Introduction to language theory and compiling

INFO-F-403

Project : Part 3

18 December 2020

BAKKALI Yahya : 000445166
HAUWAERT Maxime : 000461714

Université Libre de Bruxelles (ULB)

Contents

1 Introduction

The objective of this project is to design a compiler for a new language Fortr-S. The first and second components of a compiler, the lexical analyser and the syntax analyser, have already been implemented in the first and second part of this project. In this third and last part, the semantic analyser and the code generator, will be implemented.

2 Description

2.1 Semantic analyser

It analyses what the code means in order to prepare its translation. The semantic analysis consists of verifying three different aspects.

- Visibility and scoping : Verification here includes ensuring that all variables declared in one scope cannot be used elsewhere. Similarly, it must be ensured that the same variable name can coexist in different scopes and other similar things.
- Type control : In general, the function return and the variable have a type. Thus, when assigning a variable, for example, the value to be assigned must be checked to ensure that it has the same type as the variable or that a conversion has been explicitly made or must be made when necessary.
- Control flow : It basically verifies the order in which the instructions of the program will be executed.

2.2 Abstract syntax tree

It is a derived version of the parse tree. It is meant to delete useless nodes of the parse tree to keep only the useful elements for analysing the structure of the program, these useless nodes were useful only to check whether the input was coherent with the grammar or not.

2.3 Code generator

It converts a certain representation of the code of a program into a code executable by a computer. Generally, a code generator takes an abstract syntax tree as input and converts it into a code in an intermediate language. This can be useful to write one compiler for different architectures as only the intermediate language has to adapt itself for the targeted architecture.

2.4 Basic block

A basic block contains a list of instructions that execute sequentially. It has no branch in except at the beginning and no branch out except at the end. It is useful in the analysis process, where the code must be divided into multiple basic blocks. Each basic block corresponds to a node in a control flow graph.

3 Bonus

In this part of the project, several features can be added. It has been decided that one of them will be implemented. The chosen functionality consists of adding logical operators and adding options to run the program with.

3.1 Logical operators

The added operators will be the following ones:

Operator	**Description**
<	less than
<=	less than or equal to
>=	greater than or equal to
!=	not equal to

To do this, some lexical and syntactical changes should be made. First, the corresponding tokens of the operators will be added for them to be recognized by the deterministic automata. Then some modifications will be done to the <Comp> rules.

The starting grammar for the rule <Comp>.

<Comp>	→	=
	→	>

The modified grammar for the rule <Comp>.

<Comp>	→	=
	→	! =
	→	>
	→	>=
	→	<
	→	<=

These modifications imply that the first and follow should be recomputed in order to keep the integrity of the parser. The first and follow that had to change :

Symbol	First	Follow
<Expr>	[VarName], [Number], -, (	[EndLine], =, >,)
<Expr'>	+, -, ϵ	[EndLine], =, >,)
<Prod>	[VarName], [Number], -, (	[EndLine], +, -, =, >,)
<Prod'>	*, /, ϵ	[EndLine], +, -, =, >,)
<Atom>	[VarName], [Number], -, (	[EndLine], +, -, *, /, =, >,)
<Comp>	=, >	[VarName], [Number], -, (

The result after all the computations :

Symbol	First	Follow
<Expr>	[VarName], [Number], -, (	[EndLine], =, !=, >, >=, <, <=,)
<Expr'>	+, -, ϵ	[EndLine], =, !=, >, >=, <, <=,)
<Prod>	[VarName], [Number], -, (	[EndLine], +, -, =, !=, >, >=, <, <=,)
<Prod'>	*, /, ϵ	[EndLine], +, -, =, !=, >, >=, <, <=,)
<Atom>	[VarName], [Number], -, (	[EndLine], +, -, *, /, =, !=, >, >=, <, <=,)
<Comp>	=, !=, >, >=, <, <=	[VarName], [Number], -, (

3.2 Options

3.2.1 -o

The `o` option allows the users to save their program generated by the code generator instead of printing it on the terminal. The generating code can be run through `LLVM IR` by executing it with the `lli` command.

3.2.2 -exec

The `exec` option allows the users to directly test the program through java. In order to easily implement it, it has been decided to create a temporary file using the `File.createTempFile` function. Then the program executes the `lli` command on this file. When executing `lli` on a `.ll` file it directly interprets the code. The process executing the command inherits the standard output and input, for the users to be able to interact with the command (read) and to view the results (print). The file is then deleted.

4 Implementation

The implementation of the lexical analyser and the syntax analyser were reused for this part. So, in this section only the additions and the modifications will be shown.

4.1 AbstractSyntaxTree

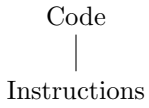
In order to have a more readable code it has been decided to add a `semantics` package with a class for each semantic. These classes allow the construction of the abstract syntax tree.

