INFO-F-403
Introduction to language theory and compiling
-
ALGOL-0 Part 3

Antoine de Selys - Arthur Pierrot

000443288 - 000422751

Teachers Gilles GEERAERTS - Raphaël BERTHON - Sarah WINTER

Academic Year 2019-2020

November 2019

Contents

1 Statement

In this third part of the project we had to augment the recursive-descent LL(1) parser we had written during the first and second parts in order to let it generate code that corresponds to the semantics of the Algol-0 program that is being compiled. The output code must be LLVM intermediary language (LLVM IR).

2 Introduction

As a reminder in the second part of the project we had different purpose concerning our ALGOL-0 compiler. First, we had to transform the ALGOL-0 grammar given. In particular, the transformation that consists in modifying the grammar to take into account priority and associativity of operators allows one to remove (some) ambiguities of grammars. In a second time, we had to check if our grammar is *LL(1)* and write the *action table* of an *LL(1)* parser for the transformed grammar.

So for this third project our main goal was that our parser had to be upgrade in order to produce a code corresponding to the code being compiled but the output code had to be LLVM.

3 Definition

LLVM IR is the backbone that connects frontends and backends, allowing LLVM to parse multiple source languages and generate code to multiple targets. Frontends produce the IR, while backends consume it. The IR is also the point where the majority of LLVM target-independent optimizations takes place. The choice of the compiler IR is a very important decision. It determines how much information the optimizations will have to make the code run faster. On one hand, a very high-level IR allows optimizer to extract the original source code intent with ease. On the other hand, a low-level IR allows the compiler to generate code tuned for a specific hardware more easily [1].

4 Choices and Hypotheses

On the one hand to obtain a LLVM code we decided to reuse the *Parser* we made in the second project and we assume that it was perfectly working.

On the other hand we followed the guidelines provided in the statement with the main goal of generating code from our parser.

4.1 Grammar Modification

In order to do the first bonus, we had to modify the grammar. Below is the grammar before the modifications.

[1]	<Program>	$\longrightarrow$	begin <Code> end
[2]	<Code>	$\longrightarrow$	ϵ
[3]		$\longrightarrow$	<InstList>
[4]	<InstList>	$\longrightarrow$	<Instruction> <NextInst>
[5]	<NextInst>	$\longrightarrow$	ϵ
[6]		$\longrightarrow$	; <InstList>
[7]	<Instruction>	$\longrightarrow$	<Assign>
[8]		$\longrightarrow$	<If>
[9]		$\longrightarrow$	<While>
[10]		$\longrightarrow$	<For>
[11]		$\longrightarrow$	<Print>
[12]		$\longrightarrow$	<Read>
[13]	<Assign>	$\longrightarrow$	[VarName] := <ExprArith>
[14]	<ExprArith>	$\longrightarrow$	<Prod> <ExprArith'>
[15]	<ExprArith'>	$\longrightarrow$	+ <Prod> <ExprArith'>
[16]		$\longrightarrow$	- <Prod> <ExprArith'>
[17]		$\longrightarrow$	ϵ
[18]	<Prod>	$\longrightarrow$	<Atom> <Prod'>
[19]	<Prod'>	$\longrightarrow$	* <Atom> <Prod'>
[20]		$\longrightarrow$	/ <Atom> <Prod'>
[21]		$\longrightarrow$	ϵ
[22]	<Atom>	$\longrightarrow$	- <Atom>
[23]		$\longrightarrow$	[Number]
[24]		$\longrightarrow$	[VarName]
[25]		$\longrightarrow$	(<ExprArith>)
[26]	<If>	$\longrightarrow$	if <Cond> then <Code> <IfSeq>
[27]	<IfSeq>	$\longrightarrow$	endif
[28]		$\longrightarrow$	else <Code> endif
[29]	<Cond>	$\longrightarrow$	<CondAnd> <Cond'>
[30]	<Cond'>	$\longrightarrow$	or <CondAnd> <Cond'>
[31]		$\longrightarrow$	ϵ
[32]	<CondAnd>	$\longrightarrow$	<SimpleCond> <CondAnd'>

[33]	<CondAnd'>	$\longrightarrow$	and <SimpleCond> <CondAnd'>
[34]		$\longrightarrow$	ϵ
[35]	<SimpleCond>	$\longrightarrow$	<ExprArith> <Comp> <ExprArith>
[36]		$\longrightarrow$	not <SimpleCond>
[37]	<Comp>	$\longrightarrow$	=
[38]		$\longrightarrow$	>=
[39]		$\longrightarrow$	>
[40]		$\longrightarrow$	<=
[41]		$\longrightarrow$	<
[42]		$\longrightarrow$	/=
[43]	<While>	$\longrightarrow$	while <Cond> do <Code> endwhile
[44]	<For>	$\longrightarrow$	for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> endwhile
[45]	<Print>	$\longrightarrow$	print([VarName])
[46]	<Read>	$\longrightarrow$	read([VarName])

Table 1: Grammar before transformations

The following rules needed to be changed because we wanted the comparators to work the same way as the arithmetic operators. The <SimpleCond> had also to accept the boolean parenthesis.

