

UNIVERSITÉ LIBRE DE BRUXELLES



ÉCOLE
POLYTECHNIQUE
DE BRUXELLES



INTRODUCTION TO LANGUAGE THEORY AND COMPILING
INFO-F403

Project Pascalmaispresque Part 2 - Parser

Students :

ACHTEN Alexandre	494484
BIENFAIT Alexandre	513930

Professors :

GEERAERTS Gilles
SASSOLAS Mathieu
BRICE Léonard

November 2023

Academic year 2023-2024

1 Lexical Analyzer

1.1 Regular expressions

VARNAME : `[a-z][a-zA-z0-9]*`

A VARNAME must be a string of letters and digits starting with a lower case letter. The corresponding regular expression ensure that a varname starts by a lowercase letter. It is followed by a Kleene closure of letters and digits

NUMBER : `[0-9]+`

A number is a string of digits. There must be at least one digit in it.

1.2 MACROS

WHITESPACE : `[\t\r\n]+`

The whitespaces have been defined as the return of line, space, tabulation and the carriage return.

1.3 States

The lexical analyzer can be in the following states.

There are 3 different states :

1. Initial state

It is the basic state. It detects every key words of the language and return a symbol with the corresponding Lexical Unit. When encountering a character that isn't in the Lexical Unit, it raises an Illegal Character message.

2. Long comment state

This state starts when there is a ' ' in the initial state. It ignores any other character except another ' ' that sends back to the initial state.

3. Short comment state

This state start with the characters `**` and end where there is a newline character (`\n`).

We could implement nested comments and recursively call the long comment state to "count" the depth of comments. But the difficulty is that in our language we identify long comments with `" ' ' "`. There is no difference between the beginning and the end of long comments.

To use nested comments, we would have to create a new character to distinguish the beginning and end of comments (e.g. `"/**` and `**/"`).

1.4 Main file

The script in the main file takes as argument a file to analyse with the LexicalAnalyzer. It prints each token and it's lexical unit to the standard output. To print the symbol table, a `HashMap` is used to store the line of the first occurrence of each variable. Then the symbol table is printed to the standard output.

1.5 Tests

Tests have been carried out to ensure that all functionalities concerning the scanner are working. Here's a list of the different tests we've done :

- Test if spaces are not taken into account (regardless of the number of spaces)
- Test with an illegal character such as "A" or "?"
- Test the recognition of variable names such as "whileif" or "alex83"
- Test the short and long comments
- Test a number followed by characters without spaces between them (the program doesn't work correctly for that case and just return a NUMBER followed by a VARNAME)
- Test if a NUMBER starts with 0 (we could change the definition of the regular expression NUMBER by $[1-9][0-9]^*$, but we don't know if it's the purpose of the lexical analyzer)

To run all the tests, please use the `make tests` command.

2 Parser

2.1 Pascalmaispresque grammar

The table of the grammar rules of Pascalmaispresque is shown in the Figure 1.

[1]	<Program>	→ begin <Code> end
[2]	<Code>	→ ε
[3]		→ <InstList>
[4]	<InstList>	→ <Instruction>
[5]		→ <Instruction> ... <InstList>
[6]	<Instruction>	→ <Assign>
[7]		→ <If>
[8]		→ <While>
[9]		→ <For>
[10]		→ <Print>
[11]		→ <Read>
[12]		→ begin <InstList> end
[13]	<Assign>	→ [VarName] := <ExprArith>
[14]	<ExprArith>	→ [VarName]
[15]		→ [Number]
[16]		→ (<ExprArith>)
[17]		→ - <ExprArith>
[18]		→ <ExprArith> <Op> <ExprArith>
[19]	<Op>	→ +
[20]		→ -
[21]		→ *
[22]		→ /
[23]	<If>	→ if <Cond> then <Instruction> else
[24]		→ if <Cond> then <Instruction> else <Instruction>
[25]	<Cond>	→ <Cond> and <Cond>
[26]		→ <Cond> or <Cond>
[27]		→ { <Cond> }
[28]		→ <SimpleCond>
[29]	<SimpleCond>	→ <ExprArith> <Comp> <ExprArith>
[30]	<Comp>	→ =
[31]		→ <
[32]	<While>	→ while <Cond> do <Instruction>
[33]	<Print>	→ print ([VarName])
[34]	<Read>	→ read ([VarName])

FIGURE 1 : Initial Pascalmaispresque grammar

In order to get an LL(1) grammar, we must first remove unproductive and unreachable expression.

The only unproductive expression is $\langle \text{For} \rangle$ so we removed it. There are no unreachable expressions.

