Problem Set 4
Due: 6pm Friday, March 4 (Independent)
11:59pm Wednesday, March 9 (Group)

Revisions: Mar 3: Added Group Problem 2. The assignment is now complete. Mar 2: (1) Added notes about testing toList, fromList, toSexp, and fromSexp to Group Problem 1. Mar 1: Added Group Problem 1.

Overview:
The individual problem on this assignment tests your understanding of first-class functions, higher-order list operations, and modules. The group problems on this assignment will give you practice with sum-of-product datatypes, s-expressions, and interpreters.

Reading:
• Handout #8 (Jason Hickey’s OCAML tutorial): Chapters 6, 7, 10 (ignore 10.4), and 11.
• Handout #21: Modules and Data Abstraction in OCAML.
• Handout #22: Sum-of-Product Data Types
• Handout #26: INTEX: An Introduction to Program Manipulation
• Handout #27: POSTFIX: A Stack-Based Language

Individual Problem Submission:
Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn’s office door by 6pm Fri. March. 4. The packet should include:
1. an individual problem header sheet;
2. your final version of ListMatrix.ml from Problem 1a;
3. a transcript showing the result of running testListMatrix();
4. your final version of FunMatrix.ml from Problem 1b;
5. a transcript showing the result of running testFunMatrix();

Each student should also submit a softcopy (consisting of your final ps4-individual directory) to the drop directory `~cs251/drop/ps4/username.

Working Together:
If you want to work with a partner on this assignment, you should try to find a different partner than you worked with on a previous assignment. If this proves difficult, please email Lyn describing your situation.

Group Problem Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn’s office door by 11:59pm on Wed. Mar. 9. The packet should include:
1. a team header sheet indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your final version of OperationTreeSet.ml for Group Problem 1;
3. your final version of PostFix.ml from Problem 2a.
4. your final version of PostFixInterp.ml from Problem 2b.
5. your final version of your POSTFIX gcd program from from Problem 2c, along with testing transcripts demonstrating that it works as expected.

Each team should also submit a single softcopy (consisting of your final ps4-group directory) to the drop directory `~cs251/drop/p4/username, where username is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet).
Individual Problem [40]: Enter the Matrix!

Matrix Definitions
A matrix is a two dimensional array of elements. A matrix $M$ with $r$ rows and $c$ columns is called an $r \times c$ matrix and is often depicted as follows:

$$
\begin{array}{cccc}
M_{(1,1)} & M_{(1,2)} & \cdots & M_{(1,c)} \\
M_{(2,1)} & M_{(2,2)} & \cdots & M_{(2,c)} \\
\vdots & \vdots & \ddots & \vdots \\
M_{(r,1)} & M_{(r,2)} & \cdots & M_{(r,c)}
\end{array}
$$

The notation $M_{(i,j)}$ denotes the element in row $i$ and column $j$ of the matrix. Row and column indices start at 1. **We will assume throughout this problem that matrices are never empty; i.e., they always have at least one row and one column.**

Here is an example of a $3 \times 4$ matrix $A$ with integer elements:

$$
\begin{array}{cccc}
61 & 64 & 69 & 76 \\
71 & 74 & 79 & 86 \\
81 & 84 & 89 & 96
\end{array}
$$

In this example, $A_{(2,3)}$ is 79 and $A_{(3,2)}$ is 84. If $M$ is an $r \times c$ matrix and $i$ is outside the range $[1 \ldots r]$ or $j$ is outside the range $[1 \ldots c]$, then the notation $M_{(i,j)}$ is undefined.

The transpose of an $r \times c$ matrix $M$ is a $c \times r$ matrix $M'$ such that $M'_{(i,j)} = M_{(j,i)}$. For example, the transpose of $A$ is the following $4 \times 3$ matrix:

$$
\begin{array}{ccc}
61 & 71 & 81 \\
64 & 74 & 84 \\
69 & 79 & 89 \\
76 & 86 & 96
\end{array}
$$

Ocaml MATRIX Signature
Fig. 1 presents an Ocaml signature `MATRIX` for a matrix abstract data type (ADT). The contract for the functions in the `MATRIX` signature is explained in Fig. 2. Examples illustrating the behavior of the matrix functions are shown in Fig. 3.

```
module type MATRIX = sig
  type 'a matrix
  val make : int -> int -> (int -> int -> 'a) -> 'a matrix
  val dimensions : 'a matrix -> int * int
  val get : int -> int -> 'a matrix -> 'a option
  val put : int -> int -> 'a -> 'a matrix -> 'a matrix
  val transpose : 'a matrix -> 'a matrix
  val toLists : 'a matrix -> 'a list list
  val map : ('a -> 'b) -> 'a matrix -> 'b matrix
end
```

