UNIVERSITÉ LIBRE DE BRUXELLES

Project part 2: Parser

Subject : introduction to language theory and compiling

TRANGANIDAS Orestis
BOUGLAM Sara

Contents

1 Introduction

The compiling process goes through several phases, starting by the lexical analysis where the deterministic finite automaton was introduced as a tool to model and execute the process, resulting in the generation of a set of finite lexical units representing the whole program; afterwards, the syntactic analysis comes in second place, which treats the set of finite symbols resulted in the prior phase.

Deterministic finite automaton might be a suitable tool to tackle the parser generation, however, it is flawed and has numerous limitations, this can be noticed in the $L_{()}$; such a simple language that is not regular, therefor, the concept of grammar is brought up, which fill the gaps of the deterministic finite automaton .

Formally, grammar simply describes how the string is formed from a language's alphabets that match the language's syntax.

In this report, parser generation process will be addressed, initially, $LL(1)$ grammar accompanied with the symbol table are generated, afterward, as a final step the parser is implemented programmatically using recursive descent, additionally , creating the parse tree regarding the inputted Fortran program which validates its syntactic correctness.

2 Background

2.1 Parser

The parser is responsible of performing the syntactic analysis of a program, where it receives as an input the set of tokens generated by the combination of the scanner and the screener and output the syntax or the parse tree that represent the structure of the whole program which would be accessed by the compiler in a later phase, two type of parser can be distinguished; *top down parser* and *bottom up parser*.

When the syntax error occurs in a program, the parser is supposed to localized and report it, optionally, diagnose it and correct it (1).

2.2 Grammar

2.2.1 Definition

The syntax analysis is characterized by the grammar, which consists of the elementary component of a program. (1)

Grammar is a quadruplet (2) $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{S})$

$\boldsymbol{V}$ is a finite set of variables

$\boldsymbol{T}$ is a finite set of terminals

$\boldsymbol{P}$ is a finite set of production rules $\boldsymbol{\alpha} \to \mathbf{B}$ such that :

- $\boldsymbol{\alpha} \,\epsilon\, (\boldsymbol{V} \cup \boldsymbol{T})* \boldsymbol{V}(\boldsymbol{V} \cup \boldsymbol{T})*$
- $\boldsymbol{B} \,\epsilon\, (\boldsymbol{V} \cup \boldsymbol{T})*$

2.2.2 Chomsky hierarchy

Grammars are divided into 4 distinguishable classes

- *Class 0*: includes all the classes(1 ,2, 3)
- *Class 1*: Context Sensitive Grammar

$$\boldsymbol{\alpha} \to \boldsymbol{B} \in \boldsymbol{P} \mid \boldsymbol{\alpha} = \boldsymbol{S} \wedge \boldsymbol{B} = \boldsymbol{\varepsilon} \vee |\boldsymbol{\alpha}| \leq |\boldsymbol{B}|/\{\boldsymbol{S}\}$$

- *Class 2*: Regular Grammar

$$\boldsymbol{\alpha} \to \boldsymbol{B} \in \boldsymbol{P} \mid \boldsymbol{\alpha} \in \boldsymbol{V} \wedge |\boldsymbol{\alpha}| = \mathbf{1}$$

- *Class3:* Context free grammar: is comprised of :
    1. *Left regular grammar* $\boldsymbol{\alpha} \to \boldsymbol{B} \mid \boldsymbol{\alpha} \in \boldsymbol{P} \wedge (\boldsymbol{B} \in \boldsymbol{T}^{*} \vee \boldsymbol{B} \in \boldsymbol{T}^{*}.\boldsymbol{V})$

    2. *Right regular grammar* $\boldsymbol{\alpha} \to \boldsymbol{B} \mid \boldsymbol{\alpha} \in \boldsymbol{P} \wedge (\boldsymbol{B} \in \boldsymbol{T}^{*} \vee \boldsymbol{B} \in \boldsymbol{V}.\boldsymbol{T}^{*})$

The syntactic phase of a compiler is written using Context free grammar.

