INFO-F403 - Introduction to Language Theory and Compiling

Compiling Project Part 2

DE VOS Sébastien

KALAI Tarik

GEERAERTS Gilles

WINTER Sarah

BERTHON Raphael

M1-IRIFS - 2021-2022

Table des matières

1 Introduction

In this report we will explain how we implemented a parser. We will also resume all the different steps that allowed us to have an LL1 grammar. And finally we will present the different tests we made to verify our parser.

First we will begin by removing the unproductive and unreachable variables (if we find any) then we make the grammar non-ambigous and finally in order to get an LL1 grammar, we remove the left recursion before applying factorisation to the grammar.

After that we implement the action table using the firsts and follows of each variables.

Finally we finish this report by explaining the implementation of the parser and the different tests realised.

2 Question 1

Remove unproductive variables

In order to remove the unproductive variables we answer to this question : "Which variables finish within a certain number of steps?" . We begin with V_0 (set of variables reachable in 0 steps) and finish, in this case, with V_4 (set of variables reachable within 4 steps). All the variables which are not present in V_4 are considered unproductive and can thus be removed.

$$V_0 = \{\emptyset\}$$

$$V_1 = \{Code, Op, Comp, ExprArith, Print, Read\}$$

$$V_2 = \{Code, Op, Comp, ExprArith, Print, Read, Program, Instruction, Assign, For, SimpleCond\}$$

$$V_3 = \{Code, Op, Comp, ExprArith, Print, Read, Program, Instruction, Assign, For, SimpleCond, InstList, Cond, SimpleCond\}$$

$$V_4 = \{Code, Op, Comp, ExprArith, Print, Read, Program, Instruction, Assign, For, SimpleCond, InstList, Cond, SimpleCond, If, While\}$$

We can see that all the variables are included in the V_4 set which means there are no unproductive variables. Thus we can not simplify the language.

Remove unreachable variables

In order to remove the unreachable variables we will answer to this question : "Which variables can be reach within a certain number of steps ?". For instance we have V_1 the set of all the variables which finish in 1 step.

$$
\begin{aligned}
V_0 =\ & \{Program\}\\
V_1 =\ & \{Program, Code\}\\
V_2 =\ & \{Program, Code, InstList\}\\
V_3 =\ & \{Program, Code, InstList, Instruction\}\\
V_4 =\ & \{Program, Code, InstList, Instruction, Assign, Id, While, For, Print, Read\}\\
V_5 =\ & \{Program, Code, InstList, Instruction, Assign, Id, While, For, Print, Read, ExprArith, Cond\}\\
V_6 =\ & \{Program, Code, InstList, Instruction, Assign, Id, While, For, Print, Read, ExprArith, Cond,\\
& Op, SimpleCond\}\\
V_7 =\ & \{Program, Code, InstList, Instruction, Assign, Id, While, For, Print, Read, ExprArith, Cond,\\
& Op, SimpleCond, Comp\}
\end{aligned}
$$

We can see than all the variables are present in V_7 so we can conclude that there are no unreachable variables and thus we can not simplify the language.

Making the Grammar non-ambiguous

[1]	$<Program>$	$\rightarrow begin <Code> end$
[2]	$<Code>$	$\rightarrow \epsilon$
[3]		$\rightarrow <InstList>$
[4]	$<InstList>$	$\rightarrow <Instruction>$
[5]		$\rightarrow <Instruction>; <InstList>$
[6]	$<Instruction>$	$\rightarrow <If>$
[7]		$\rightarrow <While>$
[8]		$\rightarrow <Assign>$
[9]		$\rightarrow <For>$
[10]		$\rightarrow <Print>$
[11]		$\rightarrow <Read>$
[12]	$<If>$	$\rightarrow if <Cond> then <Code> endif$
[13]		$\rightarrow if <Cond> then <Code> else <Code> endif$
[14]	$<While>$	$\rightarrow while <Cond> do <Code> endwhile$
[15]	$<Cond>$	$\rightarrow not <Cond>$
[16]		$\rightarrow <SimpleCond>$
[17]	$<SimpleCond>$	$\rightarrow <ExprArith><Comp><ExprArith>$
[21]	$<Comp>$	$\rightarrow =$
[22]		$\rightarrow >$
[23]		$\rightarrow <$
[24]	$<ExprArith>$	$\rightarrow <ExprArith> + <F>$
[25]		$\rightarrow <ExprArith> - <F>$
[26]		$\rightarrow <F>$
[27]	$<F>$	$\rightarrow <F> * <G>$
[28]		$\rightarrow <F> / <G>$
[29]		$\rightarrow <G>$
[30]	$<G>$	$\rightarrow - <G>$
[31]		$\rightarrow (<ExprArith>)$
[32]		$\rightarrow <H>$
[33]	$<H>$	$\rightarrow [VarName]$
[34]		$\rightarrow [Number]$
[35]	$<Assign>$	$\rightarrow [VarName] := <ExprArith>$
[36]	$<For>$	$\rightarrow for [VarName] from <ExprArith> by <ExprArith> to <ExprArith> do <Code> endfor$
[37]	$<Print>$	$\rightarrow print([VarName])$
[38]	$<Read>$	$\rightarrow read([VarName])$

