

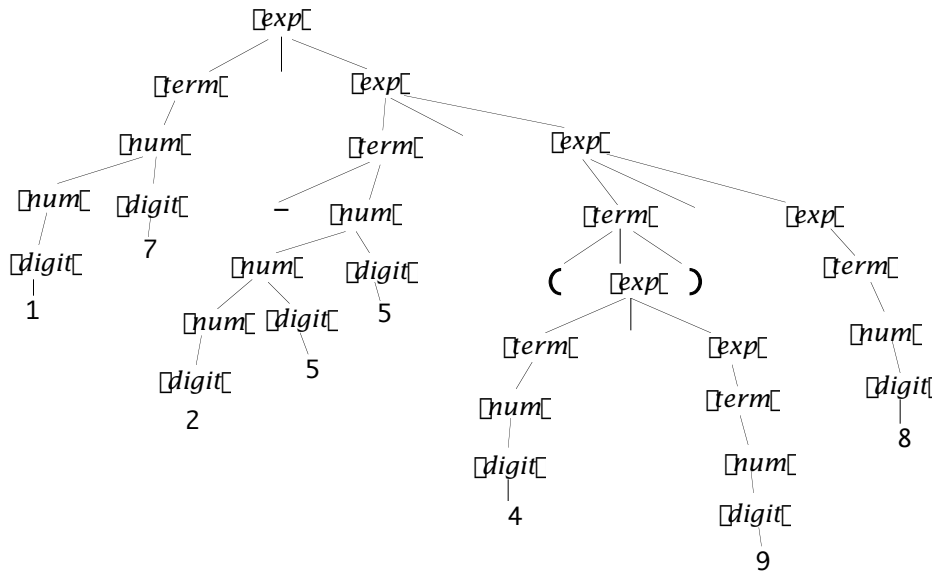
1. [15] Given the following syntax:

```

<exp> ::= <term> <exp> | <term>
<term> ::= <num> | - <num> | '(' <exp> ')'
<num> ::= <digit> | <num> <digit>
<digit> ::= 0 | 1 | ... | 9
    
```

draw a parse tree for the following expression:

17 - 255 (4 9) 8



2. [10] Translate the following function into an equivalent tail-recursive function whose body contains no assignment operations.

```

int f( int x, int k, int n )
{
    while( n > 0 )
    {
        x = k*x; n = n-1;
    }
    return x;
}
    
```

The effect of the loop is to rebind the parameters x and n to values $k*x$ and $n-1$, respectively. A straightforward translation obtains the same rebinding by means of a recursive function call:

```

int f(int x, int k, int n )
{
    if( n > 0 )
        return f( k*x, k, n-1 );
    else
        return x;
}
    
```

3. [10] Translate the following guarded-command program into C. Assume that the type of all variables is *integer*.

```

if  $x < y$  □  $x, y := x+y, x-y$  □  $x > y$  □  $x := z-y; y := x-z$  fi
    
```

A solution:

```

if ( x < y ) { int t = x+y; y = x-y; x = t; }
else if ( x > y ) { x = z-y; y = x-z; }
else exit(1);
    
```

4. [10] Give the most general type of the following expression:

$$\backslash x y z \rightarrow \text{if } y < z \text{ then } x^2 \text{ else } 0$$

Answer:

$$(\text{Num } a, \text{Ord } b) \Rightarrow a \rightarrow b \rightarrow b \rightarrow a$$

5. [15] Determine formally the truth (or otherwise) of the following proposition:

$$\{ x = 12 \wedge y = 4 \} x, y := x+y, x-6; x := y+8 \{ x+y > 4 \}$$

$$\begin{aligned} & \text{wp}("x, y := x+y, x-6; x := y+8", x + y > 4) \\ = & \quad \{ \text{def. " ;" } \} \\ & \text{wp}("x, y := x+y, x-6", \text{wp}("x := y+8", x + y > 4)) \\ = & \quad \{ \text{def. " :=" } \} \\ & \text{wp}("x, y := x+y, x-6", (x + y > 4)_{y+8}^x) \\ = & \quad \{ \text{substitution} \} \\ & \text{wp}("x, y := x+y, x-6", y + 8 + y > 4) \\ = & \quad \{ \text{simplification} \} \\ & \text{wp}("x, y := x+y, x-6", y > -2) \\ = & \quad \{ \text{def. " :=" } \} \\ & (y > -2)_{x+y, x-6}^{x, y} \\ = & \quad \{ \text{substitution, simplification} \} \\ & x - 6 > -2 \\ = & \quad \{ \text{algebra} \} \\ & x > 4 \\ \square & \quad \{ \text{algebra} \} \\ & x = 12 \wedge y = 4 \end{aligned}$$

So the proposition is true.

6. [10] Rewrite this C function

```
int pos( double a[] )
{
    int r = 0;
    for( int k = 0; k < 20; k++ )
        r += a[k] > 0;
    return r;
}
```

using pointers instead of array references. Assume that `sizeof(double) = 10`, and `sizeof(int) = 4`. For full credit, don't use an integer counter in the loop.

```
int pos( double* a )
{
    int r = 0;
    for( double* p = a; p < a+20; p++ )
        r += *p > 0;
    return r;
}
```

7. [10] Write a function `max` which can be used to find the largest of the values `arb(j)`, `arb(j+1)`, ..., `arb(k)`.

for an arbitrary function `arb :: int -> float` and arbitrary integers `j` and `k`. Your function must use *Jensen's device*.

