INFO-F-403: Project Super-Fortran Part 1
Lexer analyser

Sefu Kevin and Abou Zaidi Ahmed

November 16, 2018

1 Introduction

In this project we were requested to design and write a compiler for Super-Fortran. It is a very simple imperative language. The project is divided in 3 parts and for this second part we had to produce the parser of the compiler. The parser is the tools that perform the syntax analysis of a sequences of tokens and verify if those tokens are conform to the rules of grammar. During the parsing, the parse had to build the derivation tree.

There exist severals methods of building a parser. In our case we choose to design a recursive decent ll(1) parser.

2 Grammar

We did not find any useless rule. They all are accessible and productive.

[1] $\langle$**Program**$\rangle$ $\rightarrow$ BEGINPROG [**ProgName**] [**EndLine**] $\langle$**Variables**$\rangle$ $\langle$**Code**$\rangle$ ENDPROG

[2] $\langle$**Variables**$\rangle$ $\rightarrow$ VARIABLES $\langle$**VarList**$\rangle$ [**EndLine**]

[3] $\langle$**Variables**$\rangle$ $\rightarrow \varepsilon$

[4] $\langle$**VarList**$\rangle$ $\rightarrow$ [**VarName**] $\langle$**VarList_prim**$\rangle$

[5] $\langle$**VarList_prim**$\rangle$ $\rightarrow$, [**VarName**] $\langle$**VarList_prim**$\rangle$

[6] ⟨**VarList_prim**⟩ → ε

[7] ⟨**Code**⟩ → ⟨**Instruction**⟩ [**EndLine**] ⟨**Code**⟩

[8] ⟨**Code**⟩ → ε

[9] ⟨**Instruction**⟩ → ⟨**Assign**⟩

[10] ⟨**Instruction**⟩ → ⟨**If**⟩

[11] ⟨**Instruction**⟩ → ⟨**While**⟩

[12] ⟨**Instruction**⟩ → ⟨**For**⟩

[13] ⟨**Instruction**⟩ → ⟨**Print**⟩

[14] ⟨**Instruction**⟩ → ⟨**Read**⟩

[15] ⟨**Assign**⟩ → [**VarName**] := ⟨**ExprArith**⟩

[16] ⟨**ExprArith**⟩ → ⟨**Term**⟩ ⟨**ExprArith_prim**⟩

[17] ⟨**ExprArith_prim**⟩ → + ⟨**Term**⟩ ⟨**ExprArith_prim**⟩

[18] ⟨**ExprArith_prim**⟩ → - ⟨**Term**⟩ ⟨**ExprArith_prim**⟩

[19] ⟨**ExprArith_prim**⟩ → ε

[20] ⟨**Term**⟩ → ⟨**Atom**⟩ ⟨**Term_prim**⟩

[21] ⟨**Term_prim**⟩ → * ⟨**Atom**⟩ ⟨**Term_prim**⟩

[22] ⟨**Term_prim**⟩ → / ⟨**Atom**⟩ ⟨**Term_prim**⟩

[23] ⟨**Term_prim**⟩ → ε

[24] ⟨**Atom**⟩ → [**Number**]

[25] ⟨**Atom**⟩ → [**VarName**]

[26] ⟨**Atom**⟩ → (⟨**ExprArith**⟩)

[27] ⟨**Atom**⟩ → - ⟨**Atom**⟩

[28] ⟨**If**⟩ → IF (⟨**Cond**⟩) THEN [**EndLine**] ⟨**Code**⟩ ⟨**IfSeq**⟩

[**29**] $\langle$**IfSeq**$\rangle$ $\rightarrow$ ENDIF

[**30**] $\langle$**IfSeq**$\rangle$ $\rightarrow$ ELSE [**EndLine**] $\langle$**Code**$\rangle$ ENDIF

[**31**] $\langle$**Cond**$\rangle$ $\rightarrow$ $\langle$**AndCond**$\rangle$ $\langle$**Cond_prim**$\rangle$

[**32**] $\langle$**Cond_prim**$\rangle$ $\rightarrow$ OR $\langle$**AndCond**$\rangle$ $\langle$**Cond_prim**$\rangle$

[**33**] $\langle$**Cond_prim**$\rangle$ $\rightarrow \varepsilon$

[**34**] $\langle$**AndCond**$\rangle$ $\rightarrow$ $\langle$**SimpleCond**$\rangle$ $\langle$**AndCond_prim**$\rangle$

[**35**] $\langle$**AndCond_prim**$\rangle$ $\rightarrow$ AND $\langle$**SimpleCond**$\rangle$ $\langle$**AndCond_prim**$\rangle$