In order to make the grammar non-ambiguous we arranged the Grammar with respect to the priority and associativity of the different operators. To be more specific, the most restrictive operator must be at a high position whereas the least restrictive ones must be on a low position. Regarding the associativity we had to introduce new variables for instance we introduced F in the rules 24, 25, 26 to respect the left associativity of the + and - operators. This has also been applied to other rules which can be seen on the table above.

Remove left recursion and apply factorisation

[1]	$< Program >$	$\rightarrow begin < Code > end$
[2]	$< Code >$	$\rightarrow < InstList >$
[3]		$\rightarrow \epsilon$
[4]	$< InstList >$	$\rightarrow < Instruction >< InstTail >$
[5]	$< InstTail >$	$\rightarrow ; < InstList >$
[6]		$\rightarrow \epsilon$
[7]	$< Instruction >$	$\rightarrow < If >$
[8]		$\rightarrow < While >$
[9]		$\rightarrow < Assign >$
[10]		$\rightarrow < For >$
[11]		$\rightarrow < Print >$
[12]		$\rightarrow < Read >$
[13]	$< If >$	$\rightarrow if < Cond > then < Code >< Tail >$
[14]	$< Tail >$	$\rightarrow endif$
[15]		$\rightarrow else < Code > endif$
[16]	$< While >$	$\rightarrow while < Cond > do < Code > endwhile$
[17]	$< Cond >$	$\rightarrow not < Cond >$
[18]		$\rightarrow < SimpleCond >$
[19]	$< SimpleCond >$	$\rightarrow < ExprArith >< Comp >< ExprArith >$
[20]	$< Comp >$	$\rightarrow =$
[21]		$\rightarrow >$
[22]		$\rightarrow <$
[23]	$< ExprArith >$	$\rightarrow < A >< B >$
[24]	$< A >$	$\rightarrow < F >$
[25]	$< B >$	$\rightarrow + < F >< B >$
[26]		$\rightarrow - < F >< B >$
[27]		$\rightarrow \epsilon$
[28]	$< F >$	$\rightarrow < C >< D >$
[29]	$< C >$	$\rightarrow < G >$
[30]	$< D >$	$\rightarrow * < G >< D >$
[31]		$\rightarrow / < G >< D >$
[32]		$\rightarrow \epsilon$
[33]	$< G >$	$\rightarrow - < G >$
[34]		$\rightarrow (< ExprArith >)$
[35]		$\rightarrow < H >$
[36]	$< H >$	$\rightarrow [VarName]$
[37]		$\rightarrow [Number]$
[38]	$< Assign >$	$\rightarrow [VarName] := < ExprArith >$
[39]	$< For >$	$\rightarrow for[VarName]from < ExprArith > by < ExprArith > to$ $< ExprArith > do < Code > endfor$
[40]	$< Print >$	$\rightarrow print([VarName])$
[41]	$< Read >$	$\rightarrow read([VarName])$

In this part we did 2 different modifications to the language.

First we got rid of left factorisation. Let's take InstList for example, before InstList was :

InstList -> Instruction

InstList -> Instruction ;InstList

Here we have a problem since we only have one look ahead, we can't know in which category we have to go. To solve this problem we change InstList as the following :

InstList -> Instruction InstTail

InstTail -> ;InstList

InstTail -> ε

Thus, this way we have removed the problem since with one look ahead we always have one possibility.

