check that verifies that an object is not positioned in reference to itself.

```
// Check for autoreference positioning
  private void checkAutoreference()
  {
```

```
    for (int i=1; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      if (c instanceof State)
      {
        State state = (State) c;

        if (state.getName().equals(state.getNameRef()))
          env.semanticError.add(getErrorGenerator(i,
             AUTOREFERENCE));
      }
      else if (c instanceof Transition)
      {
        Transition transition = (Transition) c;

        if (transition.getName().equals(transition.getNameRef
           ()))
          env.semanticError.add(getErrorGenerator(i,
             AUTOREFERENCE));
      }
    }
  }
```

3.3.2.7 Check for invalid number of tokens

Semantic check that verifies that in each state of the network there is a number of token between 0 and 3.

```
// Check if the number of token is between 0 and 3 for the
   states
  private void checkInvalidTokens() {

    for (int i=0; i < env.petriNet.size(); i++)
    {
      Component c = env.petriNet.get(i);

      if (c instanceof Start)
```

```
    {
      Start start = (Start) c;
      int tokens = 0;
      if(!start.getTokens().equals(""))
        tokens = Integer.parseInt(start.getTokens());

      if (tokens < 0 || tokens > 3)
        env.semanticError.add(getErrorGenerator(i,
           INVALID_TOKENS));
    }
    else if (c instanceof State)
    {
      State state = (State) c;
      int tokens = Integer.parseInt(state.getTokens());

      if (tokens < 0 || tokens > 3)
        env.semanticError.add(getErrorGenerator(i,
           INVALID_TOKENS));
    }

  }

}
```

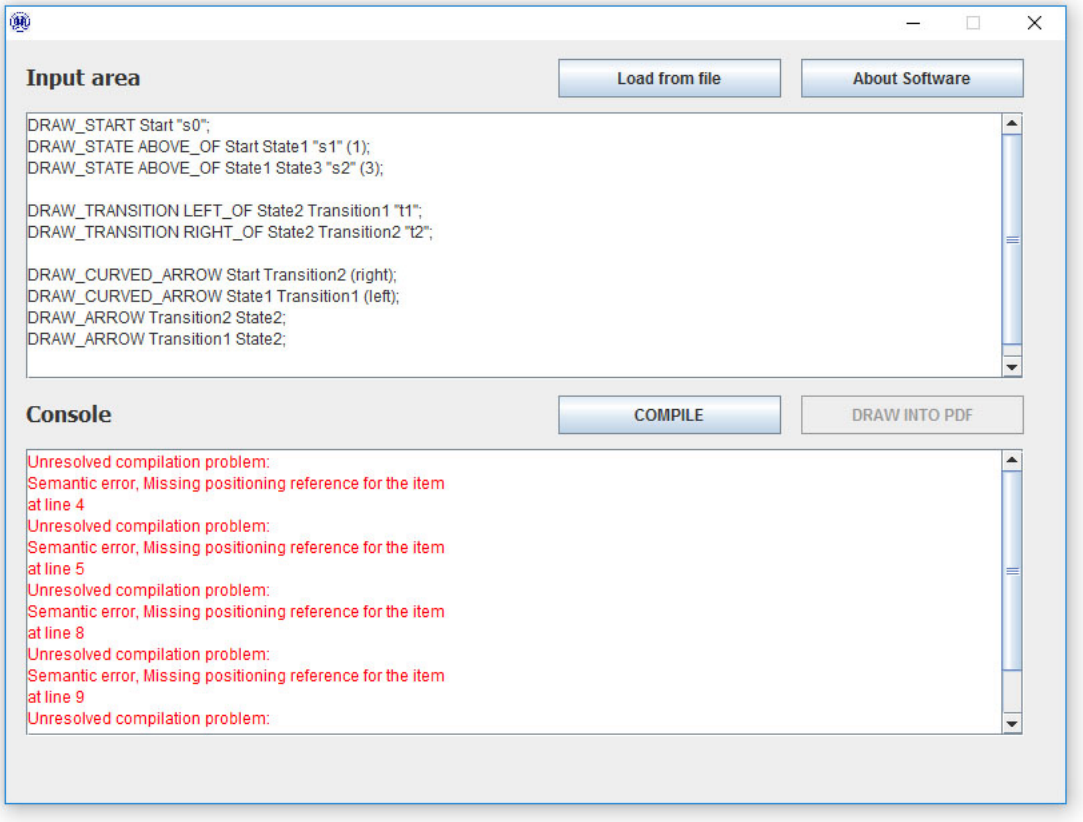

Figure 3.20: Example of semantic error.

