

Pseudo-code virtual machine. Second Interim report

Tarek Fouda

May 15, 2013

1 Introduction

Pseudo code Algorithm is a code that is taught for student who are new to computer science and coding. It simply allows them to understand how to assign variables, execute (while,for,do) loops, and how to check for a condition to execute the "if" or the "else" part. A lot of these new students face a lot of problems in understading the meaning of Pseudo Code, what does the code exactly do and how the variables are assigned and so on. Tracing such a code isn't easy for such students, and they do not know where the code went wrong except from the feed back of the instructor through grading the tests, quizzes or the assignments. It's really hard for them to understand the mechanism of Pseudo code algorithm from just a paper, as they can not see any execution happens, they are not aware of how the variables are assigned, neither they are aware of how the loop is executed and how a while loop -for example- stops when the condition is false. Their understanding to such an algorithm needed something more enhanced to teach them the mechanism of such an algorithm.

1.1 Pseudo-code interpreter approach

From this point the idea of creating a tool that helps such students literally understand what goes on inside the code, and how it's executed line by line. This interpreter simply is the mirror of the code written by the student, it highlights the errors existing in the code, shows what variables are assigned and the new values assigned to them and finally it translates what happens in the code through a graphical user Interface. This tool simply allows the student to freely write his/her code and by pressing a compile button all the errors are highlighted. Also by compiling the code an arrow should appear on the very first line, this arrow goes down when the student wants to, after every line is finished, the variables assigned, the loops executed and the conditions checked should appear in a different window and here is the magic of this tool, as it translates exactly the smallest details that happen in the code.

2 How to start on this tool?

2.1 The programming language

To start on how to implement such an interpreter with the above functions, I thought of implementing the tool either in Java or in Haskell. Haskell is a functional programming language and it is more easier to write a lot of grammars and rules in Haskell, but on the other hand, I'm used more to implementing in Java, besides this tool requires a Graphical User Interface GUI which can not be done using Haskell, so It was decided to code and implement this tool using Java programming language.

2.2 Java parser generator (Antlr)

Our mind was made up with the language used in implementing in this tool, choosing the specific and efficient Java parser generator was left to begin with the tool. There are many of Java parser generators like (Antlr, JavaCC, BYACC/J.. etc).

The two most popular Java parser generators were Antlr and JavaCC. I had to research and compare between them in terms of differences, flexibility and learning curve.

I started to search more for Antlr and JavaCC, Antlr home page provided me with more convenient documentation and explanation about how the generator and the lexer work. On the other hand JavaCC homepage was a bit difficult for me to understand the documentation out of it. Another advantage of Antlr was that it produces parser source code in various languages but on the other hand, JavaCC only produces java. By more researches I found out that Antlr was more developer-friendly than JavaCC. Moreover Antlr had Eclipse plugins and that was an advantage as the project uses Graphical User Interface and the most easiest way to integrate all the code together with the GUI is through eclipse, so Antlr having an Eclipse IDE was a huge advantage for me.

2.3 How to start with Antlr

Installing Antlr is pretty easy after following the instruction presented in the home page of Antlr website wwwantlr.org.

Installing Antlr for UNIX and WINDOWS are shown, but in my case I was using windows, installing Antlr for windows was as shown in Figure 1.

(Thanks to Graham Wideman)

0. **Install Java** (version 1.6 or higher)

1. **Download** <http://antlr.org/download/antlr-4.0-complete.jar>
Save to your directory for 3rd party Java libraries, say C:\Javalib

2. **Add antlr-4.0-complete.jar to CLASSPATH**, either:

- Permanently: Using System Properties dialog > Environment variables > Create or append to CLASSPATH variable
- Temporarily, at command line:

```
SET CLASSPATH=C:\Javalib\antlr-4.0-complete.jar;%CLASSPATH%
```

(Do we need . "dot" in there too?)

