Homework 3

This homework is due Wednesday, Feb. 21 (by 5:00 PM). Note that one of the problems must be done on the computer.

1. (3 points) Draw a contour diagram for a procedure whose outline is shown in Figure 3.5 on page 110 of MacLennan.
2. (15 points) Do exercise 3-2 on page 110 of MacLennan (explicit inheritance). Your design should be precisely described and illustrated by an example.
3. (10 points) Would an explict inheritance facility (see the previous problem) help Algol 60 programs to solve the information hiding problems discussed in MacLennan's section 3.3 (pages 111–112)? Why or why not?
4. (8 points) Do exercise 10 on page 151 in the text.
5. (10 points) This exercise concerns the **switch** declarations of Algol 60 (see pages 149–150 in the text). Would it be a good idea to restrict the **switch** declaration so that only declarations of the form

   switch name = label1, label2, label3, ...;

   are permitted? (That is, would it be a good idea to rule out expressions in **switch** declarations?) Why or why not?
6. (20 points) Do exercise 3-14 on page 124 of MacLennan (string data type design). Your design should be done on the computer and mailed to `mobasher`. Please include your name prominently with your design.

   Present your answer by first giving a brief overview of the type, describing the values abstractly, and then describing each operation by describing its arguments, precondition (if any) and postcondition in format illustrated by the following example.

```
integer = type
   Operations [add, sub, mul, div, lt, equal]

   Overview: The abstract value of an integer consists of a magnitude
             (a natural number) and a sign (either + or -).
             The magnitude of n is written mag(n), the sign of n is written
             sign(n).  Note that -0 = +0.

             The operations on natural numbers are +, -, *, <, and =,
             with their usual meanings.

   procedure add(i,j: integer) returns(n: integer)
          Effect: if sign(i) = sign(j),
                  then sign(n) = sign(i) and mag(n) = mag(i) + mag(j);
```

```
                 else if mag(i) > mag(j),
                 then mag(n) = mag(i) - mag(j) and sign(n) = sign(i);
                 else mag(n) = mag(j) - mag(i) and sign(n) = sign(j).

  procedure sub(i,j: integer) returns(n: integer)
         Effect: if not(sign(i) = sign(j)),
                 then sign(n) = sign(i) and mag(n) = mag(i) + mag(j);
                 else if mag(i) > mag(j),
                 then mag(n) = mag(i) - mag(j) and sign(n) = sign(i);
                 else mag(n) = mag(j) - mag(i) and sign(n) = sign(j).

  procedure mul(i,j: integer) returns(n: integer)
         Effect: if sign(i) = sign(j),
                 then sign(n) = (+) else sign(n) = (-);
 mag(n) = mag(i) * mag(j).

  procedure div(i,j: integer) returns(n: integer)
         Requires: not(mag(j) = 0).
         Effect: if sign(i) = sign(j),
                 then sign(n) = (+) and
                 exists (0 < r <mag(j)) mag(i) = mag(j)*mag(n) + r;
                 else sign(n) = (-) and
                 exists (0 < r <mag(j)) mag(i) = mag(j)*mag(n) - r.

  procedure lt(i,j: integer) returns(b: boolean)
         Effect: if sign(i) = sign(j),
                 then b = (mag(i) < mag(j));
                 else if mag(i) = mag(j) and mag(i) = 0, then b = false;
                 else b = (sign(i) = (-)).

  procedure equal(i,j: integer) returns(b: boolean)
         Effect: b = (mag(i) = mag(j)
                      and (sign(i) = sign(j) or mag(i) = 0)).
```

Note carefully the specification of the `div` operation. Its precondition (following `Requires:`) says that the effect of `div` is only defined when the abstract value of the second argument is not zero. That is, the programmer using this type should never divide by zero. The postcondition (following `Effect:`) describes the abstract value of the result in terms of the abstract values of the arguments and the operations defined in the overview for manipulating these abstract values. The precondition is omitted if it is vacuous (i.e., always true). The above example will be available in the course public directory, in the file `/usr1/users/class/cs342/public/integer.spec` if you wish to use it as a basis for editing.

In your design of the `string` type, use sequences of characters as the abstract values of strings. You may use the following operations on these abstract values, as well as any others you define. Sequences are written in angle brackets, so `<c,d,e>` is the sequence consisting of `c`, `d`, and `e`. The operations on sequences are `<>` which denotes the empty sequence, `cons` which constructs a sequence from a character and a sequence, `size`, `head`, `tail` and `=`. Their behavior is described by the following equations, where `s` and `t` are arbitrary sequences and `c` and `d` are arbitrary characters.

```
size(<>) = 0
size(cons(c,s)) = 1 + size(s)
head(cons(c,s) = c
tail(cons(c,s)) = s
(<> = <>) = true
(cons(c,s) = cons(d,t)) = (c = d) and (s = t)
```