Chapter 4

Execution Flow

When the main application starts, the user sees the main screen. He will mainly perform two operations:

- Enter the code to be compiled in the "Input area";

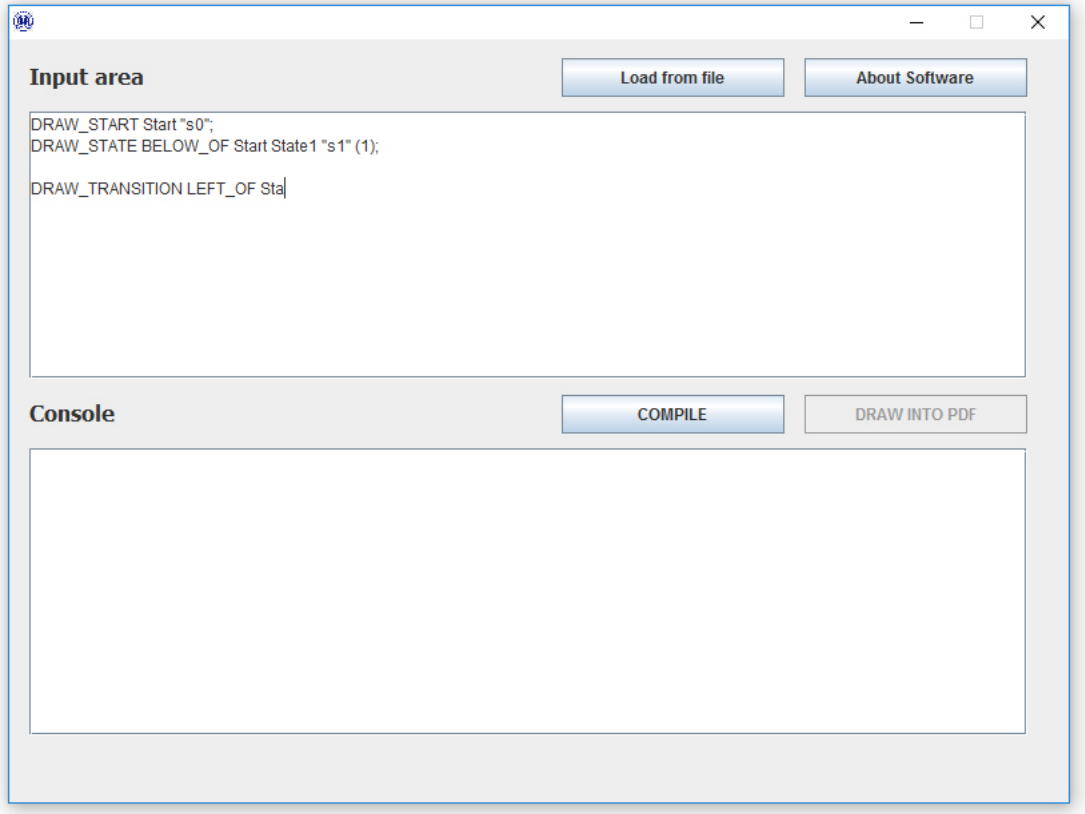

Figure 4.1: Manual typing of input code.

