Language Specification:

**1 .Language Definition**:

  1.1 Alphabet:

  1.1.a. Upper (A-Z) and lower case letters (a-z) of the English alphabet02

       b. Underline character '\_';

       c. Decimal digits (0-9);

  Lexic:

      a.Special symbols, representing:

- operators: +, -, \*, /, >, <, ==, =, >=, <=, !=, %, ++, --

- separators: space, {, }, ;

- reserved words: int, double, const, char, bool, void, struct, if, else, else if, for, while, break, return, read, write, in, continue

- keywords: \_NULL, \_NULL\_STR

\_NULL ::= -4.04

-NULL\_STR ::= “\*\_\*”

      b.identifiers

  -a sequence of letters and digits, such that the first character is a letter; the rule is:

    identifier ::= letter | [{underscore}]letter[{underscore}][{letter}][{underscore}][{digit}]

##e.g.: a, \_a, ab, a\_b, a3

    letter ::= "A" | "B" | ...| "Z"

    digit ::= "0" | "1" |...| "9"

underscore ::= “\_”

      c.constants

1.integer - rule:

      noconst:=+no|-no|no

      no:=digit{no}

2.character

    character:='letter'|'digit'

3.string

      constchar:="string"

      string:=char{string}

      char:=letter|digit

 2.2 Syntax:

The words - predefined tokens are specified between " and ":

a) Sintactical rules:

    program ::= "VAR" decllist ";" cmpd\_statement

    decl\_list ::= declaration | declaration " " decl\_list “;”

declaration ::= type “ “ IDENTIFIER| function

function ::= (type|”void”) “ “ IDENTIFIER “(“ [decl\_list] ”)” cmpd\_statement

      type1 ::= "bool" | "char" | "int" | "double"

  array\_decl ::= type1 "[" (nr|“”) "]"

aggregate\_type ::= struct

struct ::= IDENTIFIER “{“decl\_list “}”

      type  ::= type1|array\_decl | aggregate\_type

   cmpd\_statement ::= "{" statement\_list "}"

   statement\_list ::= statement | statement ";" statement\_list

       statement ::= simple\_statement | struct\_statement

  simple\_statement ::= assign\_statement | io\_statement

 assign\_statement ::= IDENTIFIER "=" expression “;”

 expression ::= constant | expression operator expression

     io\_statement ::= "read" | "write”

 struct\_statement ::= cmpd\_statement | if\_statement | while\_statement | for\_statement

     if\_statement ::= "if" condition cmpd\_statement [“else if” condition

cmpd\_statement][“else” cmpd\_statement]

  while\_statement ::= "while" condition cmpd\_statement

for\_statement ::= type IDENTIFIER in range(nr1,nr2,direction) cmpd\_statement |

type IDENTIFIER in array\_name

array\_name ::= array identifier

nr1 ::= int

nr2 ::= int

direction ::= ‘-1’ | ‘1’

‘-1’ is backwards ::= num\_two, num\_two-1, …, num\_one

‘1’ is forwards ::= num\_one, num\_one+1, …, num\_two

condition ::= expression RELATION expression

  RELATION ::= "<" | "<=" | "=" | "!=" | ">=" | ">"

(b) Lexical

* indexing: IDENTIFIER “[“ index “]”
* index :

- 0->len\_array(IDENTIFIER) (“the index’th element from the start”)

- -1->-len\_array(IDENTIFIER) (“the index’th element from the end”)

- num->the (num-1)’th element from the start

- -num->the num’th element from the end

num := int

Program: Given an array of integers, filter out the prime ones and print them

bool is\_prime(int num\_one){

if (num\_one < 2){

return false;

}

if (num\_one == 2){

return true;

}

for int i in range(2, num\_one-1, 1){

if (num\_one % i == 0){

return false;

}

return true;

}

void filter\_primes(int[] array\_one){

for i in range(0, array\_one[-1]-1,1){

if (is\_prime(array\_one[i])){

write('cmd', array\_one[i]);

}

}

}