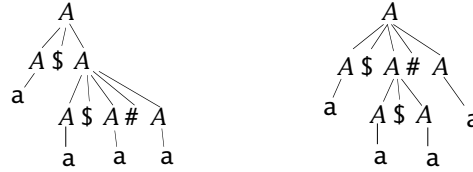


1. [15] Consider the following EBNF grammar:

$$A ::= A \$ A \mid A \$ A \# A \mid a$$

- a. Show that the grammar is ambiguous by finding a string that has two different parse trees, and draw the parse trees.

a \$ a \$ a # a



- b. Change the grammar to require parentheses that eliminate the ambiguity.

$$A ::= A \$ ('A') \mid A \$ ('A \# A') \mid a$$

- c. Write two strings in the changed grammar, one reflecting each of the two interpretations of the ambiguous string.

a \$ (a \$ (a # a))
a \$ (a \$ (a) # a)

2. [10] Translate the following expression as required for evaluation using a stack, and show the steps of its stack evaluation.

$$3 + ((4 * 5) - (9 / 3))$$

The solution (stack contents underlined):

	3 4 5 * 9 3 / - +
<u>3</u>	4 5 * 9 3 / - +
<u>3</u> 4	5 * 9 3 / - +
<u>3</u> 4 5	* 9 3 / - +
<u>3</u> 20	9 3 / - +
<u>3</u> 20 9	3 / - +
<u>3</u> 20 9 3	/ - +
<u>3</u> 20 3	- +
<u>3</u> 17	+
<u>20</u>	

3. [16] Define in Haskell a function `once` so that `once xs` is a list containing only the elements of `xs` that occur exactly once in `xs`.

- a. Do not use explicit recursion— use list comprehensions and Prelude functions only. Include the function's most general type.

```
> once :: Eq a => [a] -> [a]
> once xs = [ x | x <- xs, null (tail [ y | y <- xs, y == x ]) ]
```

- b. Again, using Prelude functions only— no recursion, and no list comprehensions.

```
> once xs = filter unique xs
> where
> unique x = null (tail (filter (==x) xs))
```

4. [15] Write a Haskell program `adder` that inputs successive lines until it inputs an empty line, at which point it outputs the sum of the lines that were integer numerals (ignoring the lines that were not integer numerals).

```
> adder :: IO ()
> adder = add 0
>   where
>     add n = do s <- getLine
>               if null s
>               then print n
>               else case reads s of
>                    [(m, "")] -> add (m+n)
>                    _         -> add n
```

5. [10] Find formally the weakest Q such that

$$\{Q\} \ c, a, b := b+2, c-3, a+b \ \{a \leq b < c\}$$

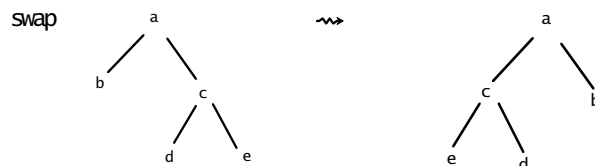
A solution:

$$\begin{aligned} & \text{wp}(\text{"c, a, b := b+2, c-3, a+b", } a \leq b < c) \\ &= (a \leq b < c)_{b+2, c-3, a+b}^{c, a, b} \\ &= c-3 \leq a+b < b+2 \end{aligned}$$

6. [10] Given this definition

```
> data Tree a = Nil | Node (Tree a) a (Tree a)
```

define the function `swap :: Tree a -> Tree a` so that `swap t` is a tree in which `t`'s sibling subtrees are interchanged. For example,



A solution:

```
swap Nil = Nil
swap (Node t1 x t2) = Node (swap t2) x (swap t1)
```

7. [15] Design a Haskell algebraic type `Angle` with instance declarations for `Eq`, `Ord`, and `Show`. Allow for an angle to be represented either in degrees as an `Int` or in radians as a `Float` (2π radians = 360°).

```
> data Angle = Deg Int | Rad Double
> rads :: Angle -> Double
> rads (Deg a) = fromInt a * pi / 180
> rads (Rad a) = a
> instance Eq Angle where
>   a1 == a2 = rads a1 == rads a2
> instance Ord Angle where
>   a1 <= a2 = rads a1 <= rads a2
> instance Show Angle where
>   show (Rad a) = show a ++ " radians"
>   show (Deg a) = show a ++ "°"
```

8. [12] For each of the following code samples, determine (i) whether a static type check would detect an error, and (ii) whether executing the code would result in a dynamic type error. Assume that the code is C, but with either static or dynamic type checking (actual C, of course, has only static checking).

Indicate your answer by circling **one** of **s** static, **d** dynamic, **b** both, **n** neither.