A context free language (**CFL**) is defined such that $\boldsymbol{L}(\boldsymbol{G}) = \boldsymbol{L}$ where G is context free grammar (2).

2.3 Grammar derivation:

The derivation of a string for a specific grammar is the process of generating a sequence of grammar rules application that does the start symbol transformation into the string (3). There exist two types of derivations, left most and right most one (2):

For a derivation$\boldsymbol{w}$, $\boldsymbol{S\,w'} \;=>\; \boldsymbol{w\,\alpha\,w'}$:

- Left most derivation requires that $\boldsymbol{w} \in \boldsymbol{T}^{*}$
- Right most derivation requires that $\boldsymbol{w'} \in \boldsymbol{T}^{*}$

In order to prove that given word belongs to the language of a grammar, *derivation tree* can be used, note that the tree is completed when leaves only contain terminals (2)

2.4 First^{k} and follow^{k}

2.4.1 First^{k}

$$\boldsymbol{First}^{\,k}\,(\boldsymbol{a}) = \{\boldsymbol{w} \;\in\; \boldsymbol{T}^{*} |\, \boldsymbol{a} =>^{*} \boldsymbol{wx} \;\wedge (\,|\boldsymbol{w}| = \boldsymbol{k} \;\vee\; |\boldsymbol{w}| < \boldsymbol{k}\;\; \boldsymbol{and}\; \boldsymbol{x} = \boldsymbol{\varepsilon})\}$$

2.4.2 follow^{k}

$$\boldsymbol{Follow}^{\,k}\,(\boldsymbol{a}) = \{\boldsymbol{w} \;\in\; \boldsymbol{T}^{*} |\{\exists\, \boldsymbol{\beta}\,,\boldsymbol{\gamma}\;|\boldsymbol{\beta} =>^{*} \boldsymbol{\beta a \gamma} \;\wedge\; \boldsymbol{w} \in \boldsymbol{First}^{\,k}\,(\boldsymbol{\gamma})\}\}$$

2.5 $LL(1)$ parser

LL refers to Left scanning Left parsing which means the input string is scanned from the left to the right. $LL(K)$ grammar uses K *look-ahead* which defines the generation of k *firsts* and *follows*, therefore, $LL(1)$ grammars can be defined as an $LL(k)$ grammar where $K \;=\; 1$

However, the grammar should adhere to a set of rules in order not to confuse the parser later on, for this, certain grammar transformation are required initially (2):

2.5.1 Useless rules removal

For a given grammar $G \;=\; (V, T\,, P, S\,)$, useless rules removal consists of eliminating the following the follow component:

- Unproductive variable: $\{\,there\; is\; no\; w \in\; T^{*} \mid\; A \;=>_{G}^{*}\;\; w\,\}$
- Unreachable symbols: $\{for\; X \in\;\; V \cup T\,,\; \nexists\;\; S =>_{G}^{*}\;\; \alpha 1\; X\; \alpha 2\,\}$

2.5.2 Ambiguity removal

Grammar is considered ambiguous In the cane when several derivative tree are extracted for the same word.

Ambiguity can be removed through adjusting the production rules such that, they respect a certain order (2).

2.5.3 Left recursion removal

Left recursion can be illustrated such that:

$$\boldsymbol{S -> Sa}$$

$$\boldsymbol{S -> c}$$

One can see that: $\boldsymbol{First\ (c)\ \in\ first(S)\ \subseteq\ first(Sa)}$ **,** which violate the $LL(1)$ grammar, this can be fixed by turning the left recursion into right recursion (2).

2.5.4 Left prefixes removal

The following grammar is left prefixed:

$$\boldsymbol{A -> aB}$$

$$\boldsymbol{A -> aC}$$

This grammar can be adjusted through factoring the common prefixes.

