ALGOL-0 Introduction to language theory and compiling Project – Part 3

Mounir Bounamar Mustapha Cherrabi

December 2019

1 Introduction

As part of this course, we were asked to implement a compiler for the ALGOL-0 language. This project was divided into 3 parts and the third part was to implement a functional compiler for basic instructions for the ALGOL-0 language. In order to compile the ALGOL-0 code, we used the LLVM intermediary language (LLVM IR) to get an output response. LLVM is a low level language which makes it possible to translate a given language into a low-level instruction for it to be executed by a computer.

2 Implementation

One of the main difficulties we encountered was the way in which we had to transform an arithmetic expression so that the priority of operations is respected. Several approaches were possible, one of them being the transformation of our parsed tree into an AST (abstract syntax tree). This approach proved to be more complex to implement because of several aspects like the handling of each sub tree or our lack of knowledge on the subject. It is for these reasons that we decided to implement the Shunting-yard algorithm.

2.1 Shunting-yard

Shunting-yard algorithm is a method used to parse a mathematical expression. This method allow us to have the priorities of each operation and most important, it keeps them in the right order. We can not only have arithmetic operation but also comparison/boolean operator (like *and*,*or*,*not*,<,>,..).

We will give an example of the application of this algorithm for a better understanding.

2.1.1 Example

Let's have a simple arithmetic expression like this one :
$a := 1 * (2 + 5) + 9 - 1 / 9$
We will use two stacks in order to backup our data :

- a general stack where we have our final expression.
- a stack for operators that will get pop each time we encounter an operator with greater priority.

We first encounter the number 1 and we simply put it in the general stack because it is not an operator.
Step 1:

General Stack	Operators Stack
1	

Then we meet the operator $*$. Two cases are possible :

- If the top of the stack operator has a lower priority than the encounter operator, we push it in the stack operator.
- In case of a higher priority in the top of the stack operator, we remove it and we add it in the general stack. We then add the encounter operator in the stack operator

It is important to note that if the stack operator is empty, we don't need to do any checking. We simply push the operator in the stack operator.
Step 2:

General Stack	Operators Stack
1	*

When we meet an open parenthesis, we put it in stack operator and we keep on handling the rest of the expression with the same rule has described before. As soon as we encounter a closing bracket, we transfer all the information between the bracket from the stack operator to the general stack, if there is any. We do not need to keep the bracket in the general stack, so we can get rid of them.
Step 3:

General Stack	Operators Stack
1	(
	*

Step 4:

General Stack	Operators Stack
2	(
1	*

Step 5:

General Stack	Operators Stack
2	+
1	(
	*

Step 6:

General Stack	Operators Stack
5	+
2	(
1	*

Step 7:

General Stack
+
5
2
1

Operators Stack
*

Step 8:

General Stack
*
+
5
2
1

Operators Stack
+

we will proceed this way until we finish reading our arithmetic expression. Once finished, we will only need to transfer the rest from the stack operator to the general one.

Step n:

General Stack
-
/
9
1
+
9
*
+
5
2
1

Operators Stack

This algorithm works perfectly for arithmetic expressions whether simple or complex. We also added the comparison operators/booleans to be able to reuse the same technique in the case of a condition (i.e *if,while,for*). All this is managed by our class **Arithmetic**. The function **transformToSYA** will be called recursively until we encounter an interesting symbol (a number/varname/operator/bracket).

In case where we have a comparison operator, we will clear all the stack operator but keep all the boolean operator because they have a higher priority

and they need to be placed right after the evaluation, e.g the condition $x > 0$ and $b < 1$ will produce a stack like this $[x, 0, >, b, 1, <, and]$ where "and" is at the top of the stack. It is important to note that the booleans operators have the highest priority (not > and > or) meaning that each time we encounter them , we need to clear all the stack and apply the same priority rule has described in the beggining of this section.

2.2 LLVM

Once we have the general stack filled, we need to write all the operation with the right LLVM syntax. Indeed we do that in the class **ArithExpression** where the function **calcul** will be called recursively until we finished reading our stack.

This function will keep on going through the stack until it meet either an arithmetic operator or a boolean/comparator operator. We will then take the 2 previous expressions, it can be a register, a varname or a number, and will apply the operator on them using the function **addInstruction** which will add the instruction to our list. After that, we have to remove these 3 elements from the stack and replace it by the register where this operation took place. At the end, we should have only one element left in the stack and it is the register where the last value is stored.

