Université Libre de Bruxelles

INFO-F403 : Introduction to language theory and compiling

Computer Science Section

Building a FORTR-S compiler : Part 3

Professor:
Gilles Geeraerts

Students :
Miró-Manuel Matagne
Linh Tran Ngoc

Academic Year 2020 - 2021

Contents

1 Introduction

The goal of this project, as part of the course Introduction to language theory and compiling, is to build a compiler for the FORTR-S language. The first part consisted in coding the scanner of this compiler. Then the second part aims to code the parser with the scanner implemented. Finally, in this third and last part of the compiler, LLVM IR code will be generated corresponding to the semantics of the Fortr-S program that will be compiled. First, the utilization of this part of the project will be explained, then a description of the different implementations will be given. Lastly, the testing implementation and different test files that our group has chosen will be explained.

2 Utilization

In order to obtain the LLVM code corresponding to a certain source file written in `FORTR-S` language, one can navigate to the `dist` folder and run the command :

```
java -jar part3.jar sourceFile.fs
```

where `sourceFile.fs` is the file we want to generate LLVM code from.

Options can be introduced in the command line. For instance if the user would like to output the LLVM code to a `.ll` file, he should run the command :

```
java -jar part3.jar sourceFile.fs -o outputFile.ll
```

where `outputFile.ll` is the file we want to store the generated LLVM code to.

If the user wants the Abstract Syntax Tree to be stored in a file, he can enter the following command :

```
java -jar part3.jar sourceFile.fs -ast file.tex
```

where `file.tex` is the file we want to store the LaTeX code corresponding to the Abstract Syntax Tree.

If the user wishes to execute the FORTR-S code directly, he should enter :

```
java -jar part3.jar sourceFile.fs -exec
```

In this last case, the standard inputs and outputs related to the execution of the `.ll` file are redirected towards the standard input/output of the running program. It must be precised that this command will only work if LLVM is installed properly on the computer that is running the program, as the command line functions `llvm-as` and `lli` will be called.

A `Makefile` is present in the root repository, and the user can enter the command `make` in order to compile the project. One can also use the commands `make testing` to execute the code on a specific file, `make output` in order to store the output code in a file called `outputFile.ll`, `make exec` in order to execute directly the generated LLVM code, or `make ast` in order to store the Abstract Syntax Tree corresponding to the input file in a file called `ast.tex`.

3 Implementation

3.1 Building the AST

The first step in order to generate the code corresponding to the FORTR-S language file being read is to generate an Abstract Syntax Tree starting from the original syntax tree returned from the parser, described in the Part2 of the project. Indeed, using the actual parse tree in order to

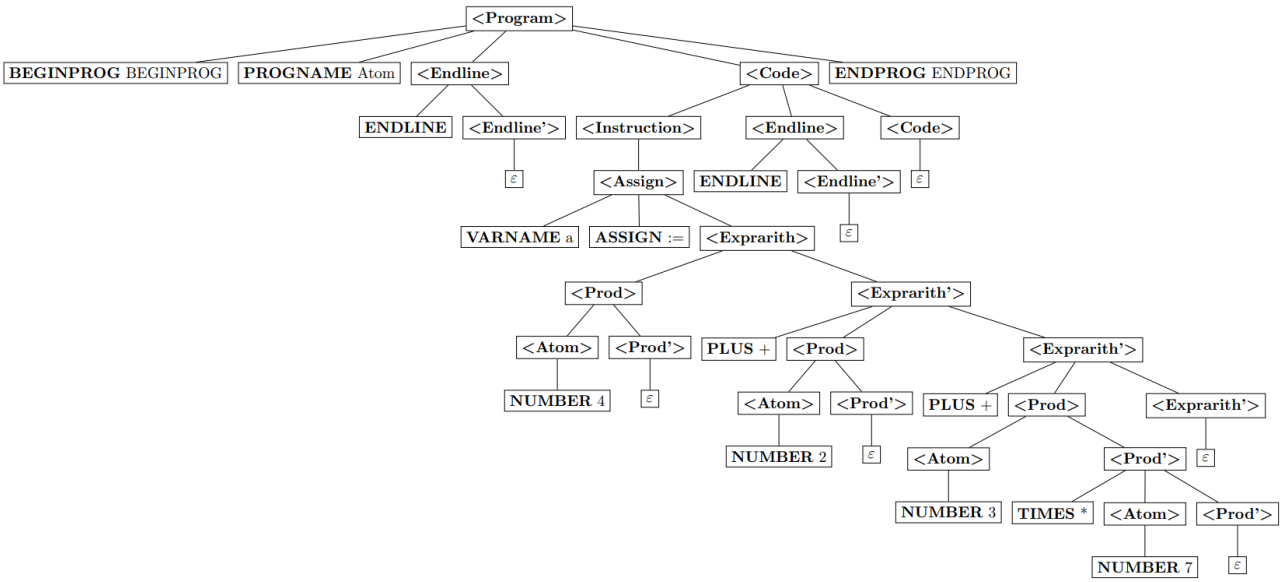
generate code is quite complicated, because there are many terminals that are not very useful, and the associativity of the operators is hard to figure out.

