Bilkent University

Programming Languages

Department of Computer Engineering

THE LANGUAGE

Lexical Analysis Report

Project Group: 25

- *Serhat Aras - 21401636 - Section 02*
- *Seyfullah Yamanoğlu - 21400697 - Section 02*
- *Talha Şeker - 21302144 - Section 02*

Supervisor:

- *H. Altay Güvenir*

BNF description of the Language

<program> ::= <statements>
<statements> ::= <statement> | <statements> <statement>
<statement> ::= <if_statement> | <non_if_statement>

<non_if_statement> ::= <while_statement> | <expression> |
<declarations> |<function_call> | <assignment> | <input_statement>
| <dowhile_statement> | <output_statement>
<while_statement> ::= while (<expression>){ <statements> }
<dowhile_statement> ::= do {<statements>} while (<expression>)

<if_statement> ::= <matched> | <unmatched>
<matched> ::= if(<expression>)<matched> else <matched>
| <non_if_statement>
<unmatched> ::= if(<expression>)<if_statement>
| if(<expression>)<matched> else <unmatched>

<input_statement> ::= readbool <ident> | readinteger <ident>
<output_statement> ::= write <expression> | write <sentence>

<ident> ::= <letter> | <ident><letter> | <ident><digit>
<digit> ::= 0 | <nonzero>
<nonzero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<truthvalue> ::= <true> | <false>
<true> ::= true | <integer>
<false> ::= false | 0

<integer> ::= <sign> <nosign> | <nosign> | 0
<nosign> ::= <nonzero> <digit> | <nonzero>

<functiondec> ::= <returntype> <ident> (<functiondec_parameterlist>){<statements> return
<expression>}
<functioncall> ::= <ident>(<functioncall_parameterlist>);
<functiondec_parameterlist> ::= <functiondec_parameter>
|<functiondec_parameter>,<functiondec_parameterlist>
<functiondec_parameter> ::= <datatype> <ident> | “”
<functioncall_parameterlist> ::= <functioncall_parameter> | <functioncall_parameter>,
<functioncall_parameterlist>
<functioncall_parameter> ::= <ident> | “”

<datamodifier> ::= public | protected | final
<returntype> ::= void | <datatype>
<datatype> ::= boolvar | intvar

<declarations> ::= <vardec>; | <constvardec>; | <constdec>; | <arrayvardec>;
<vardec> ::= <datatype> <vardeclist>
<vardeclist> ::= <ident> | <ident> , <vardeclist>
<constvardec> ::= constvar <vardec><assignment_operator><expression>
<arrayvardec> ::= <datatype>[] <vardeclist>
<variable> ::= <ident> | <arraycall>

<arraycall> ::= <ident>[<integer>]

<assignment> ::= <variable><assignment_operator><expression>
<assignment_operator>::= <-
<expression> ::= <arithmetic_exp>; | <logical_exp>; | <equality_exp>;

<arithmetic_exp> ::= <arithmetic_exp> + <term> | <arithmetic_exp> - <term> | <term>
<term> ::= <variable> | <integer> | <term> * <factor> | <term> % <factor> | <term> / <factor>
<factor> ::= (<arithmetic_exp>) | <variable> | <integer>

<logical_exp> ::= <core_exp> | <complex_exp>
<core_exp> ::= <truthvalue> | <variable> | <equality_exp>
<complex_exp> ::= <not_operator><logical_exp> | (<arithmetic_exp>)
| (<logical_exp><logical_op><logical_exp>)

<equality_exp> ::= <coreeq_exp> | <complexeq_exp>
<coreeq_exp> ::= <truthvalue> | <variable>
<complexeq_exp> ::= <not_operator><equality_exp> | (<arithmetic_exp>)
| (<coreeq_exp><equality_op><coreeq_exp>)

<logical_op> ::= <or_operator> | <and_operator> | <xor_operator>
|<if-biconditional_op> | <if-then_op>
<equality_op> ::= <isequal_operator> | <lessthan_operator>
| <greaterthan_operator>

