What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Programming (Partner Optional):

Problems: Turn in handwritten or typed answers by the due date. Be sure your work is stapled and that your answers are clearly marked and in the correct order.

Programming: Turn in a printout of your code separately from your problem answers. If you worked with a partner, only one person needs to hand in a printout, but be sure both IDs are at the top of the file. Turn in an electronic copy as well using the instructions at the end of the programming questions.

Reading

1. (Required) Read Mitchell, Chapter 8.1—8.2.
2. (Required) Mitchell, Chapter 10.

Problems

Q1. (10 points) Exceptions Consider the following functions, written in ML:

```ml
exception Excpt of int;
fun twice(f,x) = f(f(x)) handle Excpt(x) => x;
fun pred(x) = if x = 0 then raise Excpt(x) else x-1;
fun dumb(x) = raise Excpt(x);
fun smart(x) = (1 + pred(x)) handle Excpt(x) => 1;
```

What is the result of evaluating each of the following expressions?

(a) twice(pred,1);
(b) twice(dumb,1);
(c) twice(smart,0);

In each case, be sure to describe which exception gets raised and where.

Q2. (15 points) Equivalence of Abstract Data Types

Mitchell, Problem 9.2

Q3. (15 points) Expression Objects

We now look at an object-oriented way of representing arithmetic expressions given by the grammar

```
e ::= num | e + e
```

We begin with an “abstract class” called SimpleExpr. While this class has no instances, it lists the operations common to all instances of this class or subclasses. In this case, it is just a single method to return the value of the expression.
abstract class SimpleExpr {
    abstract int eval();
}

Since the grammar gives two cases, we have two subclasses of SimpleExpr, one for numbers and one for sums.

class Number extends SimpleExpr {
    int n;
    public Number(int n) { this.n = n; }
    int eval() { return n; }
}

class Sum extends SimpleExpr {
    SimpleExpr left, right;
    public Sum(SimpleExpr left, SimpleExpr right) {
        this.left = left;
        this.right = right;
    }
    int eval() { return left.eval() + right.eval(); }
}

(a) **Product Class**

Extend this class hierarchy by writing a Times class to represent product expressions of the form

\[ e ::= \ldots | e \ast e \]

(b) **Method Calls**

Suppose we construct a compound expression by

```java
SimpleExpr a = new Number(3);
SimpleExpr b = new Number(5);
SimpleExpr c = new Number(7);
SimpleExpr d = new Sum(a,b);
SimpleExpr e = new Times(d,c);
```

and send the message `eval` to `e`. Explain the sequence of calls that are used to compute the value of this expression: `e.eval()`. What value is returned?

(c) **Comparison to “Type Case” constructs**

Let's compare this programming technique to the expression representation used in ML, in which we declared a datatype and defined functions on that datatype by pattern matching. The following `eval` function is one form of a “type case” operation, in which the program inspects the actual tag (or type) of a value being manipulated and executes different code for the different cases:

```java
datatype MLExpr =
    Number of int
    | Sum of MLExpr * MLExpr;

fun eval (Number(x)) = x
    | eval (Sum(e1,e2)) = eval(e1) + eval(e2);
```

This idiom also comes up in class hierarchies or collections of structures where the programmer has included a **Tag** field in each definition that encodes the actual type of an object.
i. Discuss, from the point of view of someone maintaining and modifying code, the differences between adding the *Times* class to the object-oriented version and adding a *Times* constructor to the *MLExpr* datatype. In particular, what do you need to add/change in each of the programs. Generalize your observation to programs containing several operations over the arithmetic expressions, and not just `eval`.

ii. Discuss the differences between adding a new operation, such as `toString`, to each way of representing expressions. Assume you have already added the product representation so that there is more than one class with a nontrivial `eval` method.

---

**Programming (Partner Optional)**

**P1. (10 points) ............................................... Exceptions in ML**

Your GitLab account will have a project for your to use for this and the following programming questions. You can follow the same instructions as on HW 1 for cloning it and adding a partner.

The function `stringToNum` defined below uses two auxiliary functions to convert a string of digits into a non-negative integer.

(* Convert one character to a numeric digit. *)

```ml
fun charToNum c = ord c - ord #"0";
```

```ml
fun calcList (nil,n) = n
| calcList (fst::rest,n) =
    calcList(rest,10 * n + charToNum fst);
```

(* Convert a string of digits to a number. The `explode` function converts a string to a list of characters. *)

```ml
fun stringToNum s = calcList(explode s, 0);
```

For instance, `stringToNum "3405"` returns the integer 3405. (The function `explode` converts a string into a list of characters, and `ord` returns the ASCII integer value for a character.)

Unfortunately, `calcList` returns a spurious result if the string contains any non-digits. For instance, `stringToNum "3a05"` returns 7905, while `stringToNum "405"` returns 15595. This occurs because `charToNum` will convert any character, not just digits. We can attempt to fix this by having `charToNum` raise an exception if it is applied to a non-digit.

