

1. [10] Determine the most general type of this Haskell function:

```
> com f g (x:xs) (y:ys) = f x == g y && com f g xs ys
```

Its type is

```
com :: Eq a => (b -> a) -> (c -> a) -> [b] -> [c] -> Bool
```

2. [10] Given that the months of the year are to be represented by integers 1...12, you are implementing the type **set of month** in C++ using minimum storage.

- a. Define the types `month` and `setOfMonth` in C++. Assume that

```
sizeof( short int ) = 2
sizeof( int )       = 4
sizeof( long int )  = 8.
```

```
typedef short int month;
```

```
typedef short int setOfMonth; .
```

- b. Define the function `bool contains(setOfMonth S, month m)` which returns true iff `S` contains `m`.

```
bool contains( setOfMonth S, month m )
{
    if( 1 <= m && m <= 12 )
        return (S >> (m - 1)) & 1U ;
    return false;
}
```

3. [10] Consider this program fragment in C:

```
int a = 0;
int b = (a++ && ++a) >= (++a || a++);
printf( "%d %d", a, b )
```

Assuming that all side-effects take effect with no delay, what are the possible outputs?

For operators whose order of operand evaluation is not defined by the language, assume...

- a. right operands are evaluated first:

<code>b = (a++ && ++a) >= (++a a++)</code>	where a = 0
<code>b = (a++ && ++a) >= (1 a++)</code>	where a = 1
<code>b = (a++ && ++a) >= 1</code>	where a = 1
<code>b = (1 && ++a) >= 1</code>	where a = 2
<code>b = (1 && 3) >= 1</code>	where a = 3
<code>b = 1 >= 1</code>	where a = 3
<code>b = 1</code>	where a = 3

So the output is **3 1**

- b. left operands are evaluated first:

<code>b = (a++ && ++a) >= (++a a++)</code>	where a = 0
<code>b = (0 && ++a) >= (++a a++)</code>	where a = 1
<code>b = 0 >= (++a a++)</code>	where a = 1
<code>b = 0 >= (2 a++)</code>	where a = 2
<code>b = 0 >= 2</code>	where a = 2
<code>b = 0</code>	where a = 2

So the output is **2 0**

4. [10] Define a C++ function `sumArray` that computes the sum of elements `i` through `j` of a `double` array `X`. In your solution, do not use any array-indexing operations.

```
double sumArray( double* X, int i, int j )
{
    double sum = 0;
    double* p = X + i;
    double* q = X + j;

    while( p <= q ){ sum += *p; p++; }

    return sum;
}
```

5. [15] Assume that arrays are indexed from 0 in the following program:

```
int k = 1;
procedure p( int x, int y ) {
    x := x+2; y := 6; k := 10;
}
procedure main() {
    int array B = [0,1,2,3,4,5];
    p( k, B[k] );
    print k; print B[1]; print B[3];
}
```

In the following table, for each output put an **X** in the column of the parameter-argument binding mechanism being used in **procedure p**.

output	value	reference	value-result	name
10 6 3		X		
10 1 6				X
3 6 3			X	
10 1 3	X			

6. [10] Translate the following function into an equivalent recursive function which has the same big- O space requirements in terms of n .

```
procedure f( integer x, integer n ) : integer
{
    while ( n > 0 )
    {
        n := n-1; x := x*2;
    };
    while ( n < 0 )
    {
        n := n+1; x := x/2;
    };
    return x;
}
```

The given program's space requirements are independent of n ; the same independence in a recursive equivalent requires tail recursion. The effect of the assignments is to rebind the parameters x and n to the values of $x*2$ (or $x/2$) and $n-1$ (or $n+1$), respectively. A straightforward translation obtains the same rebindings by means of tail-recursive calls:

```
procedure f( integer x, integer n ) : integer
{
    if( n > 0 )
        return f( x*2, n-1 )
    else if( n < 0 )
        return f( x/2, n+1 )
    else
        return x;
}
```

7. [10] The following fragmentary program text contains declarations of several variables and a reference (`a[3].f2[1]`) to one of these variables.

To answer this question, determine the address information that a compiler would generate for the variable reference. Assume

- a typical stack implementation
- space is allocated for variables in the order in which they are declared, beginning with offset 0
- no requirement that variables be aligned on word boundaries.

Assume that addresses identify bytes, and that the basic data types' space requirements (in bytes) are as follows:

bool 1 **char** 1 **int** 2 **float** 4

The program:

```

program
  type B = record f1 : float [1..10]; f2 : char[0..2] end ;
  i : int ;
  y : B;
  z : int [1..10];
  j : int ;
  procedure p1
    a : B[3..12];
    y : float ;
  begin
    ...
  end p1;
  procedure p2
    m : int [0..99];
    x : B;
    a : B[1..4];
  begin
    ... a[3].f2[1] ...
  end p2;
begin
  ...
end program .

```

You can get partial credit for an incorrect answer *only* if you've shown your work.

First we build a table for **procedure** `p2` showing each variable's size and its address(es) in its block's activation record.

<code>procedure p2</code>	<code>m</code>	200	0-199
	<code>x</code>	43	200-242
	<code>a</code>	172	243-415

Then the address information for `a[3].f2[1]` is

$$\begin{aligned}
 & \text{address}(a) + 2 * \text{sizeof}(B) + \text{sizeof}(f1) + 1 * \text{sizeof}(\text{char}) \\
 & = 243 + 2 * 43 + 10 * \text{sizeof}(\text{float}) + 1 \\
 & = 243 + 86 + 10 * 4 + 1 \\
 & = 370
 \end{aligned}$$

8. [10] Suppose you are given Haskell functions

```
f1 :: a -> b
f2 :: c -> d
f3 :: e -> f
```

a. In order for the expression

```
f1 >.> f2 >.> f3
```

to be valid, what relationships must hold between a, b, c, d, e, and f?

```
b = c
d = e
```

b. Suppose you have C programs corresponding to f1, f2, and f3. How would you write a Unix pipe command corresponding to `f1 . f2 . f3`?

```
f3 | f2 | f1
```

What requirements must f1, f2, and f3 satisfy for this command to be valid?

f2 and f3 must write to standard output, and f1 and f2 must read from standard input.