- Load an external file that contains the code to be compiled using the button **"LOAD FROM FILE"**;

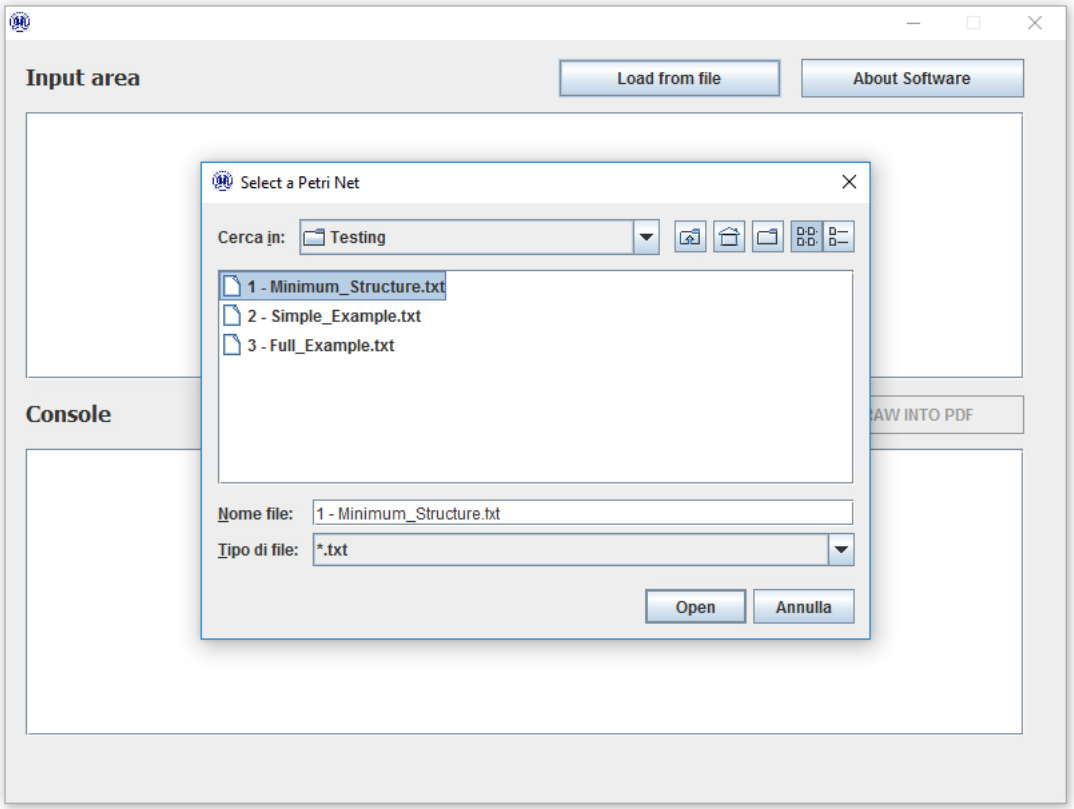

Figure 4.2: Load of input code from file.

When the user clicks on "**COMPILE**" launches a new thread that deals with the compilation of what is present in the input area.

The operations performed during compilation are:

1. Lexical analysis by the Lexer;

2. Parsing by the parser;

3. Syntactic analysis by the parser generated by ANTLR according to the grammar rules defined in the **language.g** file;

4. Checking for syntax errors: if present, is sent to the user a detailed notification , the Model and the View are updated and finally the compilation process ends;

