Bilkent Üniversitesi

Programming Languages

Department of Computer Engineering

Project 2 Report

PhysLab

Project Group: 25

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

Supervisor:

- Karani Kardeş

BNF Description

PhysLab Language

PhysLab is a drone specific language with 4 inbuilt types; **int**, **double**, **boolean**, and **string**. PhysLab has been designed with Swift, Java, and C++ languages in mind. It uses the flexibility of Java and simplicity of Swift[1]. The loop statements are similar with the loops of Swift. Moreover, the output and assignment structures are also similar. PhysLab has several built-in functions for drones only. While connect() and disconnect() functions allows drones to connect to the base station, getInclination(), getAltitude(), takePicture(), e.g. functions enables users to get data via drone's hardware.

[1] Apple's modern language designed for iOS and macOS applications in 2014.

Special Tokens

MAIN = “main()”

The comment keyword of PhysLab to start the program. Statements, which is not in this built-in function, will not be executed.

COMMENT = “//”

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.

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 = “->”

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

IN = “in”

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

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).

Conflicts Left Unresolved

1) Shift-reduce conflict on EQUAL, NOTEQUAL, GREATERTHAN, LESSTHAN, GREATERTHANOREQUAL, LESSTHANOREQUAL, AND, OR at printables and logicOperation. Here, . denotes the pointer.

```
printables : logicCondition .
logicCondition : logicCondition . logicOp notNonLogicExp
```

The problem here is that after a PRINT keyword, we always reach outStatement and that is the only way to reach printables nonterminal. So, although this is conflicting with logicCondition, we will parse such an input containing “PRINT logicCondition” as outStatement. Thus, this does not have an ambiguity. Also, after a printables nonterminal, we do not expect a logic operation. So this will be a syntax error. Thus, such a conflicting input can not be given to the parser.

2) Shift-reduce conflict on COMMA at printables and outStatement. Here, . denotes the pointer.

```
outStatement : PRINT printables .
printables : printables . COMMA logicCondition
printables : printables . COMMA STRING
```

Here, the problem is that we won’t use parse by using printable nonterminal only if we have an out statement. Thus, this does not cause any ambiguity.

3) Similarly, Shift-reduce conflict on EQUAL, NOTEQUAL, GREATERTHAN, LESSTHAN, GREATERTHANOREQUAL, LESSTHANOREQUAL, AND, OR at printables and logicOperation. Here, . denotes the pointer.

```
printables : printables COMMA logicCondition .
logicCondition : logicCondition . logicOp notNonLogicExp
```

We have a similar issue as the first conflict. Here, we do not expect a logic operation after printable nonterminal.

Program Rules

```
<program>         ::=   MAIN LB <statements> RB

<statements>      ::=   <statements> <statement> | <empty>

<statement>       ::=   <statementBlockEl> | <ifStatement>

<statementBlockEl>::=   COMMENT | <declaration> | <assignment>
                        | <loopStatement> | <functionCall>

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

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

<forLoop>         ::=   FOR VARIABLE IN <forTerm> FORDOT <forTerm>
                        [<forStepExp>] LB <statements> RB

<whileLoop>       ::=   WHILE <logicCondition> LB <statements> RB

<assignment>      ::=   VARIABLE ASSIGN <assignmentExp>

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

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

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

<functionDec>     ::=   FUNC VARIABLE <parameterExp> [<functionoutExp>] LB
                        <statements> [<returnStatement>] RB

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

<outStatement>    ::=   PRINT <printables>

<printables>      ::=   <printables> COMMA <assignmentExp>
                        | <assignmentExp>

<inStatement>     ::=   VARIABLE ASSIGN SCAN

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

<droneGet>        ::=   GETINCLINATION | GETALTITUDE | GETTEMPRATURE
                        | GETACCELERATION | GETTIMESTAMP

<cameraStatus>    ::=   TURN

<takePicture>     ::=   TAKEPICTURE

<connect>         ::=   CONNECT

<disconnect>      ::=   DISCONNECT
```

```
<returnStatement> ::=   RETURN <assignmentExp> | <empty>

<assignmentExp>   ::=   <logicCondition> | STRING

<logicCondition>  ::=   <logicCondition> <logicOp> <notNonLogicExp>
                        | <notNonLogicExp>

<notNonLogicExp>  ::=   NOT <nonLogicExp> | <nonLogicExp>

<nonLogicExp>     ::=   <functionCall> | BOOLEAN | <arithmeticExp>

<logicOp>         ::=   EQUAL | NOTEQUAL | LESSTHANOREQUAL
                        | GREATERTHANOREQUAL | LESSTHAN | GREATERTHAN | AND
                        | OR

<arithmeticExp>   ::=   <arithmeticExp> <leastPrecMathOp> <multiplyDivide>
                        | <multiplyDivide>

<multiplyDivide>  ::=    <multiplyDivide> <mostPrecMathOp> <number>
                        | <number>

<leastPrecMathOp> ::=   PLUS | MINUS

<mostPrecMathOp>  ::=   MULTIPLY | DIVIDE | REMAINDER

<matched>         ::=   IF <logicCondition> LB (<matched>
                        | <statementBlocks>) RB ELSE LB (<matched> |
                        <statementBlocks>) RB

<unmatched>       ::=   IF <logicCondition> LB <statementBlocks> RB | IF
                        <logicCondition> LB (<matched> | <statementBlocks>)
                        RB ELSE LB <unmatched> RB

<type>            ::=   INTTYPE | DOUBLETYPE | STRINGTYPE | BOOLTYPE

<statementBlocks> ::=   <statementBlocks> <statementBlockEl> | <empty>

<empty>           ::=

<number>          ::=   INTEGER | DOUBLE | VARIABLE

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

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

<functionParam>   ::=   TYPE COLON VARIABLE

<functionoutExp>  ::=   FUNCTIONOUT <functionParam>

<forStepExp>      ::=   BY INTEGER

<forTerm>         ::=   VARIABLE | INTEGER | <functionCall> | LP
                        <arithmeticExp> RP
```