Here is the representation of the example showed before written in LLVM syntax and we can see that the priority of operations is respected.

```
%1 = add i32 2, 5
%2 = mul i32 1, %1
%3 = add i32 %2, 9
%4 = sdiv i32 1, 9
%5 = sub i32 %3, %4
%a_0 = alloca i32
```

Figure 1: LLVM syntax for $a := 1 * (2 + 5) + 9 - 1 / 9$

2.3 Error Handling

LLVM handle already very well the detection of error but we decided to handle the obvious one our self. Indeed, if we use a variable, either in an assignation or in a condition, that has not been declared, we will throw an error to the user directly specifying which variable is causing the trouble. We do that in the function **loadVariable** in the class **ScopeHandler** which will simply check if the variable is present in our list that keeps track of all our declared variable.

One of the error, that we decided to not handle but LLVM do it for us, is when we declare a variable only in an instruction, i.e in a *if,else,while,for* and then decide to use it outside his declaration scope. Even if we do not handle the scope meaning each time a variable is declared/changed it impacts the variable in a global way. LLVM does not allow us to reuse the variable. This has the

effect that even if we declare the variable outside its scope again, it can not be used. This will be explained in the next part

2.4 Hypothesis

It is not specified in the grammar that a variable can not be declared multiple times. Thus, when we have a variable that has already been declared, we made the hypothesis that it is a simple assignation and the value of the variable is changed without storing again the variable in the memory as if it was a new one. Indeed, we will never have an error telling us that a variable has already been initialized. However, knowing that LLVM does not allow the use of a variable declared in an instruction scope, simply reusing the same variable outside the instruction (even if reassign it) will throw an error. This kind of code will not work because of c being initialized in the *if* statement.

```
begin Error
    x:=3;
    if x=3 then
      c:=0
    endif;
    c:=10;
    print(c)

end
```

Figure 2: Error program

Another hypothesis that has been made is that when we have an assignation, the variable is created only after the expression on the right is fully evaluated. Therefore, such an assignation will return an error: $a := 3 + a$ if a is declared for the first time.

We had the choice between returning an error or initializing all the variables to 0 as soon has we read them. We decided to throw an error because it seemed more logical to us.

2.5 Modified files

Some files have been modified during this last part in order to fit our need, we will quickly summarize them and explain why we did such a change.

Grammar.java : we added 5 more rules at the end of the grammar allowing us to do the bonuses. We will explain which bonus more in detail later in the next section.

ParserTree.java : During the part 2, we had trees that can have epsilon value. It turned out that it was easier if the node with an epsilon value was just deleted. Thus, We do not have to worry about whether a node has valid children

or not, i.e children with no epsilon value. We also added the rule next to each node's name allowing us to know directly which algorithm to apply depending on the rule we are on.

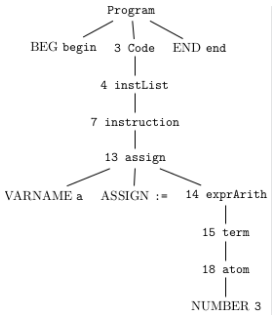

Figure 3: tree of a simple assignation

LexicalAnalyzer/LexicalUnit.java : We had to add some token in order for our lexical analyzer to detect new information like the name of the program or the *elif* statement.

3 Bonus

We decided to implement 3 bonuses that seemed interesting to us. We will explain them and give a short example in the following lines.

3.1 Several Programs

The first bonus is allowing to have multiple programs in a single file, i.e functions. Thus, the user has the choice between writing a single program, as he could still do, or writing several one. In this case, the approach is a little different. Indeed, there is a possibility for the user to write the name of each program just after the **begin** token. It is important to note that if no name were given to a program, we will give one per default. The program name must have an upper case letter for the first letter and can be followed by number, lower/upper case letter. However, the user can not give the same names to two program, if they are on the same file obviously.

We had to modify our grammar, as previously said, in order for our parser to recognize the possible program name. Thus, this change is visible by the addition of 4 new rules and a slight modification of the first one:

```
"<Program> -> begin <ProgName> <Code> <End>",
"<End>  -> end ",
"<End>  -> end <Program>",
"<ProgName> -> EPSILON",
"<ProgName> -> NameProgram"
```

Figure 4: rule for program name

It is important to note that even if we do not manage the scope, each declaration/modification of a variable is local to the program. Therefore, if a variable is declared only in the first program, it will not be usable (unless it is declared) in the other programs. This is possible thanks to the fact that we are calling the function **generateInstruct** in the **Generate** class recursively and we create a new scope handler each time.

