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

Gilles GEERAERTS Guillermo A. PÉREZ Thi-Van-Anh NGUYEN

November 7, 2016

> FORTRAN, *"the infantile disorder", by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is too clumsy, too risky, and too expensive to use.*
>
> Edsger W. Dijkstra, SIGPLAN Notices, Volume 17, Number 5

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

1. Transform the Über-FORTRAN grammar (see Figure 1) in order to: (a) Remove unreachable and/or unproductive variables, if any; (b) Remove left-recursion and apply factorisation where need be; (c) 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).
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 **recursive descent**[1] LL(1) parser for this grammar. Your parser should use the scanner that you have designed in the first part of the project in order to extract the sequence of tokens from the input. For this part of the project, the output of your parser must consist only of the *leftmost derivation* of the input string, if it is correct; or an error message if there is a syntax error.

[1] Thus, you must write the parser by hand, and *not* use a parser generator like Bison or CUP.

[1] <Program> → PROGRAM [ProgName] [EndLine] <Vars> <Code> END
[2] <Vars> → INTEGER <VarList> [EndLine]
[3] → ε
[4] <VarList> → [VarName], <VarList>
[5] → [VarName]
[6] <Code> → <Instruction> [EndLine] <Code>
[7] → ε
[8] <Instruction> → <Assign>
[9] → <If>
[10] → <Do>
[11] → <Print>
[12] → <Read>
[13] <Assign> → [VarName] = <ExprArith>
[14] <ExprArith> → [VarName]
[15] → [Number]
[16] → (<ExprArith>)
[17] → - <ExprArith>
[18] → <ExprArith> <Op> <ExprArith>
[19] <Op> → +
[20] → -
[21] → *
[22] → /
[23] <If> → IF (<Cond>) THEN [EndLine] <Code> ENDIF
[24] → IF (<Cond>) THEN [EndLine] <Code> ELSE [EndLine] <Code> ENDIF
[25] <Cond> → <Cond> <BinOp> <Cond>
[26] → .NOT. <SimpleCond>
[27] → <SimpleCond>
[28] <SimpleCond> → <ExprArith> <Comp> <ExprArith>
[29] <BinOp> → .AND.
[30] → .OR.
[31] <Comp> → .EQ.
[32] → .GE.
[33] → .GT.
[34] → .LE.
[35] → .LT.
[36] → .NE.
[37] <Do> → DO [VarName] = [Number], [Number] [EndLine] <Code> ENDDO
[38] <Print> → PRINT*, <ExpList>
[39] <Read> → READ*, <VarList>
[40] <ExpList> → <ExprArith>, <ExpList>
[41] → <ExprArith>

Figure 1: The Über-FORTRAN grammar.

Operators	Associativity
`-`, `.NOT.`	right
`*`, `/`	left
`+`, `-`	left
`.EQ.`, `.GE.`, `.GT.`, `.LE.`, `.LT.`, `.NE.`	left
`.AND.`	left
`.OR.`	left

Table 1: Priority and associativity of the Über-FORTRAN operators (operators are sorted in decreasing order of priority)

You must hand in:

- A *mandatory* PDF report containing the transformed grammar, the action table, with all the necessary justifications, choices and hypothesis;
- The source code of your parser;
- The Über-FORTRAN example files you have used to test your parser;
- All required files to evaluate your work (like a `Main.java` file calling the lexical analyser, etc).

You must structure your files in four folders:

- `doc` contains the JAVADOC and the PDF report;
- `test` contains all your example files;
- `dist` contains an executable JAR;
- `more` contains all other files.

Your implementation must contain:

1. your scanner (from the first part of the project);
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. It can be given by a set of lines where the format of each line must be:
   (a) The number of the rule (in your *transformed grammar*) (do not forget to number your rules accordingly in the report!)
   (b) The content of the rule (in your *transformed grammar*)

   The number and the content of the rule must be separated by at least one space. If there is a syntax error, your parser must throw an error message including the number of rule causing the error, e.g. "A syntax error occurs at rule [3]".

The command for running your executable must be:

```
java -jar Part2_Surname1_Surname2.jar sourceFile
```

Where `Surname1` and `Surname2` are the surnames of two members in the group (You are allowed to work in group of maximum two students).

Your folder must be named `Part2_Surname1_Surname2`. Then you will compress your folder (in the *zip* format—no *rar* or other format) and you will submit it on the Université Virtuelle by **November, 28th**. A printed copy of your report will be handed in to the Secrétariat étudiants (Mrs. Maryka Peetroons, NO8.104) for the same date.