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

<funcCallParam>   ::=   VARIABLE | CONSTANT | INTEGER | DOUBLE | STRING |
                        BOOLEAN
```

BNF Explanations

<main>: Main program statement. Every executable statement has to be in this function (e.g. main(){statements})

<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

<commentStatement>: Non-executable comment sentence (e.g. //Hello!)

<commentSentence>: Character recursive for comment sentence.

<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 <statementBlocks>

<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.

<printables>: Recursion for assignment expression, <assignmentExp>.

<**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 (e.g. "getAcceleration()").

<cameraStatus>: A built-in function allows users to turn drone's camera on/off.

<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.

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

<multiplyDivide>: 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 our 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.

<type>: Allowed primitive variable types; int, double, bool, string.

<statementBlocks>: 0 or many statements

<empty>: To declare an empty program.

<number>: Integer, double, or variable.

<parameterExp>: Write parameters between parentheses

<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. It 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 arithmeticExp 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 type check

Lex Description

```
/*lex.l file for the Project 1*/
%{
#include <stdio.h>
#include "y.tab.h"
void yyerror(char *);
%}

COMMENT                               \/\/.*
MAIN                                  main\(\)
DOT                                   \.
TAB        \\t
NL        \\n
LB                                      \{
RB                           \}
LP                              \(
RP                            \)
COMMA                  \,
ASSIGN                 \<\=
COLON              \:
FUNCTIONOUT        \-\>
FORDOT                                \.\.\.
EQUAL              \=\=
NOTEQUAL           \!\=
LESSTHANOREQUAL    \=\<
GREATERTHANOREQUAL           \=\>
LESSTHAN           \<
GREATERTHAN        \>
AND                \&
OR                 \|
NOT                                             \!
PLUS        \+
MINUS        \-
MULTIPLY      \*
DIVIDE      \/
REMAINDER      \%
INTTYPE                      int
DOUBLETYPE                            double
STRINGTYPE                            string
BOOLTYPE                     bool
BOOLEAN                               (true|false)
FOR                for
WHILE                                 while
IF                                              if
ELSE                                  else
```

```
IN                                                in
BY                                                by
CONST                                   const
FUNC                                    func
RETURN                                  return
SCAN                scan
PRINT               print
CONNECT             connect
DISCONNECT          disconnect
GETINCLINATION                getInclination
GETALTITUDE                             getAltitude
GETTEMPERATURE                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}+
STRING                                  \"(\\.|[^"\\])*\"
VARIABLE                      {low_alphabetic}{ALPHANUMERIC}*
CONSTANT            {upp_alphabetic}{ALPHANUMERIC}*

