Bilkent University

Department of Computer Engineering

CS 315

Programming Language

Lexical Analyzer for a Set Programming Language

Language name: *Cascabel*

28.02.2018

Section 1/ Group 1:

- Kazım Ayberk Tecimer: 21502531
- Fuad Aghazada: 21503691
- Eray Şahin: 21502758

Instructor:

- Halil Altay Güvenir

Table of Contents

1. **BNF**

1. <main_program> ::= execute begin <program> end
2. <program> :: = <statements> | <statement>
3. <statements> :: = <statement> | <statement><statements>
4. <statement> :: = <matched> | <unmatched>
5. <matched> :: = if (<expression>) begin <matched> end else begin <matched> end | <loop> | <single_statement>
6. <unmatched> :: = if (<expression>) begin <statements> end
   | if (<expression>) begin <matched> end else begin <unmatched> end
7. <single_statement> :: = <input_statement> | <output_statement> | <assignment_operation> | <declaration_statement> | <return_statement> | <set_statement>
8. <input_statement> :: = read <expression>
9. <output_statement> :: = print(<expression>) | println(<expression>)
10. <assignment_operation> :: = <variable_identifier> = <expression>
11. <declaration_statement> :: = <type> <variable_identifier_list> | <type> <assignment_operation>
12. <type> :: = int | double | float | long | string
13. <return_statement> :: = return <expression>
14. <set_statement> :: = <set_declaration> | <set_assignment>
15. <set_declaration> :: = @ <variable_identifier_list>
16. <set_assignment> :: = @ <variable_identifier> = { <set> }
17. <expression> :: = <term> <mid_prec_op> <expression> | <term>

18. <term> :: = <variable_identifier> <high_prec_op> <term> | <variable_identifier>
19. <high_prec_op> :: = * | /
20. <mid_prec_op> :: = + | -
21. <variable_identifier_list> :: = <variable_identifier> | <variable_identifier>, <variable_identifier_list>
22. <variable_identifier> :: = <non_digit> | <non_digit> <characters>
23. <non_digit> :: = <character> | _
24. <characters> ::= <character> | <character> <characters>
25. <character> :: = <lowercase_letter> | <uppercase_letter>| <special>
26. <special> :: = + | - | * | / | \ | ^ | ~ | : | . | ? | # | $ | &
27. <function_def> :: = <void_func_def> | <non_void_func_def>
28. <function_identifier> :: = <varible_identifier>
29. <void_func_def> :: = void <function_identifier> ([<parameters>]) begin <statements> end
30. <non_void_func_def> :: = <type> <function_identifier> ([<parameters>]) begin <statements> return <expression> end
31. <parameters> :: = <type> <identifier> | <type> <identifier> , <parameters>
32. <function_call> :: = <function_identifier> ([<variable_identifier_list>])
33. <set> ::= <set_element> | <subset>
34. <subset> :: = <set_element> | <set_element>, <set>
35. <set_element> :: = <characters> | <numbers>
36. <lowercase_letter> :: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

37. <uppercase_letter> :: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

38. <integer> :: = <abs_integer> | <sign> <abs_integer>

39. <abs_integer> :: = <digit> | <digit> <abs_integer>

40. <sign> :: = + | -

41. <digit> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

42. <set_operator> :: = UNION | INTERSECT | SET_DIFFERENCE | CARTESIAN_PRODUCT

43. <set_operation> :: = <set> <set_operator> <set> | <set> <set_operator> (<set_operation>) | (<set_operation>) <set_operator> <set>

44. <set_relation> :: = <set> <set_relations> <set>

45. <set_relations> ::= SUB_RELATION | SUPER_RELATION | ELEMENT_OF

46. <numbers> ::= <integer> | <double> | <float> | <long>

47. <loop>:: = <while> | <for>

48. <for> :: = for <variable_identifier> in range (<digit>,<digit>) begin <statements> end