2.5.5 Action table

After transforming the grammar into $LL(1)$ one, the action table is considered as the core of the parser; it is obtained through the result of generating the follow1 and first1 from the transformed grammar, it is a two dimensional table M (2) :

- The lines are indexed by the elements from $T\ \cup V$
- Rows of M are indexed by a set of T
- Cells contain the action that the parser must perform

2.5.6 Recursive descent

Recursive descent is a type of top down parser which relies on the action table obtained as well as the $LL(1)$ grammar (2):

For the input, it receives the following:

- $LL(1)\ CFG\ = (P, T\ , V, S)$ accompanied with the action table and an input word
- $w\ = w1\ , w2\ , .. \in T^*$

And outputs :$True\ iff\ \ w \in L(G)$, the sequence of rules number is printed in a left most derivation

3 Implementation

3.1 Overview

The process of generating the parser was divided into two main phases, the first phase required manual intervention which consisted of transforming the ***Fortran*** grammar to $\boldsymbol{LL}(\mathbf{1})$ and action table generation, then the second phase relied on the results obtained from the first phase, eventually, implementing the recursive descent using java programming language, then generating the parse tree.

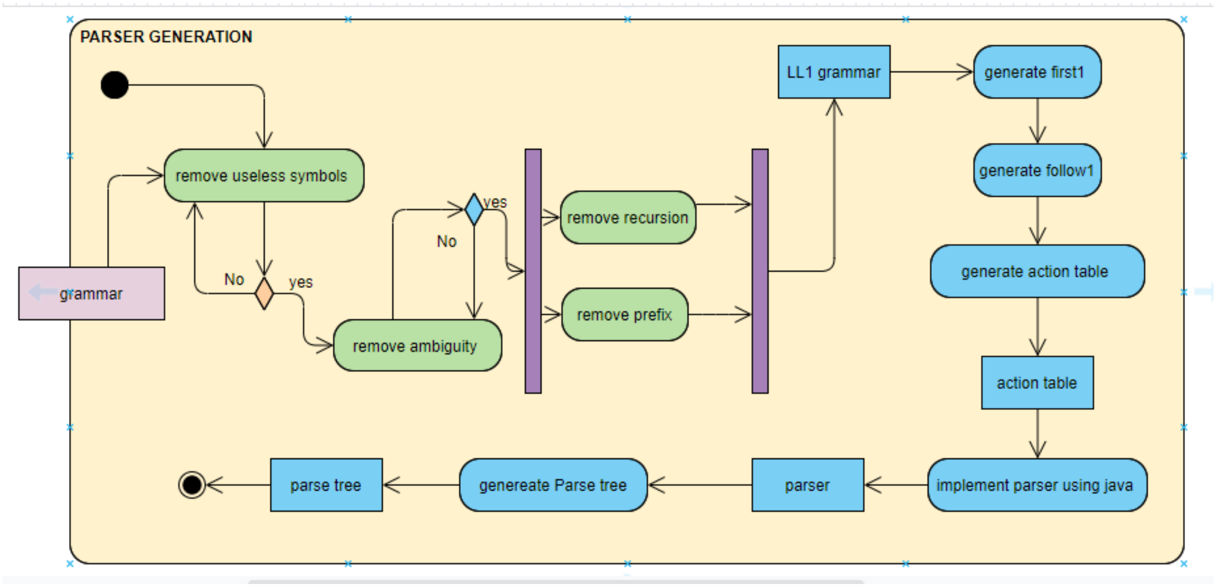


Figure 1: An activity diagram[1] that summarize the overall process *using visual paradigm web app*[2]

3.2 Phase 1

The subsequent sections explain the whole process followed in order to generate the action table

3.2.1 Unproductive characters

Considering that the set of terminals corresponds to the units contained in the Symbol Class

I	V_i
0	ϕ
1	{<Read>}
2	{<Read>, <Print> }
3	{<Read>, <Print>, <While>}
4	{ <Read>, <Print>, <While>, <Comp> }

[1] **Activity Diagram :** is a fllow chart that represents the flow from one activity to another in a system, this activity can be described as an operation (6)
[2] **Online visual paradigm:** is a web app used to generate different template and diagrams https://online.visual-paradigm.com/