An `AST` class is therefore created, and takes as argument the parse tree returned from the parser. This class contains a `getAST` method which, after calling multiple functions defined in the class, will return the appropriate Abstract Syntax Tree.

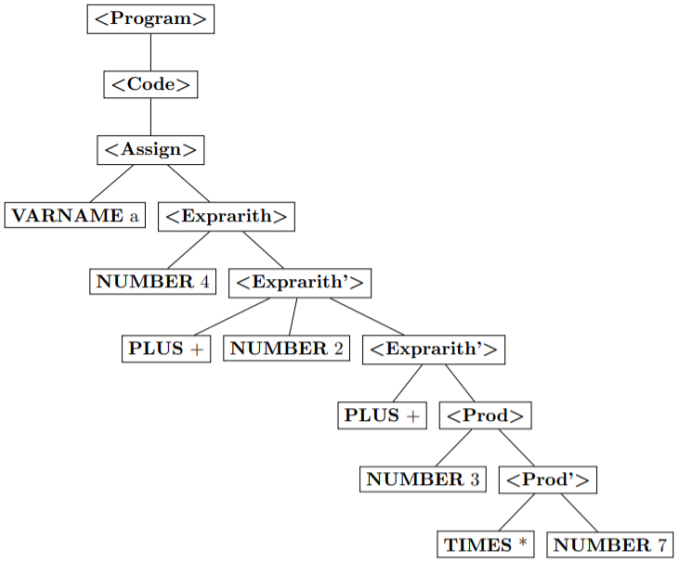
In order to explain each function used to build this AST, we will consider a simple example, corresponding to the FORTR-S code :

```
BEGINPROG Atom
    a := 4 + 2 + 3 * 7
ENDPROG
```

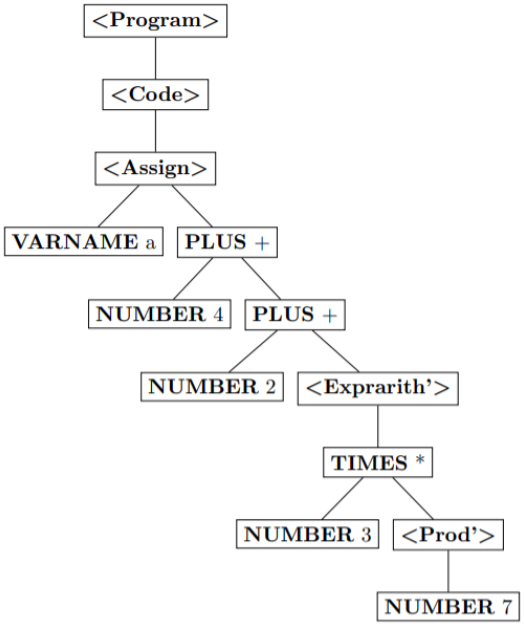
The parse tree returned from the parser is the following :



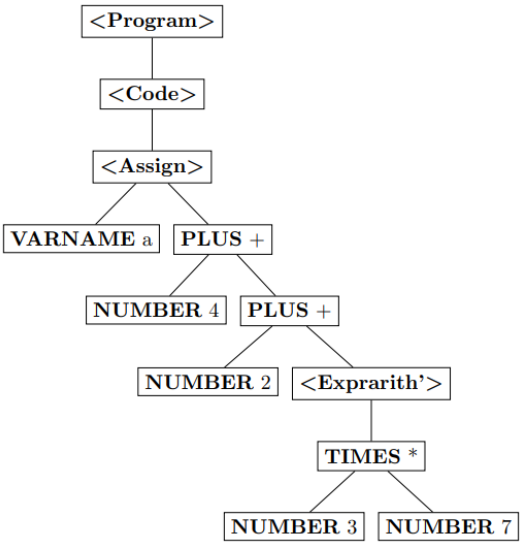
As one could imagine, it would be very difficult to translate this tree into LLVM code, even though it is such a simple code. The first step of transforming this tree into an AST is through the function `cleanTree`, which will remove unnecessary variables (terminals and non terminals). In particular all the variables whose only child was ϵ were removed, as well as the non-terminals that are useless such as `Instruction` or `EndLine'`. Some terminals such as `ENDLINE` and `ASSIGN` have also been removed as they are useless. On the example shown previously, the result of this function is as follows :



