INFO-F403 Project part 3

Vincent Gailly 547819 - Oruç Kaplan 540662

December 2022

1 Introduction

A compiler is a program that translates code written in a high-level programming language into machine code that can be executed on a computer.

After making a lexical analyser and a parser, the last step of the project is to convert the Fortress code into LLVM code. It is this one which will be treated through this report.

2 Code

To realize the translation from Fortress to LLVM machine language, we based ourselves on what we had implemented before and we notably used the parse tree. This one being relatively verbose, and not needing all the information it contained to translate in LLVM, we decided to make a first step of simplification of this tree. This step is managed by the TreeSimplifier class which will be explained in more detail below.

Once our tree was simplified, we only had to go through all the nodes of the tree and convert the program represented by this tree into LLVM language. This part is managed by the LLVMGenerator class.

2.1 TreeSimplifier class

This class is therefore in charge of simplifying what was produced as a parse tree in part 2. It is essentially in charge of removing "useless" syntactic elements for the translation into LLVM programming language.

As an example, here is the simplified tree that is produced by our code on the provided Factorial.fs file.

Since this example produces a tree that is too large to put in our report when not simplified, we decided to illustrate what was done through another test file that we created :

```
BEGIN Factorial

%% Compute the factorial of a number.
   If the input number is negative, print -1. %%

  READ(number) ,            :: Read a number from user input
  result := 1 ,

IF (number > -1) THEN
  WHILE (number > 0) DO
    result := result * number ,
    number := number - 1 ,  :: decrease number
  END ,
  ELSE                      :: The input number is negative
  result := -1 ,
END ,
PRINT(result) ,
END
```

Our code produces the following parse tree:

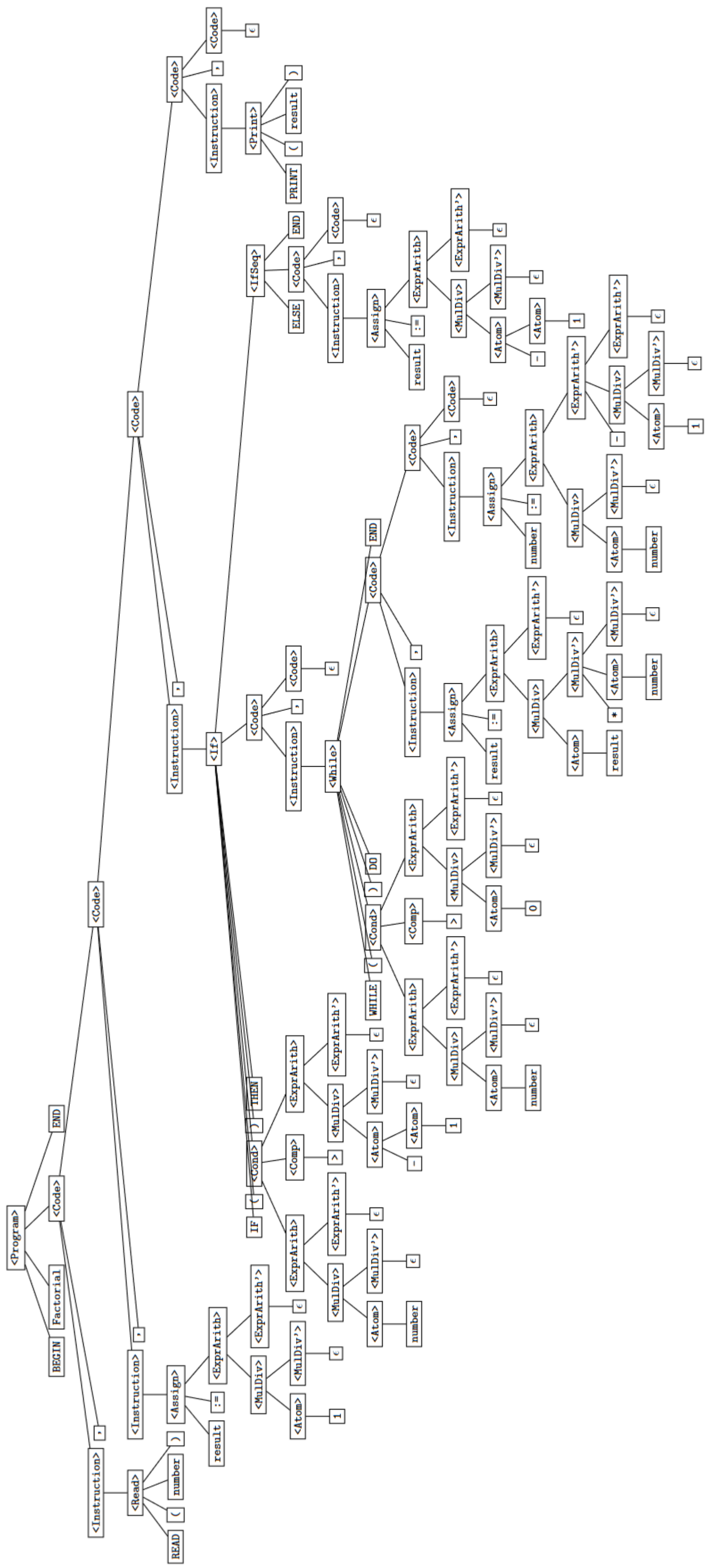

Figure 1: Basic version of the parse tree of the *Factorial.fs* file.