Figure 1: The MATRIX signature.
val make: int -> int -> (int -> int -> 'a) -> 'a matrix
   Given positive integers $r$ and $c$ and a binary function $f$, make $r \ c \ f$ returns an $r \times c$ matrix $m$ such that $m(i,j)$ is the result of the function call $f(i \ j)$. If $r$ or $c$ is $\leq 0$, the meaning of this function is undefined.

val dimensions: 'a matrix -> (int * int)
   dimensions $m$ returns a pair $(\text{rows}, \text{cols})$, where $\text{rows}$ is the number of rows in $m$ and $\text{cols}$ is the number of cols in $m$.

val get: int -> int -> 'a matrix -> 'a option
   If $m$ is an $r \times c$ matrix, $i \leq r$, and $j \leq c$ then get $i \ j \ m$ returns Some $v$, where $v$ is the value $m(i,j)$. Otherwise, get $i \ j \ m$ returns None.

val put: int -> int -> 'a -> 'a matrix -> 'a matrix
   If $m$ is an $r \times c$ matrix, $i \leq r$, and $j \leq c$, then put $i \ j \ v \ m$ returns a matrix $m'$ such that $m'(i,j) = v$ and $m'(i',j') = m(i,j)$ if $i' \neq i$ or $j' \neq j$. Otherwise, put $i \ j \ v \ m$ returns $m$ unchanged.

val transpose: 'a matrix -> 'a matrix
   Given an $r \times c$ matrix $m$, transpose $m$ returns a $c \times r$ matrix $m'$ such that $m'(i,j) = m(j,i)$.

val toLists: 'a matrix -> 'a list list
   Given an $r \times c$ matrix $m$, toLists $m$ returns a length-$r$ list of length-$c$ lists. The $k$th element of the returned list is a list of the elements in columns 1 through $c$ of row $k$ of $m$.

val map: ('a -> 'b) -> 'a matrix -> 'b matrix
   Given a function $f$ and a $r \times c$ matrix $m$, map $f \ m$ returns a $r \times c$ matrix $m'$ such that $m'(i,j) = f(m(i,j))$.

Figure 2: The contract for the seven OCAML matrix-manipulation functions.
Figure 3: Examples illustrating the behavior of the matrix functions for a module M implementing the MATRIX signature.
Matrix Implementations

In this problem, your task is to implement the MATRIX signature using two very different concrete representations for matrices.

a. [20]: Matrices as Lists

A straightforward way to implement an \( r \times c \) matrix is as a list of \( r \) elements, the \( i \)th of which is a list of the \( c \) elements in the \( i \)th row of the matrix. In this case, the type ‘a matrix is implemented as ‘a list list. For instance, in this representation, the example matrix \( A \) above would be represented as:

\[
\begin{bmatrix}
[61; 64; 69; 76];
[71; 74; 79; 86];
[81; 84; 89; 96]
\end{bmatrix}
\]

In this part you are to implement the seven functions in the MATRIX signature using the lists-of-lists representation of matrices. But there’s a catch: You are not allowed to use recursion in any of your definitions. Instead, you should define all functions using the higher order list operations in the ListUtils module.

To do this part, go to "~/cs251/ps4-individual/ListMatrix.ml" and flesh out the seven function definitions in the ListMatrix module in this file. You should load this module in OCAML via

```
#use "load-list-matrix.ml";;
```

and can test it by evaluating

```
testListMatrix();
```

Feel free to define any helper functions you find useful. However, you may not use recursion in your helper functions. You may assume that every matrix has at least one row and one column.

b. [20]: Matrices as Functions

An alternative representation of a ‘a matrix is as a function with type \( \text{int} \rightarrow \text{int} \rightarrow \text{‘a option} \) that takes two arguments, the coordinates \( i \) and \( j \), and returns an option (Some or None) of the element at these coordinates.

In this part you are to implement the seven matrix functions using this functional representation of matrices. You are not allowed to use any list operations in this part except in the implementation of toLists.