There are two groups of operations that have priority, arithmetic expressions with $\{+, -, *, /\}$ and conditions with $\{and, or\}$. To handle those priorities $\langle \text{ExprArith} \rangle$ and $\langle \text{Cond} \rangle$ were changed to the following form :

$\langle \text{ExprArith} \rangle$	$\rightarrow \langle \text{ExprArith} \rangle \langle \text{AddOp} \rangle \langle \text{MultExpr} \rangle$
	$\rightarrow \langle \text{MultExpr} \rangle$
$\langle \text{MultExpr} \rangle$	$\rightarrow \langle \text{MultExpr} \rangle \langle \text{MultOp} \rangle \langle \text{Term} \rangle$
	$\rightarrow \langle \text{Term} \rangle$
$\langle \text{Term} \rangle$	$\rightarrow (\langle \text{ExprArith} \rangle)$
	$\rightarrow - \langle \text{ExprArith} \rangle$
	$\rightarrow [\text{VarName}]$
	$\rightarrow [\text{Number}]$
$\langle \text{AddOp} \rangle$	$\rightarrow +$
	$\rightarrow -$
$\langle \text{MultOp} \rangle$	$\rightarrow *$
$\langle \text{Cond} \rangle$	$\rightarrow \langle \text{Cond} \rangle \text{ or } \langle \text{AndCond} \rangle$
	$\rightarrow \langle \text{AndCond} \rangle$
$\langle \text{AndCond} \rangle$	$\rightarrow \langle \text{AndCond} \rangle \text{ and } \langle \text{EndCondition} \rangle$
$\langle \text{AndCond} \rangle$	$\rightarrow \langle \text{EndCondition} \rangle$
$\langle \text{EndCondition} \rangle$	$\rightarrow \{ \langle \text{Cond} \rangle \}$
	$\rightarrow \langle \text{SimpleCond} \rangle$
$\langle \text{SimpleCond} \rangle$	$\rightarrow \langle \text{ExprArith} \rangle \langle \text{Comp} \rangle \langle \text{ExprArith} \rangle$

FIGURE 2 : right priority grammar

There is still ambiguity and left recursion in the grammar. It can be changed by factorising expressions and by adding intermediate states.

Factorisation has been used for $\langle \text{InstList} \rangle$ and $\langle \text{If} \rangle$ and we add new states for the different rules of arithmetic expressions and conditions.

The final grammar is presented at Figure 3.

[1]	<Program>	→ begin <Code> end
[2]	<Code>	→ <InstList>
[3]		→ epsilon
[4]	<InstList>	→ <Instruction> <Instruction_end>
[5]	<Instruction_end>	→ epsilon
[6]		→ ... <InstList>
[7]	<Instruction>	→ <Assign>
[8]		→ <If>
[9]		→ <While>
[10]		→ <Print>
[11]		→ <Read>
[12]		→ begin <InstList> end
[13]	<Assign>	→ [VarName] := <ExprArith>
[14]	<ExprArith>	→ <MultExpr><ExprArith'>
[15]	<ExprArith'>	→ <AddOp> <MultExpr> <ExprArith'>
[16]		→ epsilon
[17]	<MultExpr>	→ <Term> <MultExpr'>
[18]	<MultExpr'>	→ <MultOp> <Term> <MultExpr'>
[19]		→ epsilon
[20]	<Term>	→ (<ExprArith>)
[21]		→ - <ExprArith>
[22]		→ [VARNAME]
[23]		→ [NUMBER]
[24]	<AddOp>	→ +
[25]		→ -
[26]	<MultOp>	→ *
[27]		→ /
[28]	<If>	→ if <Cond> then <Instruction> else <If-tail>
[29]	<If-tail>	→ epsilon
[30]		→ <Instruction>
[31]	<Cond>	→ <AndCond> <OrCond>
[32]	<OrCond>	→ or <AndCond> <OrCond>
[33]		→ epsilon
[34]	<AndCond>	→ <EndCondition> <AndCond'>
[35]	<AndCond'>	→ and <EndCondition> <AndCond'>
[36]		→ epsilon
[37]	<EndCondition>	→ { <Cond> }
[38]		→ <SimpleCond>
[39]	<SimpleCond>	→ <ExprArith> <Comp> <ExprArith>
[40]	<Comp>	→ =
[41]		→ <
[42]	<While>	→ while <Cond> do <Instruction>
[43]	<Print>	→ print([VarName])
[44]	<Read>	→ read([VarName])

FIGURE 3 : Final Pascalmaispresque grammar

2.2 Grammar properties

The first thing we can deduce from the grammar rules is the First¹ and Follow¹ table. This is shown in the Table 1.