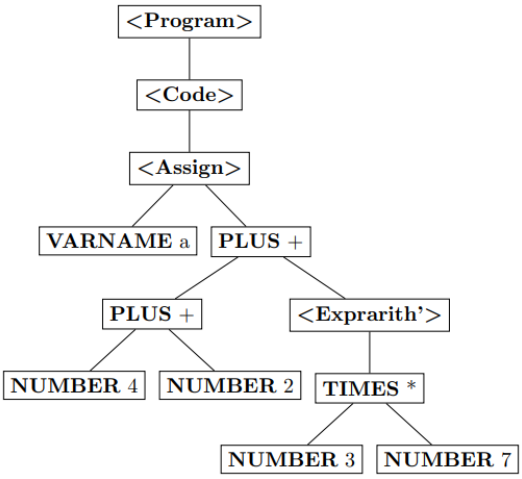
The next step is to place the operators properly, so that an operator is a non-terminal, having 2 children which represent the 2 operands. The function `setUpOperators` takes care of that, for all the operators that are present. The result on the considered example is :



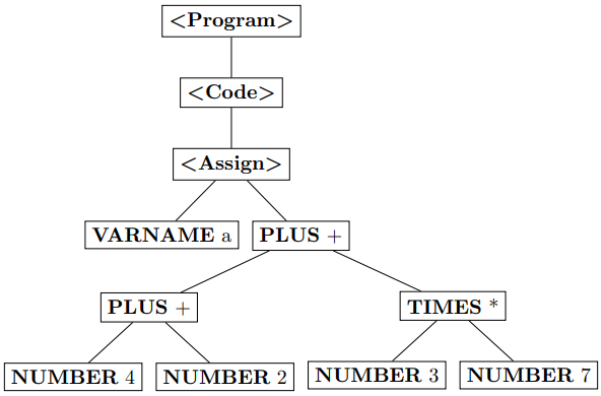
Before taking care of the associativity and priority of the operators, it is necessary to remove the `ExprArith`, `ExprArith'`, `Prod` and `Prod'` nodes whose only child is a `NUMBER` or a `VARNAME`. This is done thanks to the function `removeExprArith`, and the result on the considered example is :



Now, it is clear that on this AST, the associativity of the operators is not respected. Indeed, the addition should have left associativity, whereas the AST shows a right associativity. Therefore, the function `fixAssociativity` is used. This function calls another function called `getChildParseTree`, which establishes a list of all successive operators of the same *category* (addition and substraction or multiplication and division), and returns it in a correct order to respect left associativity. This function was a bit complicated to put in place as all the children of these operations have to be replaced accordingly. The result of applying this function to the previous example is :

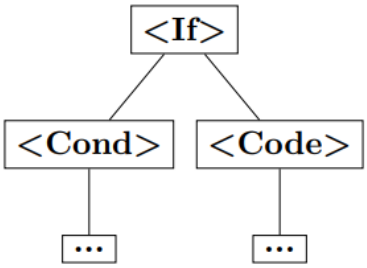


It is now clear that left associativity is respected, but there still remains a useless `ExprArith'` node in the AST. This is due to the fact there are different types of operators (or parenthesis) in the considered code. The last step is therefore to remove these nodes, calling the function `finalCleanUp`, which will return the final AST. In the case of the previously studied example, the result is :

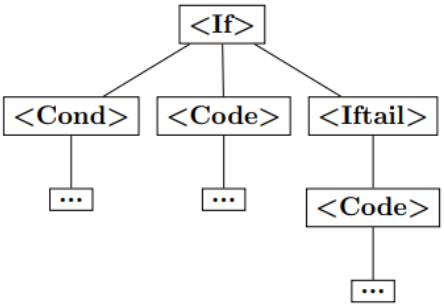


Therefore, we see that the priority and associativity of the operators are respected, and this tree will be easily readable in order to produce LLVM code.

