UNIVERSITÉ LIBRE DE BRUXELLES

FORTR-S compiler

INFO-F403 - PART 2

Baudru Julien

November 28, 2020

Contents

1 Introduction

The main goal of this project is to create a parser for our **FORTR-S** compiler. More precisely, the parser created here is a recursive top-down LL(1) parser that takes as input a list of symbols (or tokens) obtained from the lexical analyzer created in the previous phase. For this phase of the project, our compiler will output the set of grammar rules applied to the received input as well as the derivation tree.

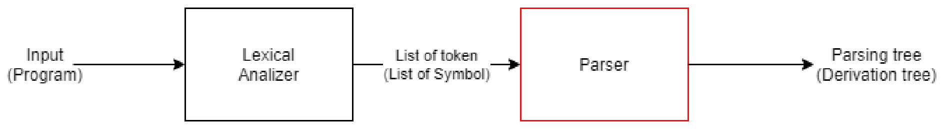

Figure 1: Step of our compiler for the moment

2 User manual

To compile this program you will need to enter the following commands :

```
(jflex src/LexicalAnalyzer.flex)
javac -d bin -cp src/ src/Main.java
jar cfe dist/Part2.jar Main -C bin
```

To run this program you can either run the following command :

```
java -jar dist/Part2.jar test/Factorial.fs
```

Or this one with the *-v* option (for *verbose*) which gives more details on the rules used during the derivation :

```
java -jar dist/Part2.jar -v test/Factorial.fs
```

Or the following one with the *-wt* option which indicates that the user wants to create a *Latex* file containing the derivation tree :

```
java -jar dist/Part2.jar -v -wt more/tree.tex test/Factorial.fs
```

When the user enters the *-v* option, the parser will print details of the rules used in the following way:

```
Detail (-v) :

1  : <Program> -> BEGINPROG <Program'>
2  : <Program'> -> [ProgName] <Program''>
3  : <Program''> -> [Endline] <Code> ENDPROG
4  : <Code> -> <Instruction> <Code'>
11 : <Instruction> -> <Read>
43 : <Read> -> READ ( [VarName] )
4  : <Code> -> <Instruction> <Code'>
...
```

3 Structure

3.1 Overall structure

The structure of the project is the following :

```
part2:
|
|---dist
|     |---part2.jar
|
|---doc
|     |---Javadoc
|     |     |...
|     |
|     |---Report.pdf
|
|---more
|     |---tree.tex
|
|---src
```

```
|     |---LexicalAnalizer.java
|     |---LexicalAnalyzer.flex
|     |---LexicalUnit.java
|     |---Main.java
|     |---Parser.java
|     |---ParseTree.java
|     |---Symbol.java
|
|---test
      |---Factorial.fs
      |---Fibonacci.fs
      |---TestOp.fs
      |...
```

The *Lexical Analyzer* and *ParseTree* classes are identical to the versions given in the correction of part 1 of this project. Concerning the *Symbol* class, a new function called *toTexString()* has been added in order to get the value of the symbol in a the *String* type.

Note that when the *-wt* option is entered by the user, the Latex file containing the generated bypass tree will be placed in the more folder of the project.

About the *Paser* class, each function in this class corresponds to variable present in the grammar. Each of these functions can be divided into three main parts, a first one for the construction of the derivation tree, a second one for displaying the rules and a last one for error handling.

Concretely, we iterate on the symbols present in the token list, if the token is recognized by a rule, then we go to the next rule and the next token (c.f. *this.index += 1;*). If this is not the case then there is a syntax error in the program and the *printError* function is called to display the source of the problem and stop the execution.

Each function takes as a parameter a list of derivation trees, i.e. an object of type *ParseTree*, the trees that we will add to this list will be the children of the calling function. The root of the tree from which we pass the children is created in the parent class, it is a symbol created from a special *Lexical Unit* named *LABELPARSE-TREE*, so these symbols will have as value the name of the rule concerned (ex : *¡Assign¿*). Since the parser we created is a recursive descent parser, the functions are called recursively .

