CSC488S 2013/2014 Final Exam Solution and Comments

1. [10 marks]
a) a = 2
b) a = 8
c) a = 3 or a = 7
    depends on order of result storing y or z first
d) a = 9
    a + b is reevaluated when used as x on line 5

2. [10 marks] As described the lexical analyzer can support this parsing scheme in two ways.
Recognizing level zero commas and equal signs Maintain a parenthesis counter. +1 for (and -1 for) When an equal sign (=) character is encountered, emit a *zeroLevelEquals* token if the counter is zero, otherwise emit a *nonZeroLevelEquals* token. Similarly when a comma (,) is encountered emit a *zeroLevelComma* token or a *nonZeroLevelComma* token..

Equal signs and commas inside strings Recognize the start of a string when ' is encountered. Don't change the parenthesis counter while inside a string. Any equal signs or commas encountered inside the string are emitted as part of the string. Recognize end of string and resume parenthesis counting.
Many solutions didn't get the details right. A lot of solutions really didn't answer the question. They described parsing schemes or other actions outside the scope of a lexical analyzer.

3. [10 marks] This innocent looking construct actually has some nasty problems:
- Need a runtime check for M <= N at the point of T's declaration
- If T is used to define array bounds, M and N may need to be stored at runtime.
- Anywhere that T is used to size an array or other data structure there will need to be runtime code to deal with a varying size.
- The RHS of any assignment to a variable of type T will need a runtime check that the value is in range for the current definition of T.
- Any array that uses T to define it's bounds may need runtime subscript checking.

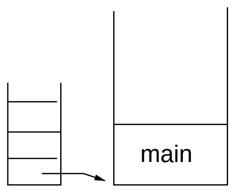
No one saw the really hard issues.
- if a variable of type T is passed as an argument to a parameter of some fixed subrange type (e.g. K : 1 .. 100) then there will have to be a runtime check that the argument variable is correct.
- type equivalence between structures may be affected if T is used to define fields in a structure.

And the really big one:
- depending on the values of M and N, T might be represented as an *signed integer* of an *unsigned integer*. This could require different code every where that a variable of type T is used.[1]

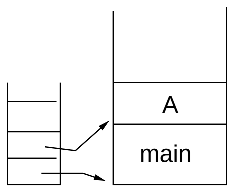
[1] When I pointed this problem out to the Euclid language designers they added a language restriction that the definition of T must result in a *signed integer*

4. [15 marks]

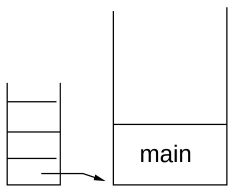
a)



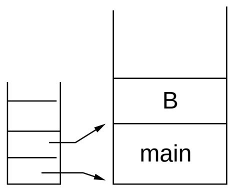
b)



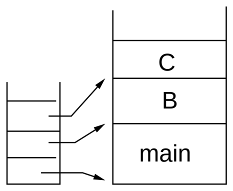
c)



d)



e)



f)

g(2) = 3

or

g(2) = undefined

Part e) illustrates a huge problem with the example[2] When C is called inside B (as g), The activation record for C's parent A is long gone, so the reference to m inside C is broken. Some answers to part e) had the activation record for A appear by magic so the reference to m was legal, other answers ignored the issue. Since the example was broken, either answer for part f) was accepted.

6. [20 marks]

a) Check that argument passed to a **var** parameter is some form of *variable* that has an address and can be stored into.

b Need new code to pass the addresses of arguments passed by reference.

c) Probably no changes since parameters/arguments aren't handled here.

d) Need to recognize arguments passed by reference and generate addresses as described in b)

e), f) Implementation specific. Any plausible code sequences was acceptable.

[2] An example of the design language in haste, repent in leisure principle. The language needs a restriction that parents can't give away their children thereby creating orphans.

