Matthew Yin

PLC Final

1. Question 1
   1. Refer to FinalQ1.java
2. Question 2
   1. Boolean expressions
      1. <boole> -> < equality > || <equality> | < equality > **&&** <equality> | <equality>
      2. <equality> -> < express > == <express> | < express > != <express> | <express>
      3. <express> -> < cond > **>** <cond> || < cond > **>=** <cond> | < cond > **<** <cond> | < cond > **<=** <cond> | <cond>
      4. <cond> -> < arith > + <arith> | < arith > - <arith> | <arith>
      5. <arith> -> < var > \* <var> | < var > / <var> | < var > % <var> | <var>
      6. <var> -> <int\_lit> | <id> | ( <boole> )
   2. Assignment statements
      1. <assign> -> <id> = <express>
      2. <express> -> < factor > + <factor> | < factor > - <factor> | <factor>
      3. <factor> -> < var > \* <var> | < var > / <var> | < var > % <var> | <var>
      4. <var> -> <id> | <int\_lit> | <bool\_lit> | ( <expr> )
   3. Mathematical Expressions
      1. <bool\_expr> -> < boole > || <bool> | <boole>
      2. <boole> -> < arith > && <arith> | <arith>
      3. <arith> -> < expon > + <expon> | < expon > - <expon> | <expon>
      4. <expon> -> < var > \* <var> | < var > / <var> | < var > % <var> | <var>
      5. <var> -> <id> | <int\_lit> | ( <bool\_expr> )
3. Question 3
   1. Boolean Expressions

/\*

<boole> -> <equality> || <equality> | <equality> && <equality> | <equality>

\*/

void boole(){

printf("Enter <boole>\n");

equality();

while (nextToken == OR\_OP || nextToken == AND\_OP){

lex();

equality();

}

printf("Exit <boole>\n");

}

/\*

<equality> -> <express> == <express> | <express> != <express> | <express>

\*/

void equality(){

printf("Enter <equality>\n");

express();

while(nextToken == EQUALITY\_OP || nextToken == INEQUALITY\_OP){

lex();

express();

}

printf("Exit <equality>\n");

}

/\*

<express> -> <cond> > <cond> || <cond> >= <cond> | <cond> < <cond> | <cond> <= <cond> | <cond>

\*/

void express(){

printf("Enter <express>\n");

cond();

while(nextToken == GREATER\_OP || nextToken == GREATEREQ\_OP || nextToken == LESS\_OP || nextToken == LESSEQ\_OP){

lex();

cond();

}

printf("Exit <express>\n");

}

/\*

<cond> -> <arith> + <arith> | <arith> - <arith> | <arith>

\*/

void cond(){

printf("Enter <cond>\n");

arith();

while(nextToken == ADD\_OP || nextToken == SUB\_OP){

lex();

arith();

}

printf("Exit <cond>\n");

}

/\*

<arith> -> <var> \* <var> | <var> / <var> | <var> % <var> | <var>

\*/

void arith(){

printf("Enter <arith>\n");

var();

while(nextToken == MULT\_OP || nextToken == DIV\_OP || nextToken == MOD\_OP){

lex();

var();

}

printf("Exit <arith>\n");

}

/\*

<var> -> <int\_lit> | <id> | ( <boole> )

\*/

void var(){

printf("Enter <var>\n");

if(nextToken == IDENT || nextTOken == INT\_LIT){

lex();

}

else if (nextToken == LEFT\_PAREN){

lex();

boole();

if(nextToken == RIGHT\_PAREN){

lex();

}

else{

error();

}

}

else{

error();

}

printf("Exit <var>\n");

}

* 1. Assignment Statements

/\*

  <assign> -> <id> = <express>

\*/

void assign(){

printf("Enter <assign>\n");

if(nextToken == IDENT){

lex();

while(nextToken == EQUAL\_OP){

lex();

express();

}

}

printf("Exit <assign>\n");

}

/\*

  <express> -> < factor > + <factor> | < factor > - <factor> | <factor>

\*/

void express(){

printf("Enter <express>\n");

factor();

while(nextToken == ADD\_OP || nextToken == SUB\_OP){

lex();

factor();