5	{<Read>, <Print>, <While>, <Comp> <Cond>}
6	{<Read>, <Print>, <While>, <Comp> <Cond>,<If>}
7	{<Read>, <Print>, <While>, <Comp> <Cond>,<If>,<Op>}
8	{<Read>, <Print>, <While>, <Comp> <Cond>,<If>,<Op>,<ExprArith>}
9	{<Read>, <Print>, <While>, <Comp> <Cond> ,<If> ,<Op> ,<ExprArith> ,<Assign>}
10	{<Read>, <Print>, <While>, <Comp> <Cond>, <If>,<Op>, <ExprArith> , <Assign> , <Instruction>}
12	{<Read>, <Print>, <While>, <Comp> <Cond>,<If> ,<Op>, <ExprArith>, <Assign> , <Instruction>,<Code>}
13	{<Read>, <Print>, <While>, <Comp> <Cond>, <If>,<Op>, <ExprArith>, <Assign> , <Instruction> , <Code>, <Program>}

Observation: There are no unproductive rules in the given grammar

3.2.2 Unreachable characters

I	V_i
0	{<Program>}
1	{<Program> , <Code>}
2	{<Program>, <Instruction>}
3	{<Program>,<Instruction>, <Assign> ,<While> ,<Print>,<Read>}
4	{<Program>,<Instruction>,
5	<Assign>,<While>,<Print>,<Read>,<ExprArith>}
6	{<Program>,<Instruction>, <Assign>,<While>,<Print>,<Read>,<ExprArith>,<Op>}
7	{<Program>,<Instruction>, <Assign>,<While>,<Print>,<Read>,<ExpArith>,<Op>, <If>}
8	{<Program>,<Instruction>, <Assign>,<While>,<Print>,<Read>,<ExprArith>,<Op>, <If>}
9	{<Program>,<Instruction>, <Assign>,<While>,<Print>,<Read>,<ExprArith>,<Op>,<If>,<Cond>}
	{<Program>,<Instruction>,<Assign>,<While>,<Print>,<Read>,<ExprArith>,<Op>, <If>,<Code>}

Observation: There are no unreachable rules in the given grammar

To sum up, there are no useless rules.

3.2.3 Ambiguous grammar

In the given grammar, there is ambiguity regarding the addition and multiplication operation which need to be adjusted according to their order

```
<ExprArith> → [VarName]

            → [Number]

            → ( <ExprArith> )

            → - <ExprArith>
            → <ExprArith> <Op> <ExprArith>
 <Op> → +
     → -
     → *
     → /
```

After the adjustment by prioritizing the multiplication and division, the above grammar is obtained

```
<ExprArith> -> <ExprArith> + <Multiplication>
           -> <ExprArith> - <Multiplication>
           -> <Multiplication>
<Multiplication> -> <Multiplication> * <Bracket>
                 -> <Multiplication> / <Bracket >
                 -> <Braquet>
<Bracket> -> (<ExprArith>)
          -> <Var>

<Var> -> [VarName]
       → [Number]
       → - <Var>
```

3.2.4 Left Factory

Now that ambiguity and useless rules haves ben eliminated, there is a noticeable prefix redundancy in the following rules:

```
<If> → IF (<Cond>) THEN [EndLine] <Code> ENDIF
<If> → IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF
```

After the removal of the left factory we get

```
<If> → IF (<Cond>) THEN [EndLine] <Code> <If”>
<If”> -> ENDIF
<If”> → ELSE [EndLine] <Code> ENDIF
```

3.2.5 Left recursion

Left recursion is noticed in the rule bellow:

```
<ExprArith> -> < ExprArith > + <Multiplication>
           -> < ExprArith > - <Multiplication>
           -> <Multiplication>
<Multiplication> -> <Multiplication> * <Braquet>
                 -> <Multiplication> / <Braquet >
                 -> <Braquet>
```

