Work Time: 2hours and 15 minutes

If a problem implementation does not compile or does not run you will get 0 points for that problem (that means no default points)!!!

1. **(0.5p by default). Problem 1: Implement Conditional Assignment statement in Toy Language.**

   a. (2.75p). Define the new statement:

   v=exp1?exp2:exp3

When exp1 is true the value of exp2 is assigned to v. Otherwise v takes the value of exp3.

Its execution on the ExeStack is the following:

- pop the statement
- create the following statement:

if (exp1) then v=exp2 else v=exp3

- push the new statement on the stack

The typecheck method of conditional assignment statement verifies if exp1 has the type bool, and also the fact that v, exp2, and exp3 have the same type.

b. (with a working GUI 1.75p, with a working text UI 0.25p). Show the step-by-step execution of the following program. At each step display the content of each program state (all the structures of the program state). The step-by-step execution must be displayed on the screen and also must be saved into a readable log text file. The following program must be hard coded in your implementation:

```
Ref int a; Ref int b; int v;
new(a,0); new(b,0);
wh(a,1); wh(b,2);
v=(rh(a)<rh(b))?100:200;
print(v);
v= ((rh(b)-2)>rh(a))?100:200;
print(v);
```

The final Out should be {100,200}

2. (0.5p by default). Problem 2:Implement a CountDownLatch mechanism in ToyLanguage.

a. (0.5p). Inside PrgState, define a new global table (global means it is similar to Heap, FileTable and Out tables and it is shared among different threads), LatchTable that maps an integer to an integer. LatchTable must be supported by all of the previous statements. It must be implemented in the same manner as Heap, namely an interface and a class which implements the interface. *Note that the lookup and the update of the SemaphoreTable must be atomic operations, that means they cannot be interrupted by the execution of the other PrgStates. Therefore you must use the lock mechanisms of the host language Java over the LatchTable in order to read and write the values of the LatchTable entrances .*

b. (0.5p). Define a new statement

newLatch(var,exp)

which creates a new countdownlatch into the LatchTable. The statement execution rule is as follows:

Stack1={newLatch(var, exp)| Stmt2|...}
SymTable1
Out1
Heap1
FileTable1
LatchTable1
==>
Stack2={Stmt2|...}
Out2=Out1
Heap2=Heap1
FileTable2=FileTable1
- evaluate the expression exp using SymTable1 and Heap1 and let be num1 the result of this evaluation. If num1 is not an integer then print an error and stop the execution.
LatchTable2 = LatchTable1 synchronizedUnion {newfreelocation ->num1}
if var exists in SymTable1 and has the type int *then*
SymTable2 = update(SymTable1,var, newfreelocation)
else print an error and stop the execution.

Note that you must use the lock mechanisms of the host language Java over the LatchTable in order to add a new latch to the table.

c. (0.5p). Define the new statement

await(var)

where var represents a variable from SymTable which is mapped to a number into the LatchTable. Its execution on the ExeStack is the following:
- pop the statement
- foundIndex=lookup(SymTable,var). If var is not in SymTable or it has not the type int then print an error message and terminate the execution.
- *if* foundIndex is not an index in the LatchTable *then*
print an error message and terminate the execution
elseif LatchTable[foundIndex]==0 *then*
do nothing

else push back the await statement(that means the current PrgState must wait for the countdownlatch to reach zero)

d. (0.5p) Define the new statement:

countDown(var)

where var represents a variable from SymTable which is mapped to an index into the LatchTable. Its execution on the ExeStack is the following:

- pop the statement
- foundIndex=lookup(SymTable,var). If var is not in SymTable or it has not the type int then print an error message and terminate the execution.
- *if* foundIndex is not an index in the LatchTable *then*

print an error message and terminate the execution

elseif LatchTable[foundIndex] > 0 *then*

LatchTable[foundIndex]=LatchTable[foundIndex]-1;

write into Out table the current prgState id

else write into Out table the current prgState id

Note that the lookup and the update of the LatchTable must be an atomic operation, that means they cannot be interrupted by the execution of the other PrgStates. Therefore you must use the lock mechanisms of the host language Java over the LatchTable in order to read and write the values of the LatchTable entrances .

e.(0.75p). Implement the method typecheck for the statement newLatch(var, exp) to verify if both var and exp have the type int. Implement the method typecheck for the statement await(var) to verify if var has the type int. Implement the method typecheck for the statement countDown(var) to verify if var has the type int.

f. (1p). Extend your GUI to suport step-by-step execution of the new added features. To represent the LatchTable please use a TableView with two columns location and value.

g. (with a working GUI 0.75p, with a working text UI 0.25p). Show the step-by-step execution of the following program. At each step display the content of each program state (all the structures of the program state). The step-by-step execution must be displayed on the screen and also must be saved into a readable log text file. The following program must be hard coded in your implementation.

```
Ref int v1; Ref int v2; Ref int v3; int cnt;
new(v1,2);new(v2,3);new(v3,4);newLatch(cnt,rH(v2));
fork(wh(v1,rh(v1)*10);print(rh(v1));countDown(cnt);
      fork(wh(v2,rh(v2)*10);print(rh(v2));countDown(cnt);
           fork(wh(v3,rh(v3)*10);print(rh(v3));countDown(cnt))));
await(cnt);
print(100);
countDown(cnt);
print(100)
```

The final Out should be {20,id-first-child,30,id-second-child,40, id-third-child, 100,id_parent,100} where id-first-child, id-second-child and id-third-child are the unique identifiers of those three new threads created by fork, while id_parent is the identifier of the main thread.