The first function (or rule) called is the *program* function, the tree will then be progressively enlarged during the different calls that will occur and finally before the return of the *program* function the bypass tree of the class (c.f. *this.parsetree*) will be initialized with the tree created during the recursive calls.

Concerning the display of rules, each time a token is recognized by a rule, the number of the rule in question is added to a list (c.f. *this.lrules*). If the user has asked for more details via the *-v* option, then the number and syntax of the rule in question is added to a second list (c.f. *this.lverbrules*).

4 Grammar

4.1 Initial grammar

Below you will find a reminder of the initial grammar rules of the **FORTR-S** language the grammar given in the statement of this project.

[1] ⟨Program⟩ → BEGINPROG [ProgName] [Endline] ⟨Code⟩ ENDPROG
[2] ⟨Code⟩ → ⟨Instruction⟩ [Endline] ⟨Code⟩
[3] ⟨Code⟩ → ε
[4] ⟨Instruction⟩ → ⟨Assign⟩
[5] ⟨Instruction⟩ → ⟨If⟩
[6] ⟨Instruction⟩ → ⟨While⟩
[7] ⟨Instruction⟩ → ⟨Print⟩
[8] ⟨Instruction⟩ → ⟨Read⟩
[9] ⟨Assign⟩ → [Varname] := ⟨ExprArith⟩
[10] ⟨ExprArith⟩ → [Varname]
[11] ⟨ExprArith⟩ → [Number]
[12] ⟨ExprArith⟩ → (⟨ExprArith⟩)
[13] ⟨ExprArith⟩ → - ⟨ExprArith⟩
[14] ⟨ExprArith⟩ → ⟨ExprArith⟩ ⟨Op⟩ ⟨ExprArith⟩
[15] ⟨Op⟩ → +
[16] ⟨Op⟩ → -
[17] ⟨Op⟩ → *
[18] ⟨Op⟩ → /
[19] ⟨If⟩ → IF (⟨Cond⟩) THEN [Endline] ⟨Code⟩ ENDIF
[20] ⟨If⟩ → IF (⟨Cond⟩) THEN [Endline] ⟨Code⟩ ELSE [Endline] ⟨Code⟩ ENDIF
[21] ⟨Cond⟩ → ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[22] ⟨Comp⟩ → >
[23] ⟨Comp⟩ → =
[24] ⟨While⟩ → WHILE (⟨Cond⟩) DO [Endline] ⟨Code⟩ ENDWHILE
[25] ⟨Print⟩ → PRINT([VarName])
[26] ⟨Read⟩ → READ([VarName])

4.2 Modified grammar

4.3 First step

Below, the grammar after removing the ambiguity and taking into account the priority and associability of the operators. The ambiguity in the original grammar was in rules 19 and 20.

[1] ⟨Program⟩ → BEGINPROG [ProgName] [Endline] ⟨Code⟩ ENDPROG

[2] ⟨Code⟩ → ⟨Instruction⟩ [Endline] ⟨Code⟩

[3] ⟨Code⟩ → ε

[4] ⟨Instruction⟩ → ⟨Assign⟩

[5] ⟨Instruction⟩ → ⟨If⟩

[6] ⟨Instruction⟩ → ⟨While⟩

[7] ⟨Instruction⟩ → ⟨Print⟩

[8] ⟨Instruction⟩ → ⟨Read⟩

[9] ⟨Assign⟩ → [Varname] := ⟨ExprArith⟩

[10] ⟨ExprArith⟩ → ⟨ExprArith⟩ + ⟨Prod⟩

[11] ⟨ExprArith⟩ → ⟨ExprArith⟩ - ⟨Prod⟩

[12] ⟨ExprArith⟩ → ⟨Prod⟩

[13] ⟨Prod⟩ → ⟨Prod⟩ * ⟨Atom⟩

[14] ⟨Prod⟩ → ⟨Prod⟩ / ⟨Atom⟩

[15] ⟨Prod⟩ → ⟨Atom⟩

[16] ⟨Atom⟩ → - ⟨Atom⟩

[17] ⟨Atom⟩ → [Varname]

[18] ⟨Atom⟩ → [Number]

[19] ⟨Atom⟩ → (⟨ExprArith⟩)