After the removal of left recursion:

```
< ExprArith > -> <ExprArith'> < ExprArith''>
< ExprArith''> -> + <Multiplication>< ExprArith''>
               -> - <Multiplication>< ExprArith''>
               ->  ε
<ExprArith'> -> <Multiplication>

<Multiplication> -> <Multiplication'> <Multiplication''>
<Multiplication''> -> * <Braquet> <Multiplication''>
                   -> / <Braquet> <Multiplication''>
                   ->ε
 <Multiplication'>-> <Bracket>
```

3.2.6 LL(1) grammar

In the end, the following $LL(1)$ grammar is obtained

```
[1] <S>  -> <Program>$
[2] <Program> → BEGINPROG [ProgName] [EndLine] <Code> ENDPROG
[3] <Code> → <Instruction> [EndLine] <Code>
[4]        → ε
[5] <Instruction> → <Assign>
```

```
[6]               → <If>
[7]               → <While>
[8]               → <Print>
[9]               → <Read>
[10] <Assign> → [VarName] := <ExprArith>
[11] <ExprArith > -> <ExprArith’> < ExprArith’’>
[12] <ExprArith’’> -> + <Multiplication>< ExprArith’’>
[13]              -> - <Multiplication>< ExprArith’’>
[14]              ->  ε
[15] <ExprArith’> -> <Multiplication>
[16] <Multiplication> -> <Multiplication’> <Multiplication’’>
[17] <Multiplication’’> -> * <Braquet> <Multiplication’’>
[18]                  -> / <Braquet> <Multiplication’’>
[19]                  ->ε
[20] <Multiplication’> -> <Bracket>
[21] <Bracket> -> (ExprArith)
[22]           -> <Var>
[23] <Var> -> [VarName]
[24]       → [Number]
[25]       → - <Var>
[26] <If> → IF (<Cond>) THEN [EndLine] <Code> <If”>
[27] <If”> -> ENDIF
[28] <If”> → ELSE [EndLine] <Code> ENDIF
[29] <Cond> → <ExprArith> <Comp> <ExprArith>
[30] <Comp> → =
[31]        → >
[32] <While> → WHILE (<Cond>) DO [EndLine] <Code> ENDWHILE
[33] <Print> → PRINT([VarName])
[34] <Read> → READ([VarName])
```

3.2.7 First^1 and Follow^1

The last step before the action table generation is the $First1$ and $Follow1$ extraction:

Symbol	**First¹**	**Follow¹**
<S>	BEGINPROG	
<Program>	BEGINPROG	$
<Code>	VarName IF WHILE PRINT READ ε	ENDPROG ENDIF ELSE ENDWHILE
<Instruction>	VarName IF WHILE PRINT READ	ENDLINE
<Assign>	VarName	ENDLINE
<ExprArith>	(VarName Number -	ENDLINE)
<ExprArith’’>	+ - ε	ENDLINE)
<ExprArith’>	(VarName Number -	+ -
<Multiplication>	(VarName Number -	+ -
<Multiplication’’>	/ * ε	+ -
<Multiplication’>	(VarName Number -	/ *
<Bracket>	(VarName Number -	/ *
<Var>	VarName Number -	/ *
<If>	IF	ENDLINE
<If”>	ENDIF , ELSE	ENDLINE
<Cond>	(VarName Number -	)
<Comp>	= >	(VarName Number -
<While>	WHILE	ENDLINE
<Print>	PRINT	ENDLINE
<Read>	READ	ENDLINE

