INFOF403 Project

Part 3

Submitted to:

Gilles GEERAERTS
Léonard BRICE
Sarah WINTER

Submitted by:

Bermudez Arroyo Loïc

Submitted on:

01/01/2023

Table of Contents

1. Introduction

This third project is about the generation of a LLVM intermediary code from the parse tree/derivation tree built from the parser implemented in the last part. To achieve this code generation, it is needed implement a semantic analyzer and finally the code generator.

2. Abstract Syntax Tree

To do a semantic analysis, the compiler must check multiple things, three in this case:

- Control flow: checks the order in which the instructions are to be executed.
- Scoping: all variables that had been declared in one scope can't be used elsewhere in the code but at the contrary a variable name can exist in multiples scopes across the code.
- Type control: when a variable is assigned, it is needed to be sure that the value associated or given to it is of the same type as the variable or that a correct conversion has been applied or must be applied.

To achieve this semantic analysis, the parse tree generated after the Fortress's code parsing, must be analyzed from top to bottom. But the problem with a parse tree is that there are a lot of nodes, that might be useless for the semantic analysis but that are useful to check whether the input is coherent with the grammar or not.

That is why I opted to first build an abstract syntax tree from the generated parse tree. An abstract syntax tree is also a parse tree but only keeping arbitrarily chosen key element useful to do the semantic analysis and following an arbitrarily chosen structure. In fine, an AST is a sharpened parse tree only keeping the essential nodes and wisely build to make the semantic analysis easier. The more sharpened is the AST, the easier the semantic analysis will be and the easier the code generation will be.

As the grammar used to parse the input file is LL(1) and deterministic, it is an easy to select the different node I wanted to keep from the parse tree. Each parse tree's nodes that are not a leaf corresponds to a non-terminal variable from the grammar (left hand-side of the grammar). And as each non-terminal produces a predefined number of elements (terminals and/or non-terminals) in a predefined order, it is easy to choose what to keep and what to discard while processing the parse tree.

```
1   [1] <Program>           → BEGIN [ProgName] <Code> END
2   [2] <Code>              → <Instruction> <CodeF>
3   [3]                     → ε
4   [4] <CodeF>             → , <Code>
5   [5]                     → ε
6   [6] <Instruction>       → <Assign>
7   [7]                     → <If>
8   [8]                     → <While>
9   [9]                     → <Print>
10  [10]                    → <Read>
11  [11] <Assign>           → <[VarName]> := <ExprArith>
12  [12] <If>               → IF (<Cond>) THEN <Code> <IfSeq>
13  [13] <IfSeq>            → END
14  [14]                    → ELSE <Code> END
15  [15] <While>            → WHILE (<Cond>) DO <Code> END
16  [16] <Cond>             → <ExprArith> <Comp>
17  [17] <Comp>             → = <ExprArith>
18  [18]                    → > <ExprArith>
19  [19]                    → < <ExprArith>
20  [20] <ExprArith>        → <Prod> <ExprArithF>
21  [21] <ExprArithF>       → + <Prod> <ExprArithF>
22  [22]                    → - <Prod> <ExprArithF>
23  [23]                    → ε
24  [24] <Prod>             → <Atom> <ProdF>
25  [25] <ProdF>            → * <Atom> <ProdF>
26  [26]                    → / <Atom> <ProdF>
27  [27]                    → ε
28  [28] <Atom>             → -<Atom>
29  [29]                    → ( <ExprArith> )
30  [30]                    → [Number]
31  [31]                    → [VarName]
32  [32] <Print>            → PRINT([VarName])
33  [33] <Read>             → READ([VarName])
```

Figure 1: LL1 grammar

2.1. Implementation

2.1.1. Abstract Syntax Tree

This class represent the structure of the abstract syntax tree. It has the same structure as the given parse tree class. A node is composed of a Label which is a Symbol and a list of children being empty if the node is a leaf or filled if not.

2.1.2. AST Generation

This class parse the parse tree and retrieve the desired symbols and set them in different nodes of the AST. The new tree will be structured in the following way:

- The “Program” node will have only one child, the Code node

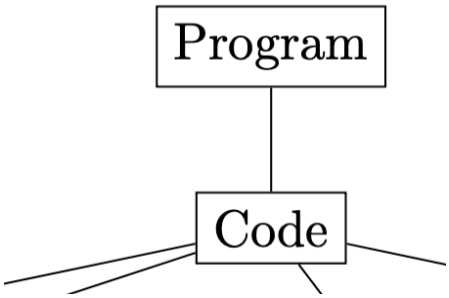


Figure 2: Program node of the Factorial.fs AST

- The “Code” node will have one/multiple child(ren, those children if there are any, is a list of independent instructions, so each code is composed of multiple or unique instruction(s)