[20] ⟨If⟩ → IF (⟨Cond⟩) THEN [Endline] ⟨Code⟩ ⟨IfSep⟩

[21] ⟨IfSep⟩ → ENDIF

[22] ⟨IfSep⟩ → ELSE [Endline] ⟨Code⟩ ENDIF

[23] ⟨Cond⟩ → ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩

[24] ⟨Comp⟩ → >

[25] ⟨Comp⟩ → =

[26] ⟨While⟩ → WHILE (⟨Cond⟩) DO [Endline] ⟨Code⟩ ENDWHILE

[27] ⟨Print⟩ → PRINT([VarName])

[28] ⟨Read⟩ → READ([VarName])

4.4 Second step

Below, the grammar after removing the left-recursion for the operator.

[1] ⟨Program⟩ → BEGINPROG [ProgName] [Endline] ⟨Code⟩ ENDPROG

[2] ⟨Code⟩ → ⟨Instruction⟩ [Endline] ⟨Code⟩

[3] ⟨Code⟩ → ε

[4] ⟨Instruction⟩ → ⟨Assign⟩

[5] ⟨Instruction⟩ → ⟨If⟩

[6] ⟨Instruction⟩ → ⟨While⟩

[7] ⟨Instruction⟩ → ⟨Print⟩

[8] ⟨Instruction⟩ → ⟨Read⟩

[9] ⟨Assign⟩ → [Varname] := ⟨ExprArith⟩

[10] ⟨ExprArith⟩ → ⟨Prod⟩ ⟨ExprArith'⟩

[11] ⟨ExprArith'⟩ → + ⟨Prod⟩ ⟨ExprArith'⟩

[12] ⟨ExprArith'⟩ → - ⟨Prod⟩ ⟨ExprArith'⟩

[13] ⟨ExprArith'⟩ → ε

[14] ⟨Prod⟩ → ⟨Atom⟩ ⟨Prod'⟩

[15] ⟨Prod'⟩ → * ⟨Atom⟩ ⟨Prod'⟩

[16] ⟨Prod'⟩ → / ⟨Atom⟩ ⟨Prod'⟩

[17] ⟨Prod'⟩ → ε

[18] ⟨Atom⟩ → - ⟨Atom⟩

[19] ⟨Atom⟩ → [Varname]

[20] ⟨Atom⟩ → [Number]

[21] ⟨Atom⟩ → (⟨ExprArith⟩)

[22] ⟨If⟩ → IF (⟨Cond⟩) THEN [Endline] ⟨Code⟩ ⟨IfSep⟩

[23] ⟨IfSep⟩ → ENDIF

[24] ⟨IfSep⟩ → ELSE [Endline] ⟨Code⟩ ENDIF

[25] ⟨Cond⟩ → ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩

[26] ⟨Comp⟩ → >

[27] ⟨Comp⟩ → =

[28] ⟨While⟩ → WHILE (⟨Cond⟩) DO [Endline] ⟨Code⟩ ENDWHILE

[29] ⟨Print⟩ → PRINT([VarName])

[30] ⟨Read⟩ → READ([VarName])

4.5 Last step

Below is the final grammar after removing the left-recursion where it was needed. The numbers of the rules in this grammar correspond to the numbers printed by our parser.

