Bilkent University

Department of Computer Engineering

CS 315

Programming Language

Parser for a Set Programming Language

Language name: *Cascabel*

19.03.2018

Section 1/ Group 1:

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

Instructor:

- Halil Altay Güvenir

Table of Contents

1. **BNF (updated)**

1. <main_program> ::= MAIN BEGINC <new_line> <program> END
2. <program> ::= <statements>
3. <statements> ::= <statement> <new_line>

   | <statement> <new_line> <statements>
4. <statement> ::= <if> | <loop> | <single_statement> | <function_def>
5. <new_line> ::= NEW_LINE | <comment> | NEW_LINE <new_line>
6. <if> ::= IF LEFT_PARANT <expression> RIGHT_PARANT BEGINC

   NEW_LINE <statements> END

   | IF LEFT_PARANT <expression> RIGHT_PARANT BEGINC NEW_LINE

   <statements> END ELSE BEGINC NEW_LINE <statements> END
7. <single_statement> ::= <input_statement>

   | <output_statement>

   | <assignment_operation>

   | <declaration_statement>

   | <set_statement>

   | <array_statement>

   | <function_call>
8. <input_statement> ::= READ <expression>
9. <output_statement> ::= <print>

   | <println>

10. <assignment_operation> ::= <variable_identifier> ASSIGNMENT_OP <expression>
11. <declaration_statement> ::= <type> <variable_identifier_list>
    | <type> <assignment_operation>
12. <type> ::= INTEGER_TYPE | FLOAT_TYPE | STRING_TYPE
13. <return_statement> ::= RETURN <expression> <new_line>
14. <set_statement> ::= <set_declaration>
    | <set_assignment>
    | <set_operation>
    | <set_relation>
15. <set_declaration> ::= SET | SET COMMA <set_declaration>
16. <set_assignment> ::= SET ASSIGNMENT_OP LEFTBRACE <set> RIGHTBRACE
    | SET ASSIGNMENT_OP <set_operation>
17. <array_statement> ::= <array_declaration>
    | <array_assignment>
18. <array_declaration> ::= ARRAY | ARRAY COMMA <array_declaration>
19. <array_assignment> ::= ARRAY ASSIGNMENT_OP LEFT_SQUARE_BRACE <array> RIGHT_SQUARE_BRACE
    | <array_access> ASSIGNMENT_OP <expression>
20. <array_access> ::= ARRAY LEFT_SQUARE_BRACE <integer> RIGHT_SQUARE_BRACE
    | ARRAY LEFT_SQUARE_BRACE <variable_identifier> RIGHT_SQUARE_BRACE

21. <array> ::= <set_element>

    | <set_element> COMMA <array>

    | SET

    | SET COMMA <array>

22. <expression> ::= <term> <mid_prec_op> <expression>

    | <term> <comparison_op> <expression>

    | <term>

23. <term> ::= <variable_identifier> <high_prec_op> <term>

    | <integer> <high_prec_op> <term>

    | <float> <high_prec_op> <term>

    | <variable_identifier>

    | <set_relation>

    | <set_operation>

    | <integer>

    | <float>

    | <string>

    | <array_access>

24. <high_prec_op> ::= MULTIPLY | DIVIDE

25. <mid_prec_op> ::= PLUS | MINUS

26. <variable_identifier_list> ::= <variable_identifier>

    | <variable_identifier> COMMA <variable_identifier_list>

27. <variable_identifier> ::= VAR_IDENTIFIER

28. <function_def> ::= <void_func_def> | <non_void_func_def>

29. <void_func_def> ::= VOID <variable_identifier> LEFT_PARANT RIGHT_PARANT BEGINC NEW_LINE <statements> END
    | VOID <variable_identifier> LEFT_PARANT <parameters> RIGHT_PARANT BEGINC NEW_LINE <statements> END
30. <non_void_func_def> ::= <type> <variable_identifier> LEFT_PARANT RIGHT_PARANT BEGINC NEW_LINE <statements> <return_statement> END
    | <type> <variable_identifier> LEFT_PARANT <parameters> RIGHT_PARANT BEGINC NEW_LINE <statements> <return_statement> END
31. <parameters> ::= <type> <variable_identifier>
    | <type> <variable_identifier> COMMA <parameters>
32. <arguments> ::= <argument> | <argument> COMMA <arguments>
33. <argument> ::= <integer> | <float> | <variable_identifier> | <string>
34. <function_call> ::= <variable_identifier> LEFT_PARANT RIGHT_PARANT
    | <variable_identifier> LEFT_PARANT <arguments> RIGHT_PARANT
35. <set> ::= <set_element>
    | <set_element> COMMA <set>
    | SET
    | SET COMMA <set>
    | empty_set
36. <set_element> ::= <string> | <numbers> | <variable_identifier>
37. <empty_set> ::= EMPTY_SET
38. <integer> ::= INTEGERC