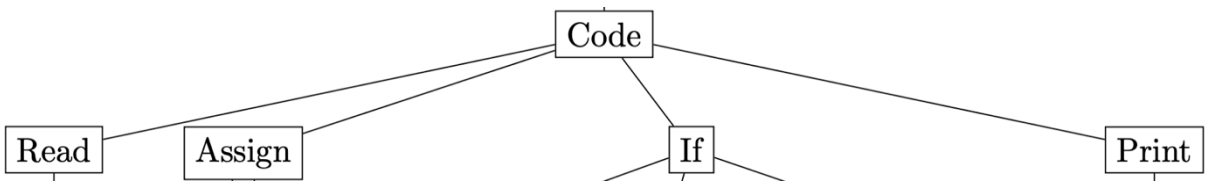
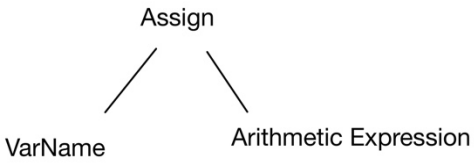
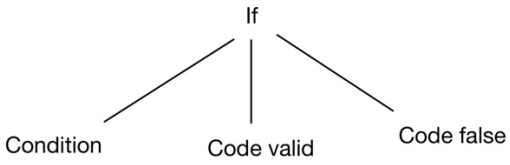


Figure 3: Code node of the Factorial.fs AST

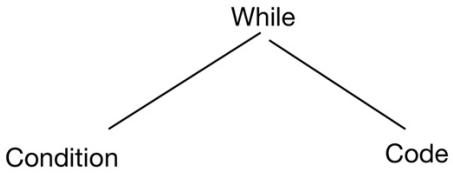
- The Assign node is the assign instruction, that node possess two children, a VarName and an arithmetic expression



- The If node is the if instruction, this node possess three children, a condition and two codes, the first is to be executed if the condition is validated and the second is to be executed if the condition isn't validated



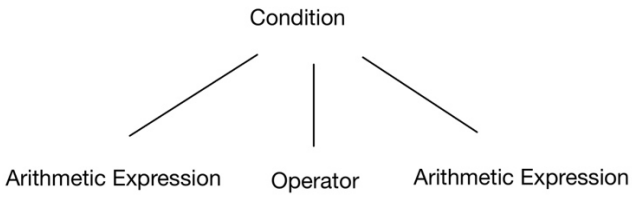
- The While node is the while instruction, it possess two children, a condition and a code that has to be executed if the condition is validated.



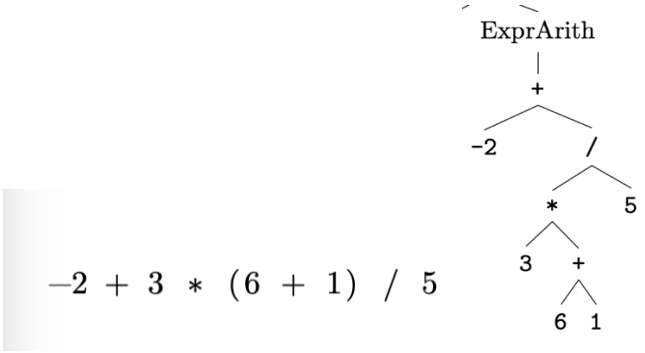
- The Print node is the print instruction, it possess only one child the VarName

- The Read node is the read instruction, it possess only one child the VarName

- The Condition node is the condition that has to be tested. This node has three children, an arithmetic expression, an operator and another arithmetic expression.



- The Arithmetic Expression node is structured like a binary tree with each root being an operator and every leaf is a VarName or a Number. For example, the following code is transformed into this structured AST.



3. LLVM Generator and SubFunctions

The " SubFunctions" class is a new class, which each object corresponds to an instruction that has to be executed. Each object possesses a label of the instruction and the content of the instruction.

The "LLVMGenerator" is the class that will create the llvm code from the abstract syntax tree previously generated. The class possess two main variables: the variable list that will store and track all the variables that appears in the AST and the list of subfunctions. The variable list keeps in check all the variables and checks if they have been used before assignment. The subfunctions list allows the instruction to be generated in the right order.

The "GenerateLLVMCode" method starts by doing a reset of every variable used by the class, as between different uses the variables weren't emptied and the llvm produced were somehow with lines of codes coming from other outputted files. The method also adds two functions: the read and the prinln, that are already in llvm format.

The "generate" method will then call the "ProcessAST" which will check the different instructions of contained in the AST and will dispatch them to the corresponding instruction processing method. Each method creates its own subfunction object as it corresponds to the instruction and give a label to the object. The instruction processing methods will then translate each subtree that was given to them into the llvm output code.

After the construction of each subfunction, the "generate" method will add each variable contained in the variable list to be written at the start of the main() function of the llvm code, that way they can be allocated at the start of the program. It will then add the content of each subfunction object in the right order to the final output string.

It will at the end of the method return the whole output as a single string variable to the main where it will be printed on the standard output.

4. Challenges

4.1. Abstract Syntax Tree generation

During that part of the development, I first tried to implement a recursive method that will delete every node I judged useless in the parse tree itself, but I faced a wall afterward: how to make certain nodes to go up in the tree while keeping the other ones in place, I then tried to do that by adding the parent element to the parse tree structure, but it didn't go well.

And then after checking attentively my LL1 grammar from the part 2, I realized that the grammar we produced needed to be deterministic in order to be parsed, I also realized that the pattern in the grammar and in the parse trees repeated each time the parser crossed path with certain values from the input file. From there I then parsed the parse tree to only retrieve the nodes I really needed.

The second challenge about the AST is how much to refine the original parse tree, as the llvm code generation will be easier the much sharper the AST is , but how much ? because too sharper also is not a good idea, if the AST is to sharp I may lose information and then have missing outputs. I then followed the AST example in the pdf to have the same structure.