[1] ⟨Program⟩ → BEGINPROG ⟨Program’⟩
[2] ⟨Program’⟩ → [ProgName] ⟨Program”⟩
[3] ⟨Program”⟩ → [Endline] ⟨Code⟩ ENDPROG
[4] ⟨Code⟩ → ⟨Instruction⟩ ⟨Code’⟩
[5] ⟨Code⟩ → ε
[6] ⟨Code’⟩ → [Endline] ⟨Code⟩
[7] ⟨Instruction⟩ → ⟨Assign⟩
[8] ⟨Instruction⟩ → ⟨If⟩
[9] ⟨Instruction⟩ → ⟨While⟩
[10] ⟨Instruction⟩ → ⟨Print⟩
[11] ⟨Instruction⟩ → ⟨Read⟩
[12] ⟨Assign⟩ → [Varname] ⟨Assign’⟩
[13] ⟨Assign’⟩ → := ⟨ExprArith⟩
[14] ⟨ExprArith⟩ → ⟨Prod⟩ ⟨ExprArith’⟩
[15] ⟨ExprArith’⟩ → + ⟨Prod⟩ ⟨ExprArith’⟩
[16] ⟨ExprArith’⟩ → - ⟨Prod⟩ ⟨ExprArith’⟩
[17] ⟨ExprArith’⟩ → ε
[18] ⟨Prod⟩ → ⟨Atom⟩ ⟨Prod’⟩
[19] ⟨Prod’⟩ → * ⟨Atom⟩ ⟨Prod’⟩
[20] ⟨Prod’⟩ → / ⟨Atom⟩ ⟨Prod’⟩
[21] ⟨Prod’⟩ → ε
[22] ⟨Atom⟩ → - ⟨Atom⟩
[23] ⟨Atom⟩ → [Varname]
[24] ⟨Atom⟩ → [Number]
[25] ⟨Atom⟩ → (⟨ExprArith⟩)
[26] ⟨If⟩ → IF ⟨If’⟩
[27] ⟨If’⟩ → (⟨Cond⟩ ⟨If”⟩
[28] ⟨If”⟩ →) ⟨If”’⟩
[29] ⟨If”’⟩ → THEN ⟨If””⟩
[30] ⟨If””⟩ → [Endline] ⟨Code⟩ ⟨IfSep⟩
[31] ⟨IfSep⟩ → ENDIF
[32] ⟨IfSep⟩ → ELSE ⟨IfSep’⟩
[33] ⟨IfSep’⟩ → [Endline] ⟨Code⟩ ENDIF
[34] ⟨Cond⟩ → ⟨ExprArith⟩ ⟨Comp⟩ ⟨ExprArith⟩
[35] ⟨Comp⟩ → >
[36] ⟨Comp⟩ → =
[37] ⟨While⟩ → WHILE ⟨While’⟩
[38] ⟨While’⟩ → (⟨Cond⟩ ⟨While”⟩
[39] ⟨While”⟩ →) ⟨While”’⟩
[40] ⟨While”’⟩ → DO ⟨While””⟩
[41] ⟨While””⟩ → [Endline] ⟨Code⟩ ENDWHILE
[42] ⟨Print⟩ → PRINT([VarName])
[43] ⟨Read⟩ → READ([VarName])

4.6 LL(1)

4.6.1 Action table

Below, the action table corresponding to the final modified grammar :

	BEGINPROG	[ProgName]	[Endline]	ENDPROG	[VarName]	:=	+	-	*	/	[Number]	(	)	IF	THEN	ENDIF	ELSE	⟩	=	WHEN	DO	ENDWHILE	PRINT	READ
⟨Program⟩	1																							
⟨Program'⟩		2																						
⟨Program''⟩			3																					
⟨Code⟩				5	4									4		5				4		5	4	4
⟨Code''⟩			6																					
⟨Instruction⟩					7									8						9			10	11
⟨Assign⟩					12																			
⟨Assign'⟩						13																		
⟨ExprArith⟩					14			14			14	14												
⟨ExprArith'⟩							15	16					17					17	17					
⟨Prod⟩					18			18			18	18												
⟨Prod'⟩									19	20			21					21	21					
⟨Atom⟩					23			22			24	25												
⟨If⟩														26										
⟨If'⟩												27												
⟨If''⟩													28											
⟨If'''⟩															29									
⟨If''''⟩			30																					
⟨IfSep⟩																31	32							
⟨IfSep'⟩			33																					
⟨Cond⟩					34			34			34	34												
⟨Comp⟩																		35	36					
⟨While⟩																				37				
⟨While'⟩												38												
⟨While''⟩													39											
⟨While'''⟩																				40				
⟨While''''⟩			41																					
⟨Print⟩																						42		
⟨Read⟩																							43	

We notice that the new grammar is LL(1) and deterministic..

4.6.2 First and Follow sets

In the current section, you will find the computation of the relevant First and Follow sets used to produce the above table.

