CSC488S 2011/2012 Final Exam Solution and Comments

1. [10 marks]

a) Benefits: all array storage can be sized and allocated at compile time.
No need for dynamic array allocation mechanisms.

b) Benefits: No need for a display mechanism, two addressing registers are sufficient.
Issues: Less flexibility for structuring software.

c) Benefits: simple to implement, easier to determine side effects.
Issues: awkward for large objects, arrays passed as parameters.
harder to build utility functions, e.g. sort

d) Benefits: sets can be implemented as a single 32-bit word bitset.

e) Benefit: no need for a complicated stack cleaning algorithm.
Issue: probably also better programming practice.

2. [10 marks]

The first step was to get the precedence of the operators correct. A lot of people got this wrong. Here is a more fully parenthesized version of the expression

(P **and (** **not (** Q **or (** I <= J **and** R **not**= Q **)))) or (** Q **and not** S **)**

A branching solution similar to those in the lecture slides looks like:

Assume a branching quadruple

label (branch , test , BRtrue , BRfalse)

A possible solution is

```
T1   ( branch , P    , T2         , T7         )
T2   ( branch , Q    , T7         , T3         )
T3   ( less   , I    , J          , R1         )
T4   ( branch , R1   , T5         , true-exit  )
T5   ( noteq  , R    , Q          , R2         )
T6   ( branch , R2   , T7         , true-exit  )
T7   ( branch , Q    , T8         , false-exit )
T8   ( branch , S    , false-exit , true-exit  )
```

3. [20 marks] The key issue here was the need for a mechanism to branch between the for statement body and each of the forSpec elements. Many people got this wrong or ignored the issue. A correct solution had to work for the case that the **while** and **step/until** forSpecs might not execute the body at all. A complete answer had to provide a translation for all three kinds of forSpecs.

We assume that the statement part of the for statement is set up as a parameterless internal routine. We assume a CALL instruction that branches to the statement leaving a return address in some standard place and a RETURN instruction that uses this return address to branch back to the instruction after the CALL. An alternative solution would be an explicit temporary variable (one per for statement) that holds the return address. Translation for the *statement*

```
        BRANCH  around
body:   statement
        RETURN
around:
```

Translation schema for the forSpecs

expression	*expressionV* **while** *expressionT*	*expressionI* **step** *expressionS* **until** *expressionL*
expression CALL body	ev: expressionV expressionT BRFALSE next CALL body BRANCH ev next:	expressionI STORE variable BRANCH test step: LOAD variable expressionS ADD STORE variable test: expressionL BRTRUE next CALL body BRANCH step next:

4. [20 marks] a) At the forward declaration
– check that it is not a duplicate declaration
– Enter the routine and it's parameter signature in the symbol table
– Mark the declaration as a forward declaration

b) at the actual declaration
– check that it is not a duplicate declaration of a non-forward routine
– if a symbol table entry for the routine is found in the local scope check that it is a forward declaration
– verify that the actual declaration occurs in same scope as the forward declaration
Almost everyone missed this check the will prevent broken routine call/return
– verify that the signature of the actual declaration matches the signature of the forward declaration.
– for functions verify that the return type of the actual declaration matches the return type from the forward declaration
– remove the forward flag from the declaration

c) Other checks
– At the end of each scope, check for forward declarations that have not been matched by actual declarations

5. [20 marks] Possible optimizations include:
– variable folding, propagate the value of n into the loop
– constant folding
– common subexpression elimination, mostly for the subscript calculations
– code motion, move loop independent code out of the inner loops
– strength reduction and test replacement on the loop variables.
Some non optimizations
– most people who transformed the loop variables forgot to fix i,j,k after the loop
– many people suggested loop unrolling, but no one actually showed an unrolled loop
– some people suggested replacing the multiply by 2.0 with a shift.
Unlikely this would be available for a floating point multiply

