Christopher Fulton

Final Exam
Programming Language Concepts
May 4, 2020

All Written Code Found on GitHub Here:

https://github.com/cfulton5/PLC-Final/

1) Listed within GitHub linked above.

2) BNF for boolean expressions, assignment, and mathematics for Java

```
<assignment> => <variable> “=” <expression> ;
<variable> => <letter> | <letter> + <variable>
               | <digit> + <variable>
<expression> => <number>
          |<number> <operator> <expression>
          |”(“ <expression> “)”
<operator> => “*” | “/” | “+” | “-“ | “%” | “>” | “<” | “==”
               | “<=” | “>=” | “!=”
<number> => <int_literal> | <float_literal>
<float_literal> => <int_literal> “.” <int_literal>
<int_literal> => <digit> | <digit> + <int_literal>
<digit> => 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<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|
```

3) Listed within GitHub linked above.

4) The four criteria to explain correctness of "while B do S end":

- P => I : Showing that the loop invariant is true.
- {I and B} S {I} : Loop invariant isn't changed by executing the loop.
- {I and (not B)} => Q : Q is true based on I being true and B being false.
- The loop terminates : Showing that the loop ends after execution.

```
a=1;
b=1;
while( b <= n ){
     a = a * x;
     b = b + 1;
}
{ a = x ^ n }
```

The loop invariant b is true before the loop commences; it is assigned the value of 1.
The loop invariant is stable and true throughout the loop and is not altered to be false within the loop.
To prove b <= n eventually is false:
b is assigned to 1;
b adds 1 each loop;
b is at some point all possible integers until n (1,2,3,...,n);
There exists an integer n+1 where the loop terminates.

5) Code listed within GitHub linked above;
The code shown in the CACM segment uses goto statements to move around the program. This is far more difficult to keep track of than the method written in Java without goto statements. Without these statements, one can easily move procedurally through loops, where goto statements makes the reader move back and forth, up and down.

6)   Code listed within GitHub linked above;
Using nested if/else statements is far more of a complex and less-readable format than the alternative of multiple-parameter if statements for each case. Where nesting has many lines of code to explain one situation in the problem, a multiple-parameter if statement sums it up quickly and concisely. Time complexity for both solutions will be identical, as there aren't any loops involved and there is a finite set of numbers. Reliability as far as code goes is identical, but from a user point of view, understandability of the non-nested solution is a more "reliable" option.

7)   #referencing definitions from chapter 6 in textbook#
Tombstones are areas in memory that act as a constant pointer within the memory. They point to dynamic variables in the heap in order to avoid the dangling pointer problem. The pointer doesn't point to the variables, and instead to the tombstone so they never point to unallocated space.
The locks-and-keys method uses integer codes for heap-dynamic variables to compare to integer codes for the pointer being used. If they match, the data is accessed; If not, there is a runtime error.
The tombstone method appears to be the safer method, as the pointer simply points to a null valued tombstone when data is not allocated properly, where the locks-and-keys method performs and illegal action. For implementation cost, the tombstone method is far worse, however, as it is always allocating space (null) no matter if the heap-dynamic variable is unallocated. The implementation cost is less for the locks-and-keys method as it has a dynamic approach to solving the dangling pointer problem.

8)   a)Code listed within GitHub linked above;
     b)Code listed within GitHub linked above;
     c)Code listed within GitHub linked above;

d)

Operational Semantics:

```
     j = -3;
     i = 0;
     loop: if i<3 == 0 goto out;
          if j+2 == 3 goto skip;
          if j+2 == 2 j = j – 1 goto skip;
          if j+2 == 0 j = j + 2 goto skip;
          j = 0
          skip:
          if j>0 == 1 goto out;
          j = 3 – I;
          i = i+1;
          goto loop
     out:
```

The program executes as follows:

```
     set j=-3
     set i=0
     repeat the following loop as long as i<3 is true
          if j+2==3, nothing
          otherwise, if j+2==2, set j=j-1
          otherwise, if j+2==0, set j=j+2
          otherwise, set j=0
          if j>0 is true, exit the loop
          set j=3-i
          set i=i+1
          repeat loop
```

9) Seed7 (released in 2005, 15 years old)

Seed7 is decently similar to other object-oriented programming languages, however it includes much more extendable functionality with user-defined statements and operators. For readability, Seed7 is surprisingly easy to read. Functions, procedures, and variables are easily distinguished from each other, and with the easy ability to create new operators and statements, the programmer can very easily read their own code. This, however, could bring up issues when sharing or selling code since user-defined operators may not be clear to a third-party reader. This problem is, as with almost all languages, negated by proper commenting and documentation. Even as someone with not many languages under his belt, Seed7 is surprisingly easy to read.

