UNIVERSITÉ LIBRE DE BRUXELLES
Faculty of Sciences
Computer Science Department

INFO-F-403

Introduction to language theory and compiling
Project - Part1 : Report

Anthony Caccia
Raphaël Vander Marcken

Academic Year 2016 - 2017

1 Added Methods

1.1 Before Lexing

```
private TreeMap<String, Integer> identifiers = new TreeMap<String, Integer>();

/**
* Add an identifier and it's declaration location to the list of identifiers
*/
private void addIdentifier(){
  if (!identifiers.containsKey(yytext())) {
    identifiers.put(yytext(), yyline+1);
  }
}

/**
* Helper function too create new symbol with position
* @param type of LexicalUnit inferred
* @return created symbol
*/
private Symbol symbolBuilder(LexicalUnit unit){
    return new Symbol(unit, yyline, yycolumn, yytext());
}

/**
* Helper function too create new symbol with position
* @param type of LexicalUnit inferred
* @param value associated
* @return created symbol
*/
private Symbol symbolBuilder(LexicalUnit unit, Object value){
    return new Symbol(unit, yyline, yycolumn, value);
}

private void log(Symbol s){
  System.out.println("line: "+ (yyline+1) + " " + s);
```

A method *symbolBuilder* is used to facilitate the creation of *Symbol*. This method is overloaded so it can be called with a value to use or to use directly the parsed text.

A method *log* is used to print and debug the lexer.

To handle identifiers, a TreeMap is declared which takes the identifier (String) as key and the line of declaration (Integer) as value. Method *addIdentifier* add the identifiers recognized in the PROGRAM state into the map if they are not already inside.

1.2 After EOF

```
System.out.println("Identifiers");
for (Map.Entry<String, Integer> i: identifiers.entrySet())
    System.out.println(i.getKey() + " " + i.getValue());
```

After *End Of File*, the word "Identifiers" is printed, followed by all the identifiers inside the map and their values. The fact that this map is a TreeMap with String keys ensure these identifiers are traversed in lexicographic order.

2 Regular Expressions

Jflex option "%ignorecase" ensure all patterns are case insensitive.

2.1 Simple patterns

```
letter = [a-zA-Z]
digit = [0-9]
identifier = {letter}({letter}|{digit})*
number = {digit}{digit}*
end_of_line = \r?\n
whitespace = [ \t]
```

letter a single latin letter, between *a* and *z*, in lower and capital case.

digit a single digit, between *0* and *9*

identifier a fortran identifier: a letter followed by any number of letters or digits

number a sequence of digit (minimum 1)

end_of_line end of line character, preceded by an optional carriage return

whitespace a space or tab character

2.2 YYINITIAL

Initial state, before a program definition.

```
^[cCdD\*!]   {yybegin(PRECOMMENT);}
program   {log(symbolBuilder(LexicalUnit.PROGRAM));}
{identifier}  {log(symbolBuilder(LexicalUnit.VARNAME)); yybegin(PROGRAM);}
{end_of_line}  {}
.     {}
```

1. handle comment: if line begins by one of the characters inside brackets, enter in precomment state
2. the word program
3. an identifier (2.1)
4. end of line character (2.1)
5. dot handle any character not yet recognized by preceding rules

2.3 PROGRAM

PROGRAM state handles all that's declared between the keywords *PROGRAM* and *END*.

```
^[cCdD\*!] {yybegin(COMMENT);}
"!"  {log(symbolBuilder(LexicalUnit.ENDLINE, " ")); yybegin(COMMENT);}
^{whitespace}*{end_of_line} {}
{end_of_line} {log(symbolBuilder(LexicalUnit.ENDLINE, " "));}
end {log(symbolBuilder(LexicalUnit.END)); yybegin(YYINITIAL);}
integer {log(symbolBuilder(LexicalUnit.INTEGER));}
if {log(symbolBuilder(LexicalUnit.IF));}
then {log(symbolBuilder(LexicalUnit.THEN));}
endif {log(symbolBuilder(LexicalUnit.ENDIF));}
else {log(symbolBuilder(LexicalUnit.ELSE));}
do {log(symbolBuilder(LexicalUnit.DO));}
enddo {log(symbolBuilder(LexicalUnit.ENDDO));}
read\* {log(symbolBuilder(LexicalUnit.READ));}
print\* {log(symbolBuilder(LexicalUnit.PRINT));}
\.not\. {log(symbolBuilder(LexicalUnit.NOT));}
\.an\. {log(symbolBuilder(LexicalUnit.AND));}
\.or\. {log(symbolBuilder(LexicalUnit.OR));}
\.eq\. {log(symbolBuilder(LexicalUnit.EQUAL_COMPARE));}
\.ge\. {log(symbolBuilder(LexicalUnit.GREATER_EQUAL));}
\.gt\. {log(symbolBuilder(LexicalUnit.GREATER));}
```

```
21 \.le\. {log(symbolBuilder(LexicalUnit.SMALLER_EQUAL));}
22 \.lt\. {log(symbolBuilder(LexicalUnit.SMALLER));}
23 \.ne\. {log(symbolBuilder(LexicalUnit.DIFFERENT));}
24 "," {log(symbolBuilder(LexicalUnit.COMMA));}
25 "=" {log(symbolBuilder(LexicalUnit.EQUAL));}
26 "(" {log(symbolBuilder(LexicalUnit.LEFT_PARENTHESIS));}
27 ")" {log(symbolBuilder(LexicalUnit.RIGHT_PARENTHESIS));}
28 "-" {log(symbolBuilder(LexicalUnit.MINUS));}
29 "+" {log(symbolBuilder(LexicalUnit.PLUS));}
30 "*" {log(symbolBuilder(LexicalUnit.TIMES));}
31 "/" {log(symbolBuilder(LexicalUnit.DIVIDE));}
32 {number} {log(symbolBuilder(LexicalUnit.NUMBER, new Integer(yytext())));}
33 {identifier} {addIdentifier(); log(symbolBuilder(LexicalUnit.VARNAME));}
34 {whitespace} {}
35 . {System.out.println("TOKEN NOT RECOGNIZED: " + yytext());}
```

lines 1-3 handle comments: line 1 is for a comment line, line 2 for a comment on the rest of a line and line 3 handle blank lines

line 4 end of line character (2.1)

lines 5-31 language keywords and symbols (for lines 13-21, the "
" is just an escape character; for lines 24-31, quotes are used to indicate that what is inside should match litterally.

line 32 a number (2.1)

line 33 an identifier (2.1): launch also a procedure to save them

line 34 a whitespace (2.1)

line 35 dot handle any character not yet recognized by preceding rules

2.4 COMMENT - PRECOMMENT

PRECOMMENT state handles all comments that occur in YYINITIAL state.

```
1 {end_of_line} {yybegin(YYINITIAL);}
2 . {}
```

line 1 end of line character (2.1) signals the end of the comment and thus switches to YYINITIAL state.

line 2 any character part of the comment is ignored

COMMENT state handles all comments that occur in PROGRAM state.
Its description is the same as PRECOMMENT state except that the end of line character switches to PROGRAM state instead of YYINITIAL state.