<SimpleCond>	$\longrightarrow$	<ExprArith> <Comp> <ExprArith>
	$\longrightarrow$	not <SimpleCond>
<Comp>	$\longrightarrow$	=
	$\longrightarrow$	>=
	$\longrightarrow$	>
	$\longrightarrow$	<=
	$\longrightarrow$	<
	$\longrightarrow$	/=

<SimpleCond>	$\longrightarrow$	<ExprArith> <SimpleCond'>
	$\longrightarrow$	not <SimpleCond>
	$\longrightarrow$	(<Cond>)
<SimpleCond'>	$\longrightarrow$	< <ExprArith>
	$\longrightarrow$	<= <ExprArith>
	$\longrightarrow$	= <ExprArith>
	$\longrightarrow$	> <ExprArith>
	$\longrightarrow$	>= <ExprArith>
	$\longrightarrow$	/= <ExprArith>

We can now see the new grammar created. With these modifications, we can rearrange the tree the same way for comparators, *and*, *or*, *not* and arithmetic operators.

[1]	<Program>	$\longrightarrow$	begin <Code> end
[2]	<Code>	$\longrightarrow$	ϵ
[3]		$\longrightarrow$	<InstList>
[4]	<InstList>	$\longrightarrow$	<Instruction> <NextInst>
[5]	<NextInst>	$\longrightarrow$	ϵ
[6]		$\longrightarrow$	; <InstList>
[7]	<Instruction>	$\longrightarrow$	<Assign>
[8]		$\longrightarrow$	<If>
[9]		$\longrightarrow$	<While>
[10]		$\longrightarrow$	<For>
[11]		$\longrightarrow$	<Print>
[12]		$\longrightarrow$	<Read>
[13]	<Assign>	$\longrightarrow$	[VarName] := <ExprArith>
[14]	<ExprArith>	$\longrightarrow$	<Prod> <ExprArith'>
[15]	<ExprArith'>	$\longrightarrow$	+ <Prod> <ExprArith'>
[16]		$\longrightarrow$	- <Prod> <ExprArith'>
[17]		$\longrightarrow$	ϵ
[18]	<Prod>	$\longrightarrow$	<Atom> <Prod'>
[19]	<Prod'>	$\longrightarrow$	* <Atom> <Prod'>
[20]		$\longrightarrow$	/ <Atom> <Prod'>
[21]		$\longrightarrow$	ϵ
[22]	<Atom>	$\longrightarrow$	- <Atom>
[23]		$\longrightarrow$	[Number]
[24]		$\longrightarrow$	[VarName]
[25]		$\longrightarrow$	(<ExprArith>)
[26]	<If>	$\longrightarrow$	if <Cond> then <Code> <IfSeq>
[27]	<IfSeq>	$\longrightarrow$	endif
[28]		$\longrightarrow$	else <Code> endif
[29]	<Cond>	$\longrightarrow$	<CondAnd> <Cond'>
[30]	<Cond'>	$\longrightarrow$	or <CondAnd> <Cond'>
[31]		$\longrightarrow$	ϵ
[32]	<CondAnd>	$\longrightarrow$	<SimpleCond> <CondAnd'>
[33]	<CondAnd'>	$\longrightarrow$	and <SimpleCond> <CondAnd'>
[34]		$\longrightarrow$	ϵ
[35]	<SimpleCond>	$\longrightarrow$	<ExprArith> <SimpleCond'>
[36]		$\longrightarrow$	not <SimpleCond>

[37]		⟶	(<Cond>)
[38]	<SimpleCond'>	⟶	< <ExprArith>
[39]		⟶	<= <ExprArith>
[40]		⟶	= <ExprArith>
[41]		⟶	> <ExprArith>
[42]		⟶	>= <ExprArith>
[43]		⟶	/= <ExprArith>
[45]	<While>	⟶	while <Cond> do <Code> endwhile
[46]	<For>	⟶	for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> endwhile
[47]	<Print>	⟶	print([VarName])
[48]	<Read>	⟶	read([VarName])