3. **Create short convenient commands for the ANTLR Tool, and TestRig, using batch files or doskey commands:**

Batch files (in directory in system PATH)

Batch file
antlr4.bat
<pre>java org.antlr.v4.Tool %*</pre>
run.bat
<pre>java org.antlr.v4.runtime.misc.TestRig %*</pre>

Or, use doskey commands:

```
doskey antlr4=java org.antlr.v4.Tool %*
doskey grun =java org.antlr.v4.runtime.misc.TestRig %*
```

Figure 1: ANTLR installation for windows users

After successfully installing Antlr on your machine you need to test your installation by typing the following two commands in your terminal in Figure 2.

Testing the installation

Either launch `org.antlr.v4.Tool` directly:

```
$ java org.antlr.v4.Tool
ANTLR Parser Generator Version 4.0
-o ____ specify output directory where all output is generated
-lib ____ specify location of .tokens files
...
```

or use `-jar` option on java:

```
$ java -jar /usr/local/lib/antlr-4.0-complete.jar
ANTLR Parser Generator Version 4.0
-o ____ specify output directory where all output is generated
-lib ____ specify location of .tokens files
...
```

Figure 2: Testing ANTLR installation

In the rest of the tutorial there is a simple grammar to teach you how to write a grammar using Antlr the grammar is simply consist of a start rule *r* and two other Lexer rules which are the identification and White spaces (*ID* , *WS*). Also there is the -gui that produces the parse tree of a grammar as shown in Figure 4.

```
In a temporary directory, put the following grammar inside file Hello.g4:
```

```

// Define a grammar called Hello
grammar Hello;
r : 'hello' ID ;           // match keyword hello followed by an identifier
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

```

```
Then run ANTLR the tool on it:
```

```

$ cd /tmp
$ antlr4 Hello.g4
$ javac Hello*.java

```

```
Now test it:
```

```

$ grun Hello r -tree
hello parrrt
^D
(r hello parrrt)

```

(That ^D means EOF on unix; it's ^Z in Windows.) The -tree option prints the parse tree in LISP notation.

It's nicer to look at parse trees visually.

```

$ grun Hello r -gui
hello parrrt
^D

```

Figure 3:

Now Antlr is installed on the machine and waiting for a grammar .g file to be written. One of the reasons we chose Antlr over JavaCC is that Antlr has eclipse plugins and IDE, we've searched for how to install this plugin on eclipse to enhance our developing infrastructure.

Thanks to Scott Stanchfield who introduced a tutorial on how to install Antlr 3.x IDE on eclipse. <http://vimeo.com/8015802>. . By following this tutorial we now have Antlr installed on eclipse and ready to be in use. Starting a new Antlr project will create a .g file where the grammar should be written in. Regarding our tool we have a grammar that covers all the combinations and tokens any Pseudo code would have.

That pops up a dialog box showing that rule `r` matched keyword `hello` followed by identifier `par`

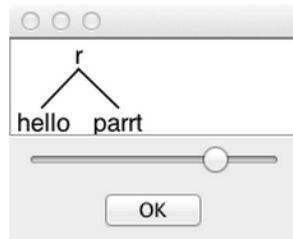


Figure 4:

2.4 GRAMMAR

The grammar was as the following:

$$\begin{aligned} \langle \text{ALGORITHM} \rangle ::= & \text{'algorithm' } \langle \text{ID} \rangle [\text{'inputs' } \langle \text{DECLIST} \rangle]? \\ & [\text{'outputs' } \langle \text{DECLIST} \rangle]? \\ & [\text{'localvar' } \langle \text{DECLIST} \rangle]? \text{'begin' } \langle \text{STATEMENT} \rangle \text{'end' } \end{aligned}$$

$$\langle \text{INDEX} \rangle ::= [\text{' } \langle \text{ARITHEXPR} \rangle \text{'}]$$

$$\langle \text{INDEXING} \rangle ::= [\text{' } \langle \text{VALUE} \rangle \text{'...'} \langle \text{VALUE} \rangle \text{'}]$$

$$\langle \text{IDLIST} \rangle ::= \langle \text{ID} \rangle \text{' , ' } \langle \text{IDLIST} \rangle \mid \langle \text{ID} \rangle$$

