

1. [15] Suppose that a Haskell script contains the following operator declarations (they're not all valid in Haskell, but assume that they are valid for the purposes of this question):

```
infixl 5 ◊, ‡
infixr 4 §, •
```

Define a BNF grammar for expressions involving the four operators given above (with the specified precedences and associativities), parentheses, and operands consisting of one or more lowercase letters.

```
<exp> ::= <term> § <exp> | <term> • <exp> | <term>
<term> ::= <term> ◊ <factor> | <term> ‡ <factor> | <factor>
<factor> ::= ( <exp> ) | <operand>
<operand> ::= <letter> | <operand> <letter>
<letter> ::= a | b | c | ... | z
```

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

|  |  |
|--|--|
| <pre>case x of   ([]:[]) -&gt; 1   []       -&gt; 2   []       -&gt; 3   [_,_]   -&gt; 4   _       -&gt; 5</pre> | <pre>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</pre> |
|--|--|

3. [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.

|   |  |                                  |
|---|--|----------------------------------|
| <pre>f x y = case x of   0 -&gt; y+2   _ -&gt; 42</pre> | <input checked="" type="checkbox"/> 1    2    3                                  | <pre>nonstrict, no sharing</pre> |
| <pre>f x y = if y&lt;0 then 3 else y^2</pre>            | <input checked="" type="checkbox"/> 1 <input checked="" type="checkbox"/> 2    3 | <pre>nonstrict, sharing</pre>    |
| <pre>f x y = if x&gt;0 then x+y else y*2</pre>          | 1 <input checked="" type="checkbox"/> 2    3                                     | <pre>strict, sharing</pre>       |
| <pre>f x y = (3+x) * (4-y)</pre>                        | 1    2    3  | <pre>strict, no sharing</pre>    |

4. [10] A list *ys* is called a *sublist* of list *xs* if *ys* can be obtained by deleting zero or more elements from *xs*. For example, “bs” is a sublist of “sublist”, and [2,5] is a sublist of [1..].

Define a function `sublists` which produces all of a list's sublists. For example, the sublists of “abc” are

```
["", "c", "b", "bc", "a", "ac", "ab", "abc"]
```

The sublists' order doesn't matter.

```
> sublists :: [a] -> [[a]]
> sublists [] = [[]]
> sublists (c:cs) = scs ++ [ c : sc | sc <- scs ] where scs = sublists cs
```

5. [10] The constant  $e$  can be defined by the power-series equation

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Define a Haskell list whose elements are the terms of the series. Then define a Haskell expression that computes an approximation of  $e$  by adding the elements of the list up to the first one whose magnitude is less than `eps`. You may use any of the “standard” functions we defined in class (such as `sum` and `takeWhile`). Note: Your answer will be judged in part on its efficiency.

```
> series :: [Double]
> series = 1 : [ t / fromInt n | (t,n) <- zip series [1..] ]

> e :: Double
> e = sum (takeWhile (>=eps) series)
```

6. [15] Find the weakest precondition such that execution of

$$a, b := 5, 7; a, b, c := b + 2, c - 3, a + b$$

establishes  $a \leq b < c$ .

$$\begin{aligned} & \text{wp}(“a, b := 5, 7; a, b, c := b + 2, c - 3, a + b”, a \leq b < c) \\ &= \text{wp}(“a, b := 5, 7”, \text{wp}(“a, b, c := b + 2, c - 3, a + b”, a \leq b < c)) \\ &= \text{wp}(“a, b := 5, 7”, (a \leq b < c)_{b+2, c-3, a+b}^{a, b, c}) \\ &= \text{wp}(“a, b := 5, 7”, b + 2 \leq c - 3 < a + b) \\ &= (b + 2 \leq c - 3 < a + b)_{5, 7}^{a, b} \\ &= 7 + 2 \leq c - 3 < 5 + 7 \\ &= 12 \leq c < 15 \end{aligned}$$

7. [15] Prove that

$$\text{length}(\text{reverse } xs) = \text{length } xs$$

for all lists `xs`.

