

1. [20] Here, as a reminder, are some examples of Haskell type expressions:

```
Int
a -> b
Tree a -> (a -> Int) -> [Employee]
((Int, Bool) -> (Int,Bool,Bool)) -> [(Int,Bool)]
```

Write an EBNF grammar for Haskell type expressions (**not** including type-class contexts such as “Eq =>”). Remember that the operator (->) is *right*-associative.

```
Type ::= Id { Id } | '[' FunType '[' | '(' FunType { , FunType } ')'
FunType ::= Type | Type -> FunType
Id ::= Letter { Letter | Digit }
Letter ::= a | b | ... | z | 'A' | 'B' | ... | 'Z'
Digit ::= 0 | 1 | ... | 9
```

2. [10] Translate the following Haskell function

```
> q :: Int -> Int -> Int
> q x d = if x<d then 0 else 1 + q (x-d) d
```

- a. into an equivalent Haskell function which uses only tail recursion

```
> q1 :: Int -> Int -> Int
> q1 x d = let quot x q = if x<d then q else quot (x-d) (q+1) in quot x 0
```

- b. into an equivalent C function which uses no recursion at all (and uses no division operators).

```
int q2( int x, int d )
{
    int q = 0;
    while( x >= d )
    {
        x = x - d;
        q = q + 1;
    }
    return q;
}
```

3. [10] Use the method of weakest preconditions to determine the truth (or otherwise) of this proposition:

$$\{x < y + 3\} \quad x, y := y - 1, x + 5 \quad \{y < x + 2\}.$$

$$\begin{aligned} & \text{wp } "x, y := y - 1, x + 5" (y < x + 2) \\ &= \quad \{\text{def. ':='}\} \\ & \quad (y < x + 2)_{y-1, x+5}^{x, y} \\ &= \quad \{\text{substitution}\} \\ & \quad x + 5 < y - 1 + 2 \\ &= \quad \{\text{simplification}\} \\ & \quad x < y - 4 \\ & \quad \square \quad \{\text{predicate calculus, counterexample: let } x = 2, y = 0\} \\ & \quad x < y + 3 \end{aligned}$$

So the proposition is false.   ■

4. [10] Translate the following guarded-command program into C. Assume that the type of all variables is *integer*.

```
do  $x < y$   $\square$   $x, y := x+y, x-y$   $\parallel$   $x > y$   $\square$   $x := z-y; y := x-z$  od
```

A solution:

```
while (  $x \neq y$  )  
{ if (  $x < y$  ){ int t = x+y; y = x-y; x = t; }  
  else      { x = z-y; y = x-z; }  
}
```

5. [10] Give the type (if any) of the following expression, assuming that **integer** and **bool** are distinct types

```
if  $x < 21$  or  $3 < 6$  then  $x+5$  else  $z < 4$ 
```

- a. assuming static type checking

type error: the true branch has type **integer**, and the false branch has type **bool**.

- b. assuming dynamic type checking

the type is **int** because the guard is true

6. [5] Given this program fragment:

```
struct R { int x; double y; };  
R x[20];  
cout << x[10];
```

and assuming that `sizeof( int ) = 4` and `sizeof( double ) = 8`, rewrite the expression `x[10]` to give the same result without using array indexing:

```
*(x+10)
```

7. [15] What is output by the following program

```
int x = 4;

function pow( int n, int exp ) : int;
{
    int r = 1;
    while( exp > 0 )
    {
        r := r * n;
        exp := exp - 1;
    }
    return r;
}

function inc( int k ) : int
{
    k := k+1;
    return k;
}

procedure main()
{
    print( pow( inc(x), 3 ) ); print( x );
}
```

- a. assuming that arguments are bound to parameters by *value*  
125 (i.e.,  $5 * 5 * 5$ )    4
- b. assuming that arguments are bound to parameters by *name*  
210 (i.e.,  $5 * 6 * 7$ )    7
- c. Rewrite the functions so that call-by-name gives the same results as call-by-value.

```
function pow( int n, int exp ) : int;
{
    int r = 1;
    int n1 = n;
    int e = exp;
    while( exp > 0 )
    {
        r := r * n1;
        e := e - 1;
    }
    return r;
}

function inc( int k ) : int
{
    return k+1;
}
```

8. [20] The following fragmentary program text contains a type definition, declarations of several variables, and five references to the variables. After each variable reference is a pair of braces containing a letter (for example, { A }); for each such letter, there is a line in the table following the program.

To answer this question, write the address information that a compiler would generate for each of the variable references in the appropriate line of the table. Assume

- a typical stack implementation
- space allocation 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**: 2            **int**: 4            **float**: 10

The program:

```

type ST = record
    n : char;
    case b : bool of
        false : (x : int)
        true  : (y : float)
    end
end;
a : int [5];  b : ST;  c : char;
procedure doit()
    b : float;  c : bool;
begin
    ... a[2] { A } ...  c { B }
end doit;
procedure main()
    r : ST;  s : char;
begin
    ... b.x { C } ...  c { D } ...  r.y { E } ...
end main;
    
```

Answer by filling in the table. Indicate the distinction between local and global variables by appending a **G** to global-variable references. If a reference is invalid, explain. Partial credit may be given for incorrect answers, but *only* if you've shown your work.

First we compute the space requirements for type ST:

n	char	2	0-1
b	bool	1	2-2
x	int	4	3-6
y	float	10	3-12

Then we build a table for each block showing each variable's size and its address(es) in its block's activation record.

