Bilkent Üniversitesi

Programming Languages

Department of Computer Engineering

Lexical Analysis Report

PhysLab

Project Group: 25

- Ömer Ünlüsoy – 21702136 – Section 3
- Cankat Tilki – 21702297 – Section 3

Supervisor:

- Karani Kardeş

BNF Description (Part A)

Types and Constants

<low_alphabetic> = [a-z]

<upp_alphabetic> = [A-Z]

<alphabetic> = [a-zA-Z]

<numeric> = [0-9]

<alphanumeric> = <alphabetic> | <numeric>

<CONSTANT> = <upp_alphabetic> <alphanumeric>*

<VARIABLE> = <low_alphabetic> <alphanumeric>*

<comment_keyword> = “//”

The comment keyword of PhysLab is “//”. Our motivation to chose this keywork was that it is a well-known keyword for comment which most languages like Java and C++ use. Simple and easy to remember.

<FORDOT> = “…”

This operator seperates the boundaries of for loops (ex: for i in 1…3). We have not used “from” and “to” keywords because this operator is shorter and more practical. Several new languages like Swift also uses this operator to shorten the for loop and make it easy to read and uderstand.

<LBRACKET> = “{”

<RBRACKET> = “}”

<SIGN> = “+” | “-”

<LP> = “(”

<RP> = “)”

<COMMA> = “,”

<ASSIGN> = “<=”

PhysLab uses “<=” operator for assignment which is not the common way to do it. Although most languages just use “=” operator which is a simple equality operator, we prefered to use “<=” operator. Our motivation was to prevent confusion with the mathematics’s equality symbol.

<COLON> = ":"

PhysLab uses colon operator in a decleration (ex: int: a <= 1). This allows developers to point the variable's type easier.

<functionout_keyword>= "->"

PhysLab uses "->" operator in the function decleration with outputs (ex: func a() -> int: b). This allows developers to point the function's outputs easier.

<TYPE> = "int" | "double" | "string" | "bool"

<INTEGER> = [<SIGN>] <numeric>+

<DOUBLE> = [numeric]* "." [numeric]+

<BOOLEAN> = "true" | "false"

<STRING> = [a-zA-Z]+

<FOR> = "for"

<WHILE> = "while"

<IF> = "if"

<ELSE> = "else"

<FUNC> = "func"

Keyword for function decleration.

<CONST> = "const"

Keyword for constant decleration

<IN> = "in"

PhysLab uses this keyword to point for loop's bounderies (ex: for i in 0…1)

<RETURN> = "return"

<BY> = "by"

PhysLab uses this keyword to specify the for loop's increment amount. Specifying it is optional (ex: for I in 0…6 by 2)(iter values: 0 -> 2 -> 4 -> 6).

<SCAN> = “scan”

PhysLab uses this keyword for input (ex: int: a <= scan).

<PRINT> = “print”

PhysLab uses this keyword for output (ex: print a).

<least_prec_math_op> = “+” | “-”

<most_prec_math_op> = “.” | “/” | “%”

<logicOp> = “==” | “!=” | “>”| “=>”| “=<”| “<”| “and”| “or”

<DRONE_GET> = “getInclination” | “getAltitude” | “getTemperature” | “getAcceleration” | “getTimestamp”

<TURN> = “turnCamera”

<TAKE_PICTURE> = “takePicture”

<CONNECT> = “connectToBase”

<DISCONNECT> = “disconnectFromBase”

Program Rules

<**program**> ::= <statements> | <empty>

<**statements**> ::= <statements> <statement> | <statement>

<**statement**> ::= <comment>| <ifStatement> | <loopStatement>
| <otherStatement>

<**comment**> ::= <comment_keyword> <alphanumeric>* <comment_keyword>

<**ifStatement**> ::= <matched> | <unmatched>

<**loopStatement**> ::= <forLoop> | <whileLoop>

<**forLoop**> ::= <FOR> <VARIABLE> <IN> <forTerm> <FORDOT> <forTerm>
[<forStepExp>] <LBRACKET> <statementBlock>
<RBRACKET>