You may use, without proving them here, any theorems proven in class or in homework exercises. For reference, here are some relevant definitions:

$$\begin{aligned} \text{length} [] &= 0 \\ \text{length} (\_ : rs) &= 1 + \text{length } rs \\ \text{reverse} [] &= [] \\ \text{reverse} (a : as) &= \text{reverse } as ++ [a] \\ [] ++ bs &= bs \\ (a : as) ++ bs &= a : (as ++ bs) \end{aligned}$$

Proof:

$$\begin{aligned} & \text{Case } []: \\ & \text{length}(\text{reverse } []) = \text{length } [] \\ &= \quad \{ \text{def. reverse} \} \\ & \text{length } [] = \text{length } [] \\ &= \quad \{ \text{reflexivity of } = \} \\ & \text{true} \end{aligned}$$

```

Case (x:xs):
  length (reverse (x:xs)) = length (x:xs)
=   { def. reverse }
  length (reverse xs ++ [x]) = length (x:xs)
=   { theorem proven in class: length (xs++ys) = length xs + length ys }
  length (reverse xs) + length [x] = length (x:xs)
=   { change notation for [x] }
  length (reverse xs) + length (x:[]) = length (x:xs)
=   { def. length }
  length (reverse xs) + (1 + length []) = 1 + length xs
=   { def. length }
  length (reverse xs) + (1 + 0) = 1 + length xs
=   { algebra }
  1 + length (reverse xs) = 1 + length xs
=   { induction hypothesis }
  1 + length xs = 1 + length xs
=   { reflexivity of = }
true      ■

```

8. [15] Suppose an algebraic type for  $n$ -ary trees with labelled leaves is defined by

```
data Gtree a = Leaf a | Node [Gtree a]
```

- a. Define a function `depthGT` that finds the depth of such a tree (i.e., the length of the longest path from the root to a leaf).

```

depthGT :: Gtree a -> Int
depthGT (Leaf _) = 0
depthGT (Node []) = 0
depthGT (Node ts) = 1 + maximum [ depthGT t | t <- ts ]

```

- b. Define a function that computes the sum of such a tree.

```

sumGT :: Num a => Gtree a -> a
sumGT (Leaf n) = n
sumGT (Node ts) = sum [ sumGT t | t <- ts ]

```

- c. Define an (`=`) operation for such a tree so that two trees are equal if they are both leaves with equal labels or they have equal subtrees.

```

instance Eq t => Eq (Gtree t) where
  Leaf m   = Leaf n   = m == n
  Node t1s = Node t2s = t1s == t2s
  _       = _         = False

```

9. [10] Translate the following C fragments into guarded-command notation.

- a. `do { x = x + a; y = x + y; } while (x - y > a);`

```

x := x + a; y := x + y;
do x - y > a → x := x + a; y := x + y; od

```

- b. `x = (a++ || ++b) + (b = x+b)`, assuming left-to-right evaluation

```

t := a; a := a + 1;
if t≠0 → b := x + b; x := b + 1
  || t=0 → b := b + 1;
  if b=0 → t := 0 || b≠0 → t := 1 fi;
b := x + b; x := t + b;
fi

```

10. [10] Given the following code

```
integer x, y = 2;

integer calvin( integer a ) {
    return a*x + y;
}

integer hobbes( integer b ) {
    integer x = 10;
    return calvin( x + b );
}

main() {
    integer y = 20;
    x := 5;
    print( hobbes( y ) );
}
```

predict the value printed, assuming functions' free variables are bound

a. statically

```
hobbes( y ) where y = 20, x = 5
hobbes( 20 ) where y = 20, x = 5
(calvin( x + b ) where x = 10, b = 20) where y = 20, x = 5
calvin( 10 + 20 ) where y = 20, x = 5
calvin( 30 ) where y = 20, x = 5
(a*x + y where a = 30, x = 5, y = 2) where y = 20, x = 5
30*5 + 2
152
```

b. dynamically

```
hobbes( y ) where y = 20, x = 5
hobbes( 20 ) where y = 20, x = 5
(calvin( x + b ) where x = 10, b = 20) where y = 20, x = 5
(calvin( 10 + 20 ) where x = 10, b = 20) where y = 20, x = 5
(calvin( 30 ) where x = 10, b = 20) where y = 20, x = 5
((a*x + y where a = 30) where x = 10, b = 20) where y = 20, x = 5
(30*x + y where x = 10, b = 20) where y = 20, x = 5
30*10 + 20
320
```