Symbol	First ¹	Follow ¹
Program	begin	ε
Code	ε [VARNAME] if while print read begin	end
InstList	[VARNAME] if while print read begin	end
Instruction_end	$\varepsilon \dots$	end
Instruction	[VARNAME] if while print read begin	\dots else end
Assign	[VARNAME]	\dots else end
ExprArith	(- [VARNAME] [NUMBER]	\dots else end) = < and or then do
ExprArith'	+ - ε	\dots else end) = < and or then do
MultExpr	(- [VARNAME] [NUMBER]	\dots else end) = < and or then + - do
MultExpr'	* / ε	\dots else end) = < and or then + - do
Term	(- [VARNAME] [NUMBER]	\dots else end) * / = < and or then + - do
AddOp	+ -	(- [VARNAME] [NUMBER]
MultOp	* /	(- [VARNAME] [NUMBER]
If	if	\dots else end
If-tail	ε [VARNAME] if while print read begin	\dots else end
Cond	{ (- [VARNAME] [NUMBER]	then } do
OrCond	or ε	then } do
AndCond	{ (- [VARNAME] [NUMBER]	or then }
AndCond'	and ε	or then } do
EndCondition	{ (- [VARNAME] [NUMBER]	and or then } do
SimpleCondition	(- [VARNAME] [NUMBER]	and or then } do
Comp	#SYNTAX	(- [VARNAME] [NUMBER]
While	while	\dots else end
Print	print	\dots else end
Read	read	\dots else end

TABLE 1 : First¹ and Follow¹ of the grammar

This grammar is strong LL(1) if :

$$First^1(\alpha_1 Follow^1(A)) \cap First^1(\alpha_2 Follow^1(A)) = \emptyset$$

With all the pair rules from the grammar under the $A \longrightarrow \alpha_n$.

PROOF :

First we check the definition on the variable **code** :

$$First^1(\varepsilon Follow^1(code)) = First^1(Follow^1(code)) = \{end\}$$

$$First^1(< InstList > Follow^1(code)) = \{[VARNAME], if, while, print, read, begin\}$$

And the intersection between these two sets is the empty set.

Now for the variable **Instruction_end** :

$$First^1(\varepsilon Follow^1(Instruction_end)) = \{end\}$$

$$First^1(... < InstList > Follow^1(Instruction_end)) = \{...\}$$

For the **Instruction** variable :

$$First^1(< Assign > Follow^1(< Instruction >)) = \{[VARIABLE]\}$$

$$First^1(< if > Follow^1(< Instruction >)) = \{if\}$$

$$First^1(< while > Follow^1(< Instruction >)) = \{while\}$$

$$First^1(< for > Follow^1(< Instruction >)) = \{for\}$$

$$First^1(< print > Follow^1(< Instruction >)) = \{print\}$$

$$First^1(< read > Follow^1(< Instruction >)) = \{read\}$$

$$First^1(begin < InstList > end Follow^1(< Instruction >)) = \{begin\}$$

For the **ExprArith'** variable :

$$First^1(< AddOp > < MultExpr > < ExprArith' > Follow^1(ExprArith')) = \{+, -\}$$

$$First^1(\varepsilon Follow^1(ExprArith')) = \{..., else, end,), =, <, and, or, then, do\}$$

$$First^1(< MultExpr > Follow^1(ExprArith')) = \{(, -, [VARIABLE], [NUMBER]\}$$

The minus sign are not the same.

For the **MultExpr'** variable :

$$First^1(< MultOp > < Term > < MultExpr' > Follow^1(MultExpr')) = \{*, /\}$$

$$First^1(\varepsilon Follow^1(MultExpr')) = \{..., else, end,), =, <, and, or, then, +, -, do\}$$

For the **Term** variable :

$$First^1((< ExprArith >) Follow^1(Term)) = \{(\}$$

$$First^1(- < ExprArith > Follow^1(Term)) = \{-\}$$

$$First^1([VARIABLE] Follow^1(Term)) = \{[VARIABLE]\}$$

$$First^1([NUMBER] Follow^1(Term)) = \{[NUMBER]\}$$

For the **AddOp** and **MultOp**, we don't need to check the definition because the terminals are different.

For the **If-tail** variable :

$$First^1(\varepsilon Follow^1(If - tail)) = \{..., else, end\}$$

$$First^1(< Instruction > Follow^1(If-tail)) = \{[VARIABLE], if, while, print, read, begin\}$$

For the **OrCond** variable :

$$First^1(or < AndCond > < OrCond > Follow^1(OrCond)) = \{or\}$$

$$First^1(\varepsilon Follow^1(OrCond)) = \{then, \}, do\}$$

For the **AndCond'** variable :

$$First^1(and < EndCondition > < AndCond' > Follow^1(AndCond')) = \{and\}$$

$$First^1(\varepsilon Follow^1(AndCond')) = \{or, then, \}$$

For the **EndCondition** variable :

$$First^1(\{ < Cond > \} Follow^1(EndCondition)) = \{\{\}$$

$$First^1(< SimpleCond > Follow^1(EndCondition)) = \{(, -, [VARIABLE], [Number]\}$$

This proves that the grammar is strong LL(1). In conclusion, the grammar is LL(1), because all strong LL(1) grammars are LL(1) grammars.

