

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

1. [5] Here are two alternative EBNF productions for the non-terminal *Exp*:

$$Exp ::= Term \{ + Term \} \mid Term \{ - Term \}$$

$$Exp ::= Term \{ + Term \mid - Term \}$$

Are these two productions equivalent? That is, do they define the same language? Explain your answer (a simple *yes* or *no* is **not** sufficient).

- No, the two productions are not equivalent. The first one defines expressions each which may contain either + or - but not both, whereas the second allows expressions containing arbitrary combinations of + and -.
2. [15] Write a BNF grammar for the syntax of telephone numbers as dialed from an ordinary residential phone in the US. Include only the keys actually pressed— that is, omit all punctuation such as '-'. Allow for both local and long-distance calls, and include the restrictions that the middle digit of an area code is either 0 or 1 and the first two digits of a local (7-digit) number are never 0 or 1. You *need not* allow for overseas calls, nor for calls to emergency services (911) or directory assistance (1411).

$$\begin{aligned} \langle phone \rangle &::= \langle local \rangle \mid \langle dist \rangle \langle local \rangle \\ \langle local \rangle &::= \langle z \rangle \langle z \rangle \langle d \rangle \langle d \rangle \langle d \rangle \langle d \rangle \langle d \rangle \\ \langle dist \rangle &::= 1 \langle d \rangle \langle a \rangle \langle d \rangle \\ \langle z \rangle &::= 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \langle a \rangle &::= 0 \mid 1 \\ \langle d \rangle &::= \langle a \rangle \mid \langle z \rangle \end{aligned}$$

3. [8] Determine which of the following grammars (1) and (2) is ambiguous.

1. $\langle X \rangle ::= A \langle X \rangle Z \mid A \langle X \rangle B \langle X \rangle Z \mid a$

2. $\langle Y \rangle ::= A \langle Y \rangle \mid A \langle Y \rangle B \langle Y \rangle \mid a$

- a. Circle the number of the ambiguous grammar: 1 2
- b. Show that grammar's ambiguity by giving two different derivations for a single expression.

$\langle Y \rangle$	$\langle Y \rangle$
$\Rightarrow A \langle Y \rangle$	$\Rightarrow A \langle Y \rangle B \langle Y \rangle$
$\Rightarrow A A \langle Y \rangle B \langle Y \rangle$	$\Rightarrow A A \langle Y \rangle B \langle Y \rangle$
$\Rightarrow A A a B a$	$\Rightarrow A A a B a$

4. [12] Consider the three evaluation strategies

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

Indicate which of them would be most efficient for each of the following functions by circling the appropriate number. If two strategies are equally efficient, circle both.

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

5. [5] Consider the following expression:

```
let f x y = 2*x + 3*y in map (f 5) [3..6]
```

If the expression is valid, give its value; otherwise, explain why it is not valid.

```
F05AB> let f x y = 2*x + 3*y in map (f 5) [3..6]
[19,22,25,28]
```

6. [10] Using `getLine :: IO String`, define `getPara :: IO [String]` which obtains the next paragraph from the standard input stream. For purposes of this question, a standard-input paragraph is defined as a sequence of lines separated from the following paragraph by an empty line. Your `getPara` should deliver the lines in a list, in the order in which they are obtained from standard input.

```
> getPara :: IO [String]
> getPara = do line <- getLine
>             if null line then return []
>             else do lines <- getPara
>                    return (line:lines)
```

7. [10] Translate the following function

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

- a. into an equivalent function that uses only tail recursion

```
quotient1 :: Int -> Int -> Int
quotient1 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 function (in C) that uses no recursion at all (nor any division operators).

```
int quotient2( int x, int d )
{
    int q = 0;
    while( x >= d )
    {
        x = x - d;
        q = q + 1;
    }
    return q;
}
```

8. [10] Write a Haskell function that generates an infinite list of the terms of the series

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots + \frac{(-1)^n x^{2n}}{(2n)!} + \dots$$

Your function should use terms already computed to reduce the cost of computing later terms, rather than computing each term independently.

```
series :: Float -> [Float]
series x = terms
  where
    terms = 1.0 : [ (-t*x*x)/(n*(n-1.0)) | (t,n) <- zip terms [2.0,4.0..] ]
```

Then use the series to define a function that computes `cos x` by adding all the terms of the series whose absolute values exceed `0.00000001`.

```
cos' x = sum (takeWhile (>0.00000001).abs) (series x)
```

9. [10] Give the most general types of the following Haskell functions

```
f (x:_) (y1:y2:_) z = if x==z then y1 else y2
g zs = f zs zs
```

Answers:

```
f :: Eq a => [a] -> [b] -> a -> b
g :: Eq a => [a] -> a -> a
```

10. [10] Find the weakest precondition such that execution of