}

printf("Exit <express>\n");

}

/\*

  <factor> -> < var > \* <var> | < var > / <var> | < var > % <var> | <var>

\*/

void factor(){

printf("Enter <factor>\n");

var();

while(nextToken == MULT\_OP || nextToken == DIV\_OP || nextToken == MOD\_OP){

lex();

var();

}

printf("Exit <factor>\n");

}

/\*

<var> -> <id> | <int\_lit> | ( <expr> )

\*/

void var(){

printf("Enter <var>\n");

if(nextToken == IDENT || nextTOken == INT\_LIT){

lex();

}

else if (nextToken == LEFT\_PAREN){

lex();

expr();

if(nextToken == RIGHT\_PAREN){

lex();

}

else{

error();

}

}

else{

error();

}

printf("Exit <var>\n");

}

* 1. Mathematical Expressions

/\*

  <bool\_expr> -> < boole > || <bool> | <boole>

\*/

void bool\_expr(){

printf("Enter <assign>\n");

boole();

while(nextToken == OR\_OP){

lex();

boole();

}

printf("Exit <assign>\n");

}

/\*

  <boole> -> < arith > && <arith> | <arith>

\*/

void boole(){

printf("Enter <boole>\n");

arith();

while(nextToken == AND\_OP){

lex();

arith();

}

printf("Exit <boole>\n");

}

/\*

  <arith> -> < expon > + <expon> | < expon > - <expon> | <expon>

\*/

void arith(){

printf("Enter <arith>\n");

var();

while(nextToken == ADD\_OP || nextToken == SUB\_OP){

lex();

expon();

}

printf("Exit <arith>\n");

}

/\*

  <expon> -> < var > \* <var> | < var > / <var> | < var > % <var> | <var>

\*/

void expon(){

printf("Enter <expon>\n");

var();

while(nextToken == MULT\_OP || nextToken == DIV\_OP || nextToken == MOD\_OP){

lex();

var();

}

printf("Exit <expon>\n");

}

/\*

  <var> -> <id> | <int\_lit> | ( <bool\_expr> )

\*/

void var(){

printf("Enter <var>\n");

if(nextToken == IDENT || nextTOken == INT\_LIT){

lex();

}

else if (nextToken == LEFT\_PAREN){

lex();

boole();

if(nextToken == RIGHT\_PAREN){

lex();

}

else{

error();

}

}

else{

error();

}

printf("Exit <var>\n");

}

1. Question 4
   1. The four criteria for proving the correctness of a loop is:
      1. The Preconditional implies that the loop invariant is true.

P -> I

I = {n >= 0}

* + 1. a = 1;
    2. b = 1;
    3. The loop invariant does not change throughout the loop. None of the statements inside the loop affect the value of n, therefore the criteria are satisfied. {n >= 0} && {b <= n}
    4. while (b <= n) {  
        a = a \* x;

b = b + 1;

}

{ a = x ^ n}

* + 1. The loop invariant and not the Boolean expression implies the post condition. {n >= 0} && {b > n} - > {a = xn}

0 <= n < b (b = 1implies that n is equal to 0)

x0 = 1 and a = 1 therefore 1 = 1 🡪 satisfied.

This proves that the loop does terminate.

