

You may use, without defining them, any functions, types, or other items defined in class lecture notes, published libraries, and published homework solutions.

**NOTE:** In any of these questions where types' sizes is relevant, you should assume the following sizes:

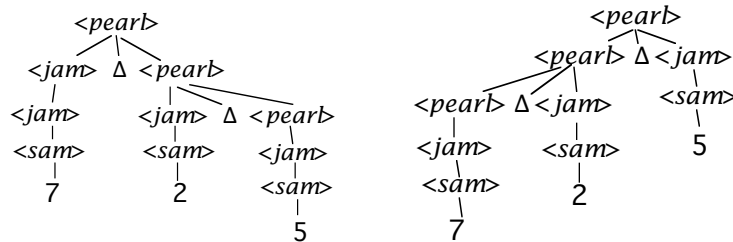
char	1 byte	pointer	4 bytes	long int	8 bytes
short int	2 bytes	int	4 bytes	double	10 bytes

1. [15] Given the following grammar:

$\langle \text{pearl} \rangle ::= \langle \text{pearl} \rangle \Delta \langle \text{jam} \rangle \mid \langle \text{jam} \rangle \Delta \langle \text{pearl} \rangle \mid \langle \text{jam} \rangle$   
 $\langle \text{jam} \rangle ::= \langle \text{jam} \rangle \Phi \langle \text{sam} \rangle \mid \langle \text{sam} \rangle$   
 $\langle \text{sam} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

a. Demonstrate the grammar's ambiguity.

For the expression  $7 \Delta 2 \Delta 5$ , the grammar allows two different parses:



b. Define an unambiguous grammar that allows exactly the same expressions as the original grammar.

$\langle \text{pearl} \rangle ::= \langle \text{pearl} \rangle \Delta \langle \text{jam} \rangle \mid \langle \text{jam} \rangle$   
 $\langle \text{jam} \rangle ::= \langle \text{jam} \rangle \Phi \langle \text{sam} \rangle \mid \langle \text{sam} \rangle$   
 $\langle \text{sam} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

2. [15] Consider the three evaluation strategies

1. innermost evaluation
2. outermost evaluation
3. lazy evaluation

For each of the following functions, indicate which strategy would be **least** efficient by circling the appropriate numbers. If two strategies are both less efficient than the third, circle both; if all three strategies are equally efficient, circle **none** of them.

$f \ x \ y = \text{if } x > 0 \text{ then } y+3 \text{ else } y-2$       1      2      3      strict, no sharing  
 $f \ x \ y = \text{if } x > 0 \text{ then } x+y \text{ else } x-y$       1      **2**      3      strict, sharing  
 $f \ x \ y = \text{if } x < 0 \text{ then } x*y \text{ else } 2-x$       **1**      **2**      3      nonstrict, sharing  
 $f \ x \ y = \text{case } x \text{ of}$   
           0  $\rightarrow y+2$   
           -  $\rightarrow 42$       **1**      2      3      nonstrict, no sharing

3. [10] Translate the following C function into one or more tail-recursive or non-recursive functions containing no loops or assignments.

```
float p( float k, int n )
{
    float r = 1;
    while( n > 0 )
    {
        if( n & 1 )
        {
            r = r*k;
            n -= 1;
        }
        else
        {
            k = k*k;
            n >>= 1;
        }
    }
    return r;
}
```

Answer:

```
float p2( float r, float k, int n )
{
    if( n > 0 )
    {
        if( n & 1 )
            return p2( r*k, k, n-1 );
        else
            return p2( r, k*k, n>>1 );
    }
    else
        return r;
}

float p1( float k, int n )
{
    return p2( 1, k, n );
}
```

4. [10] The Haskell function `unzip` can be defined this way:

```
unzip :: [(a,b)] -> ([a],[b])
unzip [] = ([],[])
unzip ((x,y):xys) = (x:xs,y:ys)
    where
        (xs,ys) = unzip xys
```

Give an equivalent definition of `unzip` using `foldr` and no explicit recursion.

```
unzip = foldr \((x,y) (xs,ys) -> (x:xs,y:ys)) ([],[])
```