To do this part, go to "~/cs251/ps4-individual/FunMatrix.ml" and flesh out the seven function definitions in the FunMatrix module in this file. You should load this module in OCAML via

```
#use "load-fun-matrix.ml";;
```

and can test it by evaluating

```
testFunMatrix();
```

Notes:

- Feel free to define any helper functions you find useful.
- You may assume that every matrix has at least one row and one column.
- Think carefully about where the bounds checking of matrix indices should occur.
- A challenging aspect of this problem is determining the number of rows and columns in a matrix represented by a function. \textit{Hint:} Use the fact that the get function returns \texttt{None} for out-of-bounds indices.
Group Problems

Group Problem 1 [25]: OperationTreeSet

Background
In this problem, you will flesh out an implementation of the SET signature in Fig. 4. This is the same as the SET signature presented in Handout #21 except that it includes a toSexp that translates a set into an s-expression representation and a fromSexp function that translates an s-expression representation into a set. Different set implementations may have different s-expression representations. However, it should be the case that for all sets s, fromSexp (toSexp s) yields a set with exactly the same elements as s.

```ocaml
module type SET = sig

  type 'a set
  val empty : 'a set (* the empty set *)
  val singleton : 'a -> 'a set (* a set with one element *)
  val insert : 'a -> 'a set -> 'a set (* insert elt into given set *)
  val delete : 'a -> 'a set -> 'a set (* delete elt from given set *)
  val member : 'a -> 'a set -> bool (* is elt a member of given set? *)
  val union: 'a set -> 'a set -> 'a set (* union of two sets *)
  val intersection: 'a set -> 'a set -> 'a set (* intersection of two sets *)
  val difference: 'a set -> 'a set -> 'a set (* difference of two sets *)
  val fromList : 'a list -> 'a set (* create a set from a list *)
  val toList : 'a set -> 'a list (* list all set elts, sorted low to high *)
  val toSexp : ('a -> Sexp.sexp) -> 'a set -> Sexp.sexp (* translates set into s-expression rep. *)
  val fromSexp : (Sexp.sexp -> 'a) -> Sexp.sexp -> 'a set (* translates s-expression rep. into set *)
  val toString : ('a -> string) -> 'a set -> string (* returns string representation of set *)

end
```

Figure 4: The SET signature.

Before beginning this problem, you should study the sorted set implementation of sets and the BST implementation of sets:

- The sorted set implementation of sets is described in Section 4.2 of the Modules handout (#21). The code for this implementation is in `/cs251/sets/SortedListSet.ml`. You can test it via the following OCAML commands:

  ```ocaml
  #cd "/students/username/cs251/sets";;
  #use "load-sorted-list-set.ml";;
  testZZZ();;
  ```

  where ZZZ is one of Tiny, Small, Medium, or Large. Each testZZZ function performs tests on word files of different sizes: the tiny file has 16 words, the small file has 476 words, the medium file has 5525 words, and the large file has 45425 words.

- The BST implementation of sets is described in Section 2.3 of the Sum of Products handout (#22). The code for this implementation is in `/cs251/sets/BSTSet.ml`. You can test it via the following OCAML commands:
where \( ZZZ \) is one of Tiny, Small, Medium, or Large.

**Important:** It turns out that some of the above testing functions (in particular, `testLarge()`) require more stack space than is provided by OCaml by default. In order to declare that OCaml should have more stack space, you need to perform the following steps exactly once (and everything should be set after that):

1. Add the following line to the end of your `~/.bashrc` file:
   ```bash
   export OCAMLRUNPARAM='l=10M'
   ```
   Note that the character ‘l’ is a lowercase ‘L’ and not the digit ‘1’. This tells OCaml to allocate 10 megawords of stack space (40 times greater than the default 250 kilowords).

2. After modifying your `~/.bashrc` file, log out of Linux and then log back in. You should now be all set.

**Operation Tree Representation of Sets**

A very different way of representing a set as a tree is to remember the structure of the set operations `empty`, `insert`, `delete`, `union`, `intersection`, and `difference` used to create the set. For example, consider the set \( t \) created as follows:

```ocaml
let t = (delete 4 (difference (union (union (insert 1 empty) (insert 4 empty)) (union (insert 7 empty) (insert 4 empty))) (intersection (insert 1 empty) (union (insert 1 empty) (insert 6 empty)))));
```

Abstractly, \( t \) is the singleton set \( \{7\} \). But one concrete representation of \( t \) is the following operation tree:

![Operation Tree Representation of Sets](image)

One advantage of using such operation trees to represent sets is that the `empty`, `insert`, `delete`, `union`, `difference`, and `intersection` operations are extremely cheap – they just create a new tree node with the operands as subtrees, and thus take constant time and space! But other operations, such as `member` and `toList`, can be more expensive than in other implementations.
Your Task

In this problem, you are asked to flesh out the missing operations in the skeleton of the OperationTreeSet module in Fig. 5. In this module, the set datatype is created by constructors Empty, Insert, Delete, Union, Intersection, and Difference. The empty, singleton, insert, delete, union, intersection, difference, and toString operations are trivial and have already been implemented. You are responsible for fleshing out the definitions of the member, toList, fromList, toSexp, and fromSexp operations.

Notes:

- You can test your implementation via the following OCAML commands:

  ```
  #cd "/students/username/cs251/ps4-group";;
  #use "load-optree-set.ml";;
  testZZZ();
  ```

  where ZZZ is one of Tiny, Small, Medium, or Large. Before trying testLarge(), you should embiggen ;-) the default OCAML stack size by following the instructions at the beginning of this problem.

- The testing code assumes that toList and fromList work correctly, so you must implement these first for any of the tests to work.

- In toList, you may find it helpful to use functions in the ListSetUtils module. (These are also used in Handout #21 to implement SortedListSet.) The declaration

  ```
  module LSU = ListSetUtils
  ```

  in OperationTreeSet allows you to use the short prefix LSU rather than the long prefix ListSetUtils to access these functions.

- In fromList, for lists with $\geq 2$ elements, you should first split the list into two (nearly) equal-length sublists and union the results of turning the sublists into sets. This yields a height-balanced operation tree.

- Before implementing toSexp and fromSexp, you should study the toSexp and fromSexp functions in the sorted list and BST implementations of sets.

- In toSexp, you should represent each non-empty node in the operation tree as an s-expression list whose first element is a lowercase symbol naming the operator and the rest of whose elements are the operands. An empty node should be represented as the symbol empty. For example, the printed representation of the s-expression shown at the beginning of this problem is:

  ```
  (delete 4 (difference (union (union (insert 1 empty)
  (insert 4 empty))
  (union (insert 7 empty)
  (insert 4 empty)))
  (intersection (insert 1 empty)
  (union (insert 1 empty)
  (insert 6 empty))))
  ```

  Note that this printed representation is a legal OCAML expression that, when evaluated, would re-create the tree!

- In fromSexp, you can use nested patterns to succinctly describe how to convert s-expressions of the form described above into a constructor tree for the set datatype. If an inappropriate s-expression is encountered, fromSexp should raise an exception using the following code:
The testing code tests toSexp and fromSexp together, so your implementation will not pass the test cases until both are working correctly.

**Group Problem 2 [75]: A Postfix Interpreter**

In this problem, you will implement an interpreter for the Postfix language described in Handout #27. As explained there, the abstract syntax of Postfix can be expressed in OCaml via the following two datatypes for programs (pgm) and commands (com), which are already defined for you in the module PostFix in the file PostFix.ml.

```ocaml
type pgm = (* PostFix programs *)
  Pgm of int * com list
and com = (* PostFix commands *)
  Int of int (* push integer literal *)
| Str of string (* push string literal *)
| Seq of com list (* executable sequence *)
| Pop (* pop top value from stack *)
| Swap (* swap top two values of stack *)
| Sel (* choose one of two values from stack *)
| Get (* push value at given stack index *)
| Put (* store top of stack at given stack index *)
| Prs (* print string *)
| Pri (* print integer *)
| Exec (* execute sequence at top of stack *)
| Add | Sub | Mul | Div | Rem (* arithmetic ops *)
| LT | LE | EQ | NE | GE | GT (* relational ops *)
```

### a. [15]: Parsing and Unparsing Postfix

Although it is possible to write all Postfix programs and commands using the above OCaml AST constructors, it is inconvenient to do so. In this part, you will implement the concrete s-expression syntax for Postfix described in Handout #27 by implementing the following four functions in the PostFix module:

- **val sexpToCom : Sexp.sexp -> com**
  
  Parse an s-expression representation of a Postfix command into a value of type com.

- **val sexpToPgm : Sexp.sexp -> pgm**
  
  Parse an s-expression representation of a Postfix program into a value of type pgm.

- **val pgmToSexp : pgm -> Sexp.sexp**
  
  Unparse a value of type pgm into an s-expression representation of a Postfix program.

- **val comToSexp : com -> Sexp.sexp**
  
  Unparse a value of type com into an s-expression representation of a Postfix command.

**Notes:**

- Before starting this problem, study the INTEX code for and examples of parsing and unparsing s-expressions in Handout #26.
module OperationTreeSet : SET = struct

module LSU = ListSetUtils

type 'a set =
  Empty
| Insert of 'a * 'a set
| Delete of 'a * 'a set
| Union of 'a set * 'a set
| Intersection of 'a set * 'a set
| Difference of 'a set * 'a set

let empty = Empty

let insert x s = Insert(x,s)

let singleton x = Insert(x, Empty)

let delete x s = Delete(x, s)

let union s1 s2 = Union(s1,s2)

let intersection s1 s2 = Intersection(s1,s2)

let difference s1 s2 = Difference(s1,s2)

let rec member x s =
  (* Replace this stub *)
  false

let rec toList s = [] (* Replace this stub *)
  (* You may use operations in ListSetUtils, using the abbreviation LSU defined above. *)

let rec fromList xs = Empty (* Replace this stub *)
  (* You should define this in terms of a "balanced" tree of Union, Insert, and Empty nodes *)

let rec toSexp eltToSexp s = Sexp.Seq [] (* Replace this stub *)
  (* Returns an s-expression that shows the structure of the tree. See the PS4 Problem 1 description for examples *)

let rec fromSexp eltFromSexp sexp = Empty (* Replace this stub *)

let rec toString eltToString s =
  StringUtils.listToString eltToString (toList s)
end

Figure 5: Skeleton of the OperationTreeSet module.
Because some sexp and com constructors have the same names (e.g. Int, Str, Seq), you will need to qualify at least one set of constructor names with an explicit Sexp. prefix for all s-expression constructors.

When a parser encounters an error, it should raise a SyntaxError exception with a single string argument. For example, if sexp is the s-expression for an unknown command, an exception can be raised as follows:

```
raise (SyntaxError ("Unknown command: "
               ^ (Sexp.sexpToString sexp)))
```

The following four functions that convert between strings and Postfix programs are included in the PostFix skeleton and will be defined once you flesh out the above four functions:

```
let stringToCom str = sexpToCom (Sexp.stringToSexp str)
let stringToPgm str = sexpToPgm (Sexp.stringToSexp str)
let comToString c = Sexp.sexpToString (comToSexp c)
let pgmToString p = Sexp.sexpToString (pgmToSexp p)
```

b. [50]: Interpreting Postfix

In this part, you will implement a Postfix interpreter in the module PostFixInterp. In addition to opening up the PostFix module to make the syntax datatypes available, the PostFixInterp module defines the following datatypes needed for the interpreter:

1. The interpreter manipulates a stack of values that may either be integers, strings, or executable sequences. These stack values are modeled by the sval datatype:

   ```
type sval = (* stack values *)
   | IntVal of int
   | StrVal of string
   | SeqVal of com list
   ```

2. The “abstract machine state” manipulated by the Postfix interpreter consists of two parts: (1) a list of commands and (2) a stack of values. In Ocaml, an easy way to model the stack is as a list. So the following state type represents the configuration manipulated by the interpreter:

   ```
type state = com list * sval list
   ```

   Note that this is not like other type definitions we have seen – it does not specify a constructor! A constructorless type definition introduces a name that abbreviates a type. So state is just a shorter name for the pair type com list * sval list.

3. The result of executing a Postfix program is either an integer or an error. This is modeled by the ans datatype:

   ```
type ans = (* answers to executing Postfix programs *)
   | IntAns of int (* integer answer *)
   | ErrAns of string (* error answer *)
   ```

   In the Postfix interpreter, it is possible to model all error situations using the ErrAns constructor. In contrast with the Intex interpreter, it is not necessary to use exceptions to model any error situations.
Your Task:

To complete the Postfix interpreter, you will need to flesh out two functions:

- **val exec : state -> ans**
  
  Given a state (coms, svals), the `exec` function executes all of the Postfix commands `coms` in the context of a stack consisting of the values `svals`. It returns an answer that is either the integer at the top of the final stack or an error describing a state that is “stuck” (i.e., a non-final state from which no progress can be made).