Table 2: Grammar before transformations

4.2 Abstract Syntax Tree

Abstract Syntax Trees are meant to abstract away some terminals and non-terminals to obtain the very structure of the program. It as only necessary information for the compiler and all the operations are represented as tree with the operator at the top and the operands as children. The creation of the AST is done in 2 steps. First it creates the tree from a ParseTree (Figure 1) by removing all the useless nodes and can be seen in Figure 2. Then the tree obtained in Figure 3 is simplified by rearranging all the operations.
The rearrangement operates in two parts. It rearranges the tree for each side of the less priority operator. All the children at the left side of the operator will be made children of a temporary node which will be the left child of the operator. Then the function it repeats itself recursively on the temporary child. The same is done for all the children at the right side of the operator. This makes the operator to have only two children transforming the tree in a binary tree below him. The only operator with one child is the *not* operator.

4.3 Compiler

The compiler use the AST to compile the code to LLVM IR language. It is done by writing the code in a file during the process. The compiler uses counters for the unnamed variables, the for and while loops and the if/else statements. For the operations, the compiler evaluate the left term and the right term then do the right operation between them (except for the *not*).

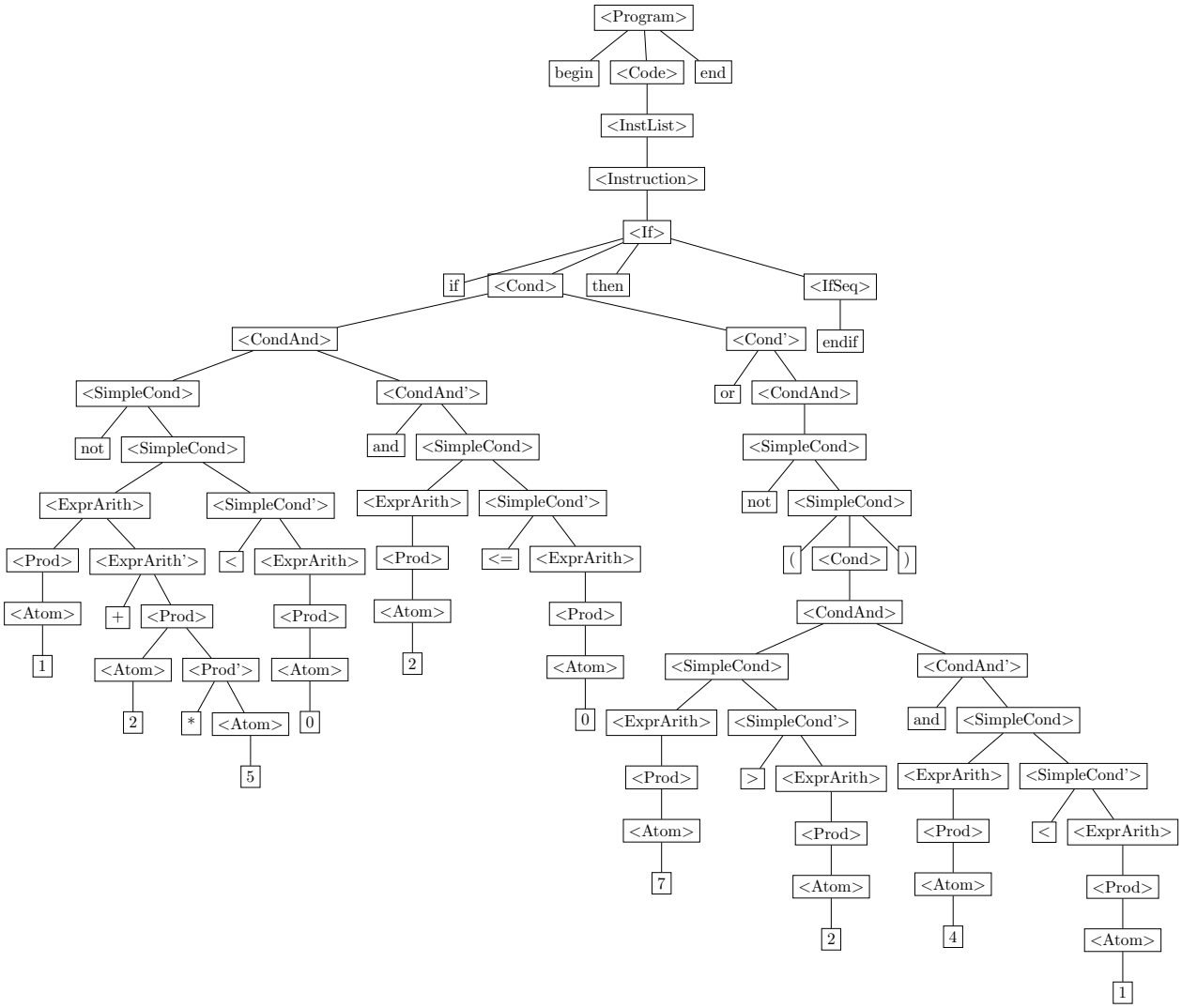

Figure 1: Derivation tree for the file *boolean-parenthesis.alg*