3.1.1 Several Programs example

There are many examples in the test folder testing different aspects of this bonus. However, we will show here the tree and the LL file that comes out of a test file having 2 programs, one without a name and the other one with a name.

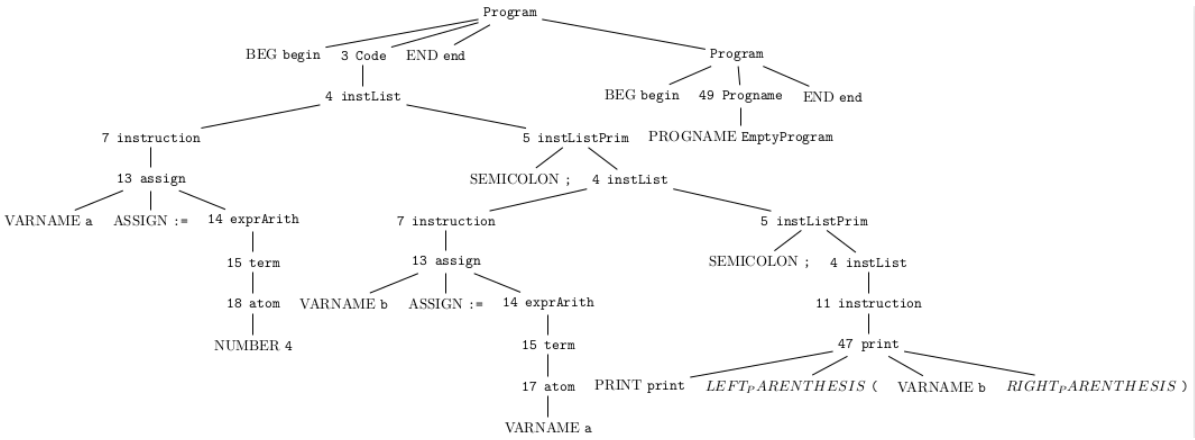

Figure 5: tree of the test file **14_SeveralPrograms_b.alg**

We can see here that the first program does not have a name but in our ll file, we will give him one per default so we can call it in our main function.

```
define void @defaultFunctionName0(){
%a_0 = alloca i32
store i32 4, i32* %a_0
%b_0 = alloca i32
%1 = load i32, i32* %a_0
store i32 %1, i32* %b_0
%2 = load i32, i32* %b_0
call void @println(i32 %2)
ret void
}

define void @EmptyProgram(){
ret void
}

define i32 @main(){
call void @defaultFunctionName0()
call void @EmptyProgram()
ret i32 0
}
```

Figure 6: LL file of the test file **14_SeveralPrograms_b.alg**

We call the function *println* which is stated no matter what happens at the very beginning of our ll file. We reuse those seen in courses (same for the function read).

3.2 For loop

The second bonus we implemented is allowing to have a *for* loop going backward , i.e : <For>: FOR [Varname] FROM <ExprArith1> BY <ExprArith2> TO <ExprArith3> means that the program should initialize the variable named $Varname$ to $ExprArith1$. Then, check if the value of $ExprArith2$ is positive or negative.

- If it is positive : we will keep on going until $Varname$ is **greater or equal** than $ExprArith3$
- If it is negative : we will keep on going until $Varname$ is **lower or equal** than $ExprArith3$

The value of $ExprArith2$ can change in the body of the for loop. We decided to adapt our code in order to switch the comparator according to the sign of $ExprArith2$. Thus, we have to be careful about how $ExprArith2$ is modified (in case where $ExprArith2$ is a variable) otherwise we can end up in an infinite loop.

Examples are available in the test file showing some for loop going backward such as $for\ x$ from 0 by -1 to -5.

3.3 Elif statement

An *elif* statement is like having another *if* so each time we have an *elif* we simply call the same function recursively, **buildIF** from the **Generate** class, until having the token signaling the end of the instruction (*endif*). We have to make sure that we do not write multiple times the code for the *else* statement. We avoid doing that by putting a boolean variable that will be true if the *else* statement has been added in the list of instruction.