Writability is in a similar vein, as many of the keywords are quite easy to understand and utilize, ex. "func" to start a function and "end func" to end it. Many of the keywords are similar to other programming languages, using the standard if, while, and for keywords as one would expect, but includes others that can act as operator symbols, introduce statements, or since it is a largely user-extensive language, many keywords can be introduced through newly defined statements. Some issues that could come from user-defined keywords could be lack of shareability, where one user expects a word to be free when it is built to be a keyword by another user. In Java, keywords are defined clearly within their compiler and hard-coded to not break the syntactical sense of the language.

Writability is also impacted by control structure, where programs could act sequentially, conditionally, or iteratively. Using the standard control scheme built in to Seed7, it acts sequentially with small breaks for different statements (like if statements or while loops). While Seed7's main goal is to be user augmentable, the control structure can be altered by introducing new statement definitions that act in a different manner to the standard way one would expect. Java follows a similar idea of control structure, barring the lack of ability to introduce newly defined statements described previously.

Data typing is one of the key draws to Seed7, where most data typing is abstract. This helps with code reliability, where having data only resolved to a type once the code is compiled allows users to manipulate it differently and act more in accordance with many of the user-defined statements (a popular example being a string array with char indices). In other languages like Java, data types are hard coded into their compiler, meaning that arrays have integer indices and parameters are expected to be a in accordance to a concrete ruleset. Java uses interface data typing, where data is set during runtime.

Expressions within Seed7 are handled in a way similar to most object-oriented program languages- contributing to the reliance of the programming language. Using their recursive descent parser, the compiler for Seed7 searches respective libraries for different objects when regards to unary, binary, or trinary expressions. These, as with most aspects of the language can be altered as well, to fully allow customization for such bit operations. Assignment is handled in the same way it handles data typing, where the assignment expressions are abstract as well. For logic and order of operations, Seed7 acts predictably and therefor reliably, using the standard logic and order that most programmers expect when utilizing a new language. The primary difference with Java and Seed7 when it comes to expressions is the lack of abstract data typing affecting assignment operations and the lack of ability to add new definitions for statements that could alter expressions for a user.

Syntactically, Seed7 breaks down key words and expressions when given a statement to parse. S7SSD(Seed7 Structured Syntax Description), the parser that the language uses, only recognizes one non-terminal symbol: (). This means that any statement or expression can be replaced by the symbol () when analyzing syntax between the key words that it checks for. When syntactically Seed7's parsing is deciding the type for an expression, it refers to semantics- whether the code makes sense when it parses. This helps avoid any hiccups involved with user-defined statements so the parser can properly interpret the syntax. After determining expressions and statements from keywords and function, the syntactical analyzer assigns the priority and associativity of the remaining expressions. It defines priority using low to high integers to determine what order of operations to execute in. For example,

```
     A = B + C * D    =>      A=(B+(C*D))
```

-from Seed7 syntax manual

This expression defined “*” as priority of 6, “+” as priority of 7, and “=” as priority of 12. Since lower priority, executes first, the example above is the result. Similar rules are applied when assigning associativity to these expressions, and associates operands from the left to the right, aiding the semantic understanding for the programming language.

Semantically, the syntax analyzer is slightly overlapped with the semantic coherence of the statements and expressions. The parser will try to analyze the semantics to determine the abstract components like data type due to their setting once compiling starts instead of at runtime. Expressions and statements are all paired with key words that have function and associativity rules to them, and semantics are extremely changeable when introducing user-defined statements into their programs.