5. [15] Use Newton's method for calculating cube roots, in which successively better approximations to the cube root of  $N$  are given by

$$a_{i+1} = \frac{2}{3} \cdot a_i + \frac{N}{3 \cdot a_i^2},$$

to define a Haskell function that (a) computes an infinite list of improving approximations and (b) selects the first approximation that differs from its predecessor by at most  $1.0e-6$ .

```
> cubrt :: Float -> Float
> cubrt n = within 1.0e-6 apps
>   where
>     apps = iterate (next n) n -- alternative: apps = n : [ next n a | a <- apps ]
>     next :: Float -> Float -> Float
>     next n a = (2/3) * a + n/(3 * a*a)
>     within :: Float -> [Float] -> Float
>     within eps (a:b:bs)
>       | abs (a-b) <= eps = b
>       | otherwise       = within eps (b:bs)
```

6. [10] Define a Haskell program `sumNints` that reads an integer, say  $n$ , from standard input, and then reads  $n$  more integers. After reading the last integer, it prints the last  $n$  integers' sum. For example,

```
F05B> sumNints
5
1
2
3
4
5
-----
15
F05B>
```

A solution:

```
> sumNints :: IO ()
> sumNints = do n <- getInt -- getInt is defined in Slide 108
>             ns <- getNints n
>             putStr ("-----\n" ++ show (sum ns))
>
>   where
>     getNints 0 = return []
>     getNints k | k>0 = do n <- getInt
>                          ns <- getNints (k-1)
>                          return (n:ns)
```

7. [5] Fill in the blank:

{ \_\_\_\_\_ } **if**  $x < 10 \rightarrow x := 10 \parallel 10 < x \rightarrow x := 3$  **fi** {  $2 < x$  }

Because an **if**-command is equivalent to **abort** if all of its guards are false, the precondition in this case must exclude all states in which  $x = 10$ .

{  $x \neq 10$  } **if**  $x < 10 \rightarrow x := 10 \parallel 10 < x \rightarrow x := 3$  **fi** {  $2 < x$  }

8. [10] Translate the following C++ statement

```
f0(x) = (f1(x) && f2(x)) || f3(x);
```

into an equivalent nested conditional statement containing no (&&) or (||) operators. Note that the functions may have side-effects.

A solution:

```
if( f1(x) )
  if( f2(x) )
    f0(x) = 1;
  else
    if(f3(x) )
      f0(x) = 1;
    else
      f0(x) = 0;
else
  if(f3(x) )
    f0(x) = 1;
  else
    f0(x) = 0;
```

9. [10] Translate this C fragment to GCN

```
while( x[j] = y[k] ) k++; j++;
```

```
x[j] := y[k]; do y[k] ≠ 0 → k := k+1; x[j] := y[k] od;   j := j+1
```

10. [5] Predict this program fragment's output:

```
typedef struct { int x; double y; } X;
X A[2]; cout << &(A[10]) - A;
```

Output:

10

11. [5] Give the type (if any) of the following expression, assuming that **integer** and **bool** are distinct types

```
if 3<2 and p<2 then x+5 else z<4
```

- assuming type checking is done statically  
type error: the true branch has type **integer**, and the false branch has type **bool**.
- assuming type checking is done dynamically  
the type is **bool** because the guard is false

12. [5] Give the most general type of the following expression:

```
\(w,x) [y] (z) → if y< then x*2 else w
```

Answer:

```
(Num a, Ord b) ⇒ (a,a) → [b] → b → a
```

13. [10] Predict what the following C++ program would print.

```
#include <iostream.h>
int y = 9;
#define CAT( x ) ((x) + y)

int dog( int x )
{
    int y = 2;
    return x + y;
}

int main()
{
    int y = 5;
    int k = 3;
    cout << dog( CAT(k) );
    return 0;
}
```

After macro substitution, the program would look like this:

```
#include <iostream.h>
int y = 9;

int dog( int x )
{
    int y = 2;
    return x + y;
}

int main()
{
    int y = 5;
    int k = 3;
    cout << dog( (k) + y );
    return 0;
}
```

The value of  $k$  is 3, and that of  $x = (k)+y$  is  $3 + 5 = 8$ , because in the environment of `main`, the value of  $y$  is 5. The value printed, which is that of  $x+y$ , is 10, because in the environment of `dog`, the value of  $y$  is 2.

