

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

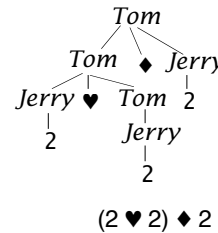
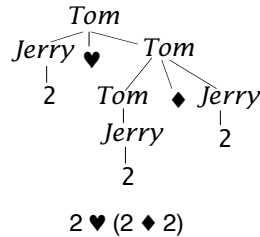
1. [15] Demonstrate, using one or more sample strings and their *parse trees*, that the following grammar is ambiguous.

$$\begin{aligned} Tom &::= Jerry \heartsuit Tom \mid Jerry \mid Tom \spadesuit Jerry \\ Jerry &::= 0 \mid 2 \mid 4 \end{aligned}$$

The string

$$2 \heartsuit 2 \spadesuit 2$$

can be parsed in two ways:



This can be seen as either a precedence problem or an associativity problem— two operators of the same precedence have opposite associativities.

Change the grammar to an unambiguous one which describes the same language.

It can be fixed by changing the precedence relationship between the operators:

$$\begin{aligned} Tom &::= Jerry \heartsuit Tom \mid Ben \\ Ben &::= Ben \spadesuit Jerry \mid Jerry \\ Jerry &::= 0 \mid 2 \mid 4 \end{aligned}$$

... or by changing their associativities:

$$\begin{aligned} Tom &::= Tom \heartsuit Jerry \mid Tom \spadesuit Jerry \mid Jerry \\ Jerry &::= 0 \mid 2 \mid 4 \end{aligned}$$

2. [10] Translate the following Haskell expression into an equivalent one that uses no patterns

```
case x of
  ([]:[]) -> 1
  []       -> 2
  []       -> 3
  [_,_]   -> 4
  _       -> 5
```

```
if null x then 3
else if null (tail x) then
  if null (head x) then 1 else 2
else if null (tail (tail x)) then 4
else 5
```

3. [10] Consider the three evaluation strategies

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

Indicate which of them would be most efficient, in terms of argument evaluations, for each of the following functions by circling the appropriate number. If two strategies are equally efficient and more efficient than the third, circle both.

$f\ a\ b = \text{if } a > 0 \text{ then } 3 \text{ else } b \div 2$	1	<input checked="" type="radio"/> 2	<input type="radio"/> 3	(no sharing, nonstrict)
$g\ x\ y = (x+y) * (x-y)$	<input type="radio"/> 1	<input type="radio"/> 2	<input type="radio"/> 3	(sharing, strict)
$h\ r\ s = \text{if } s >= 6 \text{ then } s+r \text{ else } 2$	1	<input type="radio"/> 2	<input type="radio"/> 3	(sharing, nonstrict)
$m\ p\ q = 3*p - 4*q$	<input type="radio"/> 1	<input checked="" type="radio"/> 2	<input type="radio"/> 3	(no sharing, strict)

4. [10] A 1-rotation of a list moves its first element to the end; an n -rotation of a list performs n successive rotations on it. For example,

```
rotate 3 [1,2,3,4,5,6,7,8] ~> [4,5,6,7,8,1,2,3]
rotate 8 [1,2,3,4,5]       ~> [4,5,1,2,3]
rotate (-3) [1,2,3,4,5]    ~> [3,4,5,1,2]
```

- a. Define `rotate` nonrecursively.

```
> rotate :: Int -> [a] -> [a]
> rotate n xs = drop k xs ++ take k xs where k = n `mod` length xs
```

Alternative:

```
> rotate n xs = ds ++ ts where (ts,ds) = splitAt (n `mod` length xs) xs
```

- b. Define `rotations` nonrecursively so that `rotations xs` is a list of all rotations of `xs`. For example,

```
rotations [1,2,3] ~> [[1,2,3],[2,3,1],[3,1,2]]
> rotations :: [a] -> [[a]]
> rotations xs = [ rotate n xs | n <- [0 .. length xs-1 ] ]
```

5. [10] Lists of factorials are useful in computing partial sums of series such as

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

Define the infinite list of factorials $[0!, 1!, 2!, \dots]$ so that if the elements are evaluated in order, evaluating each element after the first requires only a single multiplication. For full credit, the cost of evaluating any prefix of the list should be linear in its length.

