John Pennig

Homework 6

1. Chapter 3 Problem Set

23. a. $b > 3/2$

b. $c > 10$

c. $b > 1 - a/2$

d. $y > 6 – x/2$

24. a. $b < 1$

b. $b > (1-a)/2$

25. a. $a > 0$

b. $x < 0$

c. $x > 4/3$

26. The four criteria for proving correctness of a logical pretest loop is making sure that an invariant is true before execution of the while loop. Next, the invariant must not be modified during the loop process. Then the loop must eventually terminate satisfying the loop end criteria. Finally, the loop terminates and the postcondition must be satisfied.

28. To prove the program correct, we must verify that the invariant maintains truth throughout the execution of the program. The invariant which is the sum = the values from n to the current count + 1. Sum = 0 is correct because the value of n is less than count + 1 so

therefore its empty as Sum = 0. Now every iteration of the loop, sum is updated as current sum + count, which is still true to the invariant of values from n to the current count + 1 which is just the value of n. In every succeeding loop the value of sum is equal to the current sum + count which is still true to the invariant from n to the current count + 1. This continues until the count is = 0 in which sum holds the values of n + (n – 1) + ... + 1 where it finally terminates when count = 0. Therefore, the invariant holds before, during, and after the loop iterations. The loop condition does become false when count = 0, and the sum is then equivalent to the sum of the numbers from 1 to n.

2. **What problem is being solved in the video?** The problem being solved in the video is verifying a java program meets its specification. Is a program equal to its specification. **How do we formally specify that an array A is an exact copy of array B?** They use JML to verify that the arrays are an exact copy. They do this by checking that at each array index that the exact value is in both arrays at that index. They also make sure the arrays are the same length and that the index being checked is within the range of the array.
List all the errors that the programmer is introducing into the program, and for each error, briefly explain how it is being detected when we have a complete formal specification.

Errors: 1. The programmer makes array b 1 index shorter than array a. Therefore, the arrays are not able to be exact copies because the last array index is not being copied because it is impossible with a shorter array. It is being detected because the array lengths are not equal to eachother.

2. The programmer attempts to copy an index past the length of both arrays. The error is caught because the end index must be less than the length of the array.

3. The programmer attempts to copy from an index beginning at -1. The error is caught because the beginning index must be equal to or greater than 0.

4. The programmer makes an infinite loop by making k never increment. This is checked by making sure that the end value and k get closer each iteration.

5. The programmer makes the value at the current index of array b = to the index – 1 of array b. This error is caught because the value of array a at index i is not equal to the value of array b at index i.

6. The programmer modifies the loop condition making it not valid. The error is caught because the loop invariant is not maintained through execution of the loop.

7. The programmer modifies the starting and end indexes making them not valid again. Error is caught numerous ways by making sure start index is less than the finishing index.

3. Operational and Denotational semantics are both methods for defining the meaning of a programs. Operational semantics describes the execution of a program in a series of steps and provides a concrete, rule-based explanation of how a program behaves. Denotational semantics is a more abstract method that takes away details and attempts to map programs to mathematical functions that describe their overall effect. Operational semantics is all in the details and denotational semantics is all in the result. The two are

related by the semantics of the overall program. They are both used to prove the equivalence of implementations of a language.

4. Denotational semantics shows how a programs syntax makes up the semantics using mathematical representations. Such as the statement a = 1. It'll show that a is variable that maps to a decimal literal of 1.

5. Denotational semantics uses the state of a program to describe meaning. State changes are defined by mathematical functions in Denotational semantics. The state of a program is represented as a set of ordered pairs. VARMAP is a function of two parameters, a variable name and the program state.

6.

Denotationa Semantics notation	Explanation
M_a (x = E, s) :=	M_a denotes "the meaning/effect of the assignment," x=E is the assignment, and s is the state of the program before the assignment is done.
If (M_e (E, s) == error), then error	This statement M_e(E,s) == error checks if when E is evaluated, if results in an error.
else s' = {<i_1, v_1'>, <i_2, v_2'>, … ,<i_n, v_n'>} Where For j = 1, 2, …, n	If there is no error, it evaluates s' which is the new state of the machine to ordered pairs, i is the name of the variable and v' is the new value of that variable. j is the value of the index of the variables 1 through n in the state.
If (i_j ==x) Then v_j' = M_e(E,s)	If the variable i_j is x, then the new value of v_j is set the result of the expression M_e(E, s).
Else v_j' = VARMAP(i_j, s)	If the variable i_j is not x, then the values remain the same.

7. The abbreviation LL means left-to-right, leftmost derivation. In the context of grammars, this means that the parser processes input from the left to the right, building the parse trees using a leftmost derivation. It selects process rules based off the current input symbol and uses a stack to store other symbols. LL parsers are also referred to as recursive-parsers or top-down parsers and are often more simpler but can be slower for large languages. The abbreviation LR means left-to-right, rightmost derivation which is the reverse of LL. In the context of grammars, this means that the parser process input also from the left to the right but builds the parse trees using rightmost derivation. The rightmost nonterminal in the derivation is used to expand backwards until the original grammar is found. LR parsers can handle more complex grammars but are more complex themselves.

8. **First line:** The first line of the trace starts out with the stack in state 0. It takes in input "id + id * id $". The first id is read which is to shift 5 which is derived from the LR parse table. The parser pushes state 5 onto the stack leaving the input reading "+ id * id $".

Second line: The second line of the trace starts with the stack reading 0id5. The input remaining is "+ id * id $". The id in the stack is read and uses reduce 6 since we are in state 5. The parser uses a production rule to transform id into F using GOTO(0,F).

Third line: The third line of the trace starts with the stack reading 0F3. The input remaining is "+ id * id $". The parser uses reduce 4 since we are in state 3. The parser uses a production rule to replace F with T using GOTO(0,T).