$$\langle \text{ASSIGN} \rangle ::= \text{'set' } \langle \text{ID} \rangle \text{'to' } \langle \text{ID} \rangle$$

$$\langle \text{ASSIGNLIST} \rangle ::= \langle \text{ASSIGN} \rangle \text{' , ' } \langle \text{ASSIGNLIST} \rangle \mid \langle \text{ASSIGN} \rangle$$

$$\begin{aligned} \langle \text{DECL} \rangle ::= & \text{'number' } \langle \text{ID} \rangle [\langle \text{INDEXING} \rangle]? \mid \text{'data' } \langle \text{ID} \rangle \\ & [\langle \text{INDEXING} \rangle]? \end{aligned}$$

$\langle \text{DECLLIST} \rangle ::= \langle \text{DECL} \rangle \text{' ' } \langle \text{DECLLIST} \rangle \mid \langle \text{DECL} \rangle$

$\langle \text{STATEMENT} \rangle ::=$

$\langle \text{SEQUENTIAL} \rangle$	$\langle \text{ASSIGNMENT} \rangle$
$\langle \text{CONDITIONAL} \rangle$	
$\langle \text{ITERATIVE} \rangle$	
$\langle \text{PRINT} \rangle$	
$\langle \text{READ} \rangle$	
$\langle \text{INVOCATION} \rangle$	

$\langle \text{ASSIGNMENT} \rangle ::= \text{'set' } \langle \text{ID} \rangle [\langle \text{INDEX} \rangle] \text{'to' } (\langle \text{ARITHEXPR} \rangle \mid \langle \text{DATAEXPR} \rangle)$

$\langle \text{SEQUENTIAL} \rangle ::= \langle \text{STATEMENT} \rangle \text{' ; ' } \langle \text{STATEMENT} \rangle$

$\langle \text{CONDITIONAL} \rangle ::= \text{'if' } \langle \text{CONDITION} \rangle \text{'then' } \langle \text{STATEMENT} \rangle \text{'else' } \langle \text{STATEMENT} \rangle \text{'endif'}$
 $\mid \text{'if' } \langle \text{CONDITION} \rangle \text{'then' } \langle \text{STATEMENT} \rangle \text{'endif'}$

$\langle \text{ITERATIVE} \rangle ::= \text{'while' } \langle \text{CONDITION} \rangle \text{'do' } \langle \text{STATEMENT} \rangle \text{'loop'}$

$\langle \text{PRINT} \rangle ::= \text{'print' } (\langle \text{DATAEXPR} \rangle \mid \langle \text{ARITHEXPR} \rangle)$

$\langle \text{READ} \rangle ::= \text{'read' } \langle \text{IDLIST} \rangle$

$\langle \text{INVOCATION} \rangle ::= \text{'run' } \langle \text{ID} \rangle [\text{'inputs' } \langle \text{ASSIGNLIST} \rangle]?$
 $[\text{'outputs' } \langle \text{ASSIGNLIST} \rangle] \text{'done'}$

$\langle \text{CONDITION} \rangle ::= \langle \text{DISJUNCTION} \rangle$

$\langle \text{DISJUNCTION} \rangle ::= \langle \text{CONJUNCTION} \rangle 'or' \langle \text{DISJUNCTION} \rangle$
 $\quad | \langle \text{CONJUNCTION} \rangle$

$\langle \text{CONJUNCTION} \rangle ::= \langle \text{NEGATION} \rangle 'and' \langle \text{CONJUNCTION} \rangle$
 $\quad | \langle \text{NEGATION} \rangle$

$\langle \text{NEGATION} \rangle ::= 'not' \langle \text{ATOM} \rangle | \langle \text{ATOM} \rangle$

$\langle \text{ATOM} \rangle ::= \langle \text{BOOLEXP} \rangle | '(' \langle \text{DISJUNCTION} \rangle ')'$

$\langle \text{BOOLEXP} \rangle ::= \langle \text{ARITHEXP} \rangle ('=' | '<' | '<=' | '>' | '>=' | '=') \langle \text{ARITHEXP} \rangle$