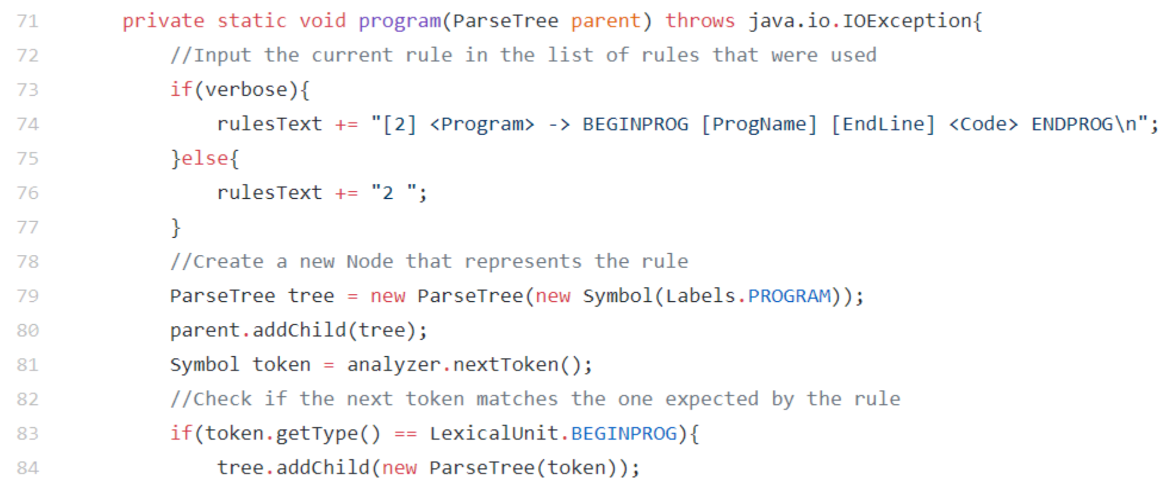
3.2.8 Action Table

	BEGINPROG	PROGNAME	(	)	+	-	*	/	=	>	IF	ELSE	ENDIF	WHILE	ENDWHILE	VARNAME	Number	PRINT	READ	ENDPROG	ENDLINE
<S>	1																				
<Program>	2																				
<CODE>											3	4	4	3	4	3		3	3	4	

<Instruction>											6			7		5		8	9		
<Assign>																10					
<ExprArith>			11			11										11	11				
<ExprArith’’>				14	12	13															14
<ExprArith’>			15			15										15	15				
<Multiplication>			16			16										16	16				
<Multiplication’’>					19	19	17	18													
<Multiplication’>			20			20										20	20				
<Bracket>			21			22										22	22				
<Var>						25										23	24				
<If>											26										
<If’’>											27	28									
<Cond>			29			29										29	29				
<Comp>									30	31											
<While>														32							
<Print>																		33			
<Read>																			34		

3.3 Phase 2

This phase consisted of implementing the recursive descent using Java programing language; the whole idea behind this mechanism consists of generating method for each production rule, and each method is contained into another which receives as a parameter the parent node of the tree and optionally the previously generated token, both of these parameters are used for generating the parse tree.



```
71      private static void program(ParseTree parent) throws java.io.IOException{
72          //Input the current rule in the list of rules that were used
73          if(verbose){
74              rulesText += "[2] <Program> -> BEGINPROG [ProgName] [EndLine] <Code> ENDPROG\n";
75          }else{
76              rulesText += "2 ";
77          }
78          //Create a new Node that represents the rule
79          ParseTree tree = new ParseTree(new Symbol(Labels.PROGRAM));
80          parent.addChild(tree);
81          Symbol token = analyzer.nextToken();
82          //Check if the next token matches the one expected by the rule
83          if(token.getType() == LexicalUnit.BEGINPROG){
84              tree.addChild(new ParseTree(token));
```

Figure 2 Excerpt from the program implemented in the *parser* class

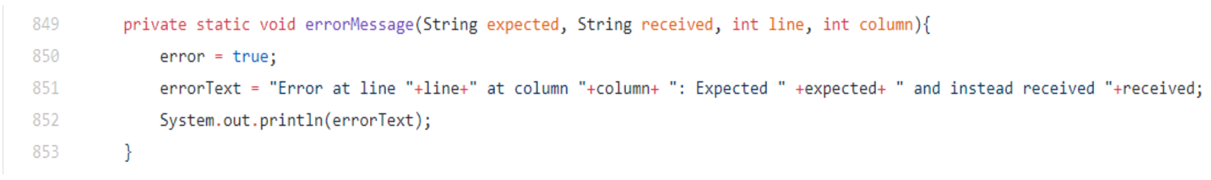
Observation:

In the excerpt above, the $< program >$ production rule is displayed, receives as a parameter the parent node which is also the start of the tree.