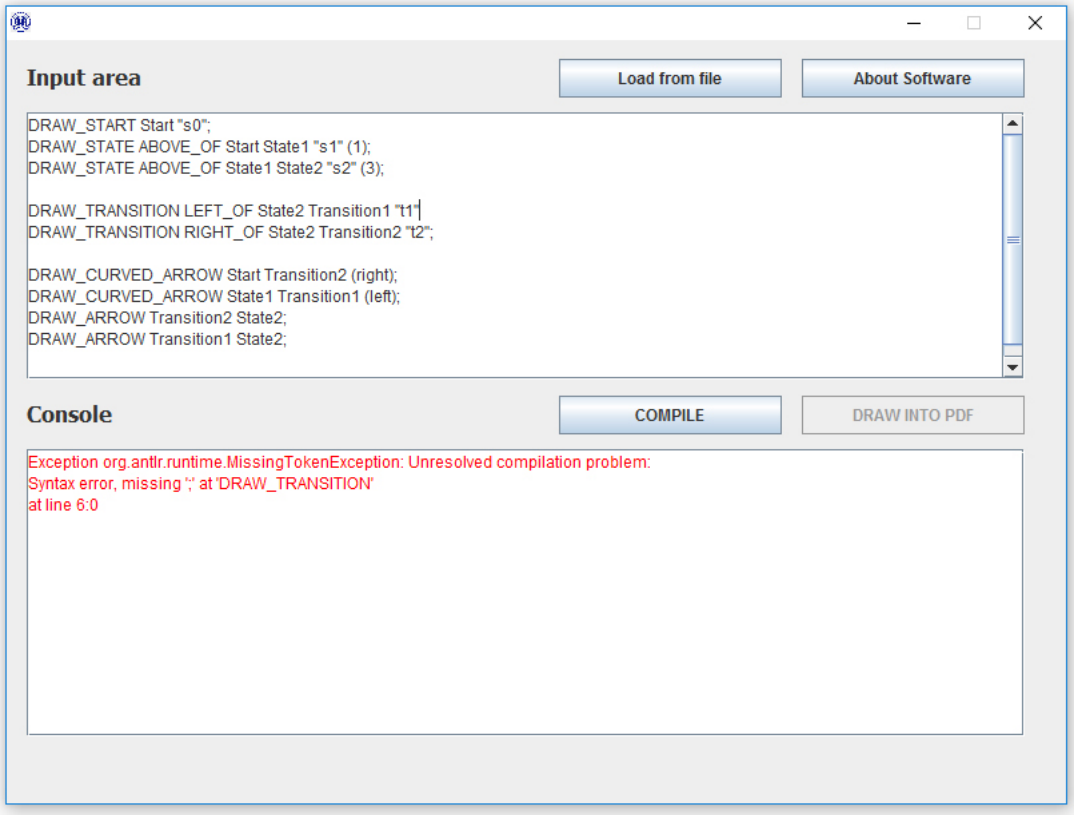

Figure 4.3: Example of compilation interrupted for syntax error.

5. If there are no syntax errors, verifies the presence of semantic errors. It is instantiated a SemanticHandler that controls all the semantic rules to check for errors: if present, is sent to the user a detailed notification, the Model and the View are updated and finally the compilation process ends;

