ECOLE
POLYTECHNIQUE
DE BRUXELLES

MA1 - IRIF

COMPILING PROJECT

INTRODUCTION TO LANGUAGE THEORY AND COMPILING

Authors

Eliot COSYN

Quentin ROELS

Course

INFO-F403

Professor

Gilles GEERAERT

Academic Year

2022-2023

Table of contents

1 Lexical Analyser

1.1 Implementation

1.1.1 Regular Expressions

To match the tokens, we first had to determine the regular expressions of the keywords, numbers, variable names and program names allowed by the Fortress language

Keywords

The number of keywords in Fortress is pretty low. Therefore, to match those tokens, we can simply ask the lexical analyser to search for them specifically instead of searching uppercase word. To do that we can use the following regular expression : "TOKEN" (where TOKEN is a known token of Fortress such as BEGIN, IF, END,...). That way we don't return any symbol for an all uppercase word thinking that it's a Fortress keyword when it's not.

Numbers

"A [Number] represents a numerical constant, and is made up of a string of digits only, without leading zeroes", we then have to match the number 0 or any string of numbers starting with a non zero symbol [1-9] and followed by any numerical symbol [0-9]*. Since the minus sign is a token on its own, we expect him to be match from the regular expression "-" of previous section and we do not need to match it with the [Number] regular expression.

$$Number = ([1-9][0-9]*|0)$$

Then, to prevent any leading zeroes, we applied a filter with the [WrongNumber] regular expression. I.e. the numbers starting with at least one zero followed by any other digit.

$$WrongNumber = 0 + [0-9] +$$

Variables name

"A [VarName] identifies a variable, which is a string of digits and lowercase letters, starting with a letter". We thus have to match any string of symbol starting with a lowercase letter [a-z] and followed by any alphanumerical (non uppercase) symbol ([a-z]|[0-9])*.

$$VarName = [a-z]([a-z]|[0-9])*$$

Program name

"A [ProgName] identifies the program name, which is a string of digits and letters, starting with an uppercase letter but not entirely uppercase (e.g. FaCTORIAL, although not very pretty, is accepted, FactorialPrgm also, but FACTORIAL is not, so that it is not confused with a keyword)". To do that we need to put an uppercase letter at the start of the word [A-Z], and force the existence of a lowercase letter [a-z] somewhere in the word. Since it could be anywhere from thes start to the end of the word, we use the Kleene closure of alphanumerical symbols ([A-Z]|[a-z]|[0-9])* before and after the mandatory lowercase letter.

$$ProgramName = [A-Z]([A-Z]|[a-z]|[0-9])*[a-z]([A-Z]|[a-z]|[0-9])*$$

Due to the program name specifications, our lexer was not able to make the difference between [ProgName] and "TOKEN"[VarName], i.e. between a program name and a Fortress token linked to a variable with no white space separating them.

To avoid that problem, we added a PROGNAME state that we enter when matching the token "BEGIN". That way, our lexer already forces a program name directly after the keyword "BEGIN", which will not have a negative impact on the compilation of a valid Fortress script.

1.1.2 Detecting comments

As mentioned earlier, the job of the Lexical Analyser is to return the tokens of the language we want to compile. Since the content of the comments is useless for the compiler, we need to drop it.

Fortress comments

There are two ways for commenting a Fortress code :

- Starting with the string "::", short comments end when the end of line "$\backslash n$" symbol is reached
- Starting and ending with "%%", long comments allow to write on multiple line until the end symbol is reached

For both type of comments we created a new jflex state that will do nothing when matching symbols and will wait for the end of comment token. When the end of comment is matched, it returns to the main state and restart searching for Fortress symbols.

Nested comments

The problem we faced with nested comments in Fortress is due to the symbol used, "%%" is both the starting and ending comment symbol. This causes the program iterating on each symbol to not be able to tell if the matched token is used to start or end a comment. Which makes it impossible to use the same solution as nested parenthesis, using a stack to count how many parenthesis are currently open, and thus, how many are yet to close.