<**whileLoop**> ::= <WHILE> <logicCondition> <LBRACKET>
<statementBlock> <RBRACKET>

<**otherStatement**> ::= <assignment> | <declaration> | <functionCall>

<**assignment**> ::= <VARIABLE> <ASSIGN> <assignmentExp>

<**declaration**> ::= <constantDec> | <variableDec> | <functionDec>

<**constantDec**> ::= <CONST> <TYPE> <COLON> <CONSTANT> <ASSIGN>
<assignmentExp>

<**variableDec**> ::= <TYPE> <COLON> <VARIABLE> | <TYPE> <COLON>
<VARIABLE> <ASSIGN> (<VARIABLE> | <assignmentExp>)

<**functionDec**> ::= <FUNC> <VARIABLE> <parameterExp> [<functionoutExp>]
<LBRACKET> <statementBlock> [<returnStatement>]
<RBRACKET>

<**functionCall**> ::= <VARIABLE> <LP> [<callParamList>] <RP>
| <outStatement> | <inStatement> | <drone_method>

<outStatement> ::= <PRINT> <printable>

<printable> ::= <printable> <COMMA>
(<INTEGER>|<DOUBLE>|<BOOLEAN>|<STRING>|<VARIABLE>)|
(<INTEGER>|<DOUBLE>|<BOOLEAN>|<STRING>|<VARIABLE>)

<**inStatement**> ::= <VARIABLE> <ASSIGN> <SCAN>

<droneMethod> ::= <droneGetMethod> | <cameraStatus> | <takePicture> |
<connect> | <disconnect>

<droneGetMethod> ::= <DRONE_GET> <LP> [<callParamList>] <RP>

<cameraStatus> ::= <TURN> <LP> [<callParamList>] <RP>

```
<takePicture>     ::=   <TAKE_PICTURE> <LP> [<callParamList>] <RP>

<connect>         ::=   <CONNECT> <LP> [<callParamList>] <RP>

<disconnect>      ::=   <DISCONNECT> <LP> [<callParamList>] <RP>

<returnStatement> ::=   <RETURN> <assignmentExp>

<assignmentExp>   ::=   <logicCondition> | <STRING> | <VARIABLE>

<logicCondition>  ::=   <logicCondition> <logicOp> <nonLogicExp>
                        | <nonLogicExp>

<nonLogicExp>     ::=   <functionCall> | <BOOLEAN> | <VARIABLE>
                        | <nonLogicMath>

<nonLogicMath>    ::=   <nonLogicMath> <least_prec_math_op> <mult_div>
                        | <mult_div>

<mult_div>        ::=   <mult_div> <most_prec_math_op> <number> | <number>

<matched>         ::=   <IF> <logicCondition> <LBRACKET> <matched>
                        <RBRACKT> <ELSE> <LBRACKET>  (<matched> |
                        <statementBlock>) <RBRACKET>

<unmatched>       ::=   <IF> <logicCondition> <LBRACKET> <statementBlock>
                        <RBRACKET> | <IF> <logicCondition> <LBRACKET>
                        <matched> <RBRACKET> <ELSE> <LBRACKET> <unmatched>
                        <RBRACKET>

<statementBlock>  ::=   <statements> | <empty>

<empty>           ::=

<number>          ::=   <INTEGER> | <DOUBLE>

<parameterExp>    ::=   <LP> [<functionParams>] <RP>

<functionParams>  ::=   <functionParams> <COMMA> <functionParam>
                        | <functionParam>

<functionParam>   ::=   <TYPE> <COLON> <VARIABLE>

<functionoutExp>  ::=   <functionout_keyword> <functionParam>

<forStepExp>      ::=   <BY> <INTEGER>

<forTerm>         ::=   <VARIABLE> | <INTEGER> | <functionCall>
                        | <LP><nonLogicMath><RP>

<callParamList>   ::=   <callParamList> <COMMA> <VARIABLE> | <VARIABLE>
```

BNF Explanations

<program>: A non-terminal consisting 0 or more statements. This allows user to define an empty program

<statements>: Set of one or more statement

<statement>: This generalizes loops if statements and other statements

<ifStatement>: This nonterminal is defining rule for if statements.

