Bilkent University

Programming Languages

Department of Computer Engineering

THE LANGUAGE

Revised and Augmented Language Design Report

Project Group: 25

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

Supervisor:

- *H. Altay Güvenir*

İçindekiler

Revised BNF description of the Language

```
program: lines;

lines:
     | lines line
    ;

line:  NL
     | statements NL
     | COMMENT
    ;

statements: statement SEMICOLON
   | statement SEMICOLON statements
  ;
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 LP expression RP LEFT_CURLY statements RIGHT_CURLY
  ;

dowhile_statement: DO LEFT_CURLY statements RIGHT_CURLY WHILE LP expression RP
  ;

if_statement : matched
   | unmatched
  ;

matched: IF LP expression RP  matched  ELSE  matched
   | LEFT_CURLY non_if_statement RIGHT_CURLY
  ;

unmatched: IF LP expression RP  if_statement
   | IF LP expression RP  matched  ELSE  unmatched
  ;


input_statement: READ_BOOLEAN IDENTIFIER
   | READ_INTEGER IDENTIFIER
   ;

output_statement: WRITE expression
   | WRITE STRING
   ;

functiondec: datatype IDENTIFIER LP functiondec_parameterlist RP LEFT_CURLY statements RETURN
expression RIGHT_CURLY
   ;

function_call: IDENTIFIER LP functioncall_parameterlist RP SEMICOLON
   ;
functiondec_parameterlist: functiondec_parameter

   | functiondec_parameter COMMA functiondec_parameter
   ;

functiondec_parameter: datatype IDENTIFIER
   | empty
   ;

empty:
   ;

functioncall_parameterlist: functioncall_parameter
   | functioncall_parameter COMMA functioncall_parameter
   ;

functioncall_parameter: IDENTIFIER
   | empty
   ;

datatype: BOOLEAN_VAR
    | INTEGER_VAR
    ;
declarations: vardec
   | constvardec
   | constdec
   | arrayvardec
   | functiondec
   ;
vardec: datatype vardeclist
   ;
vardeclist: IDENTIFIER
   | IDENTIFIER COMMA vardeclist
   ;
constvardec: CONST_VAR vardec ASSIGN expression
   ;

arrayvardec: datatype LEFT_SQUARE_BRACKET RIGHT_SQUARE_BRACKET vardeclist
   ;
```

```
variable: IDENTIFIER
   | arraycall
   ;

arraycall: IDENTIFIER LEFT_SQUARE_BRACKET INTEGER RIGHT_SQUARE_BRACKET
   ;

assignment: variable ASSIGN expression
   ;

expression: arithmetic_exp
   | logical_exp
   | NOT logical_exp
   ;

arithmetic_exp: arithmetic_exp PLUS term
   | arithmetic_exp MINUS term
   | variable INCREMENT
   | variable DECREMENT
   | term
   ;

term: factor
   | term MULT factor
   | term DIVI factor
   | truthvalue
   ;

factor: LP arithmetic_exp RP
   | variable
   | INTEGER
   ;

logical_exp:  logical_exp IF_BICONDITIONAL logical_term
   |  logical_exp IF_THEN logical_term
   ;

logical_term: logical_and
   |  logical_term OR logical_and
   |  logical_term XOR logical_and
   ;

logical_and: logical_finish
   | logical_and AND logical_finish
   ;

logical_finish: truthvalue
      | LP logical_exp RP
              | variable
              | equality_exp
equality_exp: NOT equality_exp
   | arithmetic_exp ISEQUAL arithmetic_exp
```

```
  | arithmetic_exp LESSTHAN arithmetic_exp
  | arithmetic_exp GREATERTHAN arithmetic_exp
  | arithmetic_exp GREATERTHAN_EQUAL arithmetic_exp
  | arithmetic_exp LESSTHAN_EQUAL arithmetic_exp
  ;

truthvalue: TRUE
   | FALSE
   ;

constdec: CONST STRING ASSIGN truthvalue
  ;
```

Explanations of the BNF

<program>: Program consists of lines.

<lines>: Lines consists of Lines or a line

<line>: A line can be '\n' or a statements and '\n' or Comment

<statements>: Statements can be only one statement and semicolon (;) or a statement followed by semicolon (;) and statements.

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

<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

                    }
```

<dowhile_statement>: Do-While loop statement that executes the contents of do block and check if while's expression is still valid.

<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

<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>;

<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 <string>;

<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
   }
```

<function_call>: Call to the function with the specific identifier and parameter list.

Convention: foo(functioncall_parameterlist);

<functiondec_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).

<empty>: An empty statement which includes nothing.

<functioncall_parameterlist>: It consist of multiple functioncall_parameters.