```
if  $x < y \rightarrow a, b := b, a \parallel x < z \rightarrow$  skip fi
```

establishes $a \leq b$.

$$\begin{aligned} & \text{wp}(\text{"if } x < y \rightarrow a, b := b, a \parallel x < z \rightarrow \text{skip fi"} , a \leq b) \\ = & \quad \{ \text{def. of if} \} \\ & (x < y \vee x < z) \wedge (x < y \Rightarrow \text{wp}(\text{"}a, b := b, a\text{"} , a \leq b)) \wedge (x < z \Rightarrow \text{wp}(\text{"skip"} , a \leq b)) \\ = & \quad \{ \text{defs. of ':=', skip} \} \\ & (x < y \vee x < z) \wedge (x < y \Rightarrow b \leq a) \wedge (x < z \Rightarrow a \leq b) \\ = & \quad \{ \text{predicate calculus} \} \\ & (x < y \wedge x \geq z \wedge b \leq a) \vee (x < z \wedge x \geq y \wedge a \leq b) \vee (x < y \wedge x < z \wedge a = b) \end{aligned}$$

Informally: At least one guard must hold. If exactly one guard holds, then its command's precondition is the **if...fi**'s precondition; if both guards hold, then the **if...fi**'s precondition is the conjunction of the guards' preconditions.

11. [10] If the following program compiles, what is its output? If it won't compile, what would the diagnostic say?

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

class D : public B
{
public:
    void a() { cout << "3 "; }
    void c() { cout << "4 "; }
};

void main()
{
    B b; D d;
    b.f(); b.c(); cout << endl;
    d.f(); d.c(); cout << endl;

    B* pb = new B; B* pd = new D;
    pb->a(); pb->c(); pd->a(); pd->c();
}
```

It compiles OK. The output:

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

12. [20] The following fragmentary program text contains a type definition, declarations of several variables, and eight 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 is allocated 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** 8

The program:

```

program
  type R = record x : float ; y : char[0..29] ; z : float end ;
  i : int ;
  y : R ;
  z : int [1..10] ;
  j : int ;
  procedure pushme( x : float )
    y : float ;
    a : R ;
  begin
    ... y { A } ... j { B } ...
    ... a.y[2] { C } ... z[j] { D } ...
  end p1 ;
  procedure pullyou( b : int )
    m : bool[0..99] ;
    a : R[0..3] ;
    x : float ;
  begin
    ... x { E } ... a[2].y[2] { F } ...
    ... y.z { H } ... m[2] { J } ...
  end p2 ;
  begin
    ...
  end program .

```

Answer by filling in the table. Indicate the distinction between local and global variables by prefixing 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 build a table for each block showing each variable's size and its address(es) in its block's activation record.

program	i	4	0-3
	y	76	4-79
	y.x	8	4-11
	y.y	60	12-71
	y.z	8	72-79
	z	40	80-119
	j	4	120-123
pushme	y	8	0-7
	a	76	8-83
	a.x	8	8-15
	a.y	60	16-75
	a.z	8	76-83
pullyou	m	100	0-99
	a	304	100-403
	x	8	404-311

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

	points		points
A. 0	2	E. 404	2
B. G 120	2	F. $100 + 2*76 + 8 + 2*2 = 264$	4
C. $20 (= 16 + 2*2)$	2	H. G 72	2
D. G $(80 + (\uparrow 120G-1)*4)$	4	J. $2 (= 0 + 2)$	2

- 13.** [10] Determine the most general type of the following function, showing your reasoning:

```
f m n [] = n
f m n (x:xs) = f m (m n x) xs
```

Initially, let

```
f :: a -> b -> c -> d
```

then

```
m :: a
n :: b
x:xs :: c
f m n (x:xs) :: d
f m n [] :: d
f m (m n x) xs :: d
```

Also, let

```
x :: e
```

Then

<u>type equation</u>	<u>from</u>
$c = [e]$	$x:xs :: c, x :: e, (:) :: t -> [t] -> [t]$
$a = b -> e -> b$	$n :: b, x :: e, (m n x) :: b$
$b = d$	$n :: b, f :: a -> b -> c -> d, f m n [] = n$

Substituting gives

```
f :: (b->e->b) -> b -> [e] -> b
```

i.e. (changing variables),

```
f :: (a->b->a) -> a -> [b] -> a
```

- 14.** [15] Define a C++ equivalent for the following Haskell code:

```
> module ModulePair
>   (Pair, pair, left, right)
> where
>
>   data Pair a b = Pr a b
>
>   pair :: a -> b -> Pair a b
>   pair x y = Pr x y
>
>   left :: Pair a b -> a
>   left (Pr x _) = x
>
>   right :: Pair a b -> b
>   right (Pr _ y) = y
```

A solution:

```
template < typename a, typename b >
class Pair
{
public:
    Pair( a x, b y ) : _x(x), _y(y) {}
    a left() const { return _x; }
    b right() const { return _y; }
private:
    a _x;
    b _y;
};
```

How would your solution change if the line (Pair, pair, left, right) were omitted?

A: The member variables `_x` and `_y` would become `public`.

15. [20] Given the Haskell definitions

```
data Expr = Lit Float | Add Expr Expr | Mul Expr Expr
eval :: Expr -> Float
eval (Lit v) = v
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

define a similar class `Expr` in C++. To mimic the Haskell code's three kinds of expression, use class derivation.

```
class Expr
{
public:
virtual float eval() = 0;
};

class Lit : public Expr
{
public:
Lit( float v ) :val(v) {}
float eval() { return val; }

float val;
};

class Add : public Expr
{
public:
Add( Expr* e1, Expr* e2 ) : exp1(e1), exp2(e2) {}
float eval() { return exp1->eval() + exp2->eval(); }

Expr* exp1;
Expr* exp2;
};

class Mul : public Expr
{
public:
Mul( Expr* e1, Expr* e2 ) : exp1(e1), exp2(e2) {}
float eval() { return exp1->eval() * exp2->eval(); }

Expr* exp1;
Expr* exp2;
};
```