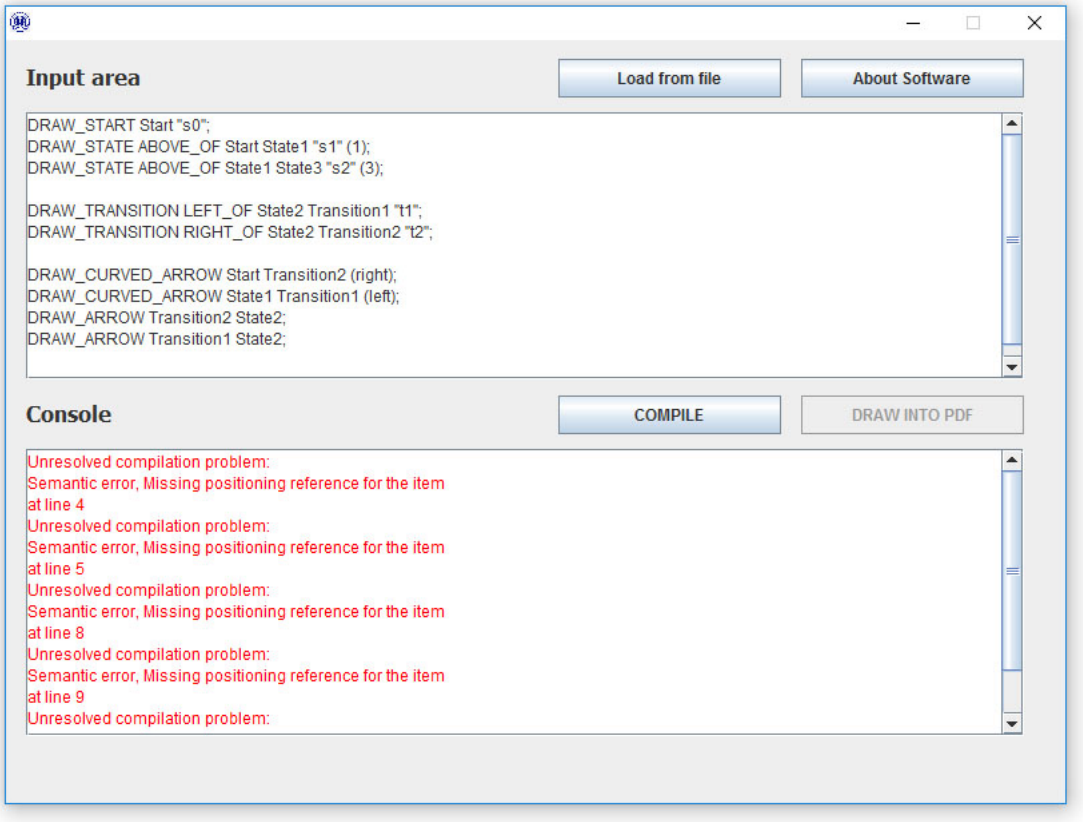

Figure 4.4: Example of compilation interrupted for semantic error.

6. If there are no syntactic and semantic errors the compilation process terminates properly and a notification "COMPILE SUCCESFULL" is sent to the user. Model and View are updated. Now the button "**DRAW ON PDF**" for the .tex file generation will also be enabled;