$\langle \text{ARITHEXP} \rangle ::= \langle \text{MULTIPLICATION} \rangle ('+' | '-') \langle \text{ARITHEXP} \rangle$
 $\quad | \langle \text{MULTIPLICATION} \rangle$

$\langle \text{MULTIPLICATION} \rangle ::= \langle \text{NEGEXP} \rangle ('*' | '/') \langle \text{MULTIPLICATION} \rangle$
 $\quad | \langle \text{NEGEXP} \rangle$

$\langle \text{NEGEXP} \rangle ::= '-' \langle \text{VALUE} \rangle | \langle \text{VALUE} \rangle '(' \langle \text{ARITHEXP} \rangle ')'$

$\langle \text{VALUE} \rangle ::= \langle \text{ID} \rangle [\langle \text{INDEX} \rangle]? | \langle \text{INTEGER} \rangle$

$\langle \text{INTEGER} \rangle ::= ('1'..'9') ('0'..'9')^*$

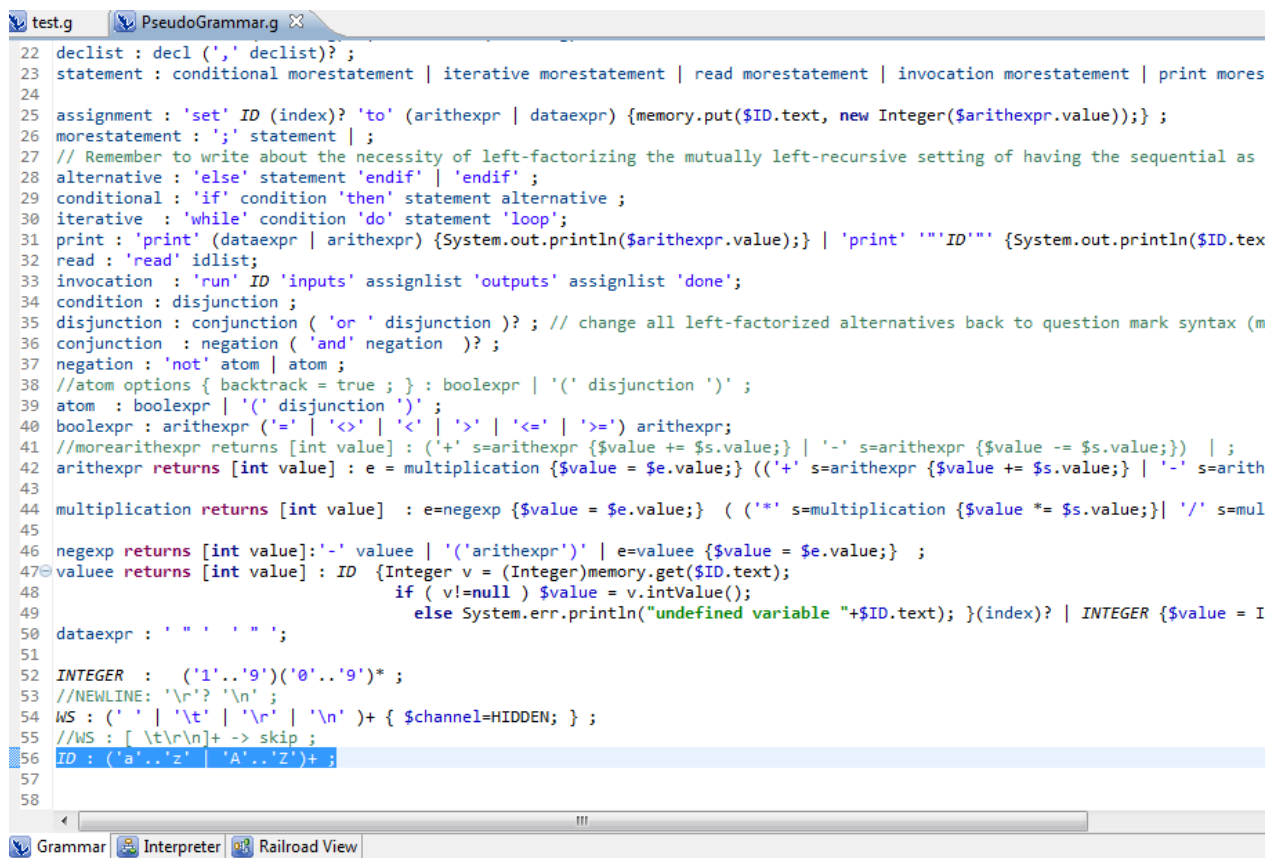
$\langle \text{ID} \rangle ::= ('a'..'z' | 'A'..'Z')^+$

