Introduction to Language Theory and compiling (INFO-F403)

Super-Fortran compiler : code generation

3th Part

Gerard Salinas, Joan S. (ID 000471612)
Geisser Aguilar, Niklaus. (ID 000469478)
Master in Computer Science ULB
29 of December, 2018

Contents

List of Figures

1 Introduction

In this last part of the project, we want to generate llvm code to be executed within the llvm command line. In order to accomplish this task, we map the already generated parse tree into an abstract syntax tree (AST) and then we walk through its branches generating the respective llvm code. For this last part of the project we did not implement any of the bonuses.

2 AST generation

In order to generate this representation, we decided to use a binary tree for the simplicity of representing all the structures with only two branches. The general idea is to study each of the branches generated by the parse tree and try to find the best way to represent it into an abstract syntax tree taking into account the operations priorities. In this report instead of detailing every map that we made for each case, we describe the more relevant conversions.

2.1 Variables declaration and assignment

Figure 1 shows how we represented the variables definition and the assignment of a variable.

```
1 BEGINPROG Conditions
2     VARIABLES a, b, c
3     a := 1 + 2 * 3 * (6 + 1)/5
4 ENDPROG
```

Listing 1: Code generating AST of figure 1

The idea is to generated a mapping for each piece of the parse tree from bottom to top and build the AST tree in the way.

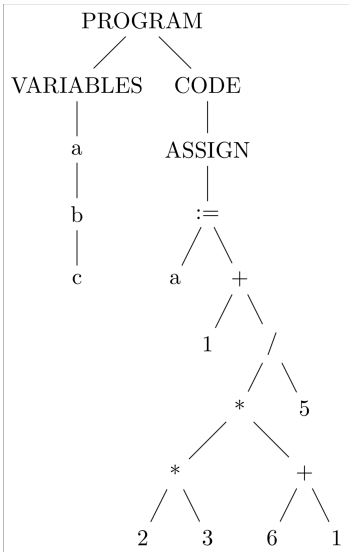

Figure 1: AST representing assignment and variables

Notice the way we represent variables declaration in the AST. We denote a variable name as the right child of another variable name, the reason of such implementation is that with a binary tree we have only two children, therefore, if we would have represented the variables name as children of the VARIABLES node, we would have been bounded by only two variables; however, this structure give us more flexibility to represent more than two variables.

In order to represent the assignment of a variable with an arithmetic expression, we should decompose this big problem into sub-problems. Figure 2 shows how the mapping works for a specific section of the parse tree. What is interesting to see here is that, in order to preserve left associativity of operations, the parse tree in rectangle number 1 generates an AST node with the division symbol containing only one right-child node which is a number (rectangle black number 1). Thus, the multiplication symbol (rectangle blue number 5) becomes left child of the "DIVIDE" symbol and the next "TIMES" symbol which is one level up in the parse tree will become left child of the last TIMES symbol in the new AST structure.

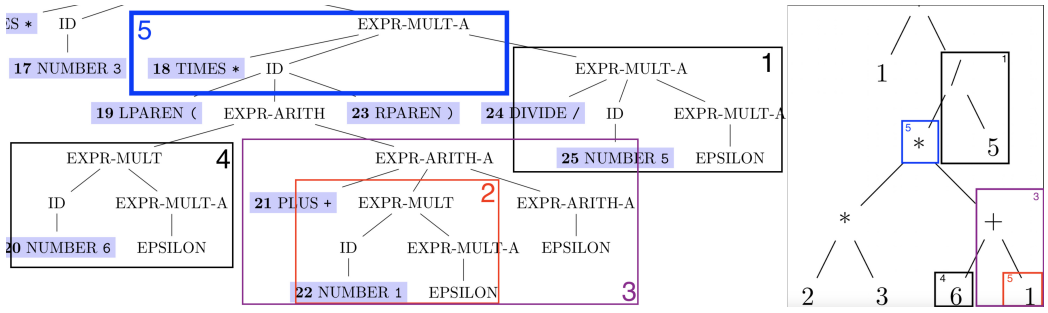

Figure 2: Left associativity in operations

2.2 Conditions

Figure 3 shows the AST structure for representing conditions. In case where there is no ELSE instruction, the header of the instruction will be IF, its right child will be the code to be executed when the given condition is met and the left child will be the condition it self. Otherwise, the ELSE instruction will be the header, its right child will be the code to be executed, its left child will be the IF statement with the code to be executed and the condition to be met. It seems odd to have the ELSE instruction in the header but this is a representation only that meets the requirements of having two children and a minimized representation of the parse tree. The generator code will know how to create the code based on the node information.

```
BEGINPROG Conditions
    IF (0 > 1 OR 5 > 3 AND 2 > 3 AND 3 > 5) THEN
        // not enter here
    ELSE
        PRINT(14)
    ENDIF

    IF (0 > 1 OR 5 > 3) THEN
        PRINT(15)
    ENDIF
ENDPROG
```

Listing 2: Code generating AST of figure 3

2.3 Cycles

Figure 4 shows the AST structure for representing both FOR and WHILE cycles. In the case of the FOR instruction, the right child contains the code to be executed inside the for and the left child contains the condition to be meet. This condition has the assignment as a left child and the value until it will iterate as a right child. The WHILE instruction has the code to be executed in the right side and the condition to be meet in the left side.