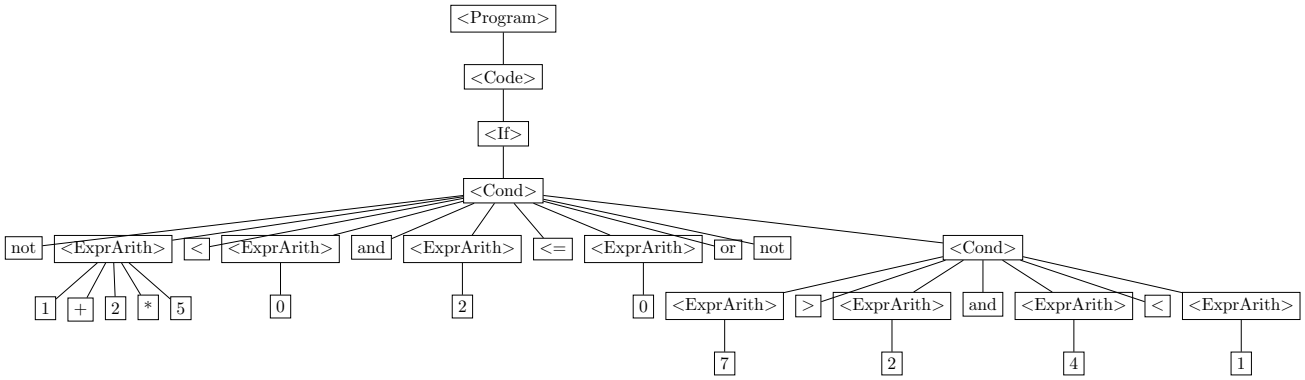

Figure 2: Tree after removing all the useless nodes

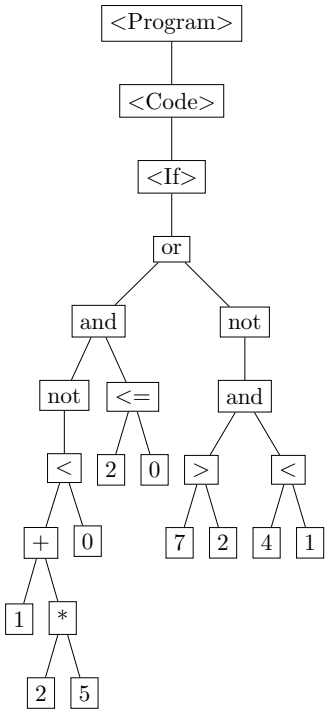

Figure 3: Abstract Syntax Tree for the file *boolean-parenthesis.alg*

4.4 Bonus

Concerning the bonus we improve the compiler for the syntactic sugar/simple extensions. Indeed the compiler allows the parentheses in the boolean expression.

The compiler can also remove empty for/while loops and if/else statements by not writing them in the LLVM IR file.

5 Difficulties encountered

We met several issues when trying to change the grammar because a change in the grammar made changes in different parts of the code. Those changes simplified mostly the compiler and the AST. Since the structure of the tree was different, the compiler had to use the tree differently.

6 Tests and Results

The command for running our executable is :
java -jar part3.jar -exec *algolFile.alg*

Several tests are available in the test folder including the *boolean-parenthesis.alg* which demonstrate the use of boolean parenthesis. The file *game.alg* contains a game written in ALGOL-0. The game contains comments to explain it in the file.

7 Conclusion

Lastly we can conclude that we accomplished the goal given in the hypotheses. Our parser generates a LLVM code in reasonable computation time. Indeed all the tests provided in the test folder are passed. The compiler allow the parentheses in the boolean expression but could had been better, for example, by implementing the functions or recursive functions.

References

[1] Gilles Geeraerts, Guillermo A.Pérez, *Introduction to language theory and compiling*, 2019.