The first step was getting the array subscript calculation right. Most people got this wrong. Amazingly some people just plugged in the array subscripting code from the example in the lecture slides, totally wrong. The stride in each dimension for A and B is 50 - 0 + 1 = 51 so an array subscript looks like

```
@A[ i , j , k ] = @( &A[ 0 , 0 , 0 ] + ( 51*51*i + 51*j + k ) * 8 )
                = @( &A[ 0 , 0 , 0 ] + 20808 * i + 408 * j + 8 * k )
```

```
30      n := 48
31      for i288 := 20808 to 998784 by 20808 do        /* 20808*48 */
            AIP := &A[0,0,0] + i288             /* A[ i , , ] */
            AIMP := AIP - 20808                   /* A[ i-1 , , ] */
            BIP := &B[0,0,0] + i288             /* B[ i , , ] */
32          for j488 := 19564 to 408 by -408 do       /* 408*48 */
                AIJP := AIP + j488                  /* A[ i , j ,] */
                AIJMP := AIJP - 408                   /* A[ i , j - 1 ,] */
                AIMJP := AIMP + j488                    /* A[ i -  1 , j , ] */
                BIJP := BIP + j488                  /* B[ i , j , ] */
                BIJMP := BIJP - 408                   /* B[ i , j - 1 , ] */
33              for k8 := 8 to 392 by 8 do             /* 8*49 */
                    AIJKP := AIJP + k8                    /* A[ i , j , k ] */
                    AIJMKP := AIJMP + k8                      /* A[ i . j - 1 , k ] */
                    AIJMKMP := AIJMKP - 8                       /* A[ i , j - 1 , k - 1 ] */
34                  @ AIJKP := @( AIJMKMP ) + @(  AIMJP + k8 )
35                  @( BIJMP + k8  ) := @( AIJMKMP ) * 2.0
36                  @( AIJKP + 8 ) := @( BIJP + k8 ) + 1.0
37              endfor
38          endfor
39      endfor
        i := 48
        j := 1
        k := 49
```

6. [10 marks] A key point was to create checking code that only evaluated the array subscript expression once to avoid side effect issues. A lot of solutions assumed that the lower and upper bounds needed to be stored with the array at runtime. There is no need for this because the bounds are compile time constants. For an array subscript `A [ expression ]` a correct solution looked something like:

```
                expression
                DUP
                DUP                     % stack contains 3 copies of expression
                PUSH A.lowerBound
                LT                      % expression < lowerBound ?
                PUSH lowOK
                BF                      % branch if expression >= lowerBound
                PUSH subsError          % error routine
                BR
lowOK           PUSH A.upperBound + 1   % upperBound + 1
                LT                      % expression < upperBound+1
                PUSH subsError          % error routine
                BF                      % branch if expression > upperBound
                . . .                   % subscript code continues here
```

7. [10 marks] This question was a test of insight into the interaction between language design and implementation. Most answers dealt correctly with need to be able to process declarations whenever they occur. Very few people recognized the serious issue shown in the example below.

```
1       var x : Integer
2       {       % new scope
3           x := 3
4           var x : Integer
5       }
```

The assignment to x on line 3 is a use-before-definition error since x is declared in the scope on line 4. Unless the compiler takes extra care in its symbol table lookup, the variable x on line 3 will get associated incorrectly with the declaration on line 1.

A correct solution to intermixed declarations and statements and the requirement for declaration before use would require:

– a two pass processing of a scope, the first pass processes all the declarations
the second pass processes all the statements.

or

– a backwards check at each declaration in a scope to ensure that the variable being
declared had not already been used in the scope.

8.[20 marks] There are a lot of semantic checks for this simple piece of code

1	class name not already in use in this package file name matches class name
2	String class/interface exists main(string[]) signature not previously declared
4	local variable limit not previously declared in this scope 100 is assignable to limit
5	System.out and System.out.println exist System.out.println takes a String argument String + int is a valid operation limit has been initialized
7	local variable i has not been previously declared in this scope 1 is assignable to i i < 100 is a valid comparison i++ is a valid operation
8	local variable isPrime has not been previously declared in this scope true is assignable to isPrime
10	local variable j has not been previously declared in this scope 2 is assignable to j i has been initialized j < i is a valid comparison j++ is a valid operation
11	i and j have been initialized i % j is a valid operation (i % j) == 0 is a valid comparison if expression value is Boolean type
12	isPrime exists false is assignable to isPrime
17	isPrime exists if expression value is Boolean type
18	System.out and System.out.print exist System.out.print takes a String argument String + int is a valid operation