First(Program) = {*BEGINPROG*}

First(Program') = {*[ProgName]*}

First(Program") = {*[Endline]*}

First(Code) = First(Instruction) & Follow(Code)
= {*[Varname], IF, WHILE, PRINT, READ, ENDPROG, ENDIF, ELSE, ENDWHILE*}

First(Code') = {*[Endline]*}

First(Instruction) = {*[Varname], IF, WHILE, PRINT, READ*}

First(Assign) = {*[Varname]*}

First(Assign') = {*:=*}

First(ExprArith) = Frist(Prod) = First(Atom) = {-, *[Varname], [Number], (*}

First(ExprArith') = {*+, -*} & Follow(ExprArith') = {*+, -,), >, =*}

First(Prod) = First(Atom) = {-, *[Varname], [Number], (*}

First(Prod') = { **, /*} & Follow(Prod') = { **, /*} & Follow(Prod) = { **, /*} & Follow(ExprArith') = { **, /,), >, =*}

First(Atom) = {-, *[Varname], [Number], (*}

First(If) = {*IF*}

First(If') = {*(*}

First(If") = {*)*}

First(If”’) = {*THEN*}

First(If””) = {*[Endline]*}

First(IfSep) = {*ENDIF, ELSE*}

First(IfSep’) = {*[Endline]*}

First(Cond) = First(ExprArith) = (Prod) = First(Atom) = {-, *[Varname], [Number], (*}

First(Comp) = {*>, =*}

First(While) = {*WHILE*}

First(While’) = {*(*}

First(While”) = {*)*}

First(While”’) = {*DO*}

First(While””) = {*[Endline]*}

First(Print) = {*PRINT*}

First(Read) = {*READ*}

5 Example N°1

5.1 Code

The only purpose of this first example is to show that the parser built for this project takes into account the priorities of the operators.

```
BEGINPROG TestOp
x := 3 + 7 * 7
y := 10 * 5 - 7
ENDPROG
```

Note : The code presented here is in the *test* folder of this project, it has the name *TestOp.fs*.

5.2 Leftmost derivation

Here is the list of rules applied during the derivation of this program :

```
1 2 3 4 7 12 13 14 18 24 21 15 18 24 19 24 21 17 4 7 12 13 14 18 24 19 24 21 16 18 24 21 17 5
```

