CSC488S 2008/2009 Final Exam Solution and Comments

1. [10 marks] A simple solution:

– Remove all 2-character lexical tokens from csc488.flex
  Add a new token DOT for a single period token.
– Replace all use of the 2-character lexical tokens in csc488.cup
  with their single character equivalents. e.g.
    ASSIGN → COLON EQUAL
    NOT_EQUAL → LESS GREATER
    DOTDOT → DOT DOT

This is all the that is required. The scanner will remove whitespace between tokens as it already does for all other tokens. The parser has no difficult parsing the revised grammar.

Comment: The question was a test of understanding of how scanners and parsers operate. Very few students came up with a correct solution. There were a lot of very broken solution involving the scanner returning a whitespace token *in this case only*.

2. [10 marks] The failure in the LL(1) property occurs during LL(1) table construction when the algorithm attempts to create a table entry that has already been created by a previous step in the algorithm. Each table entry corresponds to one member in the director set for some rule. Attempting to create the same entry more than once indicates non-disjoint director sets.

Comment: Very few students had a correct answer. Most answers simple repeated the LL(1) disjointness condition instead of answering the question.

3. [15 marks] The scanner needs to keep a counter of the number of (* that it's seen. The counter is incremented each time a (* is encountered and decremented each time a *) is encountered. If the counter is greater than zero the scanner is inside a comment and everything should be discarded.

Error detection

– if the counter becomes negative after a *) is seen this indicates an extra *)
– if the counter is greater than zero when end of file is detected, this indicates one or more unclosed comments.

4. [25 marks]

a) **string assignment** Roughly equal in implementation effort. Each requires a *copy* of the string value and for b) and c) a copy of the string length.

b) **string length** O(1)) for b) and c). O(N) for a) hence much slower.

c) **string concatenation** O((N + M) for a), need to search to find end of string before adding characters.
O(M) for b), just the cost of copying in *moreS*
O(N + M) for c), have to shift the existing string over before adding *moreS*

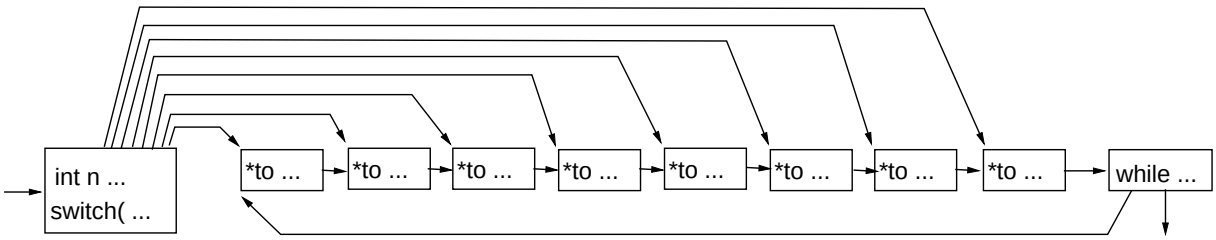
d) **substring** About the same for all alternatives, search in subject for substring, copy substring to new string.

e) **string parameters** About the same for all alternatives, if strings are passed by reference, just pass the address of the string. If strings are passed by value, same as assignment, and length is available to the function/procedure.

5. [10 marks] Possible optimizations
— Expand squareIt and cubeIt inline
— fuse loop on lines 20/21 into following loop
need X[0] = 0.0 before loop for correctness
– Move all computations that don't depend on J out of the inner loop.
i.e. X[J], X[J-1], A[J], B[J]
– Do common subexpression elimination and constant folding
– Do strength reduction and test replacement on J and K
– Possibly unroll loop on lines 23/25 5 times

```
 7      double A[ 50 ][ 50 ], B[ 50 ][ 50 ], X[ 50 ] ;
 8      int J, K ;
 9      /* Assume A and B are given values here */
        ...
21      X[ 0 ] = 0.0 ;
22      for( J8 = 8 ; J8 < 400 ; J8 += 8 )
22a        XJP = &X[ 0 ] + J8 ;
22b        @XJP = 0.0 ;
22c        XJM1 = @(XJP -8) ;
22d        J400 = 50 * J8 ;
22d        AJP = &A[0][0] + J400 ;
22e        BJP = &B[0][0] + J400 ;
23         for( K8 = 0 ; K8 < 400 ; K8 += 8 )
23a           AJK = @( AJP + K8 ) ;
23b           BJK = @( BJP + K8 ) ;
24            @XJP = @XJP + 56.0 * AJK * AJK * AJK
25                        + 5.0 * BJK * XJM1 * BJK * XJM1 + 3.0 ;
```

6. [10 marks] Combined basic blocks and dataflow diagram



Comment: most common mistake was to make case labels into basic blocks.

7. [20 marks]

a) Static semantic checks

– Check that type of *Expr* is suitable for a TYPECASE statement

– Check that the $T_i s$ are disjoint and suitable for the TYPECASE.

– Check that the $v_i s$ are not already declared and disjoint

– Create a minor scope for each S_i that contains the corresponding v_i

b) Implementation

The hard part of implementing this statement is creating a mechanism for identifying types at run-time. This gets particularly messy if separate compilation must be supported. Most solutions tagged (somehow) each type with a unique identifying integer.

Once this is done, the TYPECASE statement can be mapped into the implementation of switch statements, with the switch being made on the runtime type of *Expr*.

Comment: A lot of answers were weak on runtime type identification.

8.[20 marks] Here is a solution in (almost) C.

```
#define FAIL false
#define OK true
/* Check for expected token */
int expect( token expected ) {
      if( nextToken == expected ) {
            getNextToken() ;
            return OK
      } else
            return FAIL ;
}
/* parse one case in TYPECASE */
int oneCase() {
      if( ! type() ) return FAIL ;
      if( expect( tLeftParen ) ) {
         if( ! variable() ) return FAIL ;
         if( ! expect( tRightParen )) return FAIL ;
      }
      if( ! expect( tRightArrow )) return FAIL ;
      return statement() ;
}
```

```
parseTypeCase() {
      /* deal with initial TYPECASE */
      assert( expect( tTYPECASE ));
      if( ! expression() ) return FAIL ;
      if( ! expect( tOF )) return FAIL ;
      do
            if( ! oneCase() ) return FAIL ;
      while( expect( tVerticalBar ) ) ;
      if( expect( tELSE ) )
            if( ! statement() ) return FAIL ;
      return expect( tEND ) ;
}
```

Comment: Almost everyone forgot to process the initial TYPECASE (read the question!). This solution uses assert because it's a logic error in the higher level parser if parseTypeCase is called anywhere except at the start of a TYPECASE statement.