39. <float> ::= FLOAT
40. <numbers> ::= INTEGERC | FLOAT
41. <set_operator> ::= UNION_OP
    | INTERSECTION_OP
    | SET_DIFFERENCE_OP
    | CARTESIAN_PRODUCT_OP
42. <set_operation> ::= SET <set_operator> SET
    | SET <set_operator> LEFT_PARANT <set_operation>
    RIGHT_PARANT
    | LEFT_PARANT <set_operation> RIGHT_PARANT
    <set_operator> SET
43. set_relation ::= SET sub_relation SET
    | SET super_relation SET
    | set_element sub_relation SET
    | set_element ELEMENT_OF SET
44. <sub_relation> ::= SUBSET_RELATION
    | PROPER_SUBSET_RELATION
    | IMPROPER_SUBSET_RELATION
45. <super_relation> ::= SUPERSET_RELATION
    | PROPER_SUPERSET_RELATION
    | IMPROPER_SUPERSET_RELATION
46. <loop> ::= <while> | <for>

47. <for> ::= FOR <variable_identifier> IN RANGE LEFT_PARANT <argument> COMMA <argument> RIGHT_PARANT BEGINC NEW_LINE <statements> END
48. <while> ::= WHILE LEFT_PARANT <expression> RIGHT_PARANT BEGINC NEW_LINE <statements> END
49. <comparison_op> ::= ASSIGNMENT_OP
    | SMALLER
    | GREATER
    | EQUALITY_CHECK
    | GREATER_OR_EQUAL
    | SMALLER_OR_EQUAL
    | NOT_EQUAL
50. <comment> ::= COMMENT <new_line>
51. <string> ::= STRING
52. <print> ::= PRINT LEFT_PARANT <expression> RIGHT_PARANT
53. <println> ::= PRINTLN LEFT_PARANT <expression> RIGHT_PARANT

2. Description of non-terminals (updated)

Note: readability, writability, reliability motivations and constraints and their relation with our language is defined below with the 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.

Note that it is always required to begin with “main beginc” (to indicate the start of execution) and end with “end”. In other words, a *main_program* always starts with “main beginc”, followed by the actual code and lastly the “end” reserved word:

main beginc
code goes here
end

- <**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. A *statement* can be an *if-statement*, *loop*, *single statement* or *function definition*.

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

- <**single_statement**> : can be any of the following: *input* / *output* statement, *declaration*, *return* statement, *set statement, array statement, function call* 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*, *float, 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*), an assignment statement for sets (*set_assignment*), a set operation (*set_operation),* or a set relation *(set_relation)*.
- <**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}

- **<array>** : matches *set element(s)* or *set(s)* separated by commas.

  a) 4, @someSet, 78, 21 → these are matched by array (whole sentence)

  b) @a, @b, "setElement", -3.5

- **<array_statement>** : can be either an *array declaration* or *array assignment*.
- **<array_declaration>** : is either a single *variable identifier* preceded by an asterisk (*) or many of them separated by commas:

  a) *arr

  b) *arr1, *arr2, *arr3

- **<array_access>** : is an access to a specific *array* slot. The cell to access is identified by an *integer* or a *variable identifier*:

  a) *arr[12]

  b) *arr[varIdent]

- **<array_assignment>** : covers two cases. Firstly, it is matched when an *array* gets assigned to values provided within square brackets. Secondly, it is matched when an array cell is assigned to an *expression*:

a) *arr = [6, “abc”, -2.1, a, 56]

b) *arr[index] = 4

c) *arr[anotherIndex] = 3 * 4 + 17

- <**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

They can also be useful in the accessing arrays and set operations.

- <**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
- <**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

- **<arguments>** ::= is a collection of an arguments.
- **<argument>** ::= is part of the function declaration which have a role in the taking the arguments by (integer, float, variable identifier, or string).