49. <while> :: = while (<expression>) begin <statements> end

50. <comparison_op> :: = < | > | == | >= | <= | !=

51. <comment> :: = \# <characters>

52. <string> ::= “<characters>” | “<integer>”

53. <print> :: = print (<string> | <integer> | <expression>);

54. <println> :: = println(<string> | <integer> | <expression>);

2. Description of non-terminals

- **main_program:** The scope that where the *program* is being executed. This is corresponding to main function in C programming languages.
- <**program**> : contains the *statements* that are needed to be executed to perform some tasks. Defines the whole *program* which consist of *statements*.
- <**statements**> : list of *statements* with different types, similar to *program*. List of statements should be written between the *begin* and *end* keywords which makes the program readable.

  a)
  ```
  if (a == 7)                                → this is a statement
          begin
                  a = a + 7                  → this is a statement
          end
  print(a)                                   → this is a statement
  ```

- <**statement**> : a line/block of code, which could be either *matched* or *unmatched*, like in *if/else statements*. This helps to solve the “dangling else” problem.

  a)
  ```
  if (a == 7)
          begin
                  a = a + 7
          end
  ```

- <**matched**> : is a type of the *statement*, which could contain either another *matched* statement or a *single statement*.

  a)
  ```
  if (a == 7)
  ```

```
        begin
            a = a + 7
        end
    else
        begin
            a = a - 7
        end
```

- <**unmatched**> : is a type of the *statement*, which could contain a *matched* statement in the first block of itself and *unmatched* in the second block of itself or just a list of other statements without having a second block.

  a) if (a > 7)

```
        begin
            If (a  == 9)
                begin
                    Some statements go here
                end
        end
```

- <**single_statement**> : can be any of the following: *input* / *output* statement, *declaration*, *return* statement or an *assignment*. Examples are provided in respective sections of non-terminals.
- <**input_statement**> : is an input statement, which takes the input from the user using *read* keyword. This type of input statement makes *Cascabel* both readable and writable for the user.

  a) read x

```
>>> 5
```

x gets the value of 5

- <**output_statement**> : is an output statement, which displays the statement of the user on the console using *print* function. It looks like Python in terms of this readability and writability.

  a) x = x + 5

     print(x)

- <**assignment_operation**> : is an operation, which has a role of assigning values (more specifically, expressions) to the *variable identifiers*.

  a) varIdent = 12

  b) identifier2 = 3 * 7 + 21

  c) ident3 = ident4 → identifier assigned to the value of another identifier

- <**declaration_statement**> : is the operation where one or more new variables are declared. This type of statement also covers both the declaration and initialization of a variable.

  a) int id

  b) int studentID, luckyNumber, age

  c) int num = 11

  d) string courseName, instructorName

  e) string courseCode = “CS 319”

- <**type**> : it is the collection of keywords of *int*, *double, float, long, and string*. The type is defined with or, which leads an identifier to belong to only one type.

- <**return_statement**> : is a type of statement, which has a role of returning the value which is defined as a function type. In most of the languages that we have learned until today, the *return statement* is used like that because it makes the language considerably writable and readable.
    a) A function with a return type of int should have a return statement as follows:

```
int foo()
          begin
                    Some statements go here
                    return 3
          end
```

Note that, at the moment, the language is not able to detect the *return* statements not occurring at the end of the body of a function. In other words, the *return* statement must be the last statement before *end*.

- <**set_statement**> : is a statement which could be either a declaration statement (*set_declaration*) or an assignment statement for sets (*set_assignment*).
- <**set_declaration**> : is a statement in which the user can declare the set in the language by putting @ sign in front of the variable identifiers. Creating the set by putting only ‘@’ in front of any variable denotes the simplicity of *Cascabel.*
    a) @A, B, C → is a declaration statement where A, B and C are sets
    b) @mySet → a declaration statement with one set