<or_operator> ::= ||
<and_operator> ::= &&
<xor_operator> ::= x|
<if-then_op> ::= =>
<if-biconditional_op> ::= <=>
<not_operator> ::= !
<isequal_operator> ::= ==
<lessthan_operator> ::= <<
<greaterthan_operator>::= >>
<sign> ::= - | +
<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 | 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
<increment_opt> ::= ++
<decremenet_opt> ::= --

<constdec> ::= constant “<sentence>” <assignment_operator> <truthvalue>

<sentence> ::= <word> | <sentence> <word>
<word> ::= <character> | <word><character>
<character> ::= <digit> | <letter> | < | > | ! | @ | # | $ | % | ^ | & | * | (|) | _ | + | } | { |]
| [| ” | : | \ | ’ | ; | , | . | /

Explanations

<program> : Program consists of statements.

<statements> : Statements can be only one statement or more than one

<statement> : Statement consist of if and non-if statements, basically they contain all the statement, expression, function callings, assignments, inputs, outputs and declarations.

<if_statement> : If statement that takes an expression inside and does the corresponding statements in its statement part. Requires matched and unmatched parts to prevent the ambiguity

<non_if_statement> : All other statements that is in the language except if statement. This is required to make the if statement working properly, without ambiguity.

<while_statement> : Looping statement that takes expression inside and does the statements until the expression inside becomes false.

Convention: while (condition) {

statements

}

<expression> : Expression can be arithmetic expression, logical expression or equality expression to be used.

<declarations> : Declarations can be variable, constant variable, constant and array.

<function_call> : This is responsible for function callings, that has a identifier (which is its name actually) and function calling parameters inside.

Convention: foo(functioncall_parameterlist);

<assignment> : Assignment is conducted with the assignment operator between the variable (left hand side) and the expression (right hand side).

<dowhile_statement> : Loop statement that does the given statement at once and checks for the expression given in while part and continues till the expression becomes false.

Convention: do {

statements

}while (expression);

<matched> : If statement that is already have a matched else with itself.

Convention:
```
if( expression ) {
        statements
}
else {
        statements
}

        or

if ( expression ) {
        statements
        if( expression ) {
                statements
        }
        else {
                statements
        }
}
else {
        statements
}
```

<unmatched> :If statement that does not have a matched else with itself.

Convention:
```
if( expression ) {
        statements
}
```

<input_statement> : Predescribed usage for taking an input from the client with reserved words "readbool" and "readinteger".

Convention: readbool <identifier>; or readinteger <identifier>;

<ident> : Identifiers starts with letter and can continue with digit or letter, no identifier can start with any number or digit, starting character must be a letter.

Convention: var1, v156asd, a1, count, etc.

<output_statement> : Predescribed usage for giving an output during the program execution with the reserved word "write". That can both output expressions and sentences.

Convention: write <expression>; or write <sentence>;

<digit> : All the digits from 0 to 9 in the language

<nonzero> : All digits from 1 to 9, excluding 0

<truthvalue> : True or false values

<true> : True is defined with the reserved word “true” or any integer

<false> : False is defined with the reserved word “false” or digit “0”

<nosign> : Digit with the no sign, meaning positive (hidden +).

<functiondec> : This is responsible for the declaration of the functions. It includes return type, identifier (method name), function parameter list and the statements inside it to be held in case of calling of this method and returns expression. All the methods have to have “return” statement.

Convention: intvar foo(functiondec_parameterlist)

```
{
   statements
   return expression
            }
```

<returntype> : Return type of the methods that can be void or data type

<functiondec_parameterlist>: Parameter list which is used during the declaration of the method.

<functioncall_parameterlist>: Parameter list which is used during the calling of the method

<functiondec_parameter> : It can be identifier which has a datatype, or empty (null string)

<functioncall_parameter> : It can be identifier, empty string, integer or a sentence

<datamodifier> : Variable and constant encapsulation part. Represents public, protected, private words. This is optional and when omitted variable will be treated as public.

<returntype> : This represents return type of a predicate or function. A function can returns three types of data: void, boolvar or intvar.

<datatype> : Represents and holds all valid data types which are boolvar and integervar for the bnf form.

<declarations> : This part only groups declaration statements together.

<vardec> : Used to declare non-array types: boolvar and intvar.

<vardeclist> : Identifier list for declaration of multiple variables at once. It can be one ident or multiple idents.