When the verbose variable (line 73) is set to true, extra information about the parsing would be provided, which consists of displaying the whole production rule.

3.3.1 Error message

Whenever a syntactic error occurs in the input file, an error message is displayed to the standard output stream, which precise the error, the expected token and the exact position (line and column).



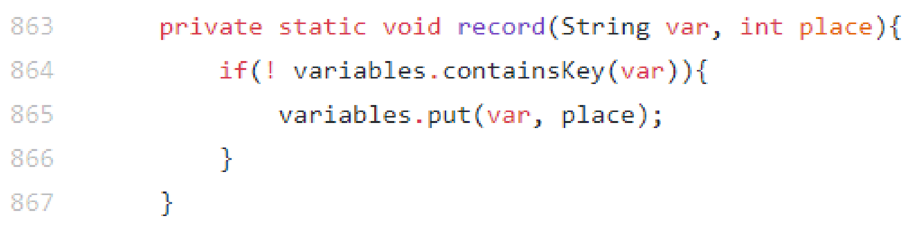
```
    private static void errorMessage(String expected, String received, int line, int column){
        error = true;
        errorText = "Error at line "+line+" at column "+column+ ": Expected " +expected+ " and instead received "+received;
        System.out.println(errorText);
    }
```

Figure 3 : The error message implemented

3.3.2 Extra feature

Additionally, another feature has been added to the $Parser$ class that is the variable recording which consists of storing the program variables in a data structure, for later use.

This feature has been implemented using the map [3]data structure; more precisely the $SortedMap$ [4], the var argument is simply map key which refers the variable name and the place $arguments$ represents the line where the variable appeared



```java
private static void record(String var, int place){
    if(! variables.containsKey(var)){
        variables.put(var, place);
    }
}
```

Figure 4: Variable recording method

3.3.3 Main class

The main class simply, put all the pieces together: receiving the following arguments from the command line:

1. $-v$ stands for the verbose(whether to display extra information about the production rules)
2. $-wt$ stands for the file to display the parse tree, $.tex$ extension is expected, else, an error is returned
3. $Fortran$ program file ($.fs$ extension is expected)

Afterward, parsing the program and outputting the parse tree into the latex file.

[3] **Map**, java util package's class data structure that consists of mapping a specific value to a key, where each key can map at most one value (4)
[4] **Sorted map** implements the *map* interface which provide a totally ordering of its keys according to a natural order or by comparator (5)

4 Conclusion

To sum up, parser generation relied on two main steps, the first step was reformulating the grammar into an appropriate one, that can be understood by the parser which is LL(1) grammar, then, filling the action table through the obtained first 1 and follow1, the second step consisted of implementing the recursive descent using java programming language, which included some additional features, namely : recognizing syntactic errors and their exact position, generating a parse tree for the programs which language is syntactically correct and recording variable appearance.

Bibliography

1. **Reinhard Wilhelm, Helmut Seidl, Sebastian Hack.** *Compiler Design .*

2. **G I L L E S G E E R A E R T S, G U I L L E R M O A . P É R E Z.** *Introduction to language theory and compilig.*

3. Context-free grammar. *Wikipedia .* [Online] https://en.wikipedia.org/wiki/Context-free_grammar#Derivations_and_syntax_trees.

4. Interface Map. *Java docs .* [Online] https://docs.oracle.com/javase/7/docs/api/java/util/Map.html.

5. SortedMap. *Java Doc.* [Online] https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html.

6. tutorials point. *UML - Activity Diagrams.* [Online] https://www.tutorialspoint.com/uml/uml_activity_diagram.htm.

7. *Introduction to language theory and compillig .*