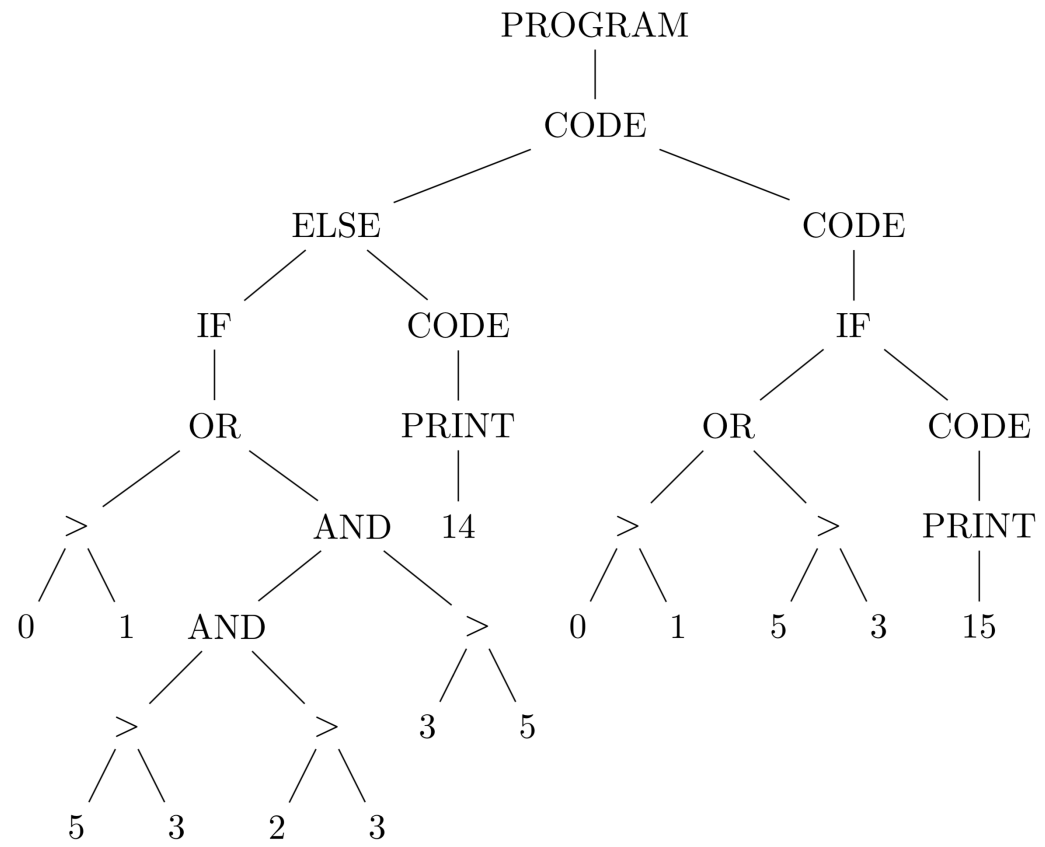

Figure 3: Conditions IF ELSE

```
BEGINPROG Counter
    VARIABLES n, a, b
    READ(a, b)

    FOR n := a TO b DO
        PRINT(n)
    ENDFOR

    WHILE( n > 1) DO
        factorial := factorial * n * (n - 1)
        n := n - 2
    ENDWHILE
ENDPROG
```