For all of the samples assume the following declarations:

```
int x = -1, y = 2; struct S{ int p, q; }; struct S z = {3,4};
```

- a. `int q = x || z;` **s** d b n
b. `int q = x && z;` s d **b** n
c. `int q = x && y;` s d b **n**
d. `int q = x ? y : z;` **s** d b n
9. [10] Predict this program's output and give a brief explanation of it:

```
#include <iostream.h>
char* s = "alligator";
int k = 5;
#define debug(n) (cout << n << " " << s << "\n");
int f ( int s )
{
    k += s;
    debug( k );
    return k;
}

int main()
{
    int k = 12;
    float s = 6.39;
    debug( s );
    f( 9 );
    return 0;
}
```

The program macro-expands to

```
#include <iostream.h>
int k = 5;
int f ( int s )
{
    k += s;
    (cout << k << " " << s << "\n");
    return k;
}

int main()
{
    int k = 12;
    float s = 6.39;
    (cout << s << " " << s << "\n");
    f( 9 );
    return 0;
}
```

so its output is

```
6.39 6.39
14 9
```

In the function `f`, static binding means that the references to `k` in `f`'s body always refer to the global variable `k`. In the macro, however, the reference to `s` is bound dynamically; in `main` it refers to the float `s`, whereas in `f` it refers to the parameter `int s`.

10. [12] Given the following fragmentary Pascal program

```
program CS345;
  var x : real;
  type S = record a, b : integer;
  y : integer;
  procedure f;
    var z : ^real;
  begin
    ...
  end;
  procedure g;
    var d : S;
        x : integer;
        b : real;
  begin
    ... x; ... y; ... d.b
  end;
begin
  ...
end.
```

assuming that integers and pointers occupy 4 bytes each, and reals occupy 8 bytes, what offsets would the compiler generate for the three variable references in procedure g? Indicate a global reference by appending G to its offset.

```
x: 8
y: 8 G
d.b: 4
```

11. [10] Suppose you have a Haskell expression

```
hf1 . hf2 . hf3
```

and you want to define an analogous Unix pipe expression using `uf1`, `uf2`, and `uf3` which are Unix analogs of `hf1`, `hf2`, and `hf3` respectively.

- a. Write the Unix pipe expression.

```
uf3 | uf2 | uf1
```

- b. Now suppose that `hf2 = unlines . map reverse . lines`. Write `uf2` in C++, assuming that you have an appropriate `reverse` function in your C++ library. Note any assumptions you have to make in your C++ code.

```
#include <iostream.h>

int main()
{
  char line[100]; // assume 100 chars is enough

  while( cin )
  {
    cin.getline( line );
    reverse( line );
    cout << line;
  }

  return 0;
}
```

12. [15] How does the following C++ class definition fail to satisfy the Law of the Big Three?

```
#include <iostream.h>

class String
{
public:
    String( const char* );
    ~String(){ free(); }
    void print() const;
private:
    char* chars;
    int length;
    void free(){ delete chars; }
};

String::String( const char* s )
{
    length = strlen( s );
    chars = new char[ length+1 ];
    strcpy( chars, s );    // copies s to chars
}

void String::print() const
{
    cout << chars << endl;
}
```

It has a destructor, but it lacks an assignment operator and a copy constructor.

Add what is needed to satisfy the law.

```
#include <iostream.h>

class String
{
public:
    String( const char* );
    String( const String& );           // copy constructor
    ~String(){ free(); }
    String& operator=( const String& ); // assignment operator
    void print() const;
private:
    char* chars;
    int length;

    void copy( const String& );
    void free(){ delete chars; }
};

String::String( const char* s )
{
    length = strlen( s );
    chars = new char[ length ];
    strcpy( chars, s );
}

String::String( const String& b )
{
    copy( b );
}

String& String::operator=( const String& b )
{
    if( chars != b.chars )
    {
        free();
        copy( b );
    }
    return *this;
}
```

```
void String::copy( const String& b )
{
    length = b. length;
    chars = new char[ length+1 ];
    strcpy( chars, b.chars );
}

void String::print() const
{
    cout << chars << endl;
}
```

13. [15] Define, in C++ and then in Haskell, a polymorphic type `Pair` whose values have two fields of the same type. In C++, you'll define a class template; in Haskell, an algebraic type.

Then for both the C++ and Haskell `Pair` types, define the operator `(<=)` so that `Pairs` are ordered lexicographically.

```
template < typename T >
class Pair
{
public:
    T a; T b;
};

template < typename T >
int operator<=( Pair<T> x, Pair<T> y )
{
    return x.a < y.a ||
           (x.a == y.a && x.b <= y.b);
}
```

```
data Pair a = Pr a a

instance Eq a => Eq (Pair a) where
    Pr xa xb = Pr ya yb = xa==ya && xb==yb

instance Ord a => Ord (Pair a) where
    Pr xa xb <= Pr ya yb
    = xa < ya || (xa == ya && xb <= yb)
```