- <**set_assignment**> : is a statement in which the user can assign value to the sets using curly brackets ({}). Here, curly brackets and commas make the language more readable owing to the similarity of notation of sets in mathematics.
    a) @A = {2, 5, “element”, “6f4\”, @B}
- <**expression**> : can address either a single term or a middle-precedence operation where the first operand is a *term* and the second operation is also an *expression*. This way, middle-precedence operations (addition or subtraction) are handled after resolving the operations of higher precedence, namely the multiplication and division.
    a) varIdent → *varIdent* is a *variable identifier* which implies that it is a *term* with the further implication that it is also an *expression*
    b) a * b + c → Since * is a *high-precedence operator*, *a * b* becomes a *term*. Further, + is a *middle-precedence operator* and c is a *term* as well. Therefore, the precedence of multiplication over addition is handled.

Note that the language is incapable of processing expressions involving parentheses currently. That is, instead of writing *middle-precedence operations* within parentheses to give them higher precedence, the user should first calculate and store the result of these *middle-precedence operations* in a variable and then use that variable in the next line of calculation.

- <**term**> : is a part of an expression in which the operations with a *higher precedence* (*, /) are solved.
    a) 3 * 5 + 2 → in this expression ‘3 * 5’ is the term part, which has a *higher precedence operator* (*)
    b) 98 / 14
- <**high_prec_op**> : is a list of operators, which include multiplication (*) and division (/)
    a) 33 * 1223
    b) 34532 / 23
- <**mid_prec_op**> : is a list of operators, which include addition (+) and subtraction (-)
    a) 123 + 23
    b) 98546 - 123
- <**variable_identifier_list**> : is a list of *variable identifiers* which may contain at least one or more identifier(s), defined recursively.
    a) int a, b, c, d
    b) float num1, num2, num3, num4
- <**variable_identifier**> : is a block of *characters*/symbols which is started by a *non digit* and continues with the *characters*.
    a) abc123
    b) $var934
    c) _myVarible98
- <**non_digit**> : is any *lowercase* or *uppercase* character, a *special* character (see below) or an underscore.

- <**characters**> : is a block of alphabetical letters (could be one or more)

  a) abcde

  b) Mehmet

- <**character**> : consists of all *lowercase* and *uppercase* characters as well as the following set of *special* characters.
- <**special**> : is a list of symbols containing +, - , * , / , \ , ^ , ~ , : , . , ? , # , $, &
- <**function_def**> : can be the definition of either a *void* or *non-void* function.
- <**function_identifier**> : is an identifier for the function, which is the same with the identifiers for the variables.

  void foo (int a, int b) → foo is a function identifier

- <**void_func_def**> : is a type of function with either *parameters* or without *parameters*, which is used when function has no value to *return*.

  a) void foo(int a)

  ```
          begin
                  statements go here
          end
  ```

  b) void foo()

  ```
          begin
                  statements go here
          end
  ```

- <**non_void_func_def**> : covers function definitions with *return* types.

  a) int func(double c)

  ```
          begin
                  statements go here
  ```

```
        return expression
    end
```

b) int func()

```
    begin
        statements go here
        return expression
    end
```

- <**parameters**> : are the parts of the functions, which have a role in the passing the data from the outside of the function to the block of function.

```
void foo (int a, double b, float c)
    begin
        statements go here
    end
```

- <**function_call**> : covers function calls as the name suggests.
    a) func1()
    b) func2(varIdent1)
    c) func3(varIdesnt1, varIdent2)
- <**set**> : could be a *set element* or a *subset*. Different set elements are separated by commas in between them.
    a) 91
    b) 1, W, 9, s
    c) $_cs*:pl, .27, -41

    Note that when ***defining*** a set, variable identifiers’ definition criteria is applied. In other words, a set cannot have a name starting with a digit.

a) @set1 ={ 45, $money$, -99.1 } → *set1* is a valid identifier

b) @23xy = { a, b, c } → not allowed (invalid set name)

- <**subset**> : is a collection of one or more *set elements* and, thus, is a *set* as well.

  a) ?q , -1 , 3.98 , a_b_c

- <**set_element**> : can be a single *character* or more *characters* or any sort of *number* (*integer*, *double*, *float*, *long*).

  a) R

  b) element

  c) 3.51

  d) .2

  e) -81.2

  f) 482

  g) c-+_ba

- <**lowercase_letter>** : this non terminal is the collection of all *lower case* letters. It starts with a, ends with z. *Lowercase* letter is defined with or which leads to one *lowercase* letter consists from only one of them.