For nested comments to be implemented in our Fortress compiler, we should be able to know if "%% A %% B %% C %%" means that A and C are two different long comments **OR** that B is nested in the primary comment.

One possible way to implement the nested comments in our compiler would be to first consider them as two separated comments and then check if B is syntactically correct in Fortress. If so, we let it slide to the next phase of the compiling process, else, we drop the tokens like we would have done with comments. The limitation is then "no valid Fortress language in nested comments" instead of "no nested comments".

1.2 Testing

To test our lexical analyser, we edited the Fortress example file in different ways to see how the lexer would react in those situations.

1.2.1 Wrong tokens

First, we wrote all sorts of bad tokenized Fortress scripts to verify that the program exits with the appropriate error code. Every kind of token was reviewed by misspelling it or writting it poorly :

1. numbers (ex: 0012)
2. variable name (ex: reSult, 7result)
3. keywords (ex: READ -> REED)
4. program name (ex: FACTORIAL)

The correct use of long comments was also tested mainly to check if the comment was closed before the end of the file.

1.2.2 Valid files

Finally, we wrote valid Fortress scripts with no spaces or new line, or even more white spaces than necessary to check if our lexer could tokenize them properly and thus met the specifications of the project.

2 Syntactic Analyser

As a preliminary note, both the action table and the modified grammar are available here : `https://drive.google.com/file/d/1yNA4TWexr7KF3uPDZnjl1rXpsVKXCDV8/view?usp=share_link`

2.1 LL(1) Grammar

2.1.1 Unproductive and Unreachable variables

After applying the corresponding algorithms, neither unproductive nor unreachable variables were found in the starting grammar. I.e. by modifying the <ExprArith> variable such that

2.1.2 Ambiguity and Priority

The main part of the grammar modification was to implement the operator priority, which was done accordingly to the exercise session. I.e. by forcing the grammar to create additions of product and products of atoms as follows :

(1)	Exp	→	Exp + Prod
(2)		→	Exp − Prod
(3)		→	Prod
(4)	Prod	→	Prod ∗ Atom
(5)		→	Prod/Atom
(6)		→	Atom
(7)	Atom	→	−Atom
(8)		→	Cst
(9)		→	Id
(10)		→	(Exp)

Figure 2.1: Implementation of operators priority

2.1.3 Left-recursion and Factorisation

The second most important modification was to make our grammar a LL(1) grammar. To do so, we first removed left recursion in our grammar by using the corresponding algorithm, giving us this result :

$$
\begin{array}{lrll}
(1) & \mathsf{Exp} & \rightarrow & \mathsf{Prod}\,\mathsf{Exp}' \\
(2) & \mathsf{Exp}' & \rightarrow & +\mathsf{Prod}\,\mathsf{Exp}' \\
(3) & & \rightarrow & -\mathsf{Prod}\,\mathsf{Exp}' \\
(4) & & \rightarrow & \varepsilon \\
(5) & \mathsf{Prod} & \rightarrow & \mathsf{Atom}\,\mathsf{Prod}' \\
(6) & \mathsf{Prod}' & \rightarrow & *\mathsf{Atom}\,\mathsf{Prod}' \\
(7) & & \rightarrow & /\mathsf{Atom}\,\mathsf{Prod}' \\
(8) & & \rightarrow & \varepsilon \\
(9) & \mathsf{Atom} & \rightarrow & -\mathsf{Atom} \\
(10) & & \rightarrow & \mathsf{Cst} \\
(11) & & \rightarrow & \mathsf{Id} \\
(12) & & \rightarrow & (\mathsf{Exp})
\end{array}
$$

Figure 2.2: Operators without left recursion

Next, we had to factorise the 'IF' expression to allow the parsing with only one symbol of look-ahead. This was done the same way as done in the exercise sessions :

$$
\begin{array}{llll}
(1) & [\text{if}] & \rightarrow & \texttt{if}\ [\text{Cond}]\ \texttt{then}\ [\text{Code}]\ \texttt{fi} \\
(2) & [\text{if}] & \rightarrow & \texttt{if}\ [\text{Cond}]\ \texttt{then}\ [\text{Code}]\ \texttt{else}\ [\text{Code}]\ \texttt{fi}
\end{array}
$$