5.3 Derivation tree

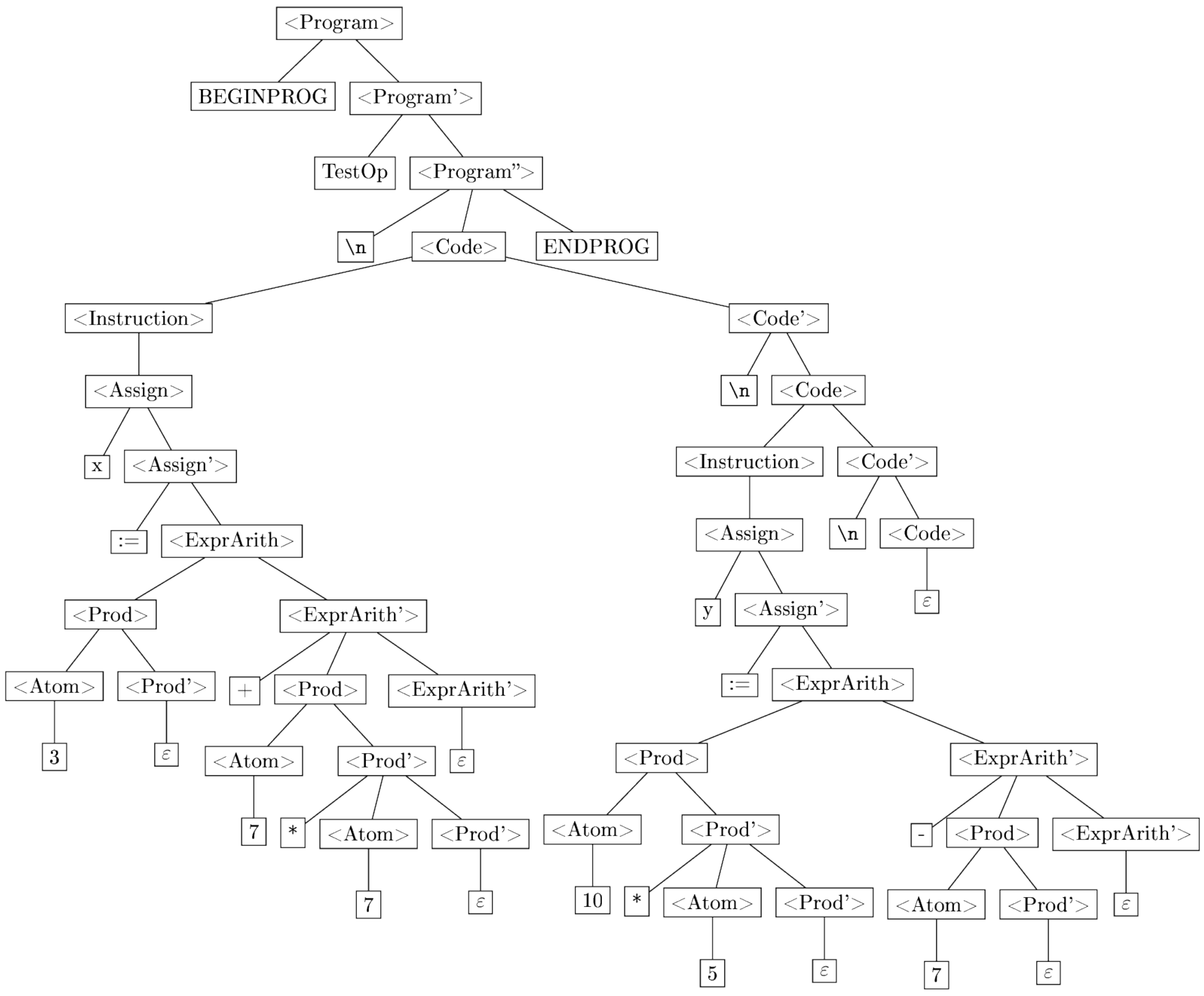

Figure 2: Derivation tree

6 Example N°2

6.1 Code

The example below is a very simple program, it asks the user for a number and if this number is positive then it prints 1, otherwise it prints -1.

```
BEGINPROG Example
/* Prints 1 if the input is a postive value,
otherwise prints -1.
*/
READ(input)
IF (input > 0) THEN //Positive value
    res := 1
ELSE                //NEgative value
    res := -1
ENDIF
PRINT(res)
ENDPROG
```

Note : The code presented here is in the *test* folder of this project, it has the name *Test.fs*.

6.2 Leftmost derivation

Here is the list of rules applied during the derivation of this program :

```
1 2 3 4 11 43 4 8 26 27 34 14 18 23 21 17 36 14 18 24 21 17 28 29 30 4 7 12
13 14 18 24 21 17 5 32 33 4 7 12 13 14 18 22 24 21 17 5 4 10 42 5
```