1. The loop terminates
2. {n > 0}
3. Since b starts at a value of 1 and n is greater than 0, then for some value of y, where b + y > n, the loop will terminate.
4. Question 5
   1. Refer to FinalQ5.java for the rewritten code.
   2. The code with the goto statement is easier to read than the rewritten code. The keyword “goto” is much easier for others to understand its functionality than the keyword “continue” used in my code. The person analyzing the code would have to know that continue means to skip the iteration of the loop. Since there is no other parameters for the function, the functionality of the keyword may be misunderstood.
5. Question 6
   1. The solution with the nested selections provides a higher complexity than the solution without nested selections. The user will have to be able to trace the program with more precision than with the non-nested selection solution. The reliability of the two are about the same, but the nested selection should be more reliable. The non-nested solution is directed towards a more generalized setup for determining the outcome, whereas the nested solution is directed towards each specific situation by comparing each variable with all the others.
6. Question 7
   1. The tombstone method creates a tombstone which is a pointer to the heap dynamic variable (when created) and stays at that memory location even after the heap dynamic variable is deallocated. Locks and Keys approach assigns each heap-dynamic variable a key with which each pointer will need to have the correct key to dereference the corresponding lock. Both methods are about equal in their safety because in both situations an error will arise if deallocated heap dynamic variables are being referenced. The costs are also similar between the two, the tombstone stays inside the memory and cannot be deallocated even if the variable is already deallocated. The address of the Lock and Key approach is equally as bad because the lock value changes and doesn’t allow for anyone to access the information of a deallocated heap-dynamic variable. The address’s value is still stored in memory but is unreachable by any pointers already set to its previous lock value. Both situations are safe because they return an error if any pointers try to reference deallocated memory but are both costly in the fact that each method takes up memory in space after the heap-dynamic variable is disposed.
7. Question 8
   1. Refer to FinalQ8P1.c
   2. Refer to FinalQ8P2.c
   3. Refer to FinalQ8P3.java
   4. The operational semantics for the segment include the for loop and the switch statement. The for loop has 3 conditions inside it’s parameters, expr1, expr2, and expr3. Expr1 is usually assigning the value of a integer variable used to iterate the loop. Expr2 is the conditional statement to see if the loop will iterate through the code inside again or not. Expr3 is the operations that are done on expr1 to make sure that the loop will terminate after a given time. The switch statement operational semantics is checking to see if the parameter passed into the keyword matches any of the cases inside the scope of the switch statement. If the parameter does match, then the code will goto that label and run the code starting from the label.
8. Question 9
   1. Language chosen: Go language
   2. Readability
      1. The language readability is on par if not better than other programming languages. The language is similar to C and Java in the fact that it will not be hard to learn other languages after learning Go Lang’s format. Go Lang compilers are set up in a way that if any variable is unused even after initialization and declaration, it will return an error prompt when the code is executed. The compiler will alert the user that the variable is not used and either needs to be used or removed, which will improve readability because it removes from the source code any unnecessary variables. Go Lang also doesn’t allow for function overloading; therefore, programmers don’t need to worry about which function is being used because there is only one definition of the function which will also decrease the amount of code in the file. This will allow for less clutter in the file, making it easier to read the code. The readability of Go’s keywords is not much different from other languages; therefore, users will not be confused if a certain keyword executes differently than expected. One of the downsides to Go Lang’s readability is the fact that the language is case sensitive, so if the code contains variables such as x and X, then the two letters may contain different values which may be confusing to users.
   3. Writability
      1. The writability of the language consists of different ways to accomplish a single task. For example, if a programmer wanted to declare a variable of type int, the programmer can specifically initialize the variable of type int and variable name x with the value of 2. However, if the programmer knows that the value of x should be 2, then the programmer can also implicitly declare the value of x as an integer by setting the variable x’s value to 2 and using the “:=” symbol declared by Go Lang. The “:=” symbol states that the variable name will implicitly inference the data type of the value and use that to declare the variable. In this case, Go Lang would be better than other languages in writability’s expressivity. Go Lang is a strongly and statistically typed language which does not allow for type changes after a variable has been declared and provides limitations on type conversions between variables. This may be a downside because users will have to be careful when declaring variables and will need to consider future use of the variable after declaration. The option of type casting is available to programmers to work around this, but users will need to remember the syntax for it.
   4. Reliability
