Test 3 - Revision

Test 3 will cover the following topics:

- Typechecking. You should be able to:
  - Define typechecking specifications for new expressions and statements.
  - Point out type errors in SPL programs i.e. you should know the typechecking rules of SPL.
- Stack Frames.
  - Stack frame layout. Given a set of SPL functions, you should be able to determine the stack frame layout of each function declaration.
  - You should be able to write TPL code that accesses the elements (local variables, parameters, etc) allocated in the stack frame.

1 Typechecking

- Write down the abstract syntax tree (AST) representation and the typecheck specification of the new expression *CondExp* (Conditional Expression) defined by the following concrete syntax:

  $$\textit{CondExp} \rightarrow \textit{Exp}\ \mathbf{?}\ \textit{Exp}\ \mathbf{:}\ \textit{Exp}$$

  CondExp behaves like a short version of the IF statement. It contains three subexpressions and is evaluated as follows:
  - The first expression is evaluated to a boolean value t.
  - If t is true then the second expression is evaluated. The result of that evaluation is the result of the evaluation of the whole expression.
  - If t is false then the third expression is evaluated. The result of that evaluation is the result of the evaluation of the whole expression.
  - *CondExp* is an expression and, therefore, returns a value.

  For example, we can use *CondExp* to write the following code:

```
float a,b;
int x,y,z;
boolean t,r;
// some code here
x := (x > 0) ? y : (y + z);  // CondExp is the right-habd side of the assignments
a := (y >=10) ? 10.0 : a * b;
t := r ? (x > 0) : (x > 1)
```

  Solution: The AST representation of *CondExp* can be defined as follows:

  CondExp(Exp b, Exp e1, Exp e2)

  The AST representation throws away unnecessary syntax (punctuation symbols, operators, etc). In this case, we only keep the three sub-expressions that make up *CondExp*.

  I have asked ypu to define the AST representation because it's the best way to express the typechecking specification (and other properties) of parts of a programming language. The typechecking specification for *CondExp* is:

```
1.  int ping(boolean t, int p1) {
2.    boolean q;
3.    int x, y;
4.    float w,z;
5.    w = 10.0;
6.    x := p1 + 5;
7.    q := t and p1
8.    if (q and (50 > x)) then {
9.      z := pong(x,w);
10.    } else {
11.     z := pong(w,x,12);
12.   }
13.   return x + z;
14. }

15. float pong(float p1, int p2, int p3) {
16.      // some code here
17.   return (int) p1 + p2*p3;
18. }
```

Figure 1: SPL Program with type errors

typecheck(CondExp(b,e1,e2), f, stable) =
t = typecheck(b,f,stable)
ReportError if t $\neq$ boolean
t1 = typecheck(e1,f,stable)
t2 = typecheck(e2,f,stable)
ReportError if (t1 $\neq$ t2)
return t1

The first expression must be boolean. The second and third expressions can be of any type, as long as they are the same. All three subexpressions must be correctly typed.

Note: Think how you can express the typing rules of repeat-until, do-while, etc.

- Consider the two SPL function declarations of Figure 1. There are four type errors: identify them.

  solution:

  - Line 7: Operator **and** expects both oprans to be of type boolean. However, variable p1 is of type int.
  - Line 9: Incorrect number of arguments sent to function **pong**. Also, the type of the first two arguments is incorrect ie.e the first argument should be float..
  - Line 13: Operator + expects the type of both operands to be the same, either integer or float. Here, we have x of type int and z of type z (we are not assuming that there's automatic conversion between types).
  - Line 17: The return type of pong is float but the expression being returned has type int (the subexpression is correctly typed though)

  Figure 2 shows are correctly typed version of this code.

```
1.  int ping(boolean t, int p1) {
2.    boolean q;
3.    int x, y;
4.    float w,z;
5.    w = 10.0;
6.    x := p1 + 5;
7.    q := t and (p1 > 0)  // both operands are boolean
8.    if (q and (50 > x)) then {
9.      z := pong(w,x,p1); // correct number and type of argument
10.    } else {
11.     z := pong(w,x,12);
12.   }
13.   return x + (int) z;  // z is casted to int. The type of the return is now int.
14. }

15. float pong(float p1, int p2, int p3) {
16.      // some code here
17.   return p1 + (float) (p2*p3);   // the return expression is now float
18. }
```

Figure 2: SPL Program with type errors

2 Stack Frames

Consider the SPL function declations in Figure 2. Assuming that there's enough registers for temporaries, even acrros function calls:

- Determine the stack frame layout for function ping.

  A stack frame (when is on top of the stack) is demarked by the Frame Pointer (FP) and the Stack Pointer. Starting from FP, the layout of ping's frame is:

  - Offset 0: local var q (boolean, size 1)
  - Offset 1: local var x (integer, size 1).
  - Offset 2: local var y
  - Offset 3: local var w (float, size 2)
  - Offset 5: local var z
  - Offset 7: Return address (address, size 1)
  - Offset 8: Returned value (from pong, float, size 2)
  - Offset 10: pong's third parameter, p3 (int)
  - Offset 11: pong's third parameter, p2 (int)
  - Offset 12: pong's third parameter, p1 (float)

- What's ping's stack frame size? It's 14. Note that the last element, a float (size 2), has offset 12.

- How does pong access its second parameter? When pong is called, it's parameters will be located above FP, that is, they will have a negative offset with respect to FP. In p2's case, the offset is -3. For example, if we want to increment the value of p1 by 1 we should write:

  ```
  ADDI FP(-3), 1, FP(-3)
  ```

- How does ping access local variable z? Local variable is lcoated an an offset of 5 from FP. If we want to read the value of x and, for example, store it in R5, we should write:

```
STORE FP(5), R5
```