(a) Revise the definition of `charToNum` to raise an exception, and then modify the function `stringToNum` so that it handles the exception, returning 1 if there is a non-digit in the string. You should make no changes to `calcList`.

(b) Implement ML functions to provide the same behavior (including returning 1 if the string includes a non-digit) as in the first part, but without using exceptions. While you may change any function, try to preserve as much of the structure of the original program as possible.

(c) Which implementation do you prefer? Why? Answer this as a comment in the code.

---

**P2. (15 points) .......................................... Tail Recursion**

(a) The dot product of two vectors \([a_1, \ldots, a_n]\) and \([b_1, \ldots, b_n]\) is the sum \(a_1 b_1 + a_2 b_2 + \cdots + a_n b_n\). For example,

\[
[1, 2, 3] \cdot [-1, 5, 3] = 1 \cdot -1 + 2 \cdot 5 + 3 \cdot 3 = 18
\]

Implement the function

```ml
dotprod: int list -> int list -> int
```
to compute the dot product of two vectors represented as lists. You should write this using tail-recursion, so your dotprod function will probably just be a wrapper function that calls a second function that does all the work. If passed lists of different length, your function should raise a DotProd exception. You will need to declare this type of exception, but you need not catch it.

- dotprod [1,2,3] [~1,5,3];
  val it = 18 : int
- dotprod [~1,3,9] [0,0,11];
  val it = 99 : int
- dotprod [] [];
  val it = 0 : int
- dotprod [1,2,3] [4,5];
  uncaught exception DotProd

(b) The numbers in the Fibonacci sequence are defined as:

\[
\begin{align*}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2)
\end{align*}
\]

Thus, the sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc.
The following defines a function that returns the n-th Fibonacci number.

fun slow_fib(0) = 0 
  | slow_fib(1) = 1 
  | slow_fib(n) = slow_fib(n-1) + slow_fib(n-2);

Unfortunately, computing slow_fib(n) requires \(O(2^n)\) time.
Define a tail recursive function fast_fib that can compute \(F(n)\) in \(O(n)\) time by using tail recursion. (As above, fast_fib will most likely be a wrapper that calls a tail-recursive function.) The tail-recursive function should have only one recursive call in its definition.

- fast_fib 0
  val it = 0 : int
- fast_fib 1
  val it = 1 : int
- fast_fib 5
  val it = 1 : int
- fast_fib 10
  val it = 1 : int

(Hint: When converting sumSquares to tail-recursive form, we introduced one auxiliary parameter to accumulate the result of the single recursive call in that function. How many auxiliary parameters do you think we will need for fibtail?)

P3. (20 points) .......................................................... Visitor Design Pattern

The code for this question is located in the ExprVisitor.java file. Include answers to the questions below as comments at the top of that file.

The extension and maintenance of an object hierarchy can be greatly simplified (or greatly complicated) by design decisions made early in the life of the hierarchy. This question explores various design possibilities for an object hierarchy (like the one above) that represents arithmetic expressions.

The designers of the hierarchy have already decided to structure it as shown below, with a base class Expr and derived classes Number, Sum, Times, and so on. They are now contemplating how
to implement various operations on Expressions, such as printing the expression in parenthesized
form or evaluating the expression. They are asking you, a freshly-minted language expert, to
help.

The obvious way of implementing such operations is by adding a method to each class for each
operation. This version is not in the starter code, but the expression hierarchy would look like
the following in this scenario:

```java
abstract class Expr {
    public abstract String toString();
    public abstract int eval();
}

class Number extends Expr {
    int n;

    public Number(int n) { this.n = n; }
    public String toString() { ... }
    public int eval() { ... }
}

class Sum extends Expr {
    Expr left, right;

    public Sum(Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }
    public String toString() { ... }
    public int eval() { ... }
}
```

Suppose there are $n$ subclasses of `Expr` altogether, each similar to `Number` and `Sum` shown here.
How many classes would have to be added or changed to add each of the following things?

(a) A new class to represent division expressions.
(b) A new operation to graphically draw the expression parse tree.

Another way of implementing expression classes and operations uses a pattern called the Visitor
Design Pattern. In this pattern, each operation is represented by a `Visitor` class. Each Visitor
class has a `visit` method for each expression class `Class` in the hierarchy. Each expression
class `Class` is set up to call the `visitClass` method to perform the operation for that particular
class. In particular, each class in the expression hierarchy has an `accept` method which accepts a
Visitor as an argument and “allows the Visitor to visit the class and perform its operation.” The
expression class does not need to know what operation the visitor is performing.

If you write a Visitor class `ToString` to construct a string representation of an expression tree, it
would be used as follows:

```java
Expr expTree = ...some code that builds the expression tree...;
ToString printer = new ToString();
String stringRep = expTree.accept(printer);
System.out.println(stringRep);
```

The first line defines an expression, the second defines an instance of your `ToString` class, and
the third passes your visitor object to the `accept` method of the expression object.