This solution's use of `(!!)` makes its cost quadratic:

```
> facs :: [Integer]
> facs = [1] ++ [ fromInt n * facs!!(n-1) | n <- [1..] ]
```

Another solution, which doesn't use `(!!)` and whose cost is linear:

```
> facs :: [Integer]
> facs = [1] ++ [ n*f | (n,f) <- zip [1..] facs ]
```

6. [15] Prove that

$$c (\text{map } f) (\text{map } g) = \text{map } (c \text{ f } g)$$

given the definitions

```
map h [] = []
map h (x:xs) = h x : map h xs
c f1 f2 n = f1 (f2 n)
```

Hint: Apply both sides to an arbitrary argument.

Because `map`'s second argument is a list, we apply both sides to an arbitrary list `xs`:

```
c (map f) (map g) = map (c f g)
= c (map f) (map g) xs = map (c f g) xs
```

Now we proceed by list induction.

Proof:

Case `[]`:

```
c (map f) (map g) [] = map (c f g) []
= { def. c }
map f (map g []) = map (c f g) []
= { def. map }
map f [] = []
= { def. map }
[] = []
= { identity }
true
```

Case `(a:as)`

```
= c (map f) (map g) (a:as)
= { def. c }
map f (map g (a:as))
= { def. map }
map f (g a : map g as)
= { def. map }
f(g a) : map f (map g as)
= { def. c }
(c f g) a : map f (map g as)
= { induction hypothesis }
(c f g) a : map (c f g) as
= { def. map }
map (c f g) (a:as) ■
```

7. [10] Determine formally whether the following predicate holds:

$$\{2*a+b > 6\} \ a, b := 2*a+b, 2*b-a; \ a := a+b \ \{a-b > 2\}$$

First we find the weakest precondition for this command and postcondition:

$$\begin{aligned} & \text{wp}(“a, b := 2*a+b, 2*b-a; \ a := a+b”, \ a-b > 2) \\ &= \{ \text{def of “;”} \} \\ & \quad \text{wp}(“a, b := 2*a+b, 2*b-a”, \ \text{wp}(“a := a+b”, \ a-b > 2)) \\ &= \{ \text{def. of “:=”} \} \\ & \quad \text{wp}(“a, b := 2*a+b, 2*b-a”, \ (a-b > 2)_{a+b}^a \ b) \\ &= \{ \text{substitution} \} \\ & \quad \text{wp}(“a, b := 2*a+b, 2*b-a”, \ a+b-b > 2) \\ &= \{ \text{algebra} \} \\ & \quad \text{wp}(“a, b := 2*a+b, 2*b-a”, \ a > 2) \\ &= \{ \text{def. of “:=”} \} \\ & \quad (a > 2)_{2*a+b, 2*b-a}^{a, b} \\ &= \{ \text{substitution} \} \\ & \quad 2*a+b > 2 \\ &\Leftarrow \{ \text{pred. calc.} \} \\ & \quad 2*a+b > 6 \end{aligned}$$

The last step determines that the weakest precondition follows from (i.e., is implied by) the given precondition, and hence that the predicate holds.

8. [15] Use a Haskell algebraic type to define a type of sets of integers. Use the same array-of-booleans (i.e., *characteristic vector*) representation as in Homework 7, using lists instead of arrays, and using the list-indexing operator (!!) :: [a] -> Int -> a. Define functions

- `makeSet :: (Int,Int) -> [Int] -> SetOfInt` so that `makeSet (3,12) [9,4,5,11]` gives a set whose “base type” (in Pascal terminology) is the subrange `[3 .. 12]` containing the integers 4, 5, 9, and 11.
- `elemS :: Int -> SetOfInt -> Bool` which returns a value indicating whether the integer is a member of the set.
- `(==) :: SetOfInt -> SetOfInt -> Bool` so that two sets can be compared using the equals operator.

```
> data SetOfInt = Set [Bool] Int Int deriving (Eq)
> makeSet :: (Int,Int) -> [Int] -> SetOfInt
> makeSet (lo,hi) ints = Set [ i `elem` ints | i <- [lo..hi] ] lo hi
> elemS :: Int -> SetOfInt -> Bool
> i `elemS` (Set bs lo hi) = lo <= i && i <= hi && bs!!(i-lo)
```

Alternative for `...deriving (Eq)`:

```
> instance Eq SetOfInt where
>   (Set b1 lo1 hi1) == (Set b2 lo2 hi2) = (b1,lo1,hi1) == (b2,lo2,hi2)
```

9. [10] Define a “short-circuit” multiplication function `sm`, which evaluates only one of its arguments when that is enough to give the answer. Choose any suitable programming language from the ones we have studied this semester.