11. [10] Given the following Haskell definitions:

```
> data Temp = Fahrenheit Float | Celsius Float
> freezing :: Temp -> Bool
> freezing (Fahrenheit t) = t <= 32
> freezing (Celsius t)   = t <= 0
```

define comparable types and a comparable freezing function in C++ using class derivation.

```
class Temp {
public:
    virtual bool freezing() = 0;
    float t;
};

class Fahrenheit : public Temp {
public:
    Fahrenheit( float temp ) : t(temp) {}
    bool freezing(){ return t <= 32; }
};

class Celsius : public Temp {
public:
    Celsius( float temp ) : t(temp) {}
    bool freezing(){ return t <= 0; }
};
```

12. [15] Suppose there are three Ada tasks A, B, and C, all competing repeatedly for some given resource. Define a task `FairShare` with entries which can be called by the tasks A, B, and C to ensure that none of them ever acquires the resource twice in succession. That is, between any two successive acquisitions by a given task, the resource must be acquired at least once by one of the other two tasks.

```
task FairShare is
  entry Acheck;
  entry Bcheck;
  entry Ccheck;
end FairShare;
```

```
task body FairShare is
  last : integer;
begin
  last := 0;
  loop
    select
      when last /= 1 =>
        accept Acheck do
          last := 1;
        end Acheck;
      or
      when last /= 2 =>
        accept Bcheck do
          last := 2;
        end Bcheck;
      or
      when last /= 3 =>
        accept Ccheck do
          last := 3;
        end Ccheck;
    end select;
  end loop;
end FairShare;
```

13. [15] The following C++ class definition lacks certain features that are necessary to prevent memory leaks and to give it value semantics. Add those features, indicating with arrows where they belong (when it matters).

```
#include <iostream.h>
#include <string.h>

class String
{
public:
  String( char* );           // initializes a String by copying a character array
  String cat( const String& ); // concatenates two strings, producing a new string
  const char* strPtr() const { return chars; }
private:
  String( int ); // not for public use
  int len;      // number of characters (does not count terminating null)
  char* chars;  // the String's characters
};

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

String String::cat( const String& b )
{
  String r( len + b.len + 1 );

  strcpy( r.chars, chars );
  strcpy( r.chars + len, b.chars );

  return r;
}

String::String( int n )
{
  len = n;
  chars = new char[ len+1 ];
}
```

Solution (added code in **bold**):

```
#include <iostream.h>
#include <string.h>

class String
{
public:
    String( char* );           // initializes a String by copying a character array

    // The Big Three
    String( const String& );   // copy constructor
    ~String(){ delete chars; }; // destructor
    String& operator=( const String& ); // assignment operator

    String cat( const String& ); // concatenates two strings, producing a new string

    const char* strPtr() const { return chars; }

private:
    String( int ); // not for public use

    int len;      // number of characters (does not count terminating null)
    char* chars;  // the String's characters
};

String::String( const String& b )           // copy constructor
{
    len = b.len;
    chars = new char[ len+1 ];
    strcpy( chars, b.chars );
}

String& String::operator=( const String& b ) // assignment operator
{
    if( chars != b.chars )
    {
        delete chars;
        len = b.len;
        chars = new char[ len+1 ];
        strcpy( chars, b.chars );
    }

    return *this;
}

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

String String::cat( const String& b )
{
    String r( len + b.len + 1 );

    strcpy( r.chars, chars );
    strcpy( r.chars + len, b.chars );

    return r;
}

String::String( int n )
{
    len = n;
    chars = new char[ len+1 ];
}
```