6.3 Derivation tree

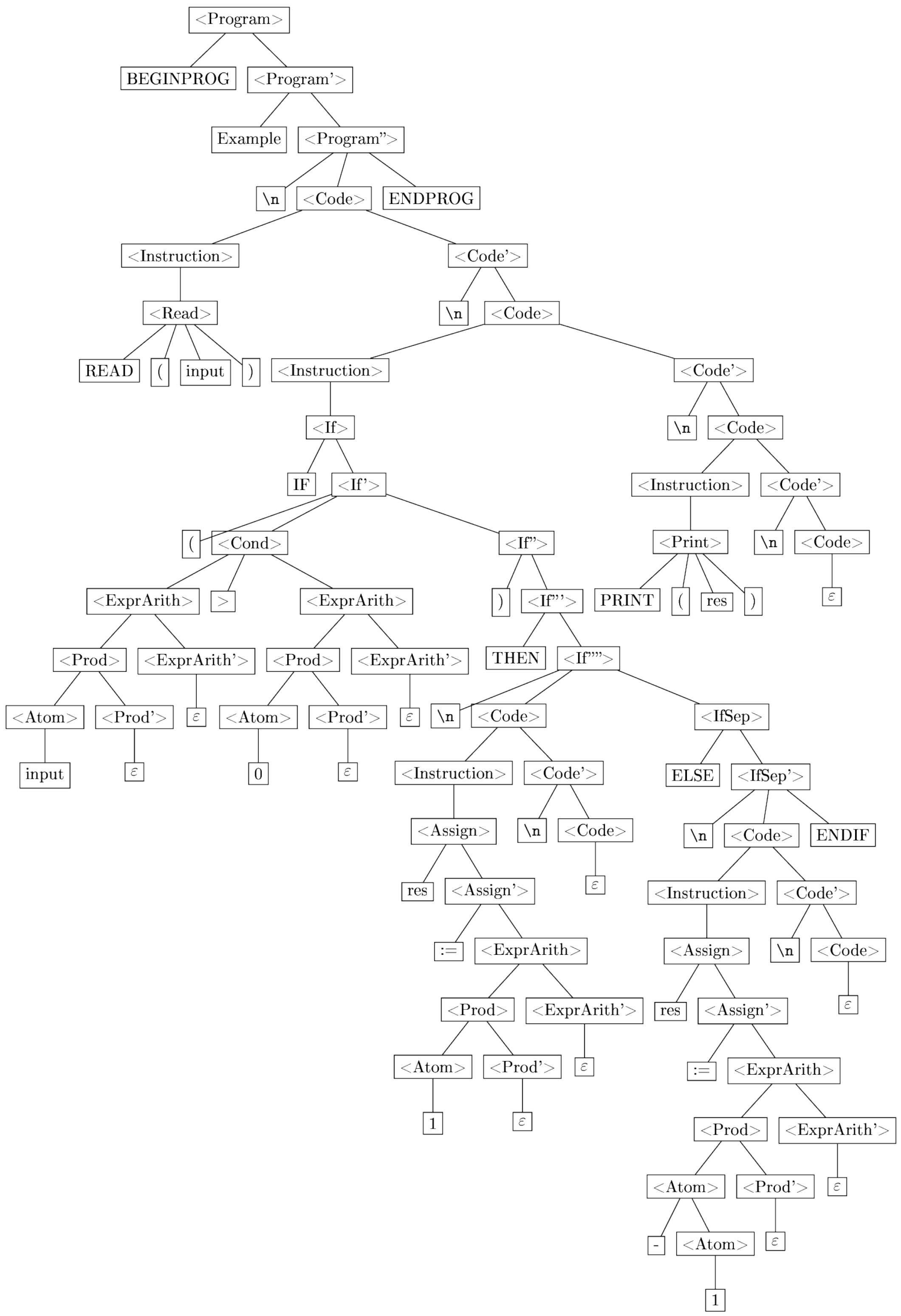

Figure 3: Derivation tree

7 Example N°3

7.1 Code

Below, the same example advanced in Part 1 of this project, this program asks the user to enter a number x and then displays all the numbers in the Fibonacci sequence up to the x^{th} number.

```
BEGINPROG Fibonacci
/* Compute and print the n first number
of the Fibonacci suite*/

READ(n) // Read the number give by the user
a := 0
olda := a
b := 1
i := 0
WHILE (n > i) DO
    olda := a
    a := b
    b := olda + b
    i := i + 1
    PRINT(a) //Print the number i of the Fibonacci suite
ENDWHILE
ENDPROG
```

Note : The code presented here is in the *test* folder of this project, it has the name *Fibonacci.fs*.