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

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

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

<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 for assigning values into the fields "<-"

Convention: variable <- expression;

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

<arithmetic_exp>: This is to construct arithmetic operations in our language. Operator precedence among arithmetic operators are saved. This is composed of

- arithmetic_exp PLUS term **or**
- arithmetic_exp MINUS term **or**
- variable INCREMENT **or**
- variable DECREMENT **or**
- term

To prevent ambiguity we decleared operator precedence and associativity. Furthermore, in order to apply these rules, we create sub-rules for maintain the unambiguity in the Language

<term> : This is the layer before factor, multiplication of a term and factor, division of a term and a factor, or a truthvalue.. It completes the transaction between the arithmetic_exp and the layers below

<factor> : This layer holds a arithmetic_ exp In between LP and RP or a variable or an INTEGER.

<logical_exp>: A logical Expression can be represented either logical_exp IF_BICONDITIONAL logical_term or logical_exp IF_THEN logical_term

<logical_term>: Logical_term stands for the logical term representation. It can represent a logical_term, logical_term OR logical_and , or logical_term XOR logical_and. This layer ensures the completeness of the logical expression representation.

<logical_and>: This is to represent an and of logical_term with logical_finish of just logical_finish.

<logical_finish> : This is refer to truthvalue, logical_exp in between parenthesis, a variable, or an equality expression.

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

<truthvalue> : TRUE | FALSE

<constdec> : This constant decleration assigns a truthvalue to a constant string by using defined assignment operator.

Lex Description of the Language:

```
%option yylineno

DIGIT  [0-9]

SIGN [+-]

OR_OPERATOR \|\|

AND_OPERATOR \&\&

XOR_OPERATOR x\|

IF-THEN \=\>

IF-BICONDITIONAL \<\=\>

NOT \!

ISEQUAL \=\=

LESSTHAN \<\<

LESSTHAN-EQUAL \<\<\=

GREATERTHAN \>\>

GREATERTHAN-EQUAL \>\>\=

LETTER [A-Za-z_$]

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

LEFT_PTH \(

RIGHT_PTH \)

LEFT_CURLY \{

RIGHT_CURLY \}

LEFT_SQR_BRACKET \[

RIGHT_SQR_BRACKET \]

ASSIGNMENT <-

TRUE true

FALSE false

TRUTH_VALUE ({TRUE}|{FALSE})

QUOT \"

ALPHANUMERIC ({LETTER}|{DIGIT})

PLUS \+

MINUS \-
```

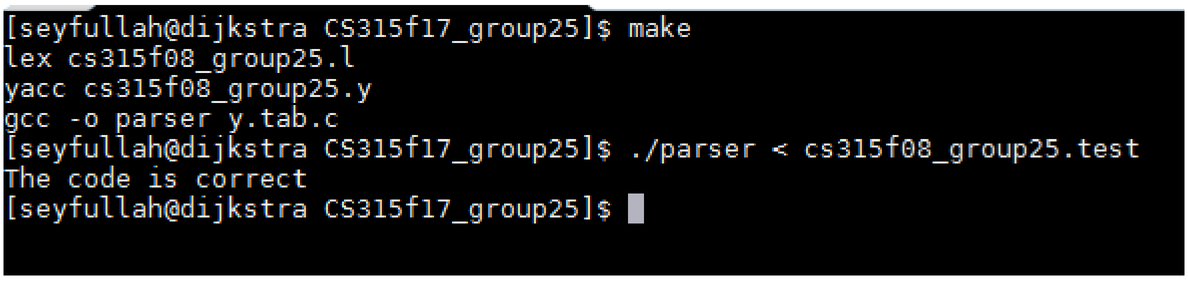
```
MULT \*
DIVI \/
SEMICOLON \;
INCREMENT \+\+
DECREMENT \-\-
WHITESPACE [ \t\n]
WHILE while
FOR for
DO do
IF if
ELSE else
CONSTANT const
READBOOL readbool
READINTEGER readinteger
WRITE write
CONSTANTVAR constantvar
BOOLVAR boolvar
INTVAR intvar
VOID void
PUBLIC public
PROTECTED protected
PRIVATE private
FIXED fixed
COMMENT [\#]([ \t]+|[a-zA-Z0-9]*)*
INTEGER {SIGN}?{DIGIT}+
EXEC exec
MAIN main
RETURN return
LOWERCASE [a-z]
UPPERCASE [A-Z]
IDENTIFIER {LETTER}|({ALPHANUMERIC}|_)*
```

```
STRING \"([^\\\"]|\\.)*\"

%%

{OR_OPERATOR} return OR;

{AND_OPERATOR} return AND;

{XOR_OPERATOR} return XOR;

{IF-THEN} return IF_THEN;

{IF-BICONDITIONAL} return IF_BICONDITIONAL;

{NOT} return NOT;

{ISEQUAL} return ISEQUAL;

{LESSTHAN} return LESSTHAN;

{LESSTHAN-EQUAL} return LESSTHAN_EQUAL;

{GREATERTHAN} return GREATERTHAN;

{GREATERTHAN-EQUAL} return GREATERTHAN_EQUAL;

{LEFT_PTH} return LP;

{RIGHT_PTH} return RP;

{LEFT_CURLY} return LEFT_CURLY;

{RIGHT_CURLY} return RIGHT_CURLY;

{LEFT_SQR_BRACKET} return LEFT_SQUARE_BRACKET;

{RIGHT_SQR_BRACKET} return RIGHT_SQUARE_BRACKET;

{ASSIGNMENT} return  ASSIGN;

{PLUS} return PLUS;

{MINUS} return MINUS;

{MULT} return MULT;

{DIVI} return DIVI;

{SEMICOLON} return SEMICOLON;

{INCREMENT} return INCREMENT;

{DECREMENT} return DECREMENT;

{TRUE} return TRUE;

{FALSE} return FALSE;

{WHILE} return  WHILE;

{FOR} return FOR;
```

```
{DO}  return DO;
{IF} return IF;
{ELSE} return ELSE;
{READBOOL} return READ_BOOLEAN;
{READINTEGER} return READ_INTEGER;
{WRITE} return WRITE;
{CONSTANTVAR} return CONST_VAR;
{BOOLVAR} return BOOLEAN_VAR;
{INTVAR} return INTEGER_VAR;
{MAIN} return MAIN;
{VOID} return VOID;
{EXEC} return EXEC;
{RETURN} return RETURN;
{PUBLIC} return PUBLIC;
{PROTECTED} return PROTECTED;
{PRIVATE} return PRIVATE;
{FIXED} return FIXED;
{CONSTANT} return CONST;
{STRING} return STRING;
{COMMENT}  return COMMENT;
{INTEGER} return INTEGER;
{IDENTIFIER} return IDENTIFIER;
[ \t\r] ;
\n { extern int lineno; lineno++;
            return NL;
           }
. { strcpy(yylval.string, yytext);
   return ERROR;
   }
%%
int yywrap(void){        return 1;}
```