%option yylineno
%%
{MAIN}                                  {return MAIN;}
{INTTYPE}                     {return INTTYPE;}
{DOUBLETYPE}                  {return DOUBLETYPE;}
{BOOLTYPE}                              {return BOOLTYPE;}
{STRINGTYPE}                  {return STRINGTYPE;}
{GETINCLINATION}    {return GETINCLINATION;}
{GETALTITUDE}                 {return GETALTITUDE;}
{GETTEMPERATURE}    {return GETTEMPERATURE;}
{GETACCELERATION}   {return GETACCELERATION;}
{GETTIMESTAMP}                {return GETTIMESTAMP;}
{TAKEPICTURE}       {return TAKEPICTURE;}
{TURN}              {return TURNCAMERA;}
{CONNECT}                     {return CONNECT;}
{DISCONNECT}                  {return DISCONNECT;}
{PRINT}                                 {return PRINT;}
{SCAN}                                  {return SCAN;}
{BOOLEAN}                     {return BOOLEAN;}
{DOT}                                   {return DOT;}
```

```
{MULTIPLY}                    {return MULTIPLY;}
{IF}                                    {return IF;}
{ELSE}                                  {return ELSE;}
{NL}                                    {return NL;}
{TAB}               {return TAB;}
{FOR}       {return FOR;}
{FORDOT}                      {return FORDOT;}
{WHILE}                       {return WHILE;}
{IN}                                    {return IN;}
{BY}                                    {return BY;}
{CONST}                                 {return CONST;}
{FUNC}                                  {return FUNC;}
{RETURN}                      {return RETURN;}
{LB}                                    {return LB;}
{RB}                                    {return RB;}
{LP}                                    {return LP;}
{RP}                                    {return RP;}
{COLON}                                 {return COLON;}
{COMMA}                                 {return COMMA;}
{FUNCTIONOUT}                 {return FUNCTIONOUT;}
{ASSIGN}                      {return ASSIGN;}
{EQUAL}                       {return EQUAL;}
{NOTEQUAL}          {return NOTEQUAL;}
{GREATERTHAN}     {return     GREATERTHAN;}
{LESSTHAN}          {return LESSTHAN;}
{GREATERTHANOREQUAL} {return GREATERTHANOREQUAL;}
{LESSTHANOREQUAL}   {return LESSTHANOREQUAL;}
{AND}               {return AND;}
{OR}                          {return OR;}
{PLUS}                                  {return PLUS;}
{MINUS}                                 {return MINUS;}
{DIVIDE}                      {return DIVIDE;}
{REMAINDER}                             {return REMAINDER;}
{COMMENT}                               {return COMMENT;}
{INTEGER}                     {return INTEGER;}
{DOUBLE}                      {return DOUBLE;}
{STRING}                      {return STRING;}
{VARIABLE}                              {return VARIABLE;}
{CONSTANT}                              {return CONSTANT;}
{NOT}                                   {return NOT;}
%%