<constvardec> : It is for constant variable declarations. Constants must be declared and initialized in the one line with the assignment operator. The reserved word for this is “constvar”
Convention: constvar intvar PI = expression;

<arrayvardec> : Array declarations. Programmer must use the data type and square brackets for the declaration.
Convention: boolvar[] arr1, arr2, arr3;

<variable> : Represent dynamic structures of <identifier> and <arraycall>. Basically translates to testVariable or arr1[0].

<arraycall> : conventional usage for getting an element of an array. For example: arr1[0]

<assignment_operator> : Operator for assigning values into the fields “<-”

<arithmetic_exp> : This is to construct arithmetic operations in our language. Operator precedence among arithmetic operators are saved. To prevent ambiguity we insisted on to use of parentheses. This is composed of <term> and <factor>.

<term> : This is the layer after <arithmetic_exp> and before the <factor> layer and represents the summations and subtractions.

<factor> : This is the down level to represent the divisions and multiplications. By using this structure we saved the operator precedence in our language.

<logical_exp> : Initial point of logical expressions and propositional calculus part. This is composed of <core_exp> or <complex_exp>.

<core_exp> : This represents core aspects of a logical expression like an identifier or a truth value like true or false. It also can be constant or array element.

<complex_exp> : Composed of <core_exp> that connected together via <logical_operator> and a set of parentheses. A conventional translation would be !(p || (q x| r)) where p,q,r are the identifiers.

<equality_exp> :This part and following two parts are the basis and description of the equality expressions which are basically comparison statements for boolvar and intvar variables. This part also uses the same construction as the <logical_exp>

<coreeq_exp> : It is described above

<complexeq_exp> : It is described above

<logical_op> : Operator for logical operations, which can be or, and, xor, biconditional and if-then operators

<equality_op> : Operator for equality operations, which can be equality checking, less than or greater than operator

<or_operator> : Operator for or operation "||"

<and_operator> : Operator for and operation "&&"

<xor_operator> : Operator for exclusive or operation "x|"

<if-then_op> : Operator for if-then operation "=>"

<if-biconditional_op> : Operator for biconditional check "<=>"

<not_operator> : Operator that reverse the false to true and vice versa "!"

<isequal_operator> : Operator for checking the equality "=="

<lessthan_operator> : Operator for less than operation "<<"

<greaterthan_operator> : Operator for greater than operation ">>"

<sign> : Plus or minus signs

<letter> : Valid letters in the language

<increment_opt> : Operator for Increment "++"

<decremenet_opt> : Operator for Decrement "--"

<comment> : all the characters, words, and sentences after hash sign will be treated as comment, for instance # COMMENT

