Problem Set 4 Solutions

Individual Problems

Individual Problem [40]: Enter the Matrix!

There are many correct implementations of the matrix functions in this problem. Here some simple and efficient implementations are presented. Many student solutions “worked” in the sense that they had the correct behavior, but they were more complex or less efficient than they could have been. Carefully study the solutions presented here and compare them to your own solutions.

a. [20]: Matrices as Lists

make

The make function can be elegantly implemented via map and range:

\[
\text{let make rows cols f =}
\]
\[
\text{map (fun r ->}
\]
\[
\text{(map (fun c -> f r c)
\text{(range 1 cols)))}
\]
\[
\text{(range 1 rows)}
\]

Of course, it is possible to use foldr instead of map and gen instead of range, but such solutions are not as easy to read.

dimensions

To find the length of a list, we can use the standard List.length function, or define our own list length function in terms of foldr:

\[
\text{let len xs = foldr (fun _ n -> 1+n) 0 xs}
\]

Then we can determine the dimensions of a list by finding the length of the rows and the columns:

\[
\text{let dimensions xss = (len xss, len (List.hd xss))}
\]

get

For the get function, imagine that we have a getNth function with type

\[
\text{int -> 'a list -> 'a option}
\]

such that getNth i xs returns Some v if v is the i\text{th} element of the list (1-indexed) and returns None if i is out of bounds. Then the get function can be implemented by using getNth twice: once on the row, once on the column:

\[
\text{let get i j xss =}
\]
\[
\text{match getNth i xss with}
\]
\[
\text{None -> None}
\]
\[
| \text{Some row -> getNth j row}
\]

There are many ways to implement getNth. Here is a version defined in terms of some2:
let getNth i xs = (* Version 1 *)
  match some2 (fun j _ -> i = j) (range 1 (List.length xs)) xs with
  None -> None
  | Some (_, v) -> Some v

Here is a version defined in terms of drop:

let getNth i xs = (* Version 2 *)
  if (i <= 0) || (i > (List.length xs)) then
    None
  else
    Some (List.hd (drop (i-1) xs))

Here, drop i xs returns a list without its first i elements. We could use the version of drop in ListUtils or we could define it on our own in terms of iterate:

let drop i xs =
  snd (iterate (fun (j,ys) -> (j-1, match ys with [] -> [] | (_::ys') -> ys'))
       (fun (j,ys) -> j = 0 || ys = [])
       (i, xs))

put

As in get, with put it is also helpful to define a helper function for the one-dimensional case. Assume that we have a putNth function with type

int -> 'a -> 'a list -> 'a list

such that putNth i v xs returns a copy of xs in which the i-th value (1-indexed) has been replaced by v. If i is out of range, xs should be returned unchanged. Using getNth and putNth, we can implement put as follows:

let put i j v xss = (* Version 1 *)
  match getNth i xss with
  None -> xss
  | Some row -> putNth i (putNth j v row) xss

An easy way to implement putNth is by using map2 to copy the list except at the specified index:

let putNth i v xs = (* Version 1 *)
  map2 (fun j x -> if i = j then v else x)
       (range 1 (List.length xs))
       xs

An alternative version is to use take and drop:

let putNth i v xs = (* Version 2 *)
  if (i <= 0) || (i > List.length xs) then
    xs
  else
    (take (i-1) xs) @ [v] @ (drop i xs)

In the second version, more structure is shared between the input list and the output list.

An important feature of the above definition of put is that it copies at most one row of the matrix. So for an \( n \times n \) matrix, it takes time proportional to \( n \). Many students defined put in a much more inefficient way using something like the following:
let valOf opt =  
  match opt with  
    None -> raise (Failure "valOf")  
  | Some v -> v

let put i j v xss = (* Version 2 *)  
let (r,c) = dimensions xss in  
  if (i <= 0) || (i > r) || (j <= 0) || (j > c) then  
    xss  
  else  
    make r c (fun i' j' -> if (i' = i) && (j' = j) then v else valOf(get i j xss))

Not only does this version have to visit each of the $n^2$ locations of the matrix, but the \texttt{get} at each location takes $O(n)$ time. So this alternative version of \texttt{put} takes time $\Theta(n^3)$ compared to the $\Theta(n^2)$ time of the first version.

\textbf{transpose}

The \texttt{transpose} function can be performed both elegantly and efficiently as follows:

\begin{verbatim}
let transpose xss = (* Version 1 *)  
  foldr (map2 cons)  
    (map (fun x -> []) (List.hd xss))  
    xss
\end{verbatim}

It is also possible to define \texttt{transpose} in terms of \texttt{make}:

\begin{verbatim}
let transpose xss = (* Version 2 *)  
  let (r,c) = dimensions xss in  
  make c r (fun i j -> valOf(get j i xss))
\end{verbatim}

However, the second version is less efficient than the first. For a $n \times n$ matrix, the first version takes time $\Theta(n^2)$ but the second takes time $\Theta(n^3)$.