<loopStatement>: This nonterminal is defining rule for loop statements.

<forLoop>: This nonterminal specifies for loop statement. After for keyword we write a variable to determine an element. Later, in keyword specifies the array of elements. This array is started from first <forTerm> and ends at second <forTerm>. Then by <forStepExp> we can change the step size (e.g. it can process every even element if <forStepExp> is 2) and later specifying statement block between "{" and "}".

<whileLoop>: While keyword specifies that it is a while loop. Later between brackets we write sequence of statements in <statementBlock>

<otherStatement>: This nonterminal specifies other statements as <assignment>, <functionCall> and <declaration>

<assignment>: This statement assigns a value to a variable. First it specifies the variable name with <variable> nonterminal. Then assign sign "<=" is written and after that we have an expression giving a value.

<decleration>: This statement declares a constant, variable or function

<constantDec>: First, we use "const" keyword to determine a constant. Then, we determine the type as "int", "double", "string", "bool". Then we use ":" and after that the name comes. Then we use "<=" to assign an assignment expression.

<variableDec>: First, we determine the type as before. Then, we use ":" and alter write the name. Here we might or might not initialize the variable. If we want to initialize, we use "<=" sign and later an assignment expression.

<functionDec>: First we type func keyword. Then, we write the name as a variable name. In our grammar, variable names start with small letter whereas constant names start with large letter. Then, we give parameters and later use "->" to specify the output type and name. Later, we write statements between brackets and a return statement optionally.

<functionCall>: This nonterminal is the rule to call a function. We first write the name of function and later write parameters inside "()". Here we extended this by adding out statement, in statement and methods specified for drone.

<outStatement>: This nonterminal is the output rule. Here we first use print keyword and later write parameters.

<printable>: Parameters of out statement

<**inStatement**>: Similar to assignment syntax, we first write name of variable and then use "<=" but lastly, we use scan keyword to specify the need to input.

<droneMethod>: All drone specific methods

<droneGetMethod>: Syntax rule for getting attributes of drones. We first write attribute_specific keywords (e.g. "getAcceleration") and then write parameters between parantheses.

<cameraStatus>: Syntax rule for changing camera status of drones. We first write turnCamera keyword and then write parameters between parantheses. With these parameters, we can specify whether to turn on or off by using a boolean.

<takePicture>: Syntax rule for taking picture. We first write takePicture keyword and then write parameters between parantheses.

<connect>: Syntax rule for connecting to a base computer. We first write connect keyword and then write parameters between parantheses to indicate to relevant address of base computer.

<disconnect>: Syntax rule for disconnecting from a base computer. We first write disconnect keyword and then write parameters between parantheses.

<returnStatement>: Syntax for returning a specific value from function. We first use return keyword and later the assignment expression.

<assignmentExp>: This is either a logicCondition, string or a variable.

<logicCondition>: This is the syntax for writing a logic operation.

<nonLogicExp>: This is either a functionCall, Boolean, variable or arithmetic expression.

<nonLogicMath>: This is the syntax for doing arithmetic expressions.

<mult_div>: We needed to divide arithmetic expressions into 2 parts because we had to show precedence. To do that, we determine first operations "+, -" in parse tree and then determine "%, ., /"operations.

<matched>: Matched case is used in ouf BNF declaration for if statement. This syntax is used when there are equal number of if and else statements.

<unmatched>: Unmatched case is used when there are unmatched if's. In this case, to eliminate dangling else problem, we used this approach.

<statementBlock>: 0 or many statements

<empty>: To declare an empty program.

<number>: Integer or double

<parameterExp>: Write parameters between parantheses

<functionParams>: Function parameter sequence syntax rule.

<functionParam>: A specific function parameter rule. Here we first write the type, then ":" and later the variable name.

<**functionoutExp**>: The expression for defining the output of the function. Is a part of <functionDec> which is explained in detail.

<forStepExp>: Expression for defining the step size in for loop. Use first by keyword and then put the step size as integer.

<forTerm>: Expression for what to put in <forTerm> in <forLoop>. Here, this can be a function, a variable, an integer or a nonLogicMath operation between paranthesis.

<callParamList>: Parameter syntax for functionCall.

Language Evaluation

1. Readability

a. Overall simplicity

i. A manageable set of features and constructs: We made out programming language minimal as possible. Only related if and loop statements are introduced along with assignment, return and declaration statements. This is done because we do not believe that we might need an advanced feature such as Object Oriented programming. Because normally in drones we expect a C type language for using sensors. Thus, we mostly do low level implementation.

ii. Minimal operator overloading: We do not let any operator to be overloaded.

b. Orthogonality

i. A relatively small set of primitive constructs can be combined in a relatively small number of ways: As said, we limited our language to a great extend to make the learning curve fast as possible.

c. Syntax considerations

i. Meaningful keywords: PhysLab uses common keywords for fundemental tokens such as for, if, func, return, print , int, string etc. However PhysLab also uses some new keywords that new langugaes like Swift uses in order to increase simlicty and understandibility. For example; in, by, scan, etc. These keywords are used by new languages like Swift. This situation is also same for operators. While some operators of PhysLab are common and convensional such as dot, comma, equality, and math operators; some of them are different from the conventional way to increase understandibility such as assignment (<=), =<, => and FORDOT (…).

2. Writability

a. Simplicity and orthogonality

i. Few constructs, a small number of primitives, a small set of rules for combining them: We minimized out set of primitives to integer, double, string and Boolean. We did not define char because we think that as 1 element string. Our rules are trivial. We defined if and loop statements as building blocks. Then we used assignment and declaration statements to determine variable and functions.

b. Expressivity

i. A set of relatively convenient ways of specifying operations: Our assignment operator is “<=”. This makes it unambiguous because this is like an arrow that shows the target. If we have used “=” then it would be ambiguous because of “=” in mathematics.

   Also, in out function definition, we use “->” to determine the output values. Here arrow is like a function whose input is parameters and output is right-hand side.

3. Reliability

a. Type checking

   i. Testing for type errors: We forced user to define types for later to typecheck these variables.

Lex Description

```
/*lex.l file for the Project 1*/

%option main

COMMENT                 \/\/.*
DOT                     \.
TAB                      \\t
NL                      \\n
LBRACKET                \{
RBRACKET                \}
LP                      \(
RP                      \)
COMMA                   \,
ASSIGN                   \<\=
COLON                   \:
FOUT                    \-\>
FORDOT                  \.\.\.
EQUAL                   \=\=
NOTEQUAL                \!\=
LESSTHANOREQUAL         \=\<
GREATERTHANOREQUAL      \=\>
LESSTHAN                \<
GREATERTHAN             \>
AND                     and
OR                      or
PLUS                    \+
MINUS                   \-
MULTIPLY                 \*
DIVIDE                  \/
MOD                      \%
int                     int
double                  double
```

string	string
bool	bool
BOOLEAN	(true|false)
FOR	for
WHILE	while
IF	if
ELSE	else
ELIF	elif
IN	in
BY	by
CONST	const
FUNC	func
RETURN	return
SCAN	scan
PRINT	print
CONNECT	connect\(\)
DISCONNECT	disconnect\(\)
GETINCLINATION	getInclination\(\)
GETALTITUDE	getAltitude\(\)
GETTEMPRATURE	getTemperature\(\)
GETACCELERATION	getAcceleration\(\)
GETTIMESTAMP	getTimestamp\(\)
TURN	turnCamera\(\)
TAKEPICTURE	takePicture\(\)
numeric	[0-9]
alphabetic	[A-Za-z]
low_alphabetic	[a-z]
upp_alphabetic	[A-Z]
alphanumeric	{alphabetic}|{numeric}
INTEGER	[+-]?{numeric}+
DOUBLE	{numeric}*"."{numeric}+

```
FUNCTION                {low_alphabetic}{alphanumeric}*\(\)
STRING                  \"(\\.|[^"\\])*\"
VARIABLE                {low_alphabetic}{alphanumeric}*
CONSTANT                {upp_alphabetic}{alphanumeric}*
%%
{int}                   {printf("INT_TYPE ");}
{double}                {printf("DOUBLE_TYPE ");}
{bool}                  {printf("BOOL_TYPE ");}
{string}                {printf("STRING_TYPE ");}
{GETINCLINATION}        {printf("GETINCLINATION ");}
{GETALTITUDE}           {printf("GETALTITUDE ");}
{GETTEMPRATURE}         {printf("GETTEMPRATURE ");}
{GETACCELERATION}       {printf("GETACCELERATION ");}
{GETTIMESTAMP}          {printf("GETTIMESTAMP ");}
{TAKEPICTURE}           {printf("TAKEPICTURE ");}
{TURN}                  {printf("TURNCAMERA ");}
{CONNECT}               {printf("CONNECT ");}
{DISCONNECT}            {printf("DISCONNECT ");}
{FUNCTION}              {printf("FUNCTION ");}
{PRINT}                 {printf("PRINT ");}
{SCAN}                  {printf("SCAN ");}
{BOOLEAN}               {printf("BOOLEAN ");}
{DOT}                   {printf("DOT ");}
{MULTIPLY}              {printf("MULTIPLY ");}
{IF}                    {printf("IF ");}
{ELSE}                  {printf("ELSE ");}
{ELIF}                  {printf("ELIF ");}
{NL}                    {printf("NL ");}
{TAB}                   {printf("TAB ");}
{FOR}                    {printf("FOR ");}
{FORDOT}                {printf("FORDOT ");}
```

```
{WHILE}                 {printf("WHILE ");}
{IN}                    {printf("IN ");}
{BY}                    {printf("BY ");}
{CONST}                 {printf("CONST ");}
{FUNC}                  {printf("FUNC ");}
{RETURN}                {printf("RETURN ");}
{LBRACKET}              {printf("LBRACKET ");}
{RBRACKET}              {printf("RBRACKET ");}
{LP}                    {printf("LP ");}
{RP}                    {printf("RP ");}
{COLON}                 {printf("COLON ");}
{COMMA}                 {printf("COMMA ");}
{FOUT}                  {printf("FUNCTIONOUT ");}
{ASSIGN}                {printf("ASSIGN ");}
{EQUAL}                 {printf("EQUAL ");}
{NOTEQUAL}              {printf("NOTEQUAL ");}
{GREATERTHAN}           {printf("GREATERTHAN ");}
{LESSTHAN}              {printf("LESSTHAN ");}
{GREATERTHANOREQUAL}    {printf("GREATERTHANOREQUAL ");}
{LESSTHANOREQUAL}       {printf("LESSTHANOREQUALOP ");}
{AND}                   {printf("AND ");}
{OR}                    {printf("OR ");}
{PLUS}                  {printf("PLUS ");}
{MINUS}                 {printf("MINUS ");}
{DIVIDE}                {printf("DIVIDE ");}
{MOD}                   {printf("REMAINDER ");}
{COMMENT}               {printf("COMMENT ");}
{INTEGER}               {printf("INTEGER ");}
{DOUBLE}                {printf("DOUBLE ");}
{STRING}                {printf("STRING ");}
{VARIABLE}              {printf("VARIABLE ");}
```