<constdec> : Constant declaration for variables

<sentence> : Group of words

<word> : Group of characters

<character> : Each character that is valid in the language

Lex Description:

```
/*lex.l for the project part 1*/

%option main

digit  [0-9]

sign  [+-]

or_operator \|\|

and_operator \&\&

xor_operator x\|

if-then_op \=\>

if-biconditional_op \<\=\>

not_operator \!

isequal_operator \=\=

lessthan_operator \<\<

lessthan-equal_operator \<\<\=

greaterthan_operator \>\>

greaterthan-equal_operator \>\>\=

letter [A-Za-z_$]

newline \n

character ( {digit}|{letter}|<|>|!|@|#|$|%|^|&|*|(|)|_|+|}|{|]|[|”|:|\|’|;|,|.|/|)

left_pth \(

right_pth \)

left_curly \{

right_curly \}

left_sqr-bracket \[

right_sqr-bracket \]

assignment_operator \<\-

true true

false false

truth_value ({true}|{false})

quot \"

alphanumeric ({letter}|{digit})
```

```
plus \+
minus \-
mult \*
divi \/
semicolon \;
increment \+\+
decrement \-\-
comment #
whitespace [ \t\n]
%%
{or_operator}                   {printf("OR_OPERATOR ");}
{and_operator}                  {printf("AND_OPERATOR ");}
{xor_operator}                  {printf("XOR_OPERATOR ");}
{if-then_op}                    {printf("IF-THEN_OP ");}
{if-biconditional_op}           {printf("IF-BICONDITIONAL_OP ");}
{not_operator}                  {printf("NOT_OPERATOR ");}
{isequal_operator}              {printf("ISEQUAL_OPERTOR ");}
{lessthan_operator}             {printf("LESSTHAN_OPERATOR ");}
{greaterthan_operator}          {printf("GREATERTHAN_OPERATOR ");}
{greaterthan-equal_operator}    {printf("GREATERTHAN_EQUAL ");}
{lessthan-equal_operator}       {printf("LESSTHAN_EQUAL ");}
{left_pth}                      {printf("LEFT_PARENTHESIS ");}
{right_pth}                     {printf("RIGHT_PARENTHESIS ");}
{left_curly}                    {printf("LEFT_CURLY ");}
{right_curly}                   {printf("RIGHT_CURLY ");}
{left_sqr-bracket}              {printf("LEFT_SQUARE_BRACKET ");}
{right_sqr-bracket}             {printf("RIGHT_SQUARE_BRACKET ");}
{assignment_operator}           {printf("ASSIGNMENT_OPERATOR ");}
{plus}                          {printf("PLUS ");}
{minus}                         {printf("MINUS ");}
{mult}                          {printf("MULTIPLICATION ");}
```

```
{divi}                          {printf("DIVISION ");}
{semicolon}                     {printf("SEMICOLON ");}
{increment}                     {printf("INCREMENT ");}
{decrement}                     {printf("DECREMENT ");}
{true}                          {printf("TRUE ");}
{false}                         {printf("FALSE ");}
while                           {printf("WHILE ");}
for                            {printf("FOR ");}
do                              {printf("DO ");}
if                              {printf("IF ");}
else                            {printf("ELSE ");}
readbool                        {printf("READ_BOOLEAN ");}
readinteger                     {printf("READ_INTEGER ");}
write                           {printf("WRITE ");}
constvar                        {printf("CONST_VAR ");}
boolvar                         {printf("BOOLEAN_VAR ");}
intvar                          {printf("INTEGER_VAR ");}
void                            {printf("VOID ");}
public                          {printf("PUBLIC ");}
protected                       {printf("PROTECTED ");}
private                         {printf("PRIVATE ");}
fixed                           {printf("FIXED ");}
{newline}                       {printf("\n");}
[\#]([ \t\n]+|[a-zA-Z0-9]*)+    {printf("COMMENT");}
{sign}{digit}+                  {printf("INTEGER ");}
{letter}|({alphanumeric}|_)*    {printf("IDENTIFIER ");}
quot.quot                       {printf("SENTENCE ");}
```

Example Program (Test):

Code:

```
# this is comment FYI
# variable declarations
boolvar bool1;
Boolvar bool2, bool3, boolN;
intvar integer1;
constvar integervar const_integer1 <- 1234;
constvar boolvar const_bool1 <- false;

boolvar[] arr1, arr2;
arr1[0] <- false;
arr2[1]    <- arr[0];

# constant sentences
constant “Man is mortal!” <- false;

bool1 <- true;
bool2 <- true;
readbool bool3;
integer1 <- 5;
# increment and decrement
integer1++;
integer1--;

if(!“Man is mortal!” && (bool1 || bool3) && !(const_integer1 >> integer1)){
   integer1 <- integer1 * (integert1 + 2) + integer1;
}else{
            write “No Way Dude!”;
}

while (boolN == true){
   readinteger integer1;
   boolN <- integer%2;
}

void foo(boolvar bool, integervar integer){
   write bool;
   write integer;
}

boolvar foo2(){
   return false;
}

foo(true, 8);
bool1 <- foo2();
```