Figure 2.3: If statement before factorisation

$$
\begin{array}{llll}
(1) & [\text{if}] & \rightarrow & \texttt{if}\ [\text{Cond}]\ \texttt{then}\ [\text{Code}]\ [\text{ifSeq}] \\
(2) & [\text{ifSeq}] & \rightarrow & \texttt{fi} \\
(3) & [\text{ifSeq}] & \rightarrow & \texttt{else}\ [\text{Code}]\ \texttt{fi}
\end{array}
$$

Figure 2.4: If statement after factorisation

2.1.4 Final Grammar

By summing up all previous modifications of the grammar, we obtained this grammar :

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

Figure 2.5: Fortress LL(1) Grammar

2.2 Action Table

2.2.1 First and Follow

Symbol	First[1]()	Follow[1]()
<Program>	BEGIN	
<Code>	[VarName] IF WHILE PRINT READ	ELSE END
<Instruction>	[VarName] IF WHILE PRINT READ	
<Assign>	[VarName]	
<Exp>	[VarName] [Number] - (	
<Exp'>	+ -	) = > < ,
<Prod>	[VarName] [Number] - (	
<Prod'>	* /	+ -) = > < ,
<Atom>	[VarName] [Number] - (	
<Comp>	= > <	
<If>	IF	
<EndIf>	ELSE END	
<Cond>	[VarName] [Number] - (	
<While>	WHILE	
<Print>	PRINT	
<Read>	READ	

Figure 2.6: First and Follow set of the grammar variables

2.2.2 Final Action Table

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

Figure 2.7: Action Table

2.3 Implementation

2.3.1 Structure

The implementation of the semantic analyser comes in a unique class named Parser. Its goal is to build a parsing tree given the lexical analyser or *lexer* built from the input file. One of its most important method is the *match* method. The goal of this method is to check the correspondence between the expected token (from the parser point of view, i.e. logical token given grammar ruler) and the token of look-ahead.

2.3.2 Recursive parsing method

The parsing is realised using recursive functions, one for each different variable of the grammar (i.e. all the different left-hand side of the grammar rules). The action table was used to set up *switch-cases* structures corresponding to different look-ahead possibilities leading to specific application of grammar rules. When a grammar rule is to be applied, the function recursively calls the variables and matches the terminals in order of appearance in the right-hand side of the grammar rule applied.

While parsing the FORTRESS file, it also builds the parse tree corresponding to the file and returns it as a ParseTree object. From the recursive structure of the code, it is quite easy to create the parse tree, each time that a symbol is matched (which means that it is a terminal), we return it to the calling function as a ParseTree leaf, adding it to the children of the node created by the calling grammar rule, which recursively lead to building the whole file parse tree.

2.4 Testing

As for testing in the section 1.2, we wrote some test files from the basic example, removing useful symbols here and there to see the behaviour of the semantic analyser in those specific cases.

3 Synthesis

3.1 Abstract Syntax Tree (AST)

To design the AST we decided to use the parsing tree as a base. Going trough the ParseTree recursively we managed to create an AST by chopping down the useless leaves and reorganising the nodes. This reorganisation lets us generate LLVM code easier by solely parsing the resulting abstract syntax tree.

3.1.1 Recursive generation

The recursion function receives a ParseTree object and act according to the label of the main node. Each possibility is treated separately in a switch case. In essence, when receiving a ParseTree, the function parses the children of the main node in a given order and create a clean branch for the AST corresponding to the part of the ParseTree that was given as argument.

3.1.2 Handling priorities of operators

The main problem we encountered was the order of operators with the same priority in arithmetic expressions. Even though they were well arranged in the ParseTree, the first method we came up with was reverting the order of application of those operators as if each operator was between parenthesis from the point of view of the following operator. To fix that behaviour, we reverted the recursion to start parsing arithmetic expression from bottom to top, giving the last operator of a sequence of same priority operators the highest place in the three, forcing the computation of previous operators before computing its own result.

