Jonathan Baird

All citations refer to the chapter PDFs uploaded to icollege. Specifically those that are almost entirely black text on a white background with very few pictures and are all bullet points.

1. Answer in q1.py, it uses python as the language to do the processing and is based on java’s rules for the literals.

2. Operator precedence from <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>

<bool\_expr> → ‘!’<expr>

|<expr> (‘>’|‘<’|‘>=’|‘<=’|’instanceof’) <expr>

|<expr> (‘==’|‘!=’) <expr>

|<expr> ‘&&’ <expr>

|<expr> ‘||’ <expr>

<assignment> →<var> (‘=’|‘+=’|‘-=’|‘\*=’|‘/=’ |‘%=’|‘&=’|‘^=’|‘|=’|‘<<=’ |‘>>=’ |‘>>>=’ ) <expr>

<math> → <expr> ( ‘\*’|‘/’|‘%’) <expr>

| <expr> (‘+’|‘-’) <expr>

3. answer in q3.c. I used rda.c and front.c as a base and modified it from there to work with my current ruleset. My code starts at the bottom under the comment saying “/\* start of my work for the three new rules\*/”

4. The four criteria for proving loop correctness are, Precondition implies the loop invariant, if the Invariant and the Boolean are true, the statement is executed and I remains true after, if the invariant is true and the Boolean is not then the loop terminates and then the post condition is true.

I = b <= n+1

B = b <= n

P = a = 1;b=1

Q = {a = x ^ n}

P implies I because there is no precondition provided so we can use I as the precondition.

If I and B then S then I is true because the statement inside of the while loop never has the opportunity to increase b past n+1 since the loop ends at b = n, b is then incremented by one giving it the value of n+1.

I and not B then Q is still true, for this to happen n must be equal to b – 1, using the opening assignments of a and b this gives us n’s value of 0, now we know in math that anything raised to the power of 0 is one. So the post condition will always be true because 1 = x ^ 0 and a is set to 1 at the start.

The loop will terminate because b will eventually reach n+1 and no longer satisfy the Boolean.

Rules from chapter 3 PDF page 48.

5. Code in q5.java. I used a break statement to break out of the outer most for loop. I also wrapped the break statement in an if statement that checks a Boolean I created. That Boolean is inited inside the outermost loop before anything else that way it maintains its true value at the start of every new array. The original if statement is used to set the new Boolean to false if a non zero is found thus not executing the break.

I feel that the goto method used in the original code is far more readable than having to insert a new if statement and Boolean value. The goto statement also kind of just acts like a normal function call that we are all used to seeing as it points to a different part of the code entirely.

6. Code in q6.python. The nested solution is probably more complex and definitely less readable than the non nested solution. The non nested solution is probably more reliable as it iterates through every possibility sequentially with no skipped ones unlike the nested one where half of the possibilities are skipped based on the outcome of one Boolean. If there was a problem with multiple solutions the nested one would probably be more reliable and less likely to overwrite one answer with another if the same conditions are met.

7. Tombstone seems to be the more secure of the two because when it’s corresponding variable is deallocated its value is set to nil and that deallocated variable can never be recovered however this also leads to it using more space than other methods. Tombstone also requires extra levels of indirection. Lock and key is harder to implement but doesn’t cause the same loss of space that tombstone does and does not have the extra level of indirection that tombstone has. However this method does still keep the address of deallocated memory cells but those cells no longer have a valid key assigned to them and thus cannot be accessed. Information from chapter 6, pages 65-67.

8. Code in files 8a.c, 8b.c, j8c.java

d. j = -3

for(i=0; i < 3; i++)

evaluate(i=0);

loop = control = evaluate(i<3)

if control == 0 goto out

switch(j+2)

case3:

case2:

j--;

break;

case0:

j += 2;

break;

default:

j = 0;

if (j>0){

break;

}

j = 3- i

goto loop

out:

9. For this question I will be comparing rust and java.

Readability: One impact is that declaring variables has many ways to do it unlike java’s strict variable declaration. For example here is a rust declaration “let a = 1.0f64” or “let a = 1.0” or “let a: f64 = 1.0” here are 3 ways to declare the exact same thing. In java you simply do “float x = 1.0” theres no other way to do it, its just that. Rust also uses curly brace segmenting like Java which in my opinion improves the readability of the code versus a system like pythons indentions as you can quickly at a glace check to see what’s in the curly braces and then you know what’s within a function. The keywords are pretty straight forward in what they do and isn’t too different from c based languages. It also shares a lot of syntax with C/C++.