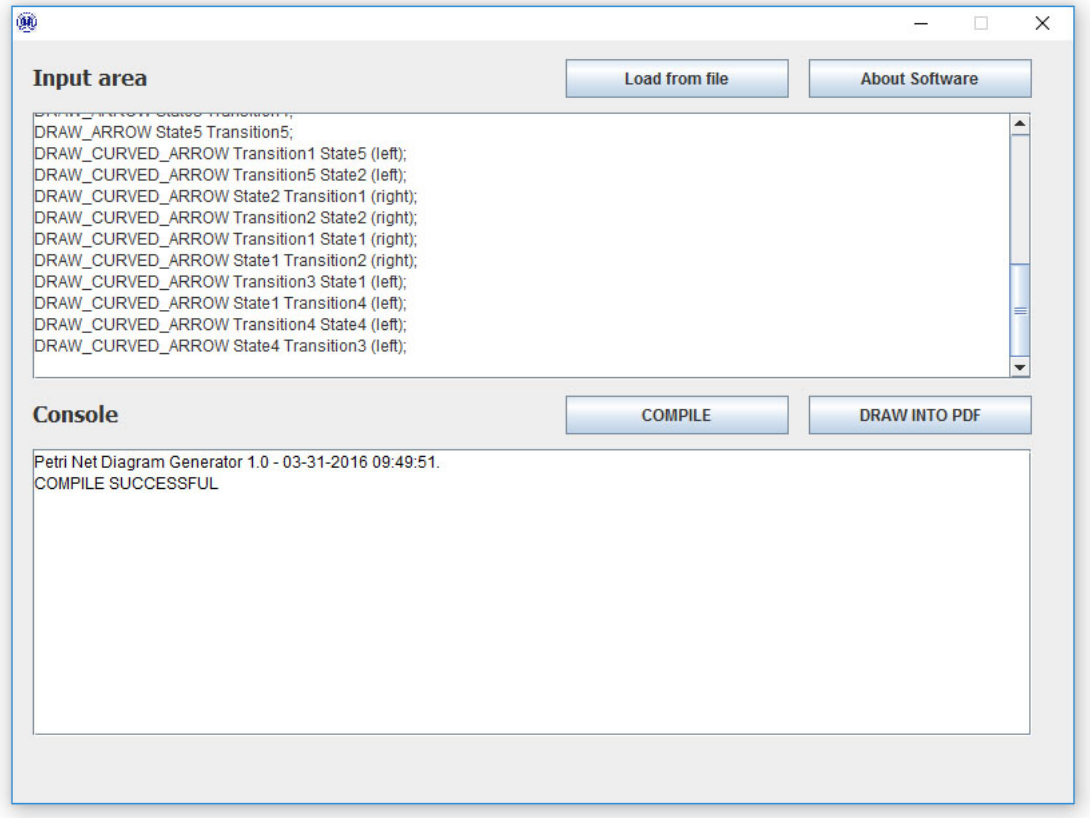

Figure 4.5: Example of compile successfully completed.

At each change of the input code or new upload from file the "**DRAW ON PDF**" button will be disabled and you will need to repeat the compilation; in this way there cannot be any inconsistencies between input and output generated from previous versions of the code files.

When the user clicks on "**DRAW ON PDF**" a new LaTeXConverter object is instantiated, with the job to handle the translation in LaTeX code.

The operations performed in this phase are the following:

1. A new **LaTeXConverter** object is instantiated and is passed to the Model that contains the compiled and validated Petri net;
2. The Petri net is converted into the equivalent LaTeX code;
3. The LaTeX code is saved in the "**PetriNet.tex**" file;
4. If during the generation phase any translation errors occur a notification is sent to the user;

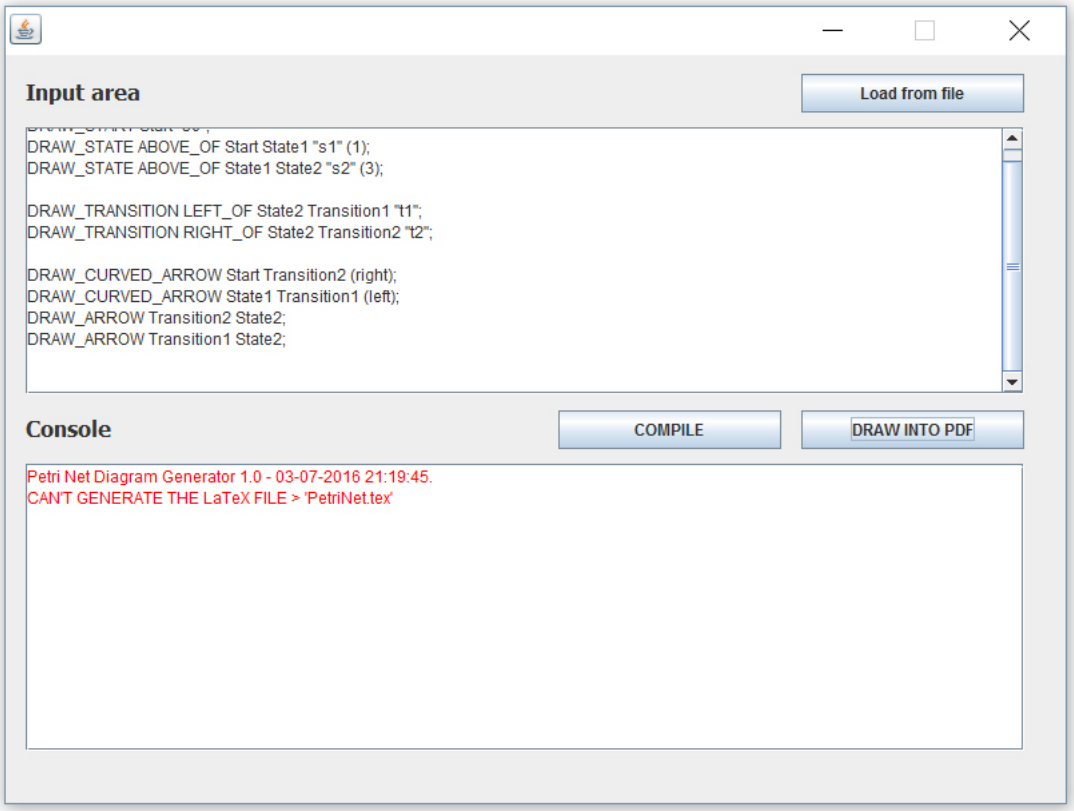

Figure 4.6: Example of error notification when generating LaTeX.

