2023-2024

INFO-F403 - Introduction to Language Theory and Compiling

PASCALMAISPRESQUE
Project – Part 1

Sacha DUYNSLAEGER

Yen Nhi NGUYEN

Gilles GEERAERTS

Léonard BRICE

Mathieu SASSOLAS

October 2023

1 PASCALMAISPRESQUE - Project Part 1

1.1 Introduction

A compiler for PASCALMAISPRESQUE is what this project aims to design and write. This simple imperative language's grammar has reserved keywords and lexical units.

This first part of the project is to produce the lexical analyzer of the compiler with JFlex.

1.2 Code Execution

The lexical analyzer is run by either

1. running the makefile at the root of the project (if we want to run the eucid.pmp) :

   1.1. make

   1.2. java -jar dist/part1.jar test/euclid.pmp

2. or by manually write in a terminal (if we want to run the eucid.pmp) :

   2.1. jflex src/LexicalAnalyzer.flex

   2.2. javac -d more -cp src/ src/Main.java

   2.3. jar cfe dist/part1.jar Main -C more .

   2.4. java -jar dist/part1.jar test/euclid.pmp

1.3 Lexical Analyzer

3 exclusive states have been defined to distinguish short comments and long comments from the main text. Exclusive because you cannot be in the main text and in a comment at the same time, and the comments were not nested.

While we are in the initial state, the scanner detects regular expressions and whenever there is a matching, the associated Java code snippet is executed.

The change of state to the short or long comment state occurs when the scanner detects ** or ” in the initial state. At the end of the line, the short comment state is modified and we return to the initial state. On the contrary, we remain in the long comment state as long as no ” is detected.

At the end of reading the file, if we are still in the long comment state, it is because this long comment has not been closed and an error message is then printed before the listed lexical units encountered and the listed variables.

1.3.1 Regular Expressions

[A-Z] and [a-z] are regular expressions for detecting an uppercase and lowercase letter respectively (called AlphaUpperCase and AlphaLowerCase in the code). To get either, the regular expression is {AlphaUpperCase}|{AlphaLowerCase} (Alpha in code).

Similarly, [0-9] is a regular expression for detecting a number (called Numeric in the code). With this, an integer has a regular expression (([1-9][0-9]*)|0), a decimal number \.[0-9]* and an exponent [eE]{Integer}.

After having defined the simplest expressions, a character which is either a letter or a number will have the expression {Alpha}|{Numeric} and we can finally define an identifier as {Alpha}{AlphaNumeric}*, the star allowing to specify that we can have zero, one or more occurrences of the previous element (here an alphanumeric character).

Other regular expressions are $\backslash r|\backslash n|\backslash r\backslash n|[\backslash t\backslash f]$ for detecting different types of space (WhiteSpace in code) and "$\backslash r$"?"$\backslash n$" for end of line detection.

1.3.2 Matched lexical units and Identifiers

In the initial state, each time the scanner detects tokens or regular expressions, the associated Java code snippet is executed. To produce the desired output, each token recognized in the initial state will be tested to see if it has already been encountered and if not, it will be added to a list of all the tokens encountered with their associated lexical unit. A similar process was used to process the variables and an additional function is used to sort the variables in lexicographical order before printing the output (the uppercase letters first in alphabetical order, then the lowercase letters in alphabetical order).

1.3.3 Test Files

Aside from the Euclid.pmp file provided at the launch of the project, other files were created to test different particular cases *(cfr section 2)* :

— A long comment has not been closed. *(cfr readAndComment.pmp)*
  "The comment with ” is not closed." is then printed before the listed lexical units encountered and the listed variables.

— Unlisted symbols are detected (i.e. @). *(cfr calculusTestAndUnknownSymbol.pmp)*
  "@ is a detected unknown symbol." is then printed before the listed lexical units encountered and the listed variables.
  NB : The quotation mark is also an unknown symbol. *(cfr readAndComment.pmp)*

— Variables do not come first in alphabetical order. *(cfr orderOfOutput.pmp)*
  The variables are in lexicographical order in the output (the uppercase letters first in alphabetical order, then the lowercase letters in alphabetical order).

1.3.4 Nested Comments

On the first hand, if the beginning and the ending elements are different, a possibility would be to have a counter. For example, a beginning element could be associated with +1, an ending element with -1, the counter cannot be negative and when the counter equals zero, then we are out of the comments. In this case, a naive solution would be to create a new state for every nested comment, but the number of nested comments cannot be predicted beforehand. Another solution would be to associate a counter to the comment state.

On the other hand, if the elements of the beginning and of the end are the same one, the problem that arises with the nested comments is that we need to count the total number of elements that delimit the comments in order to know if we are still in or outside of the nested comments. And even so, we do not know with certainty if the first symbol is associated with the second or the fourth one, or the sixth one, etc.