Result of the Test Program

*results are based on the lex tool and gcc complier in the dijkstra.ug.bcc.bilkent.edu.tr server.

```
[serhat.aras@dijkstra A]$ lex lex.l
[serhat.aras@dijkstra A]$ gcc -o test lex.yy.c
[serhat.aras@dijkstra A]$ ./test < testcode.txt
Single Line Comment
Single Line Comment
BOOLEAN_VAR  IDENTIFIER SEMICOLON
IDENTIFIER  IDENTIFIER , IDENTIFIER , IDENTIFIER SEMICOLON
INTEGER_VAR  IDENTIFIER SEMICOLON
CONST_VAR  IDENTIFIER  IDENTIFIER  ASSIGNMENT_OPERATOR  IDENTIFIER SEMICOLON
CONST_VAR  BOOLEAN_VAR  IDENTIFIER  ASSIGNMENT_OPERATOR  FALSE SEMICOLON

BOOLEAN_VAR LEFT_SQUARE_BRACKET RIGHT_SQUARE_BRACKET  IDENTIFIER , IDENTIFIER SEMICOLON
IDENTIFIER LEFT_SQUARE_BRACKET IDENTIFIER RIGHT_SQUARE_BRACKET  ASSIGNMENT_OPERATOR  FALSE SEM
ICOLON
IDENTIFIER LEFT_SQUARE_BRACKET IDENTIFIER RIGHT_SQUARE_BRACKET     ASSIGNMENT_OPERATOR  IDENTI
FIER LEFT_SQUARE_BRACKET IDENTIFIER RIGHT_SQUARE_BRACKET SEMICOLON

Single Line Comment
IDENTIFIER  ▒IDENTIFIER  IDENTIFIER  IDENTIFIER NOT_OPERATOR ▒ ASSIGNMENT_OPERATOR  FALSE SEMI
COLON

IDENTIFIER  ASSIGNMENT_OPERATOR  TRUE SEMICOLON
IDENTIFIER  ASSIGNMENT_OPERATOR  TRUE SEMICOLON
READ_BOOLEAN  IDENTIFIER SEMICOLON
IDENTIFIER  ASSIGNMENT_OPERATOR  IDENTIFIER SEMICOLON
Single Line Comment
IDENTIFIER INCREMENT SEMICOLON
IDENTIFIER DECREMENT SEMICOLON

IF LEFT_PARENTHESIS NOT_OPERATOR ▒IDENTIFIER  IDENTIFIER  IDENTIFIER NOT_OPERATOR ▒ AND_OPERAT
OR  LEFT_PARENTHESIS IDENTIFIER  OR_OPERATOR  IDENTIFIER RIGHT_PARENTHESIS  AND_OPERATOR  NOT_
OPERATOR LEFT_PARENTHESIS IDENTIFIER  GREATERTHAN_OPERATOR  IDENTIFIER RIGHT_PARENTHESIS RIGHT
_PARENTHESIS LEFT_CURLY
    IDENTIFIER  ASSIGNMENT_OPERATOR  IDENTIFIER  MULTIPLICATION  LEFT_PARENTHESIS IDENTIFIER
PLUS  IDENTIFIER RIGHT_PARENTHESIS  PLUS  IDENTIFIER SEMICOLON
RIGHT_CURLY ELSE LEFT_CURLY
WRITE  ▒IDENTIFIER  IDENTIFIER  IDENTIFIER NOT_OPERATOR ▒SEMICOLON
RIGHT_CURLY

WHILE  LEFT_PARENTHESIS IDENTIFIER  ISEQUAL_OPERTOR  TRUE RIGHT_PARENTHESIS LEFT_CURLY
    READ_INTEGER  IDENTIFIER SEMICOLON
    IDENTIFIER  ASSIGNMENT_OPERATOR  IDENTIFIER %IDENTIFIER SEMICOLON
RIGHT_CURLY

VOID  IDENTIFIER LEFT_PARENTHESIS BOOLEAN_VAR  IDENTIFIER , IDENTIFIER  IDENTIFIER RIGHT_PAREN
THESIS LEFT_CURLY
    WRITE  IDENTIFIER SEMICOLON
    WRITE  IDENTIFIER SEMICOLON
RIGHT_CURLY

BOOLEAN_VAR  IDENTIFIER LEFT_PARENTHESIS RIGHT_PARENTHESIS LEFT_CURLY
    IDENTIFIER  FALSE SEMICOLON
RIGHT_CURLY

IDENTIFIER LEFT_PARENTHESIS TRUE , IDENTIFIER RIGHT_PARENTHESIS SEMICOLON
IDENTIFIER  ASSIGNMENT_OPERATOR  IDENTIFIER LEFT_PARENTHESIS RIGHT_PARENTHESIS SEMICOLON
[serhat.aras@dijkstra A]$
```