Listing 3: Code generating AST of figure 4

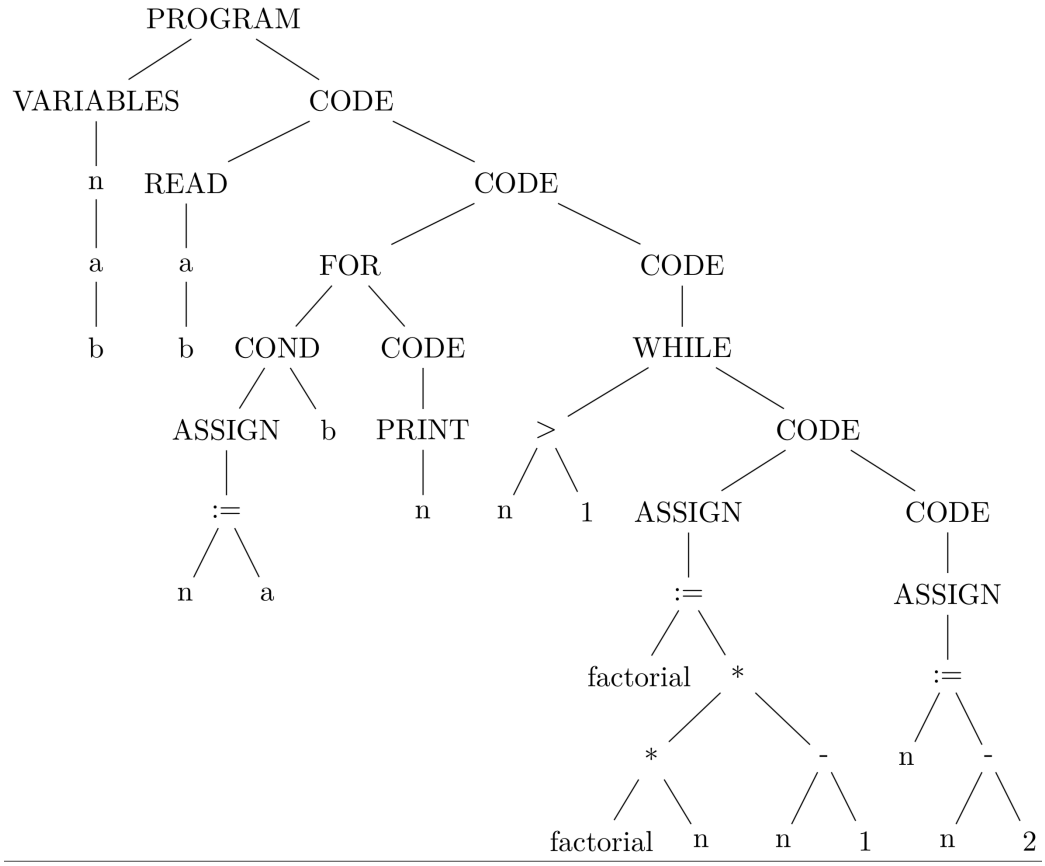

Figure 4: AST for cycles.

Notice that the AST representation for the READ and PRINT instructions are straightforward.

3 LLVM Code Generation

In this section we will describe the main idea to generate llvm code based on the abstract syntax tree. Basically, we walk trough the tree and we evaluate what the content of the node is and based on that, we take a decision to generate some code or keep walking deeper into the AST. For instance, if we have an instruction that executes an assignment of an arithmetic operation we walk through the tree until the bottom and generate the llvm code of bottom-most arithmetic operation, we keep in memory the llvm variable used and we return it to the upper tree levels (as a string) and then if there is another operation that needs the previously first computed operation we only access that variable.

The code generation includes by default pre-charged functions for printing and reading integers.

The code generation for the conditions is as follows: first, we compute the code for both branches, left and right (let's suppose that we have a simple AND condition for simplicity) so they are going to contain (in llvm code) a value of 1 if the condition was met, 0 otherwise. Afterwards in order to evaluate the AND operator we simply multiply both and we can get 0 if one of them was not met and 1 otherwise. In case we have an OR condition we make something similar but instead of multiplying we sum up both amounts, this result can be up to 2 (for instance, if both conditions were met). For multiple conditions we apply the same rule consecutively until we evaluate all the

conditions. Finally, we end up with a value greater or equals= to 0 so if it is greater than 0 means that the global condition was met.

The code generation for the NOT instruction inverts the condition before generating the code.

The code generated for the cycles are very similar to the ones we saw in the last exercise of the practice of llvm. Basically, we generate the block of code that computes the condition, then we evaluate the condition, if it is met, we jump the execution into the label that contains the piece of code that needs to be executed, otherwise we jump the execution into the label which is the end of the cycle. Notice that the FOR cycle is generated by incrementing the initial value until it reaches some given value.

4 New Line-commands

We introduced a couple of commands that help us with the execution of the llvm code and generation of the AST tree.

4.1 Generation of llvm file

In order to generate the llvm code and output into a file, we need to execute the command `-o [filename.ll]`. If no file is given, the program will create one by default named `outputFile.ll`.

4.2 Execution of .sf file

In order to execute a Super-Fortran file, we need to execute the command `-exec [code.sf]`. This will autogenerate a llvm file called `autogen.ll` and it will execute it. The output of the program will be displayed into the console. If there is some error with the llvm file (for instance, a variable was not declared properly in the Super-Fortran file to execute) the program will output the llvm error message as well. For accomplishing this command execution we use the `Runtime.getRuntime().exec(command)` command provided by the Java language. The limitation of using this approach is that if the `.sf` file executes some interaction with the user (for instance, a READ command) the program will not have a way to retrieve the input to the user and pass it to the llvm program; nevertheless, the program will output the data if there is a PRINT instruction.

4.3 See the AST generated tree

In order to see the AST tree that represents the minimized structure of the program, the command `-ast` needs to be executed, this will create the `ast.tex` file that can be opened with Latex, similar to the parse tree generated in part 2 of the project.

4.4 Final conclusions and comments

During these three phases of the project, we really understood how a compiler works and we realized that it is not easy to implement one. A big amount of aspects needs to be considered before starting the design and development part. This compiler, even if it is a very basic one, provides all the steps to build a real one and it can be improved in thousands of ways that we, as a team, would have enjoyed to implement but it is not possible due to time constrains. Additionally, it allows us to discover such powerful tools like CUP, flex, llvm and so on.