2.5 Antlr and Eclipse IDE

Now we have a grammar and an ANTLR plugin on eclipse, There are three views of the grammar (Grammar, Interpreter, RailRoad view)

2.5.1 Grammar view

The grammar view simply displays the grammar you have written with highlighting the lines that contain error. A simple grammar view is in the following figure.



```
22 declist : decl (',' declist)? ;
23 statement : conditional morestatement | iterative morestatement | read morestatement | invocation morestatement | print morestatement ;
24
25 assignment : 'set' ID (index)? 'to' (arithexpr | dataexpr) {memory.put($ID.text, new Integer($arithexpr.value));} ;
26 morestatement : ';' statement | ;
27 // Remember to write about the necessity of left-factorizing the mutually left-recursive setting of having the sequential as
28 alternative : 'else' statement 'endif' | 'endif' ;
29 conditional : 'if' condition 'then' statement alternative ;
30 iterative : 'while' condition 'do' statement 'loop' ;
31 print : 'print' (dataexpr | arithexpr) {System.out.println($arithexpr.value);} | 'print' ""ID"" {System.out.println($ID.text)} ;
32 read : 'read' idlist ;
33 invocation : 'run' ID 'inputs' assignlist 'outputs' assignlist 'done' ;
34 condition : disjunction ;
35 disjunction : conjunction ( 'or' disjunction )? ; // change all left-factorized alternatives back to question mark syntax (m
36 conjunction : negation ( 'and' negation )? ;
37 negation : 'not' atom | atom ;
38 //atom options { backtrack = true ; } : boolexpr | '(' disjunction ')' ;
39 atom : boolexpr | '(' disjunction ')' ;
40 boolexpr : arithexpr ('=' | '<' | '<=' | '>' | '>=' | '<=' | '>=') arithexpr ;
41 //morearithexpr returns [int value] : ('+' s=arithexpr {$value += $s.value;} | '-' s=arithexpr {$value -= $s.value;} | '*' s=arith
42 arithexpr returns [int value] : e = multiplication {$value = $e.value;} (('+' s=arithexpr {$value += $s.value;} | '-' s=arith
43
44 multiplication returns [int value] : e=negexp {$value = $e.value;} ( '*' s=multiplication {$value *= $s.value;} | '/' s=multiplication {$value /= $s.value;} ) ;
45
46 negexp returns [int value] : '-' valuee | '(' arithexpr ')' | e=valuee {$value = $e.value;} ;
47 valuee returns [int value] : ID {Integer v = (Integer)memory.get($ID.text);
48                                     if ( v!=null ) $value = v.intValue();
49                                     else System.err.println("undefined variable "+$ID.text); }(index)? | INTEGER {$value = Integer.parseInt($ID.text)} ;
50 dataexpr : " " ' ' ' ' ' ' ;
51
52 INTEGER : ('1'..'9') ('0'..'9')* ;
53 //NEWLINE: '\r'? '\n' ;
54 WS : ( ' ' | '\t' | '\r' | '\n' )+ { $channel=HIDDEN; } ;
55 //WS : [ \t\r\n ]+ -> skip ;
56 ID : ('a'..'z' | 'A'..'Z')+ ;
57
58
```

Figure 5: Grammar View.

2.5.2 Interpreter view

The Interpreter view is simply consisted of two main things, the first thing is a Text Field where the developer could write a sample code or a mini code to test whether everything is write or not. The second thing is a Displaying windows that shows the output of the entered code mentioned above. In other words,

The developer types a sample code in the text field as shown in the below figure and the Interpreter produces the Parse Tree of this code as shown below.