2.3 Action Table

The action table completely describes the behaviour of the parser. Each cell indicates which rule should be used depending on the line (potential tops of stack) and rows (potential look-aheads). The action table of the grammar in Figure 3 is shown

in Figure 4.

To build the action table we use the First¹ and Follow¹. If a cell is empty, it means that this case is not possible, an error is raised.

Variable	begin	end	...	VARNAME	NUMBER	:=	if	then	else	while	do	print	read	+	-	*	/	()	and	or	[]	=	<
Program	1																								
Code	2	3		2			2			2		2	2												
Instlist	4			4			4			4		4	4												
Instruction_end		5	6						5																
Instruction	12			7			8			9		10	11												
Assign				13																					
ExprArith				14	14										14			14							
ExprArith'		16	16				16				16			15	15				16	16	16		16	16	16
MultiExpr				17	17													17							
MultiExpr'		19	19				19				19			19	19	18	18		19	19	19		19	19	19
Term				22	23											21		20							
AddOp														24	25										
MultiOp																26	27								
If							28																		
If-tail	30	29	29	30			30		29	30		30	30												
Cond				31	31										31			31				31			
OrCond							33				33									32		33			
AndCond				34	34										34			34				34			
AndCond'							36				36								35	36		36			
EndCondition				38	38										38			38				37			
SimpleCondition				39	39										39			39							
Comp																								40	41
While										42															
Print												43													
Read													44												

FIGURE 4 : Action table

2.4 Implementation of the parser

To create our parser, we needed to incorporate our grammar and action table into our program. For this purpose, we created three classes : **ActionTableReader**, **GrammarAnalyzer**, and **State**.

The **State** class is essentially an enumeration of all possible variables, referred to as states.

The purpose of the **ActionTableReader** class is to create a table in which it is possible to retrieve the number of a rule based on a state and a lexical unit.

The **GrammarAnalyzer** class serves a similar purpose. It creates a table in which one can retrieve all the right-hand side elements of a rule based on a rule number and a state.

The arrays in the **ActionTableReader** and **GrammarAnalyzer** classes are created from the corresponding CSV files located in the 'doc/grammar_doc' directory.

Finally, we created the **Parser** class, in which we implemented a recursive descent parser. This parser is capable of generating the leftmost derivation as well as the parse tree of the input.

The parser is initialized with a starting state <Program> and the first symbol of the input is read thanks to the **LexicalAnalyzer**.

The parse function serves as the core of the parser implementation. Given a specific parsing state, it realize the recursive parsing process. The function begins by updating the current state and retrieving the corresponding rule number from the action table using the **ActionTableReader**. The rule number is recorded in the *usedrules* list for reference.

To handle potential errors, the function checks if a rule is found for the current state and symbol. If no rule is found, it means that we are in a grammatically

incorrect situation. An error message is then displayed, and the program terminates.

Subsequently, the function retrieves the elements produced by the identified rule from the grammar using the **GrammarAnalyzer**. These elements represent the components of the rule, including states and lexical units. The function then constructs a parse tree, initializing a node with the current state.

The parsing process iterates through each element of the rule. If the element is a state, the function recursively calls itself to parse that state, generating a subtree. This subtree will be added to the children of the tree created above. If the element is a lexical unit, it is compared to the current symbol. If a match occurs, a corresponding node is created in the parse tree, and the parser advances to the next symbol. ε elements, representing empty productions, are appropriately handled.

In case of a symbol mismatch, the function reports an error and terminates the program.

Finally, the function returns the constructed parse tree, representing the hierarchical structure of the input based on the applied grammar rules.

With the Euclid.pmp file we get the following list of rules and tree :

```
1 2 4 11 44 6 4 11 44 6 4 9 42 31 34 38 39 14 17 23 19 16 41 14 17 22 19 16 36
33 12 4 7 13 14 17 22 19 16 6 4 9 42 31 34 38 39 14 17 22 19 16 41 14 17 22 19 15
24 17 23 19 16 36 33 7 13 14 17 22 19 15 25 17 22 19 16 6 4 7 13 14 17 22 19 16 6 4
7 13 14 17 22 19 16 5 6 4 10 43 5
```

The parse tree is represented at the Figure 5.

