Super-FORTRAN
Introduction to language theory and compiling
Project – Part 2

Gilles GEERAERTS Marie VAN DEN BOGAARD Léo EXIBARD

October 25, 2018

For this second part of the project, you will write the *parser* of your Super-FORTRAN compiler. More precisely, you must:

1. Transform the Super-FORTRAN grammar (see Figure 2 at the end of the statement) in order to: (a) Remove unproductive and/or unreachable variables, if any; (b) Make the grammar non-ambiguous by taking into account the priority and the associativity of the operators. Table 1 shows these priorities and associativities: operators are sorted by decreasing order of priority (with two operators in the same row having the same priority). (c) Remove left-recursion and apply factorisation where need be;

2. Give the *action table* of an LL(1) parser for the transformed grammar. You must justify this table by giving the details of the computations of the relevant First and Follow sets.

3. Write, in Java, a parser for this grammar. We strongly recommend you to design a recursive descent LL(1) parser, since it is the easiest way; you can either code it by hand or write a parser generator which will generate it from the action table. You can also design a pushdown automaton for this, or any other solution you have in mind (for "uncommon" solutions, please ask us first), as long as it is as efficient – ie a linear-time algorithm – as a recursive descent LL(1) parser. Whatever the chosen solution, you should explain it in detail in the report. In all cases, you must implement your parser *from scratch*, in the sense that you are not allowed to use tools such as `yacc`, or existing implementations (e.g. for pushdown automata).

   If your scanner from Part 1 worked correctly, your parser should use it in order to extract the sequence of tokens from the input. Otherwise, you can use the scanner which will be provided on the Université Virtuelle by **Wednesday, October 24th**[1].

   For this part of the project, your program should output on `stdout` *the leftmost derivation* of the input string if it is correct; or an (explanatory) error message if there is a syntax error. The format for such leftmost derivation should be a sequence of rule numbers (do not forget to number your rules accordingly in the report!) separated by a space. For instance, if your input string is part of the grammer, as witnessed by a successful derivation composed of rules 1,2,5,6,8,14,15,7, your program should output `1 2 5 6 8 14 15 7`. If you deem relevant, your program can also offer, as an *option*[2] called by adding `-v` to the command (cf Figure 1 page 3 for details), a more verbose output explicitly describing the rules which are used (the exact format is up to you).

   Additionally, your program should build the parse tree (aka the "derivation tree") of the input string and, when called by adding `-wt filename.tex` to the command (again, cf Figure 1 for

[1] Unless some groups could not finish Part 1 in time.

[2] By default, when called as in Figure 1, your program should simply output the sequence of rules!

details), write it as a LaTeX file called `filename.tex`. To this end, a `ParseTree.java` class is provided on the Université Virtuelle. *You are free to modify* this class as you wish, or even design your own from scratch (as usual, explain what you did and why you did it in the report).

Operators	Associativity
- (unary), NOT	right
*, /	left
+, - (binary)	left
>, <, >=, <=, =, <>, comma (,)	left
AND	left
OR	left

Table 1: Priority and associativity of the Super-FORTRAN operators (operators are sorted in decreasing order of priority). Note the difference between *unary* and *binary* minus (-). COMMA (,) is derived using rules 4 and 42.

You must hand in:

- A PDF report containing the modified grammar, the action table, with all the necessary justifications, choices and hypotheses, as well as descriptions of your example files;
- *Bonus:* For this part, no bonus is *a priori* specified, but recall that initiative is encouraged for this project: do not hesitate to explain why some features would be hard to add at this point of the project, or to add relevant features to Super-Fortran. Preferably ask us before, just to check that the feature in question is both relevant and doable.
- The source code of your parser in a JAVA source file called `Parser.java`, as well as all the auxiliary classes (`ParseTree.java` etc);
- The Super-FORTRAN example files you have used to test your parser. It is strongly recommended to provide example files of your own;
- All required files to evaluate your work (like a `Main.java` file calling the parser, etc).

You must structure your files in five folders:

- `doc` contains the JAVADOC and the PDF report.
- `test` contains all your example Super-FORTRAN files.
- `dist` contains an executable JAR **that must be called** `part2.jar`;
- `src` contains your source files
- `more` contains all other files.

Your implementation must contain:

1. Your scanner (from the first part of the project), or the correction if yours did not work properly;
2. Your parser;
3. An executable public class `Main` that reads the file given as argument and writes on the standard output stream the leftmost derivation, possibly with some options to be more verbose or to write/print the parse tree (cf Figure 1). The command for running your executable must be as follows: `java -jar part2.jar sourceFile.sf` (cf Figure 1 for the options).

```
java -jar part2.jar [OPTION(S)] [FILE]
e.g. java -jar part2.jar -v -wt tree.tex sourceFile.sf
```

Figure 1: The generic form of the command for running your executable, as well as an example

You will compress your folder (in the *zip* format—no *rar* or other format), **which is called according to the following regexp**:

```
Part2_Surname1(_Surname2)?.zip
```

where `Surname1` and, if you are in a group, `Surname2` are the last names of the student(s) (in alphabetical order), and you will submit it on the Université Virtuelle before **November, 21st**. You are allowed to work in group of maximum two students. Note that the statement of Part 3 will be available from *November, 13th* (the deadline will be *December, 22nd*), to give you a bit of freedom on how you want to organise your work.

[1] <Program> → BEGINPROG [ProgName] [EndLine] <Variables> <Code> ENDPROG
[2] <Variables> → VARIABLES <VarList> [EndLine]
[3] → ε
[4] <VarList> → [VarName], <VarList>
[5] → [VarName]
[6] <Code> → <Instruction> [EndLine] <Code>
[7] → ε
[8] <Instruction> → <Assign>
[9] → <If>
[10] → <While>
[11] → <For>
[12] → <Print>
[13] → <Read>
[14] <Assign> → [VarName] := <ExprArith>
[15] <ExprArith> → [VarName]
[16] → [Number]
[17] → (<ExprArith>)
[18] → - <ExprArith>
[19] → <ExprArith> <Op> <ExprArith>
[20] <Op> → +
[21] → -
[22] → *
[23] → /
[24] <If> → IF (<Cond>) THEN [EndLine] <Code> ENDIF
[25] → IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF
[26] <Cond> → <Cond> <BinOp> <Cond>
[27] → NOT <SimpleCond>
[28] → <SimpleCond>
[29] <SimpleCond> → <ExprArith> <Comp> <ExprArith>
[30] <BinOp> → AND
[31] → OR
[32] <Comp> → =
[33] → >=
[34] → >
[35] → <=
[36] → <
[37] → <>
[38] <While> → WHILE (<Cond>) DO [EndLine] <Code> ENDWHILE
[39] <For> → FOR [VarName] := <ExprArith> TO <ExprArith> DO <Code> ENDFOR
[40] <Print> → PRINT(<ExpList>)
[41] <Read> → READ(<VarList>)
[42] <ExpList> → <ExprArith> , <ExpList>
[43] → <ExprArith>

Figure 2: The Super-FORTRAN grammar.