      1. Go Lang provides a better than average reliability than other languages, since it was developed and launched in 2009, which is more recent than the other popular languages such as Java and C. Since the language provides a higher expressivity than other languages, the compile time necessary for code is shorter than other languages. The language provides for type casting, which programmers will need to do if they want to convert a data type to another inside the code. The compiler does not implicitly type cast for the programmer, which helps aid in the language’s type checking reliability. The language uses error handling which allows users to declare the error that occurred using the errors.New method. This method allows programmers to printout what error has occurred in the code due to bad parameter passthrough. The reliability allows for aliasing, which is an upside because it allows users to access the same memory location with different variables. One of the advantages to Go Lang’s reliability is the fact that the language includes an inbuilt garbage collection. Since Go Lang accounts for pointers, there should be no issue for any dangling pointer problem because of the garbage collection which will clear up space. Since the readability and writability is on par if not better than popular languages, a programmer will have an easier time debugging and modifying any written code.
   5. Keywords
      1. Go Lang’s choices for reserved words seem to indicate exactly what they are named after. One of the downsides of Go Lang compared to other languages is the fact that there is no keyword “while” in the language whereas other languages include the while loop. This may cause troubles for programmers who are starting to learn Go Lang and want to incorporate a while loop into their code. Another downside to Go Lang’s keywords is that the data types are not included as keywords, in the code they appear to be written in blue text to appear as a keyword, but they aren’t. The choice of keywords should incorporate the data types, because once the code becomes more complex and has more difficult readability this variable name will make the code more confusing.
   6. Data Types
      1. Go Lang seems to have a decent selection of Data types in the language, compared to their Keywords. The language incorporated 4 main types that branch off into multiple different types. The four main types are Boolean, Numeric, String, and Derived types. Each one breaks into smaller definitions that will end up incorporating mostly everything necessary for data types. The problem with having everything incorporated into the language as a specific data type is the fact that the language is a strong and static language. The only way for data types to change in value is by type casting the intended data type onto the variable.
   7. Control Structures
      1. The control structures are very similar to other languages in that it includes if, for, and switch statements. The one different between languages is the fact that Go Lang does not include a while loop but instead a select statement for different channel cases. Since there is no while loop that means there is also no do while loop, therefore programmers will have to think of a way to get around that dilemma. Also, switch statements inside Go Lang are not programmed the same as other languages, after each case a break is not needed inside Go Lang. The language implemented the switch statement with a break after each case, but if the programmer wanted the code to continue running into the next case that is also an option with the fallthrough keyword.
   8. Expressions
      1. Expressions in Golang rely on operator precedence the same as other languages would. The only difference is that since Golang has the possibility for bitwise operations the bitwise operators are included in the operator precedence. Expressions are evaluated like any other programming language except for the fact that bitwise operations are executed before logical and assignment expressions. Golang does not include any ternary operators and only includes the simple unary and binary operators.
   9. Compared to Java
      1. Compared to Java, Golang is quite similar. The language itself compiles faster and has less clutter in the workspace due to the higher expressivity and compile-time checking. Another difference between the two languages is the fact that Java has error and exception handling where as Golang only includes error handling in its implementation. This may be a downside to Golang because the user will have to know what error might happen at the specific code they’re testing on. Golang also changed their switch statement so that each case breaks after the statements are executed instead of naturally falling through to the next case like Java. However, Golang does provide the keyword if the programmer wanted the code to fallthrough to the next case. Java also has two separate loop functions, one for a for loop and another for a while loop. Golang has decided against that and seems to have coded the for loop in a way to incorporate both functions into one keyword to make the code simpler. The variable declaration is different between the two because Java needs to specifically declare if the variable is an integer or a double without using a user-defined function. Golang can accomplish this easily with the use of the “:=” operator implemented in the code, which allows for implicit declaration of a data type for a statement like variable := value. Another difference between Golang and Java is that Java needs to specifically code a semi-colon after every line, whereas Golang does this implicitly in the background.
   10. Syntax and Semantics
       1. The syntax of Golang seems to flow easier than other languages. For example, to declare a variable you must type the following: var variable\_name variable\_type = value. The first token is telling the user that it’s a variable, the second one is the name of variable, and the third token is the type, which is concluded by the value of the variable. The syntax is a lot easier to read and understand what the specific line of code is doing. Another example of the syntax is with a function header, which contains: func function\_name (parameter\_list) [return\_types]. The function declaration can be easily understood just like the variable declarations. The syntax is very similar to that of a specification list of a new product, where it contains the name of the product and all of the details afterwards. The semantics of Golang is as straightforward as it can get, all of the keywords perform a task directly with the specified name. For example, the “fallthrough” keyword allows for switch statement cases to fallthrough the current case into the next one.

**GitHub Link**

1. <https://github.com/myin3/PLC.git>

**Sources**

1. <https://www.tutorialspoint.com/go/index.htm>
2. <https://tour.golang.org/flowcontrol/12>