What mechanism for binding arguments to parameters does your solution require?

call by name

Given integers 0 and 10, and function `fun :: int -> float`, how would `max` be called?

`max(i, 0, 10, fun(i))` // where `i` is a variable in the environment of the call

The definition of `max`:

```
float max( int s, int r, int t, float f )
{
  s := r;
  float m = f;
  s := r+1;
  while( s <= t )
  {
    float n = f;
    if( m < n ) m := n;
    s := s+1;
  }
  return m;
}
```

8. (deleted)

9. [10] Suppose variables A and B both have the following type:

record a, b, c, d : **integer** **end**;

and the following code is executed:

A.a := 0; A.b := 0; A.c := 0; A.d := 0;

A.b := 2;

B := A;

A.c := 3;

B.d := 4

Give the post-execution values of A and B by filling in the following tables, assuming the variables have

- a. value semantics.

	a	b	c	d
A	0	2	3	0
B	0	2	0	4

- b. reference semantics.

	a	b	c	d
A	0	2	3	4
B	0	2	3	4

10. [10] Suppose global variables `grade`, `count`, and `interval` are declared as follows:

```
double grade [M];
int count [N];
double interval [N];
```

and our C++ program is to establish the postconditions

```
count[0] = (∑j | 0 ≤ j < M ∧ grade[j] ≤ interval[0] : 1)
(∑i | 0 < i < N : count[i] = (∑j | 0 ≤ j < M ∧ interval[i-1] < grade[j] ≤ interval[i] : 1))
```

That is, `count[0]` is the number of grades less than or equal to `interval[0]`, and for $0 < i < N$, `count[i]` is the number of grades between `interval[i-1]` and `interval[i]`.

At the core of the program is this loop:

```
for( int k = 0; k < M; k++ )
    bucket( grade[k] )++; // increments the count corresponding to grade[k]
```

The question is how to define `bucket()`.

A solution:

```
int& bucket( double value )
{
    int i = 0;
    while( i < N && interval[i] < value ) i++;
    return count[i];
}
```

An alternative solution:

```
class bucket
{
public:
    bucket( double g ): value(g) {}
    double value;
    void operator++( int )
    {
        int i = 0;
        while( i < N && interval[i] < value ) i++;
        count[i]++;
    }
};
```

11. [10] Define in Haskell a parser which parses strings consisting of one or more matching pairs of parentheses. For each '(', there should be a corresponding ')' somewhere to its right. Examples:

```
((()()()))
(())()((()()))
```

The parser's purpose is simply to recognize strings that satisfy the specification, so it should deliver just `()`.

The first step is to define the grammar for these strings:

$$\textit{Parens} ::= '(\{ \textit{Parens} \})' \{ \textit{Parens} \}$$

Then we define a parser corresponding to the grammar:

```
> parens :: Parser ()
> parens = do char '('
>           many parens
>           char ')'
>           many parens
>           return ()
```

12. [10] Predict what is output when this C++ program runs:

```
#include <iostream>

class A
{ public:
    virtual void fO{ cout << "Af "; }
    void gO{ cout << "Ag "; }
};

class B : public A
{ public:
    void fO{ cout << "Bf "; }
    virtual void gO{ cout << "Bg "; }
};

void test( A x, A& y, A* z )
{ x.fO; y.fO; z->fO;
  cout << endl;
  x.gO; y.gO; z->gO;
}

int mainO
{
    B b;

    test( b, b, &b );

    return 0;
}
```

The output:

```
Af Bf Bf
Ag Ag Ag
```

13. [15] The Ackermann function

```
> ack :: Int -> Int -> Int
> ack 0 n          = n+1
> ack m 0 | m>0   = ack (m-1) 1
> ack m n | m>0 && n>0 = ack (m-1) (ack m (n-1))
```

is known for its extremely high computational complexity. Write a memoized version of it.

```
> fack :: Int -> Int -> Int
> fack 0 n = n+1
> fack m 0 | m>0 = acks!!(m-1)!!1
> fack m n | m>0 && n>0 = acks!!(m-1)!!(acks!!m!!(n-1))

> acks :: [[Int]]
> acks = [ [ fack m n | n <- [0..] ] | m <- [0..] ]
```

14. [10] The function `pSums` is defined so that `pSums xs` computes the *partial sums* of `xs`. That is, the n th element of the result is the sum of the first n elements of `xs`. For example,

```
pSums [3, 2, 4, 1] = [3, 5, 9, 10]
pSums [2.5, 1.2, 3.3] = [2.5, 3.7, 7.0]
```

Complete the definition of `pSums` by filling in the blanks in the following code:

```
> pSums :: _____
> pSums [] = []
> pSums (x:xs) = ps
>   where
>     ps = x : [ _____ | _____ <- _____ ]
```

The completed definition of `pSums`:

```
> pSums :: Num a => [a] -> [a]
> pSums [] = []
> pSums (x:xs) = ps
>   where
>     ps = x : [ n + p | (n,p) <- zip xs ps ]
```

15. [10] In the following code, redefine class `Delta` so that it is no longer 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 mystery(); // defined elsewhere
private:
    int bn;
};

class Delta : Beta {
public:
    Delta( int n ): Beta(n) {}
    Beta::mystery;
};
```

A solution:

```
class Delta {
public:
    Delta( int n ): b(n) {}
    int mystery(){ return b.mystery(); }
private:
    Beta b;
};
```