There are 2 more cases where the construction of the AST is quite important for the LLVM code generation. First of all, the `IF` statements are written as follows in the AST if there is no `ELSE` corresponding to the statement :



Meanwhile if an `ELSE` statement is present, then the AST will have this form :



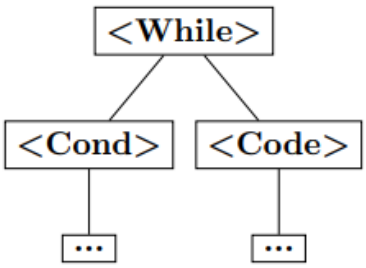
Notice that it is important to leave the `Iftail` node, even though it might seem useless. Indeed, the grammar allows to write a code like this :

```
BEGINPROG Prog
    a := -3
    IF(a > 0) THEN

    ELSE
        PRINT(a)
    ENFIF
ENDPROG
```

As the code following the `THEN` statement is empty (thanks to the rule 3 of the grammar), it is necessary to explicitly indicate which Code corresponds to the `ELSE` statement, and this is done by keeping the `Iftail` node. This allows to generate LLVM IR code without any trouble.

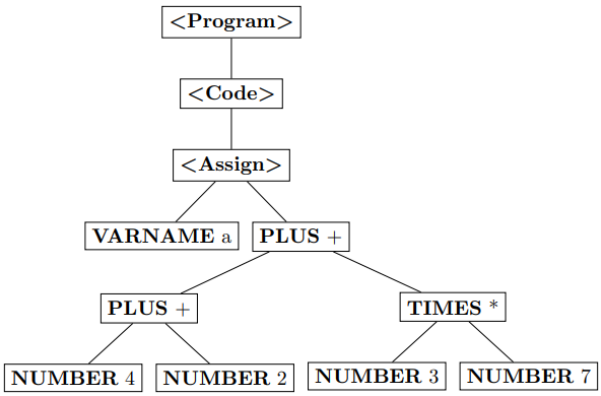
For the `WHILE` loops, the AST is represented like this :



3.2 Generating LLVM IR code

Once the AST is built, the LLVM IR code can be generated. In order to generate the code, a `LLVM` class is created, it takes as argument the Abstract Syntax Tree returned from the `AST` class. This `LLVM` class provides a `llvmCode` string variable which contains the corresponding LLVM IR code created by the `toLlvm` method and returned by the `getLlvm` method. The `toLlvm` method writes the LLVM IR code by calling the `analyze` method. It walks through the given AST by using recursively the `analyze` method for each node of the AST met.

In order to explain the process of the `analyze` method, the same example as above will be considered. The parse tree returned from the AST is the following :



As requested in the guidelines, the leftmost node represented by "**VARNAME** a" is evaluated first, i.e. an allocated variable is created, then the expression "**NUMBER** 4" is evaluated, it means that an unnamed variable is created to store the value of the number detected. Then the second "**NUMBER** 2" is also evaluated in the same way, the result of the addition of these last two expressions is computed and is stored in an unnamed variable whenever their "parent" node "**PLUS** +" is met. Notice that those unnamed variables correspond to a counter that is incremented (variable `line` in the `LLVM` class).

The same processes occur for the "**TIMES** *" node and its children.
The two `getCounter` and `setCounter` methods used in the `LLVM` class are created in the `ParseTree` class to have access on the corresponding unnamed `line` counter. Furthermore, to assign the total result of the last child of <Assign>, i.e. "**PLUS** +", in "**VARNAME** a", the total result is stored in the previously allocated variable "a" which, as for each **VARNAME** met, is added in the `values` list to memorize that it is already initialized. By creating this list, errors are avoided, for instance this mathematical expression :

```
BEGINPROG Example
    a := 3
    c := (a*c)*2
    a := b
    IF(a > d) THEN
        a := a + 1
    ENDIF
```

```
ENDPROG
```

It will return "Error : undefined variable", as the variable "c" is not initialized nor the variable "b" as well as the variable "d" in the condition of the IF statement.

In the cases where `<IF>` and `<WHILE>` are met, four methods are also created in the `ParseTree` class (`getIfCounter`, `setIfCounter`, `getWCounter` and `setWCounter`) to manage the two `ifCounter` and `wCounter` counters corresponding respectively to `<IF>` and `<WHILE>`. Those two counters allow to count the numbers of `<IF>` and `<WHILE>` met in the given AST and thus, writing the output LLVM IR code in consequence. In addition, two methods are also implemented in the `ParseTree` class (`getComp` and `setComp`) to write the right lexical unit of comparison.