Running the LEX and YACC on Dijkstra

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



```
[seyfullah@dijkstra CS315f17_group25]$ make
lex cs315f08_group25.l
yacc cs315f08_group25.y
gcc -o parser y.tab.c
[seyfullah@dijkstra CS315f17_group25]$ ./parser < cs315f08_group25.test
The code is correct
[seyfullah@dijkstra CS315f17_group25]$
```

Example Program (Test):

```
main{
boolvar bool1, bool2, bool3, boolN;
boolvar bool4;
intvar int1;
intvar var1, var2, var3;
#comment is here

intvar integer1, intger2;
constantvar intvar const_integer1 <- 1234;
constantvar boolvar const_bool1 <- false;
boolvar[] arr1;
boolvar[] arr2;

arr1[0] <- false;
arr2[1] <- arr[0];
bool1 <- true;
bool2 <- true;

readbool bool3;
integer1 <- 5;

# increment and decrement
integer1++;
integer1--;

while(term == 5){
        readinteger integer1;
};

do{
        a++;
}while(a << final);

do{
        a++;
        if(x == y){
                write "done";
        };
}while(a << final);


intvar foo(boolvar bool, intvar inter){
        write bool;
        write integer;
        return 5;
};

boolvar foo2(){
        return false;
};

foo(myBoolValue, 5);
bool1 <- foo2();

if(a << 5){
        write "it is true";
        while(a >>= 5){
                integer3++;
        };
}else{
        write "provide a new value";
};
}
```

Resolving Conflicts and problems in the Language:

In our implementation, we got both shift/reduce and reduce/reduce conflicts. This problem caused due to the fact that we got expressions on the both sides in the aritmetic_expression, in addition, the operator presidences and operation assosiativities are not defined very clear. Therefore we obtain multiple parse trees for one expression in our early implementation.

To resolve the reduce/reduce and shift/reduce conflicts, we defined the operation presidence and operation assosiativity between operators and operations of the Language. Furthermore, in order to aid the reduce multiple parse tree generation, we devide the aritmetic rules in to multiple sub-rules. This help to implement the Language more clearly.

Making the language unambigous was the biggest problem of the language alongside with the writing the grammer to handle such operations.

To summerize, we face with many problems during the definition phase and implementation state of the Language. Due to fact that we have multiple and various operations, declarations, collection and primitive types, it’s our main focus to keep the language maintained and clear, therefore, the biggest effort goes to the resolving conflicts that is generated by the YACC when we tried to compile the Language since there is no special tool to recognize or detect the conflicts and errors, therefore, this operation and resolving step done by manual debugging.