5. [15 marks]

```
boolean function expect( token ) {
      return nextToken = token
         and getToken()
}


boolean function caseStatement() {
     assert( expect( Tcase ) )
      return expression()
         and expect( Tin )
         and caseList()
         and expect( Tesac )
}

boolean function caseList() {
      if  not  oneCase()  then
         return false
      while nextToken  not = Tesac  do {
         if  not  oneCase()  then
            return false
      }
      return true
}
```

```
boolean function caseLabel() {
      if  not  pattern()  then
         return false
      if expect( Tvbar )  then
         return caseLabel()
      return true
}

boolean function oneCase() {
      return caseLabel()
         and expect( TrightParen )
         and caseCommands()
         and expect( Tsemicolon )
         and expect( Tsemicolon )
}

boolean function caseCommands() {
      while nextToken  not = Tsemicolon  do
         if  not  command()  then
            return false
      return true
}
```

A lot of solutions didn't get the syntax right. Either allowing too much or not enforcing all the constraints in the grammar. The sample solution above is inspired by several solutions that made clever use of the conditional properties of **and** .

7. [20 marks] A lot of checking is required

- in the declarations on lines 2 and 3
    check that *in , x, a, b ,c* have not be previously declared.
    check that the value of *N* results in a valid array declaration
- everywhere that *N* is used on lines 3,63,69
    check that *N* is declared, accessible and a scalar value
    (N might have been substituted by the C preprocessor)
- everywhere *in* is used, lines 63,64,65,66,69,71
    check that *in* is declared, accessible and a scalar value
- everywhere *x* is used, lines 61,66,71
    check that *x* is declared, accessible and a one dimensional array
- everywhere *a* is used, lines 64,66
    check that *a* is declared, accessible and a one dimensional array
- everywhere *b* is used, lines 60.61,64
    check that *b* is declared, accessible and a one dimensional array
- everywhere *c* is used, lines 60,64,65,71
    check that *c* is declared, accessible and a one dimensional array
- for the assignment statements on lines 60,61,63,64,65,66,69,71
    check that the left hand side is a variable
    check that the value being assigned is compatible with the variable
- for the subtraction operator (–) on lines 64,66,69,71
    check that the operands are a suitable numeric type
- for the divide operator (/) on lines 60,61,64
    check that the operands are a suitable numeric type
- for the multiply operator (*) on lines 64,65,66,71
    check that the operands are a suitable numeric type
- for the less than (<) compare on line 63
    check that the operands of the operator are suitable
- for the increment operator (++) on line 63
    check that its operand is a suitable numeric type
    check that its operand is a variable
- for the decrement operator (– –) on line 69
    check that its operand is a suitable numeric type
    check that its operand is a variable
- for the declaration on line 64
    check that m has not been previously declared
- for the array subscript operator ([ ]) on lines 60,61,64.65,66,71
    check that the subscript is a suitable numeric value

8.[20 marks]

Possible optimizations include
- constant folding
- common subexpression elimination
- strength reduction and test replacement

Some solutions suggested loop unrolling although few actually did it
Loop fusion wasn't a good choice due to loop carried dependencies.
A good solution had to show at least the final result, like the one below

```
        AP =  &a[1]        /* a[ 1 ] */
        BP =  &b[0]
        CP =  &c[0]
        XP =  &x[0]
        XLIM = XP + 792        /* x[ N – 1 ] */
 60     @CP = @CP / @BP
 61     @XP = @XP  / @BP
 63     for ( XP += 8 , CP += 8 , BP += 8 ; XP <= XLIM ; XP += 8 ) {
 64             double m = 1.0 / ( @BP – @AP * @( CP – 8 ) )
 65             @CP = @CP * m ;
 66             @XP = ( @XP – @AP * @(XP – 8 ) ) * m ;
                AP += 8 ;
                BP += 8 ;
                CP += 8 ;
 67             }
        XP =  &x[0] + 792 ;        /* x[ N – 1 ] */
        CP =  &c[0] + 792 ;        /* c[ N – 1] */
 69     for ( ; XP >  &x[0] ; XP –= 8 ) {
 71             @XP = @XP – @CP * @( XP – 8 )
                CP –= 8 ;
 72     }
        in = 0 ;
```