In order for the *elif* statement to be recognize by our parser, we had to change a bit the grammar. We added one new rule very similar to the *if* rule.

```
"<ifPrim> -> elif <cond> then <code> <ifPrim> ",
```

Figure 7: *elif* rule

Here we have a tree with several *elif* statement, some with code and other without :

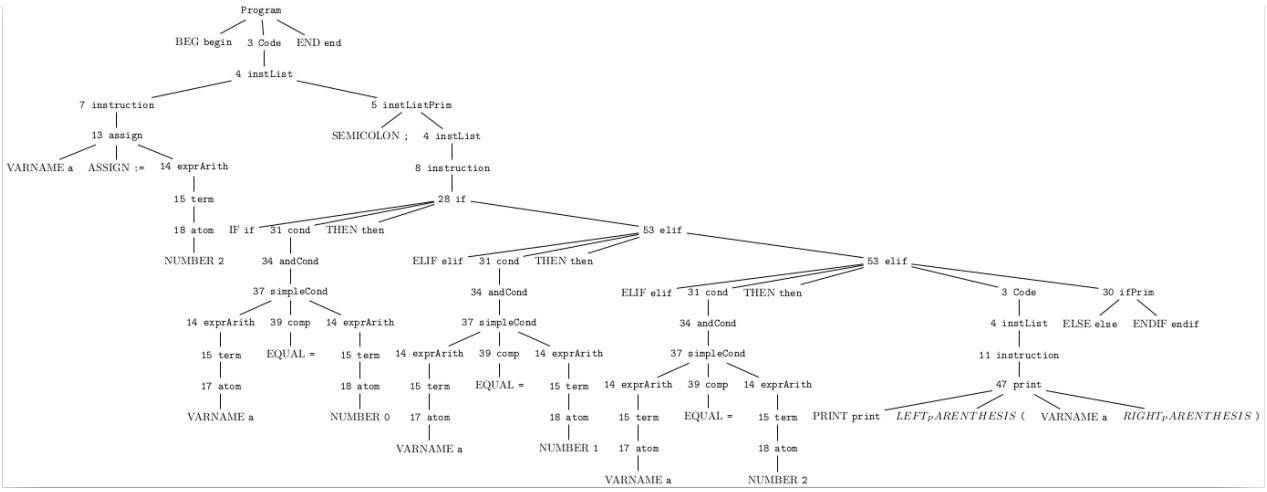

Figure 8: *elif* tree of the test file **07_EmptyIf.alg**

4 Javadoc and tests files desciption

We generated the javadoc automatically thanks to the javadoc library made available by java. Javadoc is very useful for monitoring our project and can be consulted to stay updated about the project. In addition, it gives a good support for someone outside the project wanting to understand our code. It is important to note that we are using *java 11 (11.0.4)* and *llvm-as-3.7*.
We have several **.alg** files to test our implementation and the bonuses. The

name of each file corresponds to the test it does. These test files are in the test folder. The command for running our executable is : **java -jar part3.jar testFile.alg [OPTION(S)]** where **OPTION(S)** is **-o outputFile.ll**. Here a some examples that we judged interesting to explain :

1. **02_complexAssign.alg** : This test aims to show that our Shunting-yard algorithm works well even if we have several complex assignations. We can put as many minus in front of a number as we want, it will be considered at the end as a positive/negative number thanks to the fact that we consider having a negative number like having $0-$ the number. We used the same logic for the *not*. Indeed, having a *not* is like having 1 *xor* something.

2. **09_nestedIf_b.alg** : This example is interesting in the sens that *elif* statement are not initially part of the grammar but we decided to add it because it makes the *if* statement easier to read. We have here several *elif* some with code and other without. We decided to add nested *if* and *elif* simply to show that our implementation works properly.

3. **13_Severalprograms_a.alg** : This file will be used to test the case where we have several programs. Some will have a program name, others will not. As explained before, each variable is local to his program. Thus, some variables will have different values depending on their declaration in their scope (program).

5 Conclusion

During these 3 parts, we have been confronted with several aspects that have been unknown until now such as the use of **jflex** and **LLVM** tools as well as building a recursive descent LL (1) parser.
The fact that the project is divided into several parts, allowed us to understand the essential aspects in the implementation of a compiler for a given language. We have been confronted with the basic aspects up to the most complex aspects. The fact of having certain freedom in the implementation, in the grammar and in the bonuses have been beneficial to us. This allowed us to use some approaches that we find more appropriate in order to solve some problems.