\textbf{map}

The matrix \texttt{map} function can be performed by using nested list maps:

\begin{verbatim}
let map f xss = map (map f) xss (* Version 1 *)
\end{verbatim}

This version of \texttt{map} takes time $\Theta(n^2)$ on a $n \times n$ matrix. As with \texttt{transpose}, it is also possible to define \texttt{map} in terms of \texttt{make}, but such a definition takes time $\Theta(n^3)$:

\begin{verbatim}
let map f xss = (* Version 2 *)  
  let (r,c) = dimensions xss in  
  make r c (fun i j -> f (valOf(get i j xss)))
\end{verbatim}

\textbf{toLists}

For the list representation of matrices, the implementation of \texttt{toLists} is trivial:

\begin{verbatim}
let toLists xss = xss
\end{verbatim}
b. [20]: Matrices as Functions

**make**

In the functional representation of matrices, the `make` function returns a function of two indices that performs bounds-checking on the indices before applying the given `f`:

```ocaml
define make rows cols f =
  fun i j ->
    if (i <= 0) || (i > rows) || (j <= 0) || (j > cols) then
      None
    else
      Some (f i j)
```

**dimensions**

Finding the dimensions of a matrix in the functional representation requires searching for the first row and column indices at which the abstract matrix contains `None`. It is helpful to define an auxiliary function that returns the first index (1-indexed) of a list at which a predicate is true:

```ocaml
(* Find the first integer >= 1 satisfying predicate *)
define first pred = iterate ((+) 1) pred 1
```

Then `dimensions` can be defined by using `first` twice:

```ocaml
define dimensions m =
  ((first (fun i -> (m i 1) = None))-1,
   (first (fun j -> (m 1 j) = None))-1)
```

It takes $\Theta(n)$ time to perform `dimensions` on a $n \times n$ matrix. (This assumes that the underlying function `f` supplied to `make` takes constant time.)

Note that `first` uses linear search to find the index. For the case of matrices, we could instead use a strategy of doubling the indices until a `None` is reached, and then use binary search between this index and the index of the last non-`None` index to find the first `None` index.

**get**

The `get` function is trivial in this representation:

```ocaml
define get i j m = m i j
```

Note that since bounds checks are already performed within the function returned by `make`, there is no need for bounds-checking in `get`.

**put**

The challenging aspect of implementing `put` is determining whether the given indices are in bounds. A cheap way to do this is to call `get` on the indices, as in the following implementation:

```ocaml
define put i j v m =
  (* Version 1 *)
  match get i j m with
    (* perform the bounds checks *)
    None -> m
  | Some _ -> fun i' j' -> if (i=i') && (j=j') then Some v else m i' j'
```

Alternatively, `dimensions` can be used for bounds-checking:
let put i j v m = (* Version 2 *)
    let (rows, cols) = dimensions m in
    if (i <= 0) || (i > rows) || (j <= 0) || (j > rows) then
        m
    else
        fun i' j' -> if (i=i') && (j=j') then Some v else m i' j'

However, for an \( n \times n \) matrix, the version using \texttt{dimensions} takes \( \Theta(n) \) time for the bounds checks while the other version takes \( \Theta(1) \) time for the bounds checks. Things are even worse if \texttt{dimensions} and the bounds checks are moved inside the body of \texttt{fun i’ j’ -> ...}, because then they must be performed on every lookup from the resulting matrix.

\texttt{transpose}

In the functional representation, \texttt{transpose} can be elegantly and efficiently defined as follows:

\[
\texttt{let transpose m = fun i j -> m j i} (* Version 1 *)
\]

It is also possible to define \texttt{transpose} in terms of \texttt{make}:

\[
\texttt{let transpose m = (* Version 2 *)}
    \texttt{let (r,c) = dimensions m in}
    \texttt{make c r (fun i j -> valOf(m j i))}
\]

However, (1) this version must use \texttt{valOf} for correct typing (not required by the first version) and (2) for an \( n \times n \) matrix, this version takes time \( \Theta(n) \) for the bounds-checking performed by \texttt{dimensions} (not required by the first version).

\texttt{map}

Here is a version of \texttt{map} for the functional representation:

\[
\texttt{let map f m = fun i j -> match m i j with}
    \texttt{None -> None}
    \texttt{| Some v -> Some (f v)}
\]

It is also possible to define \texttt{map} in terms of \texttt{make}, but it has the same problems as the \texttt{make}-based version of \texttt{transpose}:

\[
\texttