

1. [15] Given the following grammar:

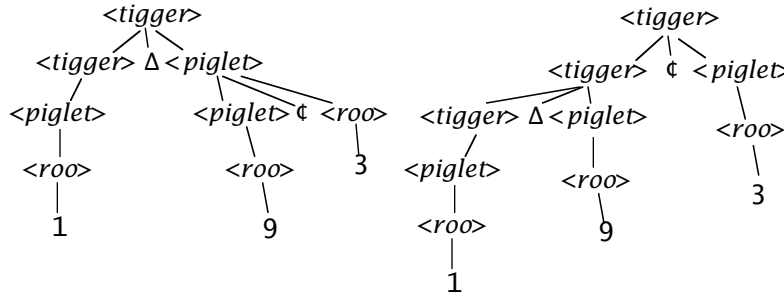
$$\langle tigger \rangle ::= \langle tigger \rangle \Delta \langle piglet \rangle \mid \langle tigger \rangle \text{ † } \langle piglet \rangle \mid \langle piglet \rangle$$

$$\langle piglet \rangle ::= \langle piglet \rangle \Delta \langle roo \rangle \mid \langle piglet \rangle \text{ † } \langle roo \rangle \mid \langle roo \rangle$$

$$\langle roo \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

- a. Demonstrate the grammar's ambiguity.

For the expression $1 \Delta 9 \text{ † } 3$, the grammar allows two different parses:



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

$$\langle tigger \rangle ::= \langle tigger \rangle \Delta \langle piglet \rangle \mid \langle piglet \rangle$$

$$\langle piglet \rangle ::= \langle piglet \rangle \text{ † } \langle roo \rangle \mid \langle roo \rangle$$

$$\langle roo \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

An alternative:

$$\langle tigger \rangle ::= \langle tigger \rangle \text{ † } \langle piglet \rangle \mid \langle piglet \rangle$$

$$\langle piglet \rangle ::= \langle piglet \rangle \Delta \langle roo \rangle \mid \langle roo \rangle$$

$$\langle roo \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

2. [15] Define `isSorted` in Haskell so that `isSorted xs` is `True` if `xs` is strictly increasing, and `False` otherwise. Include `isSorted`'s most general type, and give two definitions— one using explicit recursion and one without.

```
> isSorted :: Ord a => [a] -> Bool
> isSorted [] = True
> isSorted [x] = True
> isSorted (x:y:ys) = x < y && isSorted (y:ys)
```

two nonrecursive versions:

```
> isSorted1 xs = and (zipWith (<) xs (tail xs))
> isSorted xs = and [ x<y | (x,y) <- zip xs (tail xs) ]
```

Note: `and = foldr (&&) True`

3. [15] The Taylor series for $\sin x$ is given by

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+1}}{(2n+1)!}$$

Define a function `sinTerms` so that `sinTerms x` is an infinite list of the terms of this series for `x`. In defining the function, minimize its cost by avoiding recomputation wherever possible.

<pre>> sinTerms :: Float -> [Float] > sinTerms x = zipWith (/) nums denoms > where > xsq = x*x > nums = x : [-xsq * num num <- nums] > > denoms :: [Float] > denoms = map fromInt (alts facts) > facts :: [Int] > facts = 1 : zipWith (*) facts [2..] > alts (x:_:xs) = x : alts xs</pre>	<pre>> sinTerms :: Float -> [Float] > sinTerms x = terms > where > terms = x : [-xsq * t / (n*(n-1)) > (t,n) <- zip terms [3,5..] >] > > xsq = x*x</pre>
--	---

4. [15] Given the following Unix shell program

A | B

where A and B are defined by the C++ programs:

```
// file A.cpp
int main()
{ string sa;
  for(;;)
  {
    ... produce( sa ) ...
    cout << sa << endl;
  }
}
```

```
// file B.cpp
int main()
{ string sb;
  for(;;)
  {
    cin.getLine( sb );
    ... consume( sb ); ...
  }
}
```

define a pair of Ada tasks A and B which are related to each other in the same way as in the Unix shell program.

Assume that C++ “code” like

... produce(sa); ...

translates to Ada “code” that looks identical:

... produce(sa); ...

The Ada tasks:

```
task body A is
  sa : string;
begin
  loop
    ... produce( sa ); ...
    B.input( sa ); ...
  end loop
end A;
```

```
task body B is
  sb : string;
begin
  loop
    accept input(s : in String) do
      sb := s;
    end input;
    ... consume( sb ); ...
  end loop
end B;
```

An alternative solution:

```
task body A is
  sa : string;
begin
  loop
    ... produce( sa ); ...
    accept output( s : out String ) do
      s := sa;
    end output;
  end loop
end A;
```

```
task body B is
  sb : string;
begin
  loop
    A.output( sb );
    ... consume( sb ); ...
  end loop
end B;
```

5. [15] Predict the output of the following program:

```
#include <iostream.h>

class X
{ public:
  X(){ cout << 1 << ' '; }
  X( const X& ){ cout << 2 << ' '; }
  ~X(){ cout << 3 << ' '; }
  X& operator=( const X& ){ cout << 4 << ' '; }
};

X f( X x ){ return x; }

X& g( X& x ){ return x; }

int main()
{
  X a;
  X b = a;
  cout << endl;
  a = b;
  cout << endl;
  a = f( b );
  cout << endl;
  b = g( a );
  cout << endl;
  return 0;
}
```

The output:

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

6. [10] The following function searches an array for a given item.

```
int* find( int array[], int low, int high, int item )
{
  for( int k = low; k <= high; k++ )
    if( array[k] == item ) return _____;
  return 0;
}
```

- a. Fill in the blank two different ways.

The missing expression is
array + k or &(array[k])

- b. Rewrite the function so that it returns a reference, *or* explain why that's not possible.

It's not possible. There is evidently no guarantee that the array elements specified contain the given item. The return of a null pointer indicates that the item was not found, but there is no way to return a null reference.

7. [15] Given these three type definitions:

```
class creature
{ public:
  string cry( ) = 0;
};

class mouse : public creature
{ public:
  string cry( ){ return "squeak"; }
};

class lion : public creature
{ public:
  string cry(){ return "ROAR"; }
};
```

write a Haskell type definition in which `Creature`, `Lion`, and `Mouse` are related in the same way, and include a definition of `cry` (don't forget its type).

Two solutions:

<pre>> data Creature = Mouse Lion > cry :: Creature -> String > cry Mouse = "squeak" > cry Lion = "ROAR"</pre>	<pre>> class Creature a where > cry :: a -> String > data Mouse = Mouse > data Lion = Lion > instance Creature Mouse where > cry _ = "squeak" > instance Creature Lion where > cry _ = "ROAR"</pre>
---	---

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

```
(f1() && f2()) || f3()
```

into an equivalent nested conditional expression containing no `&&` or `||` operators. Note that the functions may have side-effects, so your solution must call exactly the same functions, in exactly the same order, as the original expression.

```
(f1() ? f2() : 0) ? 1 : (f3() ? 1 : 0)
```

9. [10] Give the type (if any) of the following expression, assuming that **integer** and **bool** are distinct types. Explain your answers.

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

- a. assuming static type checking
type error: the true branch has type **integer**, and the false branch has type **bool**.
- b. assuming dynamic type checking
the type is **bool** because the guard is false

10. [15] The following fragmentary C program text contains a type definition, declarations of several variables, and some references to the variables. The variable references are tabulated in a 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:

char 2 **int** 4 **float** 8

The program:

```

int a[10];
float b;
typedef struct{ int a, int b[3]; } S;
char c;

void f()
{
    char x[5]; S r[3]; int t;
    static int b;
    ... t ... c ... b ...
}

int main()
{
    S p[2];
    ... p[1].b[2] ... b ...
    return 0;
}

```

Answer by filling in the table. Indicate the distinction between local and global variables by appending 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 *clearly*.

First we calculate the space requirements for values of type S:

```

a      4  0-3
b     3*4 4-15

```

Then we build a table for each block showing each variable's size and its address(es) in its block's activation record.

```

global  a   10*4  0-39
        b    8   40-47
        c    2   48-49
        f::b  4   50-53

f       x    5*2   0-9
        r   3*16  10-57
        t    4   58-61

main   p   2*16  0-31

```

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

t	58
c	48 G
b (in f)	50 G
p[1].b[2]	1*16 + (4 + 2*4) = 28
b (in main)	40 G

- 11.** [20] Design an Ada task `RunningAvg` which accepts calls to `Reset`, `Put`, and `GetAvg`. Each `Put` call transmits a floating-point number to `RunningAvg`. Each `GetAvg` call transmits the average of all the numbers received by `RunningAvg` since the last call to `Reset` (or the task's creation, if `Reset` has not been called); until at least one number has been received, `GetAvg` calls should not be accepted.

```
task RunningAvg is
  entry Reset;
  entry Put( x : in real );
  entry GetAvg( avg : out real );
  sum : real; count : integer;
begin
  sum := 0.0; count := 0;
  loop
    select
      accept Put( x : in real ) do
        sum := sum + x;
        count := count + 1;
      end Put;
    or
      when count > 0 =>
        accept GetAvg( avg : out real ) do
          avg := sum / count;
        end GetAvg;
    or
      accept Reset do
        sum := 0.0; count := 0;
      end Reset;
    end select;
  end loop
end RunningAvg;
```

12. [10] If the following program compiles, what is its output? If it won't compile, what would the diagnostic say?

```
#include <iostream.h>

class Simon
{
public:
    void s1O{ cout << "1 "; }
    virtual void s2O{ cout << "2 "; }
    void fO{ s1O; s2O; }
};

void paul( Simon& r, Simon v )
{
    r.fO; v.fO;
}

class Garfunke1 : public Simon
{
public:
    virtual void s1O{ cout << "3 "; }
    void s2O{ cout << "4 "; }
};

int mainO
{
    Simon s; Garfunke1 g;

    s.fO; s.s1O; s.s2O; cout << endl;
    g.fO; g.s1O; g.s2O; cout << endl;

    paul( g, g ); cout << endl;

    return 0;
}
```

It compiles OK. The output:

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