2 Test files

2.1 euclid.pmp

```
'' Euclid's algorithm ''
begin
  read(a)...
  read(b)...
  while 0 < b do
    begin
      c := b...
      while b < a+1 do      ** computation of modulo
        a := a-b...
      b := a...
      a := c
    end...
  print(a)
end
```

This file produces the output :

```
token: begin      lexical unit: BEG
token: read       lexical unit: READ
token: (          lexical unit: LPAREN
token: a          lexical unit: VARNAME
token: )          lexical unit: RPAREN
```

```
token: ...      lexical unit: DOTS
token: read     lexical unit: READ
token: (        lexical unit: LPAREN
token: b        lexical unit: VARNAME
token: )        lexical unit: RPAREN
token: ...      lexical unit: DOTS
token: while    lexical unit: WHILE
token: 0        lexical unit: NUMBER
token: <        lexical unit: SMALLER
token: b        lexical unit: VARNAME
token: do       lexical unit: DO
token: begin    lexical unit: BEG
token: c        lexical unit: VARNAME
token: :=       lexical unit: ASSIGN
token: b        lexical unit: VARNAME
token: ...      lexical unit: DOTS
token: while    lexical unit: WHILE
token: b        lexical unit: VARNAME
token: <        lexical unit: SMALLER
token: a        lexical unit: VARNAME
token: +        lexical unit: PLUS
token: 1        lexical unit: NUMBER
token: do       lexical unit: DO
token: a        lexical unit: VARNAME
token: :=       lexical unit: ASSIGN
token: a        lexical unit: VARNAME
token: -        lexical unit: MINUS
token: b        lexical unit: VARNAME
token: ...      lexical unit: DOTS
token: b        lexical unit: VARNAME
token: :=       lexical unit: ASSIGN
token: a        lexical unit: VARNAME
token: ...      lexical unit: DOTS
```

```
token: a        lexical unit: VARNAME
token: :=       lexical unit: ASSIGN
token: c        lexical unit: VARNAME
token: end      lexical unit: END
token: ...      lexical unit: DOTS
token: print    lexical unit: PRINT
token: (        lexical unit: LPAREN
token: a        lexical unit: VARNAME
token: )        lexical unit: RPAREN
token: end      lexical unit: END



Variables
a 6
b 7
c 10
```

2.2 calculusTestAndUnknownSymbol.pmp

```
1-2
a-(-2)

1*3
1*a
a*b

1/2
1/a
a/b

@
```

This file produces the output :

```
@ is a detected unknown symbol.
```

```
token: 1          lexical unit: NUMBER
token: -          lexical unit: MINUS
token: 2          lexical unit: NUMBER
token: a          lexical unit: VARNAME
token: -          lexical unit: MINUS
token: (          lexical unit: LPAREN
token: -          lexical unit: MINUS
token: 2          lexical unit: NUMBER
token: )          lexical unit: RPAREN
token: 1          lexical unit: NUMBER
token: *          lexical unit: TIMES
token: 3          lexical unit: NUMBER
token: 1          lexical unit: NUMBER
token: *          lexical unit: TIMES
token: a          lexical unit: VARNAME
token: a          lexical unit: VARNAME
token: *          lexical unit: TIMES
token: b          lexical unit: VARNAME
token: 1          lexical unit: NUMBER
token: /          lexical unit: DIVIDE
token: 2          lexical unit: NUMBER
token: 1          lexical unit: NUMBER
token: /          lexical unit: DIVIDE
token: a          lexical unit: VARNAME
token: a          lexical unit: VARNAME
token: /          lexical unit: DIVIDE
token: b          lexical unit: VARNAME


Variables
a 2
b 6
```

2.3 readAndComment.pmp

```
** Short comment read(b)
**read(c)
read(a)
"Not a comment"

''Long comment should not be analyzed by the the lexer''

'' Multiline long comment
should not be analyzed by the lexer ''

'' Comment not closed
```

This file produces the output :

```
" is a detected unknown symbol.
" is a detected unknown symbol.
The comment with '' is not closed.
token: read      lexical unit: READ
token: (         lexical unit: LPAREN
token: a         lexical unit: VARNAME
token: )         lexical unit: RPAREN
token: Not       lexical unit: VARNAME
token: a         lexical unit: VARNAME
token: comment   lexical unit: VARNAME


Variables
Not 4
a 3
comment 4
```

2.4 orderOfOutput.pmp

```
BUppercase
AUppercase
CUppercase
aLowercase
cLowercase
bLowercase
```

This file produces the output :

```
token: BUppercase        lexical unit: VARNAME
token: AUppercase        lexical unit: VARNAME
token: CUppercase        lexical unit: VARNAME
token: aLowercase        lexical unit: VARNAME
token: cLowercase        lexical unit: VARNAME
token: bLowercase        lexical unit: VARNAME


Variables
AUppercase 2
BUppercase 1
CUppercase 3
aLowercase 4
bLowercase 6
cLowercase 5
```