Writability: The ability to change how you declare things does allow you to do implied typing like python which can improve writability. You can also overshadow variables by declaring them again, this can be used to change their type, you cannot do this in java. Rust also doesn’t need explicitly declare return types for functions like java does. As mentioned earlier the syntax is very similar to C/C++ and its keywords are straight forward and easy to understand if you have prior coding knowledge. One interesting feature is that you can actually still use reserved keywords where you normally wouldn’t be able to by giving it the r# prefix. Allowing you more wiggle room on variable/function/whatever names.

Reliability: Rust was written to be as safe and reliable as possible. It doesn’t allow for null pointers, dangling pointers, or data races. Java implicitly dereferences so dangling pointers are impossible same as Rust(Chapter 6, 62). Rust does not however use automatic garbage collection like java, instead it uses resource acquisition is initialization or RAII with optional reference counting. The language was also designed around maintaining safety in concurrent processes. Rusts raw identifier system allows for backwards compatibility with earlier versions of rust with a different set of keywords.

Keywords: Rust’s keywords like most other languages cannot be used as identifiers except if they are prefixed with r#. If they are prefixed they become raw identifiers and can be used as an identifier. This is important for version changes that add new keywords that weren’t present in previous versions because it allows you to make those keywords into identifiers. You cannot do this in java.

Data Type: Rust has signed and unsigned integers, floating point, char, bool, and the unit type who can only be an empty tuple. You can explicitly define the size of these types at declaration from 8-128 and have it be defined by the pointer size as well. You cannot do this in java. There are also array and tuple types as well. Integers can be expressed as hex, octal or binary. You can also have underscores between numbers to denote separations just like java. For the numeric data types you can also use the size types mentioned earlier as pre and post fixes in literals and not just when assigning them to variables. Tuples are basically arrays of whose entries have different types. Arrays are collections of objects who share the same type and have a size defined at declaration. Rust also has user definable types in structs and enums. Structs have a few types, unit structs which are field less, c style structs, and tuple structs which are the same tuples mentioned previously. These types similarity to C is and other languages like Java aid in its readability and writability since Java and C are massive languages across the planet. You can also cast between types and you can even give aliases to types to heavily aid in the readability and writability of complex programs as it lets you name them to things that will be more beneficial for you.

Control Structures: If else statements are similar to other languages however the condition doesn’t need to be surrounded by parenthesis (improving writability), each condition is followed by a block, and all branches must have the same return type. Return types will be covered in expressions. Loops are very similar to while loops from Java except loops are inherently infinite unless ended with break or continue statements. While loops are extremely similar to java with minor syntax differences in that the condition doesn’t require wrapped parenthesis’s. For loops in rust are nearly identical to those in python except for Rust’s use of curly braces to denote blocks. There is an interesting control method called match, which is a bit like a switch case but much more condensed than the bulky switch cases of java and other C languages. If let is like the match statement but works for enum types. While let is similar to if let but it’s just a different way to handle awkward match situations.

Expressions: Every expression ends with a semi colon. Blocks are also expressions and can be used in assignments, so instead of having to define a return value or return type code blocks will automatically return the end value of last line as long as the last expression of the block doesn’t have a semi colon after it.

Assignment: Some of this has been covered already but Rust has many options for assignment. You can let the complier decide the type of an assignment by leaving out the type, you can define the type on the left hand side or you can define the type on the right hand side as a post fix to the value.

Order of operations: Rusts OOO are similar to that of other C-Like languages. Its differences in java come down to Rust’s exclusion of some operations and condenses down some of the rules to have the same precedence. Rust uses left to right associativity with the exception of assignment operations.

Logic: Logic expressions seem to be no different than java or any other c based language.

Unary, binary, trinary, combinations: Unary operations are allowed in the form of – (negation), ! (logical or bitwise complement), \* (deference), & (borrow). I saw no mentions of trinary in the rust docs so I cannot speak it here.

Overall from this session of research and analysis of Rust I feel it has some good readability and writability features like the ability to do implied typing and explicit, the ability to overshadow variables to change types of previously made variables letting you over write variables you are done with. It has superb reliability because it was built from the ground up to be reliable and it also has keyword backwards compatibility in it with the raw identifier system. It is definitely more readable and writable than Java.

Sources on rust(<https://en.wikipedia.org/wiki/Rust_(programming_language)>, <https://doc.rust-lang.org/book/>, <https://llogiq.github.io/2016/02/28/java-rust.html>)