5. If the build is successful, it launches the LaTeX compiler MiKTeX and a notification is sent to the user;

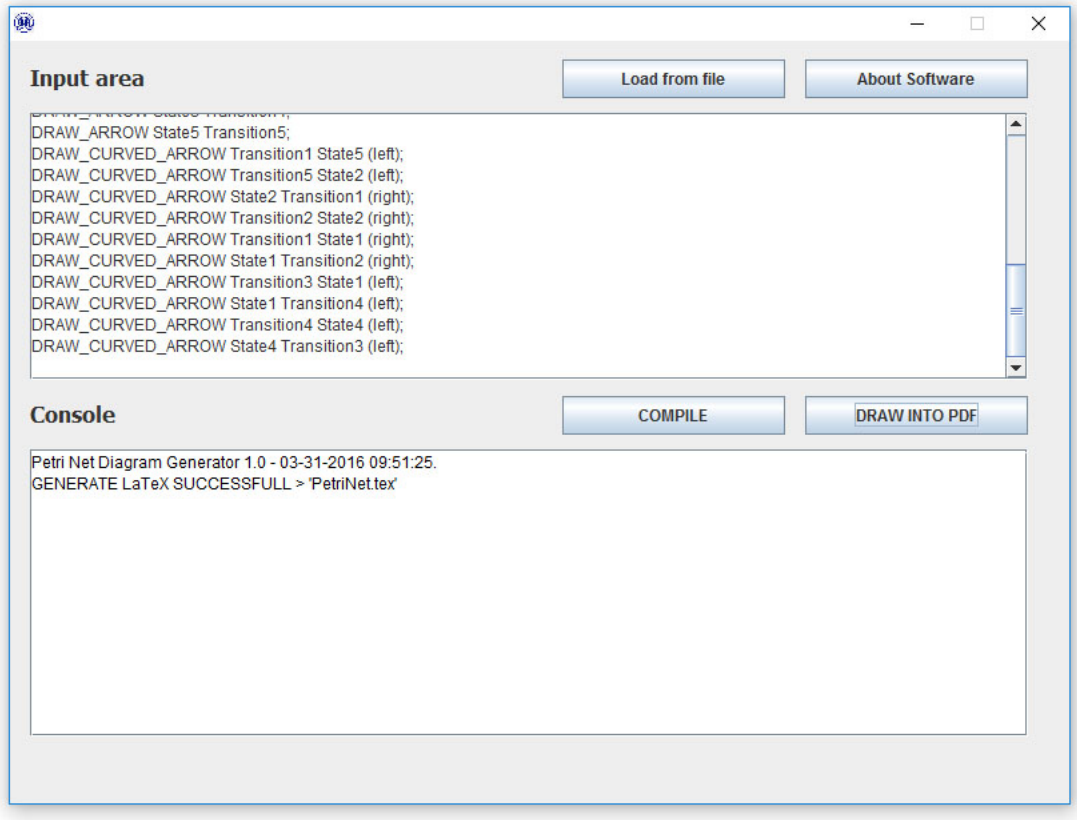

Figure 4.7: Example of successful LaTeX generation.

Conclusions

This project has been created as a basic tool to be extended for specific purposes. In fact, it is simply adjustable to satisfy specific intention such as, for example, adding new declarations in the input file or creating new objects or rules for the resulting network.

To do so, it is suggested to work on language.g to create or modify Tokens and syntactic rules, on ParserEvnironment class to create or modify fields and variables and on CompilerThread class to create or modify new methods to manage the notification and error systems.

Also it is possible to add additional rules of Petri nets, like the timing of the transitions or the weight of the connectors.

Please leave a review if you have enjoyed this project at:

`https://sourceforge.net/projects/petri-net-diagram-compiler/`