14. [5] The Haskell Prelude defines

```
max :: Ord a => a -> a -> a
max x y | x >= y    = x
        | otherwise = y
```

which returns the larger of its arguments. Define `max` in C++, making its meaning as similar as possible to the Haskell definition.

```
template <class T>
T max( T x, T y ){ return x >= y ? x : y; }
```

15. [15] The C++ function `find`

```
int find( float x ) {  
    int loc = A.lookup( x );  
    if( loc < 0 ) loc = B.lookup( x );  
    if( loc < 0 ) loc = 0;  
    return loc;  
}
```

is designed to use a function

```
int Table::lookup( float );
```

which indicates lookup failure by returning a negative result.

The latest version of the `lookup` function never returns a negative result; in case of failure, it throws an exception instead. Its declaration:

```
int Table::lookup( float ) throw ( Table::NotFound );
```

Rewrite `find` to use the new version of `lookup`.

A solution:

```
int find( float x ) {  
    int loc = 0;  
    try {  
        loc = A.lookup( x );  
    }  
    catch( Table::NotFound ){  
        try {  
            loc = B.lookup( x );  
        }  
        catch( Table::NotFound ) {  
            loc = 0;  
        }  
    }  
    return loc;  
}
```

16. [10] In the following code, redefine class `Delta` so that instead of having a *member variable* of type `Beta`, it is *derived* from class `Beta`. The new version of `Delta` should behave in all respects like the original one.

```
class Beta {  
public:  
    Beta( int n ): bn(n) {}  
    int value(){ return bn; }  
private:  
    int bn;  
};  
  
class Delta {  
public:  
    Delta( int n ): b(n) {}  
    int value(){ return b.value(); }  
private:  
    Beta b;  
};
```

A solution:

```
class Delta : Beta {  
public:  
    Delta( int n ): Beta(n) {}  
    Beta::value;  
};
```

- 17.** [10] A histogram is a bar chart like the ones in which I've presented the score distributions for the midterm exams in this class. In such a chart, each bar corresponds to a range of grades, and the height of the bar indicates the number of papers whose grades are within that range. Here is a fragment of a C++ program that computes a histogram for a collection of grades. Array variables `range` and `count` are global, as is the `const int N`. Each time `getGrade()` is called it returns the next paper's grade.

```
const int N = ...
int range[N];          // global
int count[N+1];       // global
// count[0] = number of grades g such that          g ≤ range[0]
// count[i] = number of grades g such that range[i-1] < g ≤ range[i], for 0 < i < N
// count[N] = number of grades g such that range[N-1] < g
```

At the core of the program is this loop:

```
for( ... )
{
    g = getGrade();
    bar( g )++; // increments the count corresponding to g
}
```

The question is how to define `bar()`.

A solution:

```
int& bar( int grade )
{
    int i = 0;
    while( i < N && range[i] < grade ) i++;
    return count[i];
}
```

- 18.** [10] Consider a Haskell function

```
pipe = h0 . h1 . h2
```

and define a Unix-shell pipe program which is analogous to `pipe`, assuming that you have Unix programs `u0`, `u1`, and `u2` analogous to `h0`, `h1`, and `h2`.

```
u2 | u1 | u0
```

Suppose that `h1` is defined by

```
h1 :: [String] -> [String]
h1 = map toUpper
```

Write the analogous program `u1` in C or C++. You may assume

- the maximum length of a `String` in `h1`'s argument is 100
- the C function `char toupper( char )` mimics the `toUpper` function of Haskell.

```
#include <iostream.h>
int main()
{
    char string[101];
    while( cin )
    {
        cin.getLine( string, 100 );
        for( int k = 0; string[k] != 0; k++ )
            string[k] = toupper( string[k] );
        cout << string << endl;
    }
    return 0;
}
```