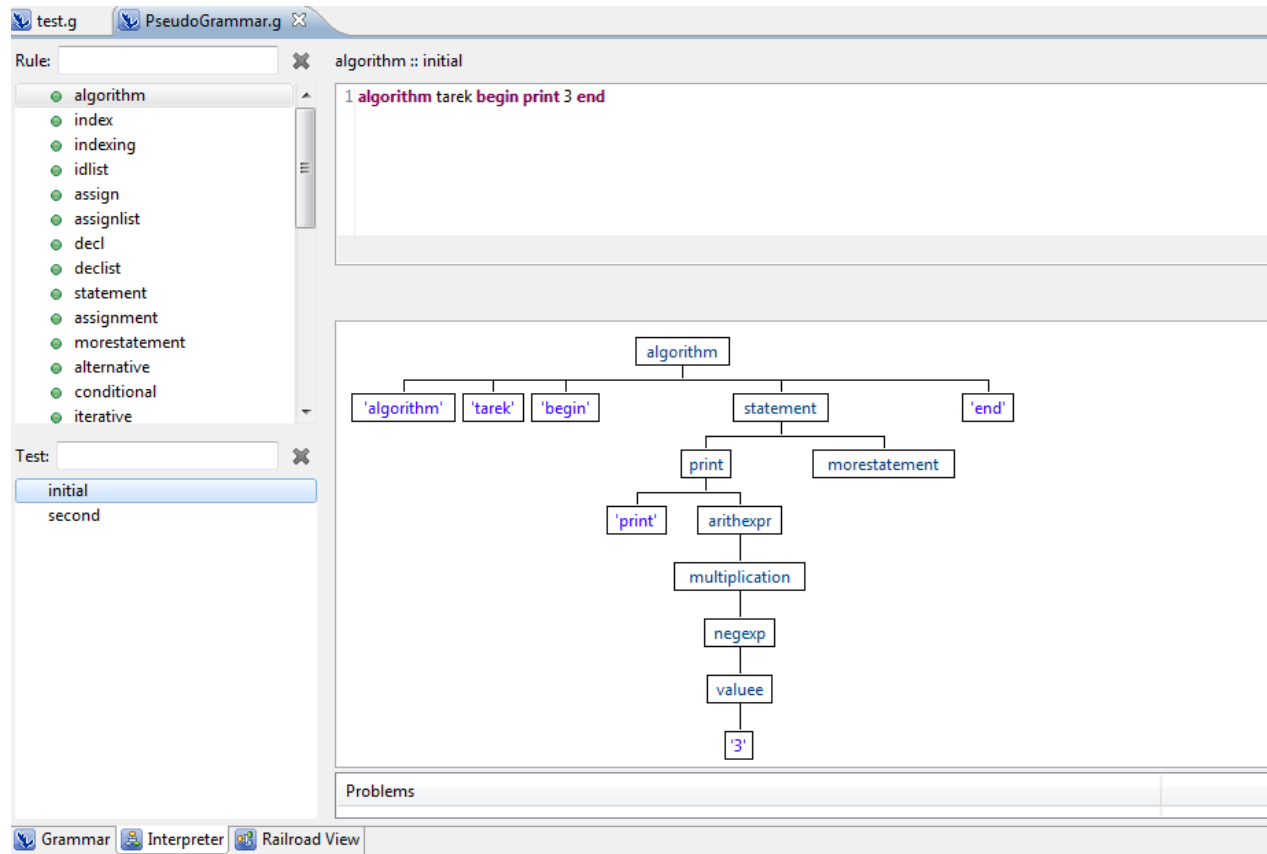


Figure 6: Interpreter View

2.5.3 Rail Road view

The Rail Road view is a view which the developer could check what every rule in the grammar do and require. for example *algorithm* rule mentioned in this section, This rule needs '*algorithm*' and an *ID*. The '*inputs*' and the '*outputs*' are not required to be there, they are optional. The Rail Road view shows that all. Check the below figure to understand the task of this view more precisely and accurately.