(global)	a	int[5]	4*5 = 20	0-19
	b	ST	13	20-32
	c	char	2	33-34
doit	b	float	10	0-9
	c	bool	1	10-10
main	r	ST	13	0-12
	s	char	2	13-14

Then we use the address information to fill in the references' activation-record addresses:

- A. 8 G (= 0+2\*4)
- B. 10
- C. 23 G
- D. 33 G
- E. 3

9. [10] Suppose variables A and B are both 4-element arrays of integers, and the following code is executed:

```

for i = 0 to 3 do A[i] := 1;
A[1] := 2;
B := A;
A[2] := 3;
B[3] := 4

```

Give the post-execution values of A and B by filling in the following tables, assuming the variables have

- a. value semantics.

	0	1	2	3
A	1	2	3	1
B	1	2	1	4

- b. reference semantics.

	0	1	2	3
A	1	2	3	4
B	1	2	3	4

10. [10] This C++ struct represents fractions by their integer numerators and denominators.

```

struct Fraction
{
    int numerator, denominator;
};

```

The following code normalizes fractions:

```

Fraction f;
if( den( f ) < 0 )
{
    num( f ) = - num( f );
    den( f ) = - den( f );
}

int cd = gcd( num( f ), den( f ) );
num( f ) = num( f ) / cd;
den( f ) = den( f ) / cd;

```

Define the functions `num` and `den` so that this code would work.

```

int& num( Fraction& f ) { return f.numerator; }
int& den( Fraction& f ) { return f.denominator; }

```

11. [5] You are given the following C++ definition:

```

template <typename T>
T min( T x, T y ){ return x < y ? x : y; }

```

Define `min` in Haskell, making its meaning as similar as possible to the C++ definition. Include `min`'s type.

```

min :: Ord a => a -> a -> a
min x y = if x < y then x else y

```

12. [10] Predict what is output when this C++ program runs:

```
#include <iostream>

class A
{ public:
    virtual void f() { cout << "Af "; }
    void g() { cout << "Ag "; }
};

class B : public A
{ public:
    void f() { cout << "Bf "; }
    virtual void g() { cout << "Bg "; }
};

void test( A x, A& y, A* z )
{ x.f(); y.f(); z->f();
  cout << endl;
  x.g(); y.g(); z->g();
}

int main()
{
    B b;

    test( b, b, &b );

    return 0;
}
```

The output:

```
Af Bf Bf
Ag Ag Ag
```

13. [15] Newton's method can be used to find roots of arbitrary order. If  $x_n$  is an approximation to the  $k$ th root of  $c$ , then a better approximation is given by

$$x_{n+1} = \frac{1}{k} \left( (k-1)x_n + \frac{c}{x_n^{k-1}} \right)$$

[Note that setting  $k = 2$  gives the square-root formula we considered in class.] Write a Haskell function `approx` which computes an infinite list of better and better approximations to the  $k$ th root of  $c$ . Use this function to define another function `root` so that `root k c` returns the first approximation to the  $k$ th root of  $c$  that differs from its predecessor by less than 0.001%. A good value for the first approximation  $x_0$  is 1.

```
> approx :: Int -> Float -> [Float]
> approx k c = iterate next 1
>   where
>     next x = (1-(1 / fromInt k)) * x + c / ( fromInt k * x^(k-1) )

> root :: Int -> Float -> Float
> root k c = re| 0.00001 (approx k c)
>   where
>     re| eps (a:b:bs)
>       | abs(a/b-1) > eps = re| eps (b:bs)
>       | otherwise       = b
```

An alternative definition of `approx`:

```
> approx :: Int -> Float -> [Float]
> approx k c = as
>   where
>     as = 1 : [ next a | a <- as ]
>     next x = (1-(1 / fromInt k)) * x + c / ( fromInt k * x^(k-1) )
```

14. [15] This question concerns a function `mdif` such that `mdif as bs` contains those elements of list `as` which differ from every element of list `bs`. For example,

```
mdif [1..10] [2, 4, 7, 12] = [1,3,5,6,8,9,10]
```

```
mdif [1,2,3,4,3,2,1] [4,3] = [1,2,2,1]
```

- a. Give `mdif`'s most general type.

```
> mdif :: Eq a => [a] -> [a] -> [a]
```

- b. Define `mdif` nonrecursively, using one or more list comprehensions.

```
> mdif as bs = [ a | a <- as, null [ b | b <- bs, a==b ] ]
```

An alternative definition:

```
> mdif as bs = [ a | a <- as, not (a `elem` bs) ]
```

15. [15] Identify the errors in the following code by listing their line numbers. For each error give a brief explanation.

```
#include <iostream>

class A
{ public:
  void fO{ cout << "Af "; }
  private:
  void gO{ cout << "Ag "; }
};

class B : public A
{ public:
  void gO{ cout << "Bg "; }
};

class C : private A
{ public:
  void gO{ cout << "Cg "; }
};

void test( A x )
{ x.fO;
  x.gO;
}

int main()
{ A a;
  B b;
  C c;

  b.fO;
  b.gO;

  c.fO;
  c.gO;

  test( a );
  test( b );
  test( c );

  A ab = b;
  A ac = c;

  return 0;
}
```

// 1  
// 2  
// 3  
// 4  
// 5  
  
// 6  
// 7  
// 8  
  
// 9  
// 10  
  
// 11  
// 12  
// 13      g is a private member of x  
  
// 14  
// 15  
// 16  
  
// 17  
// 18  
  
// 19      f is a private member of c  
// 20  
  
// 21  
// 22  
// 23      C is not a subtype of A  
  
// 24  
// 25      C is not a subtype of A