[**36**] $\langle$**AndCond_prim**$\rangle$ $\rightarrow \varepsilon$

[**37**] $\langle$**SimpleCond**$\rangle$ $\rightarrow$ $\langle$**ExprArith**$\rangle$ $\langle$**Comp**$\rangle$ $\langle$**ExprArith**$\rangle$

[**38**] $\langle$**SimpleCond**$\rangle$ $\rightarrow$ NOT $\langle$**SimpleCond**$\rangle$

[**39**] $\langle$**Comp**$\rangle$ $\rightarrow=$

[**40**] $\langle$**Comp**$\rangle$ $\rightarrow<=$

[**41**] $\langle$**Comp**$\rangle$ $\rightarrow<$

[**42**] $\langle$**Comp**$\rangle$ $\rightarrow>=$

[**43**] $\langle$**Comp**$\rangle$ $\rightarrow>$

[**44**] $\langle$**Comp**$\rangle$ $\rightarrow<>$

[**45**] $\langle$**While**$\rangle$ $\rightarrow$ WHILE ($\langle$**Cond**$\rangle$) DO [**EndLine**] $\langle$**Code**$\rangle$ ENDWHILE

[**46**] $\langle$**For**$\rangle$ $\rightarrow$ FOR [**VarName**] := $\langle$**ExprArith**$\rangle$ TO $\langle$**ExprArith**$\rangle$ DO [**EndLine**] $\langle$**Code**$\rangle$ ENDFOR

[**47**] $\langle$**Print**$\rangle$ $\rightarrow$ PRINT ($\langle$**ExpList**$\rangle$)

[**48**] $\langle$**Read**$\rangle$ $\rightarrow$ READ ($\langle$**VarList**$\rangle$)

[**49**] $\langle$**ExpList**$\rangle$ $\rightarrow$ $\langle$**ExprArith**$\rangle$ $\langle$**ExpList_prim**$\rangle$

[**50**] $\langle$**ExpList_prim**$\rangle$ $\rightarrow$, $\langle$**ExprArith**$\rangle$ $\langle$**ExpList_prim**$\rangle$

[**51**] $\langle$**ExpList_prim**$\rangle$ $\rightarrow \varepsilon$

3 Explanation of choices and hypotheses

3.1 Recursive descent

We choose to build a recursive descent parser because we found it to be more intuitive in the sense that the derivation tree correspond directly (except for epsilon)to the recursive calls and we also found this method interesting in the sense that it offer easy debugging.

3.1.1 Errors handling

In our recursive decent parsing, when we expect a specific terminal from the input but we receive a different one we throw a syntax error, that tells to the user which terminal was expected.

3.2 Arbitrary number of endlines handling

The grammar does not allow to have mutliples endlines betweens instructions. But we supposed that the mutliples endline are important tools for the user to make the code more readable. For instance if the user want to make a clean documentation of his code.

To solve this problem, we could modify the grammar but this solution will overload the grammar's rules. That's why we decided to handle this problem at the lexer level.

Start file endlines All endlines before any other type of token are ignored.

End file endline All endlines after the (first occurence) ENDPROG token are ignored.

Consecutive endlines All others consecutives endlines are considered as a one endline token.

3.3 Endlines after short comments

In order to have the same output as the one in provided the .out file, the part 1 lexer skips enldine after short comments. But this forces the user to put an additional empty line after each short comments. We changed the behaviour of the lexer.

3.4 Modification of provided classes

LexicalUnit We decided to add a LexicalUnit instance called `EPSILON` for the ε because we need to implement the `isEpsilon` method called in the provided `ParseTree` class. We also added a method named `getVerbose` to print more readable errors.

ParseTree We added a method called `labelToTex` that gives better formatting for terminals in the derivation tree than the `toString` method.

4 Tests files desciption

`00-Factorial.sf` This files is provided by the teacher assistant.

`01-err-Print.sf` On more file that contains an error to show the error message.

`02-err-Cond.sf` This file contains an error where the condition is just a number and not a comparison of two arithmetics expressions. It shows an error that occur on a step (for the ll(1) parser) where multiple terminal values are expected for the lookahead.

`03-newlines.sf` This file show how the parser (and lexer) support an arbitrary number of empty lines or lines with only comments.

5 Action table

6 Conclusion

As we said in the introduction we implemented a LL(1) parser for the Super-Fortran programming language.

During this second part of the project we learned how to make a syntaxic analysis by the building of a parser. One major difficulty we faced was to transform the initial ambigous grammar to a none ambigous one and to check if our transformation of the grammar was good. A second difficulty was the verification of the firsts and follows because its easy to make mistakes because the grammar had many produces and terminals.

But finally, this part of the project allowed us to learn more about the parsing step of a compiler and to understand deeply how it works.