int yywrap(void){
          return 1;
}
```

Yacc

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(char* s);
extern int yylineno;
%}

%token INTTYPE DOUBLETYPE BOOLTYPE STRINGTYPE VARIABLE MAIN INTEGER LP RP LB RB
%token COMMENT CONST FOR IN FORDOT WHILE FUNC RETURN COMMA COLON ASSIGN NOT
%token GETINCLINATION GETALTITUDE GETTEMPERATURE GETACCELERATION GETTIMESTAMP TAKEPICTURE TURNCAMERA CONNECT DISCONNECT PRINT SCAN
%token BOOLEAN DOT MULTIPLY IF ELSE NL TAB BY FUNCTIONOUT EQUAL NOTEQUAL GREATERTHAN LESSTHAN GREATERTHANOREQUAL LESSTHANOREQUAL AND OR
%token PLUS MINUS REMAINDER DIVIDE DOUBLE STRING FUNCTION CONSTANT

%start program
%%

program: MAIN LB statements RB;

statements: statements statement | empty;

statement: statementBlockEl | ifStatement;

ifStatement: matched | unmatched;

loopStatement: forLoop | whileLoop ;

forLoop: FOR VARIABLE IN forTerm FORDOT forTerm forStepExp LB statements RB | FOR VARIABLE IN forTerm FORDOT forTerm LB statements RB;

whileLoop: WHILE logicCondition LB statements RB;

assignment: VARIABLE ASSIGN logicCondition | VARIABLE ASSIGN STRING;

declaration: constantDec | variableDec | functionDec;

constantDec: CONST type COLON CONSTANT ASSIGN logicCondition | CONST type COLON CONSTANT ASSIGN STRING;

variableDec: type COLON VARIABLE
                            | type COLON VARIABLE ASSIGN logicCondition
                            | type COLON VARIABLE ASSIGN STRING
                            | type COLON VARIABLE ASSIGN SCAN
                            ;
```

```
functionDec: FUNC VARIABLE parameterExp functionoutExp LB statements returnStatement RB
                             | FUNC VARIABLE parameterExp LB statements returnStatement RB
                             ;

functionCall: VARIABLE LP RP | VARIABLE LP callParamList RP | outStatement | inStatement | drone_method LP RP;

outStatement: PRINT printables;

printables: printables COMMA logicCondition | printables COMMA STRING | logicCondition | STRING;

inStatement: VARIABLE ASSIGN SCAN;

drone_method: droneGet | cameraStatus | takePicture | connect | disconnect;

droneGet: GETACCELERATION | GETALTITUDE | GETINCLINATION | GETTEMPERATURE | GETTIMESTAMP;

cameraStatus: TURNCAMERA;

takePicture: TAKEPICTURE;

connect: CONNECT;

disconnect: DISCONNECT;

returnStatement: RETURN logicCondition | RETURN STRING | empty;

logicCondition: logicCondition logicOp notNonLogicExp | notNonLogicExp;

notNonLogicExp: NOT nonLogicExp | nonLogicExp;

nonLogicExp: functionCall | BOOLEAN | arithmeticExp;

logicOp: EQUAL | NOTEQUAL | GREATERTHAN | GREATERTHANOREQUAL | LESSTHAN | LESSTHANOREQUAL | AND | OR;

arithmeticExp: arithmeticExp leastPrecMathOp multiplyDivide | multiplyDivide;

multiplyDivide: multiplyDivide mostPrecMathOp number | number;

leastPrecMathOp: PLUS | MINUS;

mostPrecMathOp: MULTIPLY | DIVIDE | REMAINDER;

matched: IF logicCondition LB statementBlocks RB ELSE LB statementBlocks RB
                   | IF logicCondition LB matched RB ELSE LB statementBlocks RB
                   | IF logicCondition LB statementBlocks RB ELSE LB matched RB
                   | IF logicCondition LB matched RB ELSE LB matched RB;
```

```
unmatched: IF logicCondition LB statementBlocks RB
                | IF logicCondition LB statementBlocks RB ELSE LB unmatched RB
                | IF logicCondition LB matched RB ELSE LB unmatched RB;

type: INTTYPE | DOUBLETYPE | STRINGTYPE | BOOLTYPE;

statementBlocks: statementBlocks statementBlockEl | empty;

statementBlockEl: COMMENT | declaration | assignment | loopStatement | functionCall;

empty: ;

number: INTEGER | DOUBLE | VARIABLE;

parameterExp: LP RP | LP functionParams RP;

functionParams: functionParams COMMA functionParam | functionParam;

functionParam: type COLON VARIABLE;

functionoutExp: FUNCTIONOUT functionParam;

forStepExp: BY INTEGER;

forTerm: VARIABLE | INTEGER | functionCall | LP arithmeticExp RP;

callParamList: callParamList COMMA funcCallParam | funcCallParam;

funcCallParam: VARIABLE | CONSTANT | INTEGER | DOUBLE | STRING | BOOLEAN;

%%
void yyerror(char *s) {
        fprintf(stdout, "line %d: %s\n", yylineno,s);
}
int main(void){
 yyparse();
if(yynerrs < 1){
                printf("Parsing: SUCCESSFUL!\n");
        }
 return 0;
}
```

Makefile

```
LEX = lex

YACC = yacc -d


CC = gcc



all: parser clean


parser: y.tab.o lex.yy.o

	$(CC) -o parser y.tab.o lex.yy.o

	./parser < CS315f20_team25.test.txt


lex.yy.o: lex.yy.c y.tab.h

lex.yy.o y.tab.o: y.tab.c



y.tab.c y.tab.h: CS315f20_team25.yacc.y

	$(YACC) -v CS315f20_team25.yacc.y



lex.yy.c: CS315f20_team25.lex.l

	$(LEX) CS315f20_team25.lex.l


clean:

	-rm -f *.o lex.yy.c *.tab.* parser *.output
```

Test Code

```
main(){
 // 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()

 double: level
 level <= scan
 print level

 for iter in 0...2 {
		print "iter: "
  print iter
 }

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

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

```
    int: light
    light <= 7
    print light
    bool: isLight <= false
    bool: isDark <= true

    int: temperature <= getTemperature()
    if isLight & !isDark & light % 2 == 1 {
                    light <= 0
    }

    else {
                    print light
    }
  }

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

  int: dummy1 <= dummy(3)

  func dummy2() {
    int: a <= scan
    int: b <= scan
    if a =< 2 & b < 2{
      print a
  }
    else{
      print b
    }
  }
  dummy2()

  disconnect()
}
```