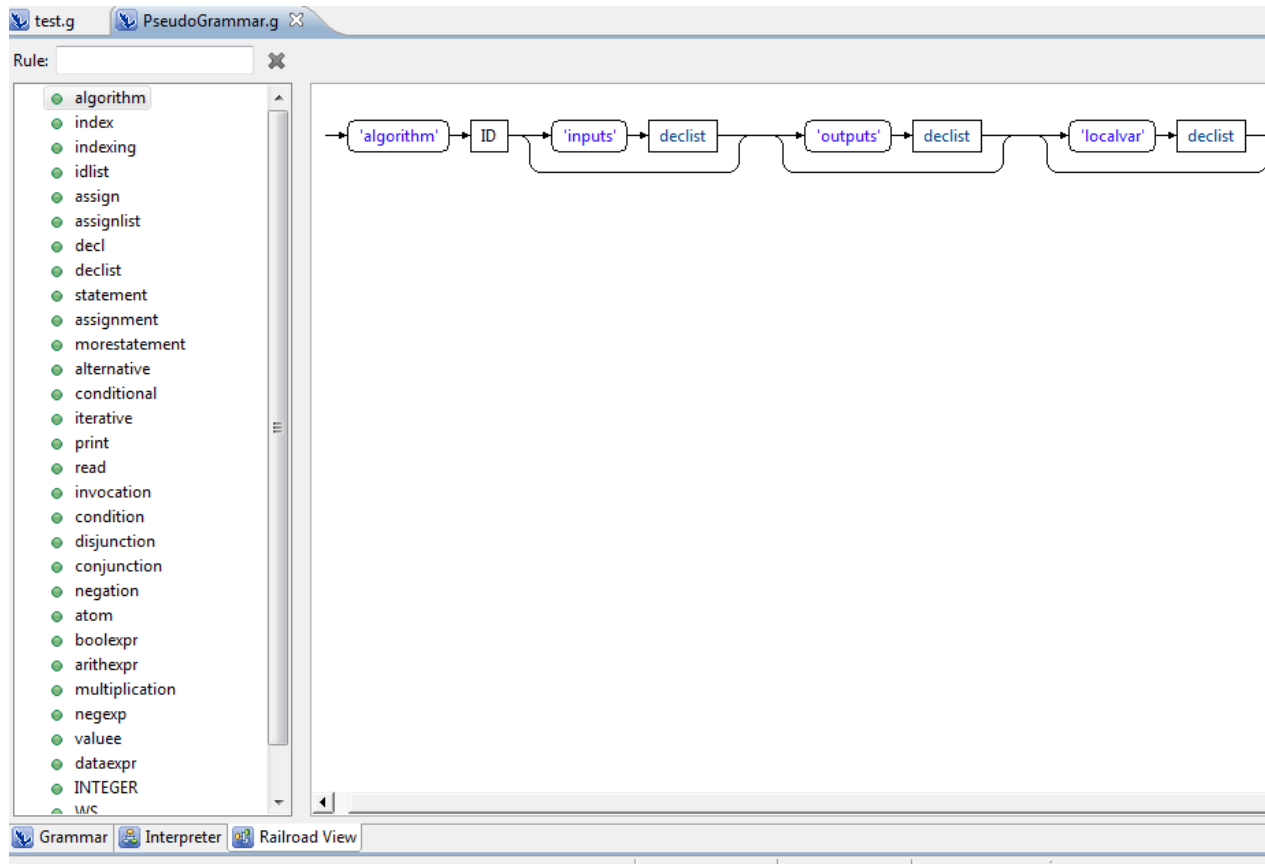


Figure 7: Rail Road View.