- <**uppercase_letter**> : this non terminal is the collection of all *uppercase* letters. It starts with A, ends with Z. *Uppercase* letter is defined with or which leads to one *uppercase* letter consists from only one of them.

- <**integer**> : can be an integer with a sign (plus or minus) in front of it or just the number itself without any signs.

  a) +64

  b) -32

c) 128

- <**abs_integer**> : is an intermediate form to designate the integer form better. More specifically, a number with a single digit or more is considered to be in this form.
    a) 4
    b) 17
    c) 6125
- <**sign**> : can either be plus (+) or minus (-).
    a) +5
    b) -5
- <**digit**> : covers all the *digits* in the decimal system: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- <**set_operator**> : is defined using one of the following keywords: UNION, INTERSECTION, SET_DIFFERENCE, CARTESIAN_PRODUCT.
- <**set_operation**> : can be any operation defined on two different sets. Parentheses are used to define associativity.
    a) @setA *UNION* @setB
    b) @setName1 *INTERSECTION* @setName2
    c) (@abc123 *UNION* @def456) *SET_DIFFERENCE* @s1
    d) @set1 *CARTESIAN_PRODUCT* (@S_1 *SET_DIFFERENCE* @S_2)
- <**set_relation**> : is defined as a pattern of *set relation* so that there can be a relation between the *sets*.
    a) @abc123 *RELATION_NAME* @def456
- <**set_relations**> : is type of the relations between the *sets* or the *subsets*: SUB_RELATION, SUPER_RELATION.

a) @abc123 SUB_RELATION @def456

b) @abc123 SUPER_RELATION @def456

- <**numbers**>**:** numbers is one of types of the *integer*, *double*, *float*, *long*
- <**loop**>: loop consists from *while* and *for* statements as mentioned below
- <**for**>: Defines the for loop statement which starts with the initially keyword *for* followed by variable identifier and followed by keyword in range and takes 2 digits inside parentheses. This *for* statement is similar to for statement in Python. Note that the simplicity is emphasized here through specifying a “range” rather than a complex expression. This simple approach also helps to make Cascabel more readable.

```
for x in range (0,3)
begin
        x = x + 3
end
```

- <**while**>: Defines the *while* loop statement which starts with the initially keyword *while* followed by parentheses which takes **expression** in it. It continues with the keyword *begin* inside this it takes statement and finishes with *end.* Again having the similar syntax with the other imperative languages, Cascabel has a characteristics of simplicity, readability and writability in declaring the while loop too. However, in terms of orthogonality like the C group languages that we have learned, Cascabel shows the same performance by having two different loop types.

```
while (x <= 3)
begin
```

```
        x = x - 1
    end
```

- <**comparison_op**> : this non-terminal is for to make comparisons between identifiers. Such as, greater, greater and equal, equals, less etc.
- <**comment**> : A line of comment in code that enable users red the explanation of code. Comments make Cascabel more readable language. We are inspired by Assembly.
  a) # *write comment here*
- <**string>** : Description of *string* which consists of any *characters* and numbers except for endline *character* between the quotation marks.
  a) “cs-315”
  b) “mahmut”
  c) “21314”
- <**print**> : is used to display values on the screen. A *print* call may receive a *string*, an integer or an *expression* (whose result is to be printed).
  a) print(“a string of characters”)
  b) print(-16)
  c) print(9 * 7)
- <**println**> : is almost the same as *print* with the exception that the cursor is moved to the *next line* after printing. Therefore, it is called in the same way as *print*.