And here is its version simplified by the TreeSimplifier class :

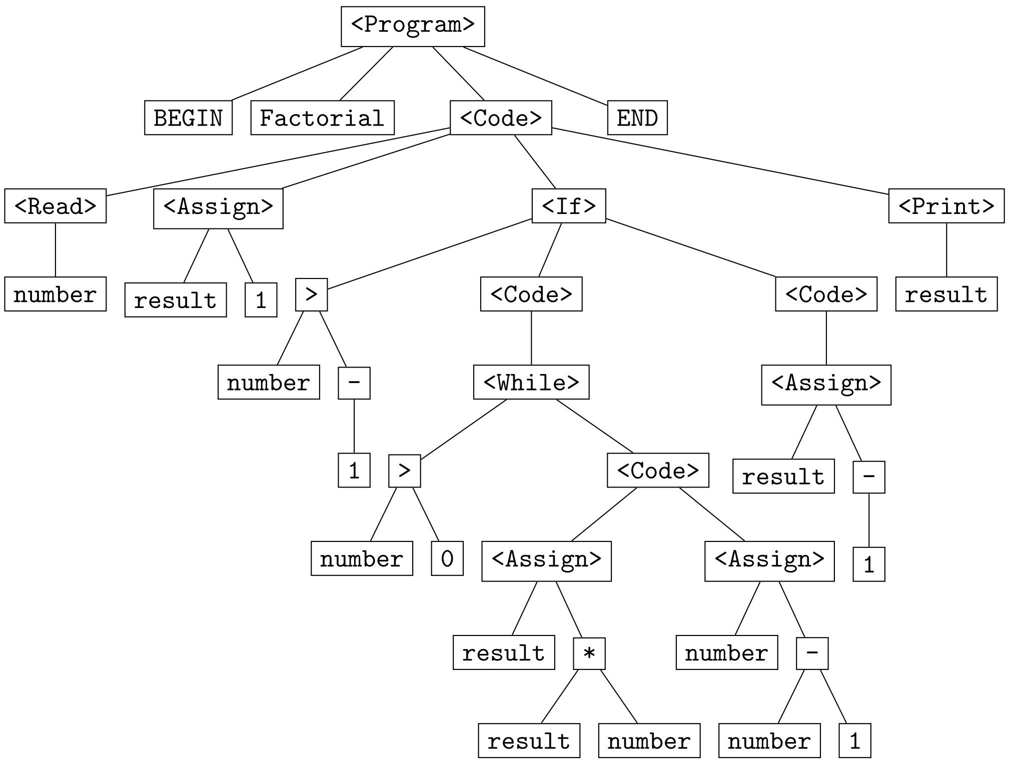

Figure 2: Simplified version of the parse tree of the *Factorial.fs* file.

You can see that there is a huge difference between figure 1 and 2. Even if this step is not necessary, the fact of having created an abstract syntex tree has greatly simplified the task for the rest of the code.

2.2 LLVMGenerator class

As its name indicates, this class is in charge of generating the LLVM code. It has a main method named generateCorrespondingLLVM which is in charge of displaying the whole LLVM code corresponding to the parse tree (simplified by our TreeSimplifier class) of a program realized with the Fortress language.

In order to handle all possible scenarios in our parse tree, a method was created to handle the conversion of each type of symbol that can be encountered in the parse tree into LLVM code. In order to be sure not to miss any possibilities, we have created all the code of this class by using our grammar (realized in the previous part). Indeed, this grammar allows us to have a synthetic vision of the code that could follow a node of the tree.

In addition to all these methods for converting Fortress code to LLVM code, we have created a method for generating automatic variable names. Indeed LLVM being a language of lower level, some more complex operations easily performed in Fortress must be transformed into a subroutine of several instructions in LLVM. Each of these subroutines must keep the sub-calculations performed in "temporary" variables that will be created with our "generateNewVariableName" method. This method is very careful not to create a variable name that may have already been created by the user and thus create a conflict.

2.3 Main class

This class handles the parsing of the arguments given to the program (e.g. : read the input file, create in a tex file the parsing tree or not)

It will use the LexicalAnalyzer and the Parser realized in the previous parts of the project and will thus recover the parseTree. It will then simplify this parse tree with the TreeSimplifier class (if the -wt option has been specified, it will also create a "simple_FILENAME.tex" file which will contain the simplified tree in latex format. It will finish by using the LLVMGenerator class to get the LLVM code and then display it in the console.

2.4 Usage

In order to use more easily the whole code, a Makefile has been provided :

- To compile the whole project :
  - You can run the makefile using the command : *make*
- To run the whole project:
  - You can run the makefile using the command : *make basic*
    Will run our compiler on the *Factorial.fs* file.
  - You can run the makefile using the command : *make testing*
    Will run our compiler on all the *.fs* file of the test directory.
  - You can use the command :
    java -jar dist/part3.jar <your code file>
    Will run our compiler on your own code.
  - You can use the command :
    java -jar dist/part3.jar <your code file> -wt <fileName>.tex
    Will run our compiler on your own code and generate the parse tree in the corresponding tex file (and the simplified parse tree in the "simple_<fileName>" tex file) .
- To run the whole project and to compile and run the output LLVM code (You need to be on a Linux distribution for this one):

- You can run the makefile using the command : *make checkBasic*
  Will run our compiler on the *Factorial.fs* file and put the output in a *Factorial.ll* file, then it will compile it and run it with llvm.
- Or you can use the three following commands :
  * *java -jar dist/part3.jar <your code file> | tee <nameFile>.ll*
    Will run our compiler on your own code.
  * *llvm-as <nameFile>.ll -o=<nameFile>.bc*
    Will compile the ll file.
  * *lli <nameFile>.bc*
    Will run the bc file.

3 Conclusion

This project allowed us to understand in a more practical way how a compiler works. Throughout the project we were able to discover the different steps involved in creating a compiler, the more complicated steps and their solutions.