3 Left Factorizing the mutually left recursive tokens

Writing grammars, one will encounter left-recursive rules. As ANTLR is a recursive-descent parser, it cannot cope with them and thus they have to be removed. But what is a left-recursive rule exactly? It is a recursive rule which calls itself on the left-edge of a global alternative. An example would be:

$$a : ab|b;$$

ANTLRworks can resolve this problem and turns rule a into

$$a : b+;$$

But ANTLRworks can do this automatically only for the above kind of left-recursive rules, which exhibit the left recursion within the implementation of the recursive rule itself. I would call these single left-recursive (SLR) rules. The other kind are mutually left-recursive (MLR) rules where the left-recursion occurs over several rule invocations. The simplest example would be the following:

$a : b|c;$

$b : a|c;$

Rule a references rule b, while rule b references rules a. A more complex example would show such a 're-reference' taking place after many subsequent rule invocations. As stated above, the resolution has to be done manually here.

That was said by *JohannesLuber* on May 2, 2008. He was declaring on www.antlr.org website that ANTLR 3.x can not resolve the problem of the left recursive rules. On the other hand, ANTLR 4 was capable of resolving such a problem but unfortunately there was no ANTLR 4 plugin on eclipse, only ANTLR 3.x existed. That was the reason the above grammar mentioned in the previous section needed modification and left factorizing to eliminate the left-recursive problem existed in the grammar.

Simply left factorizing the rules was kind of an easy task for lots of rules, for example, rules like :

$$\langle \text{DISJUNCTION} \rangle ::= \langle \text{CONJUNCTION} \rangle 'or' \langle \text{DISJUNCTION} \rangle \\ | \langle \text{CONJUNCTION} \rangle$$

$$\langle \text{CONJUNCTION} \rangle ::= \langle \text{NEGATION} \rangle 'and' \langle \text{CONJUNCTION} \rangle \\ | \langle \text{NEGATION} \rangle$$

Eclipse highlighted these two rules as they were left-recursive so this problem was to be resolved manually by gathering the common things in the rule and put it in a temporary rule as follows:

$$\langle \text{MORECONJUNCTION} \rangle ::= 'and' \langle \text{CONJUNCTION} \rangle | \text{Epsilon}$$

$$\langle \text{CONJUNCTION} \rangle ::= \langle \text{NEGATION} \rangle \langle \text{MORECONJUNCTION} \rangle$$

and the same goes for *DISJUNCTION* rule:

$$\langle \text{MOREDISJUNCTION} \rangle ::= 'or' \langle \text{DISJUNCTION} \rangle | \text{Epsilon}$$

$$\langle \text{DISJUNCTION} \rangle ::= \langle \text{CONJUNCTION} \rangle \langle \text{MOREDISJUNCTION} \rangle$$

By this we eliminated the problem of left-recursive but this was not the efficient solution, because an additional rule for all the rules having left-recursive problem was not the optimal solution. Instead these two rules were modified to be as following:

$$\langle \text{CONJUNCTION} \rangle ::= \langle \text{NEGATION} \rangle ['and' \langle \text{CONJUNCTION} \rangle]?$$

$$\langle \text{DISJUNCTION} \rangle ::= \langle \text{CONJUNCTION} \rangle ['or' \langle \text{DISJUNCTION} \rangle]?$$

Also an obvious problem appeared in *SEQUENTIAL* rule as this rule references *STATEMENT* rule and referenced by *STATEMENT* rule as well.

```

<STATEMENT> :=                                <ASSIGNMENT>
              | <SEQUENTIAL>
              | <CONDITIONAL>
              | <ITERATIVE>
              | <PRINT>
              | <READ>
              | <INVOCATION>

```

```

<SEQUENTIAL> ::= <STATEMENT> ';' <STATEMENT>

```

A solution was proposed to get over such a problem and that was to eliminate

SEQUENTIAL

rule and replace it by another rule that does not reference

STATEMENT

. The following was the solution of such a problem:

```

<STATEMENT> :=                                <ASSIGNMENT> <MORESTATEMENT>
              | <CONDITIONAL><MORESTATEMENT>
              | <ITERATIVE><MORESTATEMENT>
              | <PRINT> <MORESTATEMENT>
              | <READ><MORESTATEMENT>
              | <INVOCATION><MORESTATEMENT>

```

```

<MORESTATEMENT> ::= ';' <STATEMENT> |

```

Grammar is now not left-recursive and contain no errors