  E.g. func(1,2) here both 1 and 2 are arguments
- <**function_call**> : covers function calls as the name suggests.
  a) func1()
  b) func2(varIdent1)
  c) func3(varIdesnt1, varIdent2)
- <**set**> : could be one or more *set_elements* or *empty_set*. Different set elements are separated by commas in between them.
  a) 91
  b) 1, W, 9, s
  c) $_cs*:pl, .27, -41
  d) @0 → denotes empty set

  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)
- <**set_element**> : can be a string, variable identifier or any sort of *number* (*integer*, *float*).
  a) "I need rest"

b) element

c) 3.51

d) .2

e) -81.2

f) 482

g) C-+_ba

- **<empty_set>**: is a set which is empty. It is denoted by @0.
- <**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

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

  d) +1.2

  e) -81.2

  f) 93.1

- <**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
- <**sub_relation**> : is type of the subset* relations between the *sets* or the *subsets*: SUB_RELATION, PROPER_SUB_RELATION, IMPROPER_SUB_RELATION.

    Note*: There are two types of subset relation: proper/improper. A proper subset of A is a subset that is strictly contained in A and so necessarily excludes at least one member of S. An empty set is therefore a proper subset of any nonempty set. In contrast, a subset containing all elements of the given set is called improper subset.

    a) @abc123 SUB_RELATION @def456
    b) @abc123 PROPER_SUB_RELATION @def456
    c) @abc123 IMPROPER_SUB_RELATION @def456
- <**super_relation**> : is type of the superset* relations between the *sets* or the *supersets*: SUPER_RELATION, PROPER_SUPER_RELATION, IMPROPER_SUPER_RELATION.

    Note*: A proper superset, is a superset which is not entire the set. In contrary, if the superset is entire set, it is improper superset.

    d) @abc123 SUPER_RELATION @def456
    e) @abc123 PROPER_SUPER_RELATION @def456
    f) @abc123 IMPROPER_SUPER_RELATION @def456
- <**numbers**>**:** numbers is one of types of the *integer*, *float*

- <**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. More, in the stage of 2nd project new line option added for comments.
    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*.

3. Important Notes About Updates

Changes

- We considered the difference between proper and improper subset\superset.
- We explained for subset superset in sufficient way.
- Support for constant variables are added.
- Support for array types is added.
- Arrays may appear in several operations as a *term* and/or *variable identifier*.
- *Unmatched* and *matched* statements are removed, because our block statements always require begin/end.
- *Non_digit* is removed.
- Two new variables called *arguments* and *argument* are added to identify *integer*, *float*, *variable identifier* and *string* variables (the addition of this new non-terminal proves useful in *function calls*).

4. Known Conflicts-Warnings-Bugs

● There is no conflict or warning given by yacc. All conflicts and warnings are eliminated during development.

● With several different inputs we didn’t encounter with bug. All non working functions, bugs are resolved during development.

5. Some restrictions of *Cascabel*

In this section of report as the designers of the language we would like to show the restrictions that the users may face.

- Some statements which may include blocks (set of statements starting with “beginc” and ending with “end”) require the insertion of at least one new line character after the reserved word “beginc”. Also, “beginc” must be right after the *statement* requiring the block, without any new line interrupting (no new line between the *statement* and “beginc”).

  For instance, when writing a *for* loop, “beginc” should appear right after the *loop* declaration. Also, the first line of the body must be starting after “beginc”:

  a) for x in range (2, upperLimit) beginc → this is valid
     print(“within range”)
     end

  b) for x in range (2, upperLimit)
     beginc → this is invalid (“beginc” not in the same line as *for*)
     print(“within range”)
     end

  c) while (value <= 20) beginc print(“less than or equal”) → this is invalid

end

The main motivation for this application was readability and we are inspired by Python.

- The primary restriction about the language is about the writing comments. In most of the imperative languages the user can write plenty of comments in different sections of the code. However, in Cascabel the user is restricted to put the comments on the line directly after the statement. To exemplify, the following piece of code (a) works properly:

  a) read inputX # getting the input from the user and assign it inputX

  b) And the next example is an erroneous case:

  if (x != 5) begin # beginning of if statement

  print(x)

  end

  This restriction may create some difficulties for the user, so in the next versions of the Cascabel, we will provide our users with much more flexible language.

6. Project folder location

Project folder: https://drive.google.com/open?id=1ExUL01TVEQL4FerI7t711z9pk5Ub47Cs