The only suitable language is Haskell, because all of the others bind arguments to parameters by value (by reference wouldn’t work here, because it requires arguments to have *l*-values).

```
sm :: Num a => a -> a -> a
0 `sm` y = 0
x `sm` y = x*y
```

10. [15] Given the following program:

```

integer x; integer y = 20; integer z = 30;
integer function p ( integer y )
{
    return x + y;
}
integer function q ( integer x )
{
    return p ( x );
}
main ()
{
    x := 10;
    print p ( x );
    print q ( z );
}
    
```

predict the output assuming

a. static (lexical) binding

```

p ( x ) where x = 10
p ( 10 )
x + y where y = x, x = 10
10 + 10
20
    
```

```

q ( z ) where z = 30
q ( 30 )
p ( 30 )
x + y where y = z, x = 10, z = 30
10 + 30 where y = z, x = 10, z = 30
40
    
```

b. dynamic binding

```

p ( x ) where x = 10
p ( 10 )
x + y where y = x, x = 10
10 + 10
20
    
```

```

q ( z ) where z = 30
q ( 30 )
x + y where y = z, x = z, z = 30
30 + 30
60
    
```

11. [20] Many airplanes are equipped with *spoilers*, which can be deployed to eliminate the wings' lift. Spoilers are used to ensure that, once an airplane has touched down, it stays down, rather than bouncing into the air again. For obvious reasons, it's not good for the spoilers to be deployed while the airplane is still flying, and requiring the crew to command spoiler deployment at the moment of touchdown would add to their responsibilities at a time when they are already busy enough.

Both of these problems can be solved by an Ada task `Spoilers` which delays actual spoiler deployment until the plane actually touches down. The task should handle the following requests from clients:

- **Deploy:** cockpit-control client requests spoiler deployment; this request is accepted only when the plane's current altitude is zero. When task `Spoilers` accepts this request, it calls the procedure `DeploySpoilersNow` which causes the spoilers to be deployed immediately (you should assume that `DeploySpoilersNow` is provided for you to use).
- **Altitude:** radar-altimeter client reports to task `Spoilers` the current altitude above the ground.
- **Deployed:** cockpit-display client inquires whether spoilers have been deployed (task `Spoilers`' response is boolean).

```

task Spoilers is
  entry Deploy;
  entry Altitude ( alt : in integer );
  entry Deployed ( dep : out boolean );
end Spoilers;

task body Spoilers is
  deployed : boolean;
  altitude : integer;
begin
  deployed := false;
  altitude := 10000;
  loop
    select
      when altitude = 0 =>
        accept Deploy do
          DeploySpoilersNow();
          deployed := true;
        end Deploy;
      or
        accept Altitude ( alt : in integer ) do
          altitude := alt;
        end Altitude;
      or
        accept Deployed ( dep : out boolean ) do
          dep := deployed;
        end Deployed;
    end select;
  end loop;
end Spoilers;

```

12. [10] In the following C++ 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) {}
  void update( 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();
};

```

Alternative solution:

```

class Delta : private Beta {
public:
  Delta( int n ){ update( n ); }
  int value(){ return Beta::value(); }
};

```

13. [15] If the following C++ program would compile, what would be its output? If it would not compile, what would the diagnostic say?

```
class Alpha
{
public:
    virtual void a() { cout << "1 "; }
    virtual void c() { cout << "2 "; }
    virtual void f() { a(); c(); }
};

void coca( Alpha v, Alpha& r )
{
    v.f(); r.f();
}

class Bravo : public Alpha
{
public:
    virtual void a() { cout << "3 "; }
    virtual void c() { cout << "4 "; }
};

int main()
{
    Alpha x; Bravo y;

    x.f(); x.a(); x.c(); cout << endl;
    y.f(); y.a(); y.c(); cout << endl;

    coca( y, y ); cout << endl;

    return 0;
}
```

It compiles OK. The output:

```
1 2 1 2
3 2 3 4
1 2 3 2
```