3.2 Generating LLVM code

The main goal of this part of the porject is to generate a LLVM code that corresponds to the input Fortress program. To do so, we decided to generate the code sequentially while reading the AST. We thus only needed to make the program read the AST in the right order and generate the corresponding parts of code. The easiest way to do so is by doing it recursively, the compiler generates the code by reading the AST recursively from the left-most to the right-most leaf (which is in fact, the same order as the instructions in the input Fortress program).

3.2.1 Handling variables

Temporary variables

When writing LLVM code, we need a lot of different variables and labels to compute the operations made by the program. The number of variable not being known a priori, we decided to use temporary variables (%name), those are simple numbers that must be used sequentially, which is exactly how we are generating the code, it is thus easy to implement the variables via a global counter kept as attribute of the compiler class. Each time we need a new variable, we give it the current number of the counter and we increment the counter, the next variable will be named the same way. Since the variables are most of time used to compute the next expression we needed to be able to refer to previous variables, we can do so by calling the the variable "%(counter - 1)". And the one before by subtracting one more unit to the counter.

We used the same system for label names using the nomenclature "labelX" where X is the label counter.

Named variables

For named variables the implementation was easier since we could refer directly to it through its name in the LLVM code, allowing to force the code to use exactly the variable it is supposed to use without having to keep a variable table. When the variable appears, either we are creating it, or we are using it from previous context. We thus use a set to insert each new variable created, if the insertion returns true, we achieved to insert the variable in the set, thus it was not defined, else, the variable was already defined. We can then choose between raising an error if the statement is not supposed to create variables, or to create the variable if the statement has the right to do so.

3.2.2 Statements

While assignation and computing of arithmetic expressions were easy to implement through their sequential nature as operations; print, read, while and if statement needed a bit more of thought. Here is how we implemented these statements :

IF statement

The IF statement, as it is referenced by the grammar of Fortress, contains, at best, two possibilities: both a “IF statement” and a “ELSE statement” containing code, or only a “IF statement” containing code and no “ELSE statement”. If the condition stated after the keyword “IF” is true we go to the first statement else, if it exists, we do the second statement instead. Since we do not know a priori if the condition is going to be true or false during the program execution, we must generate both possibilities. We thus separate them exactly as stated before using labels, jumping to the *if* or the *else* label depending on the condition result.

After both statement, whatever the execution, we need to come back to the main code after executing them. We thus ask at the end of the statements code to jump to a generic “restart label” that is generated at the end of the synthesis of the if statement. After jumping to this common label, for the LLVM it is exactly as if nothing had happened and keep generating the code as before in this new label that would be reached whatever the condition result.

The case we did not talk about yet is the case that does not contain an else statement. To simplify the generation, we do it exactly as if there was a statement, but we print code only if there is one, else we directly generate the “go to jump restart label”, making the first jump useless, and thus recreating a if statement without else statement.

WHILE loops

For the WHILE statement we decided to split the job in two distinct label :

- one label is used to test the condition
- the other is the code to execute while the condition is met.

This way, to start the execution of the while statement, we jump to the condition label. There, the condition is computed, if it is true we jump to the second label, else, we skip to the restart label (cf. IF statement). When entering the code section, we execute the sequentially until we reach the end of the loop. There we put a jump to the condition label, verifying the condition once more before

deciding to execute the code again or going back to the execution of the code outside the while statement.

PRINT/READ

Both print and read functions were implemented in LLVM during the second computer practical. To allow for their use in the code, not knowing a priori if it is going to be used, we decided to always include both functions definition at the start of the LLVM generated script.

3.3 Testing

The testing of this part was straight forward since each and every statement that exists in our Fortress language appears in the given example program "Factorial.fs". However, to ensure that our code was doing the right choices on operators priorities, we added a simple program with a completely random arithmetic expression to compute and print.