The group has made the choice of not allowing declaration of variables inside an `IF`, `ELSE` or `WHILE` statement, as common programming languages. To do so, a boolean variable `inLoop` is initialized at `false` and set to `true` whenever the current child is in one of the <IF>, <IFTAIL> or <WHILE> nodes. The example of code below is an error. As a result, the compiler will return "Error : undefined variable".

```
BEGINPROG Error
    a := -1
    IF (a > 0) THEN
        b := 0
    ELSE
        c := 10
    ENDIF
ENDPROG
```

Indeed, as the two variables "b" and "c" are not initialized before the `IF` and `ELSE` statement, it is considered erroneous.

Finally, the two functions `@readInt` and `@println` are taken from the computer practical sessions in this `LLVM` class, and are added at the beginning of the generated `.ll` file.

3.3 Main file

The `Main` class first of all checks which parameters were given to the program through the command line, and finds out if any options such as `-o` or `-exec` have been entered by the user. Some very basic error handling is implemented at this step.

Then, the scanner (`LexicalAnalyzer` class) is used in order to retrieve the tokens corresponding to this specific FORTR-S code. The specifications of the scanner have been detailed in the Part 1 of the project, and have not been modified since. Next, the token list is given to the parser, which builds a Syntax Tree, such as described in Part 2 of the project.

The `AST` class is now called in order to transform the Syntax Tree into an Abstract Syntax Tree as described previously. Once this is done, the `LLVM` class is used in order to produce the LLVM code corresponding to the input file. This code is then systematically printed to the standard output, if there were no errors in the process. If the user specified an output file, the LLVM code is now written into the specified file.

If the user specified he wanted to execute the generated LLVM code directly, it is done by creating 2 files, called `source-code.ll` and `source-code.bc`, and the 2 following command lines are called :

```
llvm-as source-code.ll -o=source-code.bc
lli source-code.bc
```

The input and output streams related to the LLVM code execution are redirected to the standard input/output stream so that when a `READ` command is present for example, the user can enter the data directly in the same terminal where the program is being run. Similarly, if a `PRINT` command is present, the displaying will be made on the standard output.

Finally, the AST is converted to LaTeX code and written to the file specified by the user if the `ast` option was chosen.

4 Testing

As it has been done in Part1 and Part2 of this project, we have implemented an automated testing system in order to make sure our compiler works the way it should, especially when we make significant changes or refactorings. Therefore, a large number of test files were created and stored in the `test` folder, grouped by repositories testing similar features. In addition to the test files used in the previous parts of the project, other files have been added. Indeed, it proved to be a little tricky to implement nested `IF` conditions or nested `WHILE` loops, or combinations of both. Therefore, inside the `Instructions` folder, 2 new folders named `IF` and `WHILE` were created, and contain several `.fs` files.

For each file to be tested, a corresponding `.output` file containing the expected result is created, just like in the previous parts of the project. In this case, these files contain the LLVM code corresponding to the FORTR-S file.

Using the UNIX command `diff`, we make sure that the result of the compilation is actually exactly the same as the corresponding output file. If the test is successful, a success message is printed on the standard output. If there was an error, it is indicated on the standard output as well, with an indication to where there was a dissimilarity.

The `FortrS` folder contains complete FORTR-S codes that mix up all the different elements and rules of the language. These programs compute for instance the average of numbers entered by the user, or the Fibonacci sequence up to a certain index, etc... These are the most useful test files as they combine all codes tested in the separate folders.

A `Makefile` was created in the `test` folder in order to make this testing easier. Therefore, the commands to execute are the following :

- `make atom`
- `make comments`
- `make average`
- `make exponent`
- `make factorial`
- `make fibonacci`
- `make maximum`
- `make if`
- `make while`
- `make program`

One can also run all the previously listed tests with the command `make test`, or choose to run only the files contained in the `FortrS` folder using `make fortrs`.

All the tests executed on our various test files are successful, which leads us to think our compiler works well for FORTR-S code.

5 Conclusion

This last part of the project is the final part in the elaboration of a compiler for the FORTR-S language, whose role is to convert FORTR-S code into executable LLVM IR code. This is done by the means of an Abstract Syntax Tree, and then parsing it in order to produce the desired output. All our tests work properly, and we tried to make our test files as diversified as possible. Therefore, we think we have produced a working compiler for FORTR-S language. A field in which we could maybe have invested more work into would be error handling, which is present in our project but could probably be more efficient. Nevertheless, on a correct FORTR-S code, the compiler seems to work perfectly.