7.2 Leftmost derivation

Here is the list of rules applied during the derivation of this program :

```
1 2 3 4 11 43 4 7 12 13 14 18 24 21 17 4 7 12 13 14 18 23 21 17 4 7 12 13 14
18 24 21 17 4 7 12 13 14 18 24 21 17 4 9 37 38 34 14 18 23 21 17 36 14 18 23
21 17 39 40 41 4 7 12 13 14 18 23 21 17 4 7 12 13 14 18 23 21 17 4 7 12 13 14
18 23 21 15 18 23 21 17 4 7 12 13 14 18 23 21 15 18 24 21 17 4 10 42 5 5
```

7.3 Derivation tree

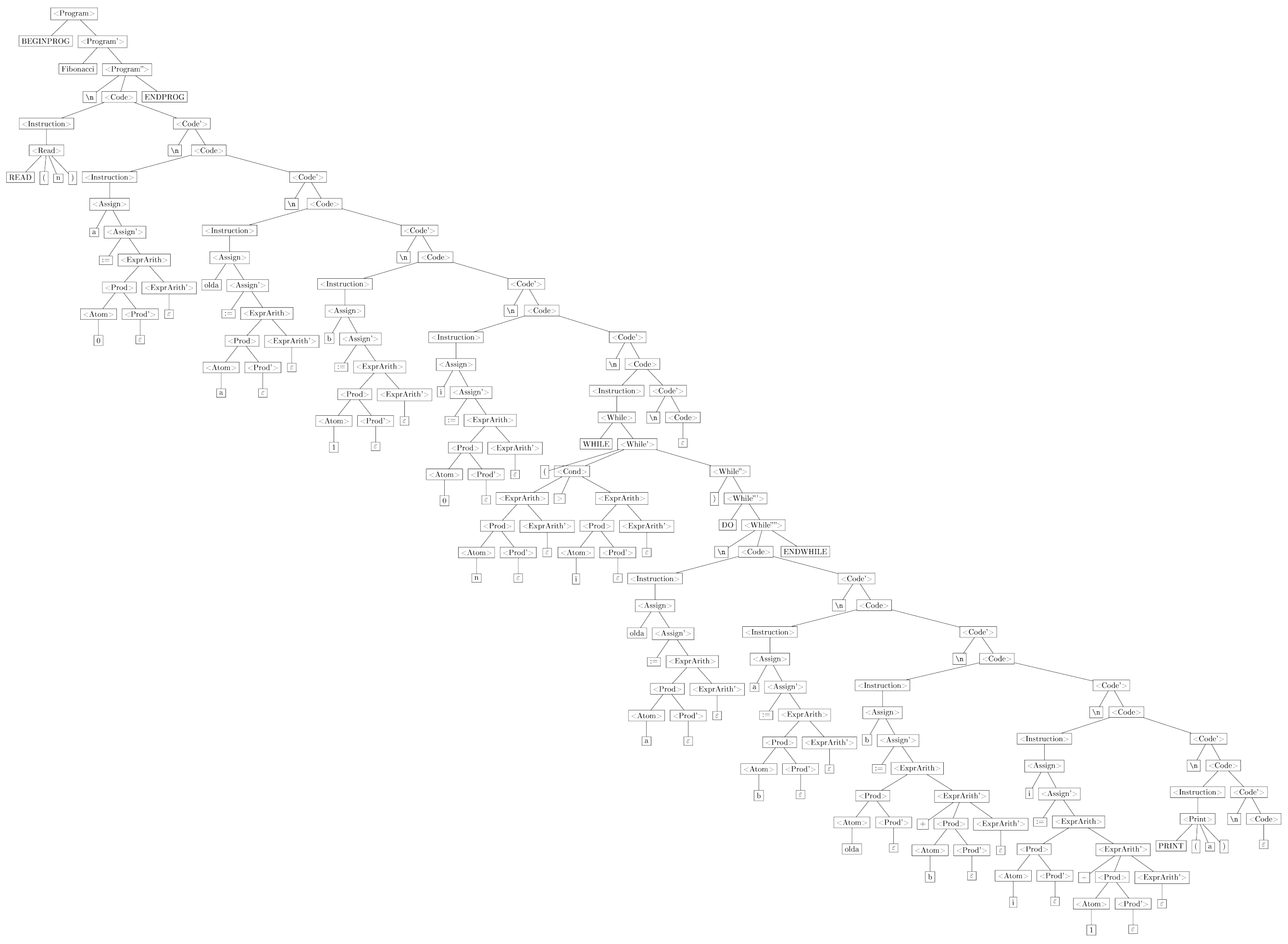

Figure 4: Derivation tree