- **val run : PostFix.pgm -> int list -> ans**
  
  Executes the given Postfix program on an integer list of arguments.

Your error messages should be chosen to accurately describe the type of error encountered. There are *lots* of different error situations; make sure you handle them all!

Notes:

- As usual, start this problem set by performing a `cvs update -d`, and perform an update every time you log in to work on this problem set.
- After launching Ocaml, connect to the ps4-group directory via `#cd "/students/your-account-name/cs251/ps4-group"`.
- To load all the appropriate files into Ocaml, execute `#use "load-postfix.ml"`. You will need to execute this every time you want to test changes to your code. This loads many files, including PostFix.ml, PostFixInterp.ml, and PostFixInterpTest.ml. Carefully look at the output of the `#use` command each time you invoke it. Even if it finds and reports an error in PostFix.ml, say, it will continue to load the other files, and it’s easy to miss the error, especially if the error message has scrolled off the screen.
- Use `run` or `runString` to test individual programs. E.g.:
  ```
  # run (Pgm(1, [Int 2; Mul])) [3];;
  - : PostFixInterp.ans = IntAns 6
  # runString "(postfix 1 2 mul)" [3];;
  - : PostFixInterp.ans = PostFixInterp.IntAns 6
  ```
- Use `test()` to test your interpreter on a large suite of test programs in `PostFixInterpTest`. The suite includes all the examples in Handout #27 and many additional error programs. In order for your interpreter to pass the test cases, it will be necessary for you to give the same error messages as those in test suite.
- The `#trace` directive can be very helpful for debugging a non-working `exec` function. The tracing output is easiest to read if you invoke it as follows:
  ```
  # open PostFix;;
  # open PostFixInterp;;
  # trace exec;;
  ```
- You *should* use any standard library functions from the standard modules that you find helpful.
- You *should not* use any imperative features of Ocaml (e.g. reference cells, array) to implement your interpreter. Even though the `state` type is immutable (consisting of two immutable lists), you can simulate changing state by creating a sequence of new state pairs in a tail recursion rather than modifying existing ones in a loop.
Pattern matching is your friend here. Use it to simplify your definitions.

In addition to defining `exec` and `run`, you may need to define some auxiliary functions.

When implementing the commands `Prs` and `Pri`, use `StringUtils.print` instead of `print_string`. The problem with `print_string` is that it “buffers” strings (potentially for a long time) before printing them. If you want to ensure that something gets printed on the console immediately, use `StringUtils.print`.

When printing values, it is necessary to sequence expressions. There are two styles of evaluating expressions $e_1, e_2, \ldots, e_n$ in order in OCAML:

1. In the *semi-colon style*, semi-colons are used to sequence statements (like in Java and C). It is often necessary to parenthesize the sequenced expressions as well. E.g.
   
   $(e_1; e_2; \ldots; e_n)$
   
   The value of such a sequence is the value of the last expression ($e_n$).

2. In the *let style*, `let`s are used to sequence the expressions used to sequence statements (like in Java and C). It is often necessary to parenthesize the sequenced expressions as well. E.g.
   
   ```ocaml
   let _ = e_1 in
   let _ = e_2 in
   ...
   let _ = e_{n-1} in
   e_n
   ```

It’s a good idea to implement, test, and debug a few commands at a time rather than attempting to handle all the commands at once.

c. [10]: GCD

In this part, your task is to write a two-argument POSTFIX program that implements the following OCAML `gcd` function:

```ocaml
let rec gcd a b =
  if b = 0 then
    a
  else
    gcd b (a mod b)
```

As part of this problem, you will need to figure out how to express a recursive function in POSTFIX. *(Hint: study the factorial example in Handout #27 and think very carefully about the problem.)* You should test your `gcd` program on a wide variety of inputs.
Name:

Date & Time Submitted:

By signing below, I attest that I have followed the policy for individual problems set forth in the Course Information handout. In particular, I have not consulted with any person except Lyn about these problems and I have not consulted any materials from previous semesters of CS251.

Signature:

In the Time column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading your problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1a [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1b [20]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Names of Team Members:

Date & Time Submitted:

Collaborators (anyone you or your team collaborated with):

By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.
Signature(s):

In the Time column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the Score column when grading you problem set.

<table>
<thead>
<tr>
<th>Part</th>
<th>Time</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Reading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 1 [25]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2a [15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2b [50]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Problem 2c [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>