Secondly we applied the left recursion on ExprArith because he was calling himself on the left part, we can see here how it was before :

ExprArith -> ExprArith - F

ExprArith -> ExprArith + F

ExprArith -> F

Thus to resolve this issue we changed it to the following.

ExprArith -> A B

A -> F

B -> + F B

B -> - F B

B -> ε

By doing so an expression will extend itself on the right and no more on the left. this has been applied to multiple cases, see the table above (With the 41 lines).

3 Question 2

First and Follows

Variables	First	Follow
Program	begin	ϵ
Code	if, while, [VarName], for, print, read	end, endif, else, endwhile, endfor
InstList	if, while, [VarName], for, print, read	end, endif, else, endwhile, endfor
InstTail	;	end, endif, else, endwhile, endfor
Instruction	if, while, [VarName], for, print, read	;, end, endif, else, endwhile, endfor
If	if	;, end, endif, else, endwhile, endfor
Tail	endif, else	;, end, endif, else, endwhile, endfor
While	while	;, end, endif, else, endwhile, endfor
Cond	not, - , (, [VarName], [Number]	then, do
SimpleCond	- , (, [VarName], [Number]	then, do
Comp	=, >, <	-, (, [VarName], [Number]
ExprArith	- , (, [VarName], [Number]	=, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
A	- , (, [VarName], [Number]	+, -, =, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
B	- , +	=, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
F	- , (, [VarName], [Number]	=, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to, +, -
C	- , (, [VarName], [Number]	*, /, +, -, =, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
D	*, /	=, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to, +, -
G	- , (, [VarName], [Number]	*, /, +, -, =, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
H	[VarName], [Number]	*, /, +, -, =, >, <, then, do,), ;, end, endwhile, endif, else, endfor , by, to
Assign	[VarName]	;, end, endif, else, endwhile, endfor
For	for	;, end, endif, else, endwhile, endfor
Print	print	;, end, endif, else, endwhile, endfor
Read	read	;, end, endif, else, endwhile, endfor

Example on how we computed the Follows and Firsts for the variable Code :

Firsts We take on the right-end side the first element (if it is a terminal) for each rule where Code appears on the left side. For the rule 2 the first element of Code is IntList which brings us to Instruction which then brings us to If, While, Assign, For, Print, Read which give us in order : if, while, [VarName], for, print, read. Thus we have First(Code) = {if, while, [VarName], for, print, read}.

Follows we check on the right-end side where Code appears and for each one we take the following element. In the first rule we have Code followed by the terminal end. In the rule 13 it is followed by Tail, in this case we will take the First(Tail) as follows for ExprArith so we have : endif and else. In the rule 15 Code is followed by endif (which we already added so it does not change our set). In rule 16 Code is followed by the terminal endwhile. Finally in the rule 39 we have Code followed by the terminal endfor. Thus in the end Follow(Code) = {end, endif, else, endwhile, endfor}

We encountered more complex problem during the implementation of this table one of them was the Follow(Assign). So we firstly check where Assign appears on the right-end side, it happens for the rule 9. Nothing follows Assign so in this case we have to go up in the "tree". In other words we check where Instruction appears on the right-end side we can see it happening for the rule 4. it is followed by InstTail which gives us ";"(the first element on the right-end side of the rule 5). In this case we also have ϵ which means that we have to keep going up in tree : thus we see that InstTail appears on the right-end side for the rule 4. It is followed by nothing so we keep going up, InstList appears on the right-end side for the rule 2 it is again followed by nothing (and we also have the ϵ) so keep going up and finally we can take all the follows of Code which are end, endif, else, endwhile, endfor. Thus we have Follow(Exprarith) = {;, end, endif, else, endwhile, endfor }