Test Code

```
// Test Code
connect()

int: i1 <= 1
double: d1 <= 1.1
bool: b1 <= false
string: s1 <= "String 1"

const int: I2 <= 2
const double: D2 <= 3.1234
const bool: B2 <= true
const string: S2 <= "String 2"

int: firstSensorData <= getInclination()
int: secondSensorData <= getAltitude()
double: thirdSensorData <= getTemperature()
double: forthSensorData <= getAcceleration()
int: fifthSensorData <= getTimestamp()

string: url <= getURL()
sendURL(var, 3)
double: level
level <= scan
print level

for iter in 0...2 {
        print "iter: ", iter
}
for iter in 0...6 by 2 {
        print "iter: ", iter
```

```
}
for iter in 0...7 {
  while takePicture() < 123456789 {
        if turnCamera() == 0 {
              x <= x + (y / z)
         }
         else {
              x <= x / z + y
         }
  }
  level <= getSoundLevel()
  int: light
  light <= 7
  print light

  int: temperature <= getTemperature()
  if temperature > getTemperature() {
        light <= getLight()
  }

  if light != 3 {
        bool: air <= getAirQuality()
  }

  else {
        print light
  }
}
func dummy (int: x) -> int: result {
        x <= x + 1
  return x
}
```

```
float: humidity <= getHumidity()


func dummy2() {
  int: a <= scan
  int: b <= scan
  if a =< 2 and b < 2{
    print a or b
  }
  elif b == 2{
    print a and b
  }
  elif a => 4{
    print a
  }
  else{
    print b
  }
}

disconnect()
```