The expression class hierarchy using the Visitor Design Pattern has the following form, with
an `accept` method in each class and possibly other methods. Since different kinds of visitors
return different types of values, the `accept` method is parameterized by the type that the visitor
computes for each expression tree:
abstract class Expr {
    abstract <T> T accept(Visitor<T> v);
}

class Number extends Expr {
    int n;

    public Number(int n) { this.n = n; }

    public <T> T accept(Visitor<T> v) {
        return v.visitNumber(this.n);
    }
}

class Sum extends Expr {
    Expr left, right;

    public Sum(Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }

    public <T> T accept(Visitor<T> v) {
        T leftVal = left.accept(v);
        T rightVal = right.accept(v);
        return v.visitSum(leftVal, rightVal);
    }
}

The associated Visitor abstract class, naming the methods that must be included in each visitor, and the ToString visitor, have this form:

abstract class Visitor<T> {
    abstract T visitNumber(int n);
    abstract T visitSum(T left, T right);
}

class ToString extends Visitor<String> {
    public String visitNumber(int n) {
        return "" + n;
    }

    public String visitSum(String left, String right) {
        return "(" + left + " + " + right + ");"
    }
}

Here is an example of using the visitor to evaluate and print an expression.

class ExprVisitor {
    public static void main(String s[]) {
        Expr e = new Sum(new Number(3), new Number(2));
        ToString printer = new ToString();
        String stringRep = e.accept(printer);
        System.out.print(stringRep);
    }
}

(c) Starting with the call to e.accept(printer), what is the sequence of method calls that will occur while building the string for the expression tree e?

(d) Add the following classes to the source file. You will need to change some of the existing classes to accommodate them.
i. An **Eval** visitor class that computes the value of an expression tree. The visit methods should return an **Integer**. Recall that Java 1.5 has auto-boxing, so it can convert int values to Integer objects and vice-versa, as needed.

ii. **Subtract** and **Times** classes to represent subtraction and product expressions.

iii. A **Compile** visitor class that returns a sequence of stack-based instructions to evaluate the expression. You may use the following stack instructions (Refer back to HW 3 if you need a refresher on how these instructions operate):

   - **PUSH**(n)
   - **ADD**
   - **MULT**
   - **SUB**
   - **DIV**
   - **SWAP**

   The visit methods can simply return a **String** containing the sequence of instructions. For example, compiling \(3 \times (1 - 2)\) should return the string

   ```
   PUSH(3) PUSH(1) PUSH(2) SUB MULT
   ```

   The instruction sequence should just leave the result of computing the expression on the top of the stack. Hint: the order of instructions you need to generate is exactly a post-order traversal of the expression tree.

   Aside: Most modern compilers (including the Sun Java compiler) are implemented using the **Visitor Pattern**. The compilation process is really just a sequence of visit operations over the abstract syntax tree. Common steps include visitors 1) to resolve the declaration to which each variable access refers; 2) to perform type checking; 3) to optimize the program; 4) to generate code as above; and so on.

Suppose there are \(n\) subclasses of **Expr**, and \(m\) subclasses of **Visitor**. How many classes would have to be added or changed to add each of the following things using the Visitor Design Pattern?

(e) A new class to represent division expressions.

(f) A new operation to graphically draw the expression parse tree.

The designers want your advice.

(g) Under what circumstances would you recommend using the standard design?

(h) Under what circumstances would you recommend using the Visitor Design Pattern?

**What To Turn In.**

- Your code for these questions should be documented, and include your ID (and that of your partner) at the top of each file.
- Turn in a printout of your code file separate from the answers to the written problems. If you worked with a partner, please turn in only one copy.
- Also, commit and push all of your code to our Gitlab server.

**P4.** (**15 points**) .......................................................... Five 5’s (Bonus Question)

Consider the well-formed arithmetic expressions using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators “+”, “-”, “*”, and “/”, and properly placed parentheses, such as “5”, “5+5”, “(5 + 5) \times (5 - 5 / 5)”, and so on. (Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5.) Write a ML program to answer one or more of the following questions:

(a) What is the smallest positive integer than cannot be computed by an expression involving exactly five 5’s?
(b) What is the largest prime number that can computed by an expression involving exactly five 5's?

(c) Exhibit an expression that evaluates to that prime number.

You should start by defining a datatype to represent arithmetic expressions involving 5 and the arithmetic operations, as well as an eval function to evaluate them. This question involves only integer arithmetic, so be sure to use div for division. You should then write code to generate all expressions containing a fixed number of 5's.

Answering the questions will then involve an exhaustive search (for all numbers that can be computed with a fixed number of 5's), so good programming techniques are important. Avoid any unnecessarily time-consuming or memory-consuming operations. You may also have to do something special to avoid division by zero inside eval, perhaps with exception handlers.

(For those who have taken or are taking 256: while there are many possible ways to solve this problem, it is one very well-suited for dynamic programming...)

Write your solution in a file name "fives.sml" and submit it along with the rest of your programming questions.