Action Table

	begin	end	;	if	then	endif	else	while	do	endwhile	not	=	>	<	+	-	*	/	(	)	[VarName]	[Number]	:=	for	from	by	to	endfor	print	read
Program	**1**																													
Code		**3**		**2**		**3**	**3**	**2**		**3**											**2**			**2**				**3**	**2**	**2**
InstList				**4**				**4**													**4**			**4**					**4**	**4**
InstTail		**6**	**5**			**6**	**6**			**6**																		**6**		
Instruction				**7**				**8**													**9**			**10**					**11**	**12**
If				**13**																										
Tail						**14**	**15**																							
While								**16**																						
Cond											**17**					**18**			**18**		**18**	**18**								
SimpleCond																**19**			**19**		**19**	**19**								
Comp												**20**	**21**	**22**																
ExprArith																**23**			**23**		**23**	**23**								
A																**24**			**24**		**24**	**24**								
B		**27**	**27**		**27**	**27**	**27**		**27**	**27**		**27**	**27**	**27**	**25**	**26**				**27**						**27**	**27**	**27**		
F																**28**			**28**		**28**	**28**								
C																**29**			**29**		**29**	**29**								
D		**32**	**32**		**32**	**32**	**32**		**32**	**32**		**32**	**32**	**32**	**32**	**32**	**30**	**31**		**32**						**32**	**32**	**32**		
G																**33**			**34**		**35**	**35**								
H																					**36**	**37**								
Assign																					**38**									
For																								**39**						
Print																													**40**	
Read																														**41**

Let's take a closer look to how we completed this action table. We will take the example of Code. To know which numbers we have to put in which column we have to take a look at the firsts and follows if needed. Since we access all the firsts by rule 2 we will put at the intersection in the table between code and the firsts of code the rule number 2. Since rule number 3 is an ε we have to take a look at the follows of Code. We will thus put the number 3 in the intersection between Code and the follow of code. And this is the general idea for the rest of the table. First you take a look at the first of a variable, then you look which rule led to which first and you put it accordingly in the table. Finally if your variable gives ε then you will put the rule of ε in the follows of the variable.

4 Question 3

Implementation Choice We implemented a recursive descent LL1 parser.

We created as much functions as we have variables and we named them accordingly.

Whenever there are different choices, to change between them we use a switch case. When a variable can be derived as an ε we make the follows of this variable return cases of the given function.

For the other cases we just go through them and use *break* whenever we are finished. We used *break* when we just want to get out of the *switch case* but we still continue running the given function whereas we use return when we want to end the function.

If we encounter an error a function [1] is called, it throws an error and stops the code.

We implemented a *getToken* function which simply changes the token only and only if the previous token has been matched even when the *getNextToken* function is called.

In the *match* function we always try to update the next token, if the expected token doesn't match the token we currently have then we throw a syntax error otherwise, we update the previous token called "matched" and we add the current token at the root of our tree and we return the given root.

Each of our variable functions returns a *parseTree*.

During this whole process each time we enter a function or a switch case we add the given rule to the arraylist "leftMostD". When the parsing finishes we print the list of left most derivative : *leftMostD*.

Main In the main the first thing we check is if the length of the argument passed in the command line are lower than 1 or greater than 3. If this is the case we throw error exceptions and exit the code.

After that we will collect the arguments in the command line correctly filtered. Such as the argument after "-wt" is the "texFile".

1. syntaxError()

And by default if no "-wt" is present the input file will be the only argument.

The input file will be the other argument wether it is on the frst place or the last one it only depends on the position of "-wt". Which says that the file following it is the "texFile".

Then if the input file is null (has not been declared) such that we only have 2 arguments (-wt and texFile) then an error will be thrown.

Once all the filtering occured we try to open the input file and if we fail we catch the exception. Then if succeeded we start the Parser with the input file when the Parser will be finished if the texFile has been stipulated by command line then we will try to writte the parseTree inside it. Finally we close the texFile and the SourceFile.

Tests We kept the same tests (test_1.io, euclid.co) as in the first part of the project and we added two more.

We had to correct $test\_1.io$ as it did not get accepted by the grammar (which assured us that we had implemented a correct parser), after correcting it we didn't have any more problems.

We also implemented a test specially made to verify if the brackets were correctly detected : *multiplebracket.co*

We also discussed with another group (David and Aurélien) which implemented the parser with an action table which is a different implementation than us, thus we asked them to share one of their tests to make sure that our code wasn't in the wrong. The test in question is "primeNumbers.co".

5 Conclusion

During this part of the project we put into practice what we learned during the exercise sessions : we firstly changed the Grammar in order to have an LL1 grammar and we implemented the First and Follow of each variable of the grammar, we then implemented the Action table.

Afterwards we realised the code of the parser for this particular language (AlCol), we came aware of the fact that it was truly a parser when the test (test_1.co) we used for the first part of the project was not accepted in the language because of some errors (which did not matter in the first part).

To go a little bit further we added some features to the code, for instance an error will be launched if there are too many (or too few) arguments given as input, we also joined PDF files of the different trees we got for each test in the folder "more".