Results of the Test Code

```
[omer.unlusoy@dijkstra ~]$ lex lex.l
[omer.unlusoy@dijkstra ~]$ gcc -o lex lex.yy.c
[omer.unlusoy@dijkstra ~]$ cat test.txt | ./lex
COMMENT

CONNECT

INT_TYPE COLON  VARIABLE  ASSIGN  INTEGER
DOUBLE_TYPE COLON  VARIABLE  ASSIGN  DOUBLE
BOOL_TYPE COLON  VARIABLE  ASSIGN  BOOLEAN
STRING_TYPE COLON  VARIABLE  ASSIGN  STRING

CONST  INT_TYPE COLON  IINTEGER  ASSIGN  INTEGER
CONST  DOUBLE_TYPE COLON  DINTEGER  ASSIGN  DOUBLE
CONST  BOOL_TYPE COLON  BINTEGER  ASSIGN  BOOLEAN
CONST  STRING_TYPE COLON  SINTEGER  ASSIGN  STRING

INT_TYPE COLON  VARIABLE  ASSIGN  GETINCLINATION
INT_TYPE COLON  VARIABLE  ASSIGN  GETALTITUDE
DOUBLE_TYPE COLON  VARIABLE  ASSIGN  GETTEMPRATURE
DOUBLE_TYPE COLON  VARIABLE  ASSIGN  GETACCELERATION
INT_TYPE COLON  VARIABLE  ASSIGN  GETTIMESTAMP

STRING_TYPE COLON  VARIABLE  ASSIGN  FUNCTION
VARIABLE LP VARIABLE COMMA  INTEGER RP

DOUBLE_TYPE COLON  VARIABLE
VARIABLE  ASSIGN  SCAN
PRINT  VARIABLE

FOR  VARIABLE  IN  INTEGER FORDOT INTEGER  LBRACKET
       PRINT  STRING COMMA  VARIABLE
RBRACKET

FOR  VARIABLE  IN  INTEGER FORDOT INTEGER  BY  INTEGER  LBRACKET
       PRINT  STRING COMMA  VARIABLE
RBRACKET

FOR  VARIABLE  IN  INTEGER FORDOT INTEGER  LBRACKET
  WHILE  TAKEPICTURE  LESSTHAN  INTEGER  LBRACKET
       IF  TURNCAMERA  EQUAL  INTEGER  LBRACKET
                VARIABLE  ASSIGN  VARIABLE  PLUS  LP VARIABLE  DIVIDE  VARIABLE RP
        RBRACKET
        ELSE  LBRACKET
                VARIABLE  ASSIGN  VARIABLE  DIVIDE  VARIABLE  PLUS  VARIABLE
        RBRACKET
  RBRACKET

  VARIABLE  ASSIGN  FUNCTION
  INT_TYPE COLON  VARIABLE
  VARIABLE  ASSIGN  INTEGER
  PRINT  VARIABLE

  INT_TYPE COLON  VARIABLE  ASSIGN  GETTEMPRATURE
  IF  VARIABLE  GREATERTHAN  GETTEMPRATURE  LBRACKET
       VARIABLE  ASSIGN  FUNCTION
  RBRACKET

  IF  VARIABLE  NOTEQUAL  INTEGER  LBRACKET
```

```
FOR  VARIABLE  IN  INTEGER FORDOT INTEGER  BY  INTEGER  LBRACKET
       PRINT  STRING COMMA  VARIABLE
RBRACKET

FOR  VARIABLE  IN  INTEGER FORDOT INTEGER  LBRACKET
  WHILE  TAKEPICTURE  LESSTHAN  INTEGER  LBRACKET
       IF  TURNCAMERA  EQUAL  INTEGER  LBRACKET
                VARIABLE  ASSIGN  VARIABLE  PLUS  LP VARIABLE  DIVIDE  VARIABLE RP
        RBRACKET
        ELSE  LBRACKET
                VARIABLE  ASSIGN  VARIABLE  DIVIDE  VARIABLE  PLUS  VARIABLE
        RBRACKET
  RBRACKET

  VARIABLE  ASSIGN  FUNCTION
  INT_TYPE COLON  VARIABLE
  VARIABLE  ASSIGN  INTEGER
  PRINT  VARIABLE

  INT_TYPE COLON  VARIABLE  ASSIGN  GETTEMPRATURE
  IF  VARIABLE  GREATERTHAN  GETTEMPRATURE  LBRACKET
       VARIABLE  ASSIGN  FUNCTION
  RBRACKET

  IF  VARIABLE  NOTEQUAL  INTEGER  LBRACKET
       BOOL_TYPE COLON  VARIABLE  ASSIGN  FUNCTION
  RBRACKET

  ELSE  LBRACKET
       PRINT  VARIABLE
  RBRACKET
RBRACKET

FUNC  VARIABLE  LP INT_TYPE COLON  VARIABLE RP  FUNCTIONOUT  INT_TYPE COLON  VARIABLE  LBRACKET
       VARIABLE  ASSIGN  VARIABLE  PLUS  INTEGER
  RETURN  VARIABLE
RBRACKET

VARIABLE COLON  VARIABLE  ASSIGN  FUNCTION

FUNC  FUNCTION  LBRACKET
  INT_TYPE COLON  VARIABLE  ASSIGN  SCAN
  INT_TYPE COLON  VARIABLE  ASSIGN  SCAN
  IF  VARIABLE  LESSTHANOREQUALOP  INTEGER  AND  VARIABLE  LESSTHAN  INTEGER LBRACKET
    PRINT  VARIABLE  OR  VARIABLE
  RBRACKET
  ELIF  VARIABLE  EQUAL  INTEGER LBRACKET
    PRINT  VARIABLE  AND  VARIABLE
  RBRACKET
  ELIF  VARIABLE  GREATERTHANOREQUAL  INTEGER LBRACKET
    PRINT  VARIABLE
  RBRACKET
  ELSE LBRACKET
    PRINT  VARIABLE
  RBRACKET
RBRACKET

DISCONNECT
[omer.unlusoy@dijkstra ~]$
```