4.1.1 Program

The program is a simple node with only one child, the code of the program.

4.1.2 Code

The code is composed of a list of instructions.

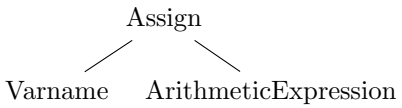


4.1.3 Instructions

All the instruction implements the `Instruction` interface. This interface allows these instructions to be represented together as known as `Instruction`. This interface has a function `dispatch`. This function allows the generator to generate the right code for the specific instruction. This removes the need of using the non-recommended `instanceof` functions.

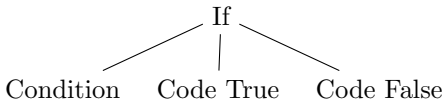
4.1.3.1 Assign

The assign instruction is composed of a `varname` and an arithmetic expression.



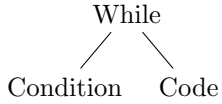
4.1.3.2 If

The if instruction is composed of a condition and two codes. One code that has to be executed when the condition is true and the other one has to be executed when the condition is false.



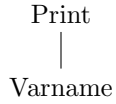
4.1.3.3 While

The while instruction is composed of a condition and a code. The code is the one that has to be executed while the condition is true.



4.1.3.4 Print

The print instruction is composed of a `varname`.

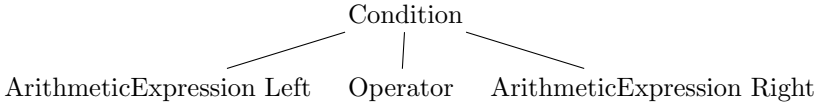


4.1.3.5 Read

The read instruction is composed of a `varname`.

4.1.4 Condition

The condition is composed of two arithmetic expressions and an operator. The operator is the boolean operation that has to be executed on the two arithmetic expressions, it is either =, ! =, >, >=, < or <=.

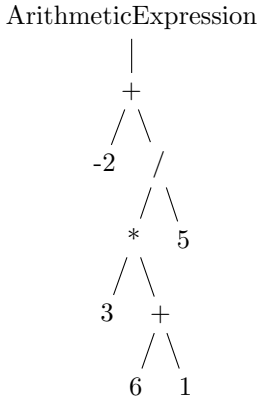


4.1.5 ArithmeticExpression

In order to store the arithmetic expression, it has been decided to use a binary tree. The interior nodes of this binary tree will have as data an operator and the leaves will have a `varname` or a `number`.

The following arithmetic expression is transformed into the following binary tree.

$-2 + 3 * (6 + 1) / 5$



4.2 CodeGenerator

The new class `CodeGenerator` has been added. It performs the semantic analysis while generating the code. Its `generate` function generates the code with the specified program as known as the root of the abstract syntax tree. It adds at the beginning of the generated code the code of two functions : read and println. While parsing the abstract tree it keeps a list of all variables. It allows the code generator to allocate memory for all the variables in the code and to detect when a variable is used before its assignment.

For each variable the following instruction is added at the beginning of the main functions.

```
%variable = alloca i32
```

`where`

- variable is the name of the variable.

4.2.1 Program

It runs the code generation of its code and then adds the following instruction in order to end the program.

```
ret i32 0
```

4.2.2 Code

It asks the different instructions it has to dispatch themselves to the right functions to generate them.

4.2.3 Assign

It assigns the result of its arithmetic expression to its `varname`.

```
store i32 %tempVariable , i32 * %variableName
```

`where`

- tempVariable is the variable containing the result of its arithmetic expression.
- variableName is the name of its variable.

4.2.4 If

First it runs the code generation of its condition.

Then it creates a new basic block, launches the code generation of its code when the condition is true then adds the following instruction.

```
br label %endLabel
```

`where`

- endLabel is the name of the basic block containing the code after this if instruction.

After that, it does the same thing with the code when the condition is false.
Finally, it creates a new basic block that will contain the code after this if instruction.

4.2.5 While

First it adds the following instruction.

```
br label %condLabel
```

`where`

- condLabel is the name of the basic block containing the code of its condition.

Then it runs the code generation of its condition.
After that it creates a new basic block, launches the code generation of its code and adds the following instruction.

```
br label %condLabel
```

`where`

- condLabel is the name of the basic block containing the code of its condition.

Finally, it creates a new basic block that will contain the code after this while instruction.

4.2.6 Print

It prints the value of its `varname` on the standard output.

```
%tempVariable = load i32 , i32 * %variableName
call void @println(i32 %tempVariable)
```

`where`

- tempVariable is a free variable.
- variableName is the name of its variable.

4.2.7 Read

It reads the value put on the standard input and assigns it to its `varname`.

```
%tempVariable = call i32 @readInt()
store i32 %tempVariable , i32 * %variableName
```

`where`

- tempVariable is a free variable.
- variableName is the name of its variable.

4.2.8 Condition

It takes the values of its two arithmetic operations, then executes its operator on the two values and finally redirects the code accordingly to the value of the condition.

```
%condVariable = icmp operator i32 %variableA , %variableB
br i1 %condVariable , label %trueLabel , label %falseLabel
```

`where`

- variableA is the variable that contains the result of its left arithmetic expression.
- variableB is the variable that contains the result of its right arithmetic expression.
- condVariable is a free variable.
- operator : `eq` for =, `ne` for ! =, `sgt` for >, `sge` for >=, `slt` for < and `sle` for <=.
- trueLabel is the label of the basic block to be executed when the conditon is true.
- falseLabel is the label of the basic block to be executed when the conditon is false.

4.2.9 ArithmeticExpression

It takes its binary tree and runs the `node` method on its root. Finally, it returns the temporary variable where its value has been saved to.

4.2.9.1 Node

It returns the temporary variable where its value has been saved to.

Its value is the result of its operator on the values of its left child and its right child.

```
%result = operator i32 %resultLeft , %resultRight
```

`where`

- result is a free variable.
- operator : `add` for +, `sub` for -, `mul` for * and `sdiv` for /
- resultLeft is the result of the left child.
- resultRight is the result of the right child.

4.2.9.2 Leaf

It returns the temporary variable where its value has been saved to.

The leaves can have two different types of data :

- `Number`

  ```
  %result = add i32 0 , number
  ```

  `where`

  - result is a free variable.
  - variableName is the value of its number.

- `Varname`

  ```
  %result = load i32 , i32 * %variableName
  ```

  `where`

  - result is a free variable.
  - variableName is the name of its variable.

4.3 Binary tree

The class `BinaryTree<T>` is an implementation of a binary tree. A binary tree is a tree that has at most two children. The T represents the type of the data it contains. In this project the T will always be `Symbol` so it will be omitted in this report.

4.4 SemanticException

A new type of exceptions `SemanticException` has been added to the `exception` package. As the other exceptions, its constructor only needs a message describing the error. It is only thrown when a variable is used before being defined as it is the only possible semantic error.

4.5 Program manual

NAME
part3.jar – A compiler generator that generates the FORTR-S language code and interprets it.

SYNOPSIS
java -jar part3.jar inputFile.fs
java -jar part3.jar inputFile.fs [-o outputFile.ll]
java -jar part3.jar inputFile.fs [-exec]

DESCRIPTION
The inputFile.fs specifies the path of the file to use, also the following options are available:

-o Save the LLVM IR code in a file called outputFile.ll instead of printing it

-exec Generate and directly interpret the LLVM IR code (without printing it or saving it in a file)

EXAMPLES
java -jar part3.jar Factorial.fs -exec

5 Description of example files

Four example files were created.

- ASSIGNexample.fs : This example contains an error as there is a print of a variable before its assignment.
- ASSIGN.fs : This example contains also an assignment error as a variable has been used in an arithmetic operation before its assignment.
- Largest.fs : This example does not contain any error. It also shows that the -exec option works flawlessly, as the IO operations between the processes are well detected.
- OPERATORSexample.fs : This example does not contain any error. It shows only that all the logical operators work fine.

For each source code example that does not contain any error there is a ".ll" file that can be executed with the *lli* command.

6 Conclusion

In this part of the project, a semantic analyser has been implemented. This parser is the continuation of the other parts and the final compiler phases. The purpose of the semantic parser is to take the syntax analysis tree generated by the parser and transform it into an abstract syntax tree in order to be able to interpret it and generate executable code using the lli command. By implementing this semantic analyser, the construction of a complete compiler for the FORTR-S language is well finished.

To summarize, this project has been divided into three main parts, each one dealing with a compiler component.

The first part deals with the lexical analysis and the use of Jflex to create a deterministic automata to check the input file and generate its tokens.

The second part was the syntax analyser where the notion of grammar was introduced. The grammar is intended to deal with rules that cannot be verified by the first part, for example the problem of multiple nested comments. The grammar uses for this purpose the first and follow which will also be reused to complete the action table. Once the action table has been defined, the time comes to create a parse tree that represents the left-most derivation obtained by the syntax analyser.

Finally, the last part concerns the semantic analyser. In this part, the goal was to gather all the other parts and to interpret them to obtain an executable code for the input file.

As it can be seen, building a compiler is a difficult task that requires a lot of work to handle all possible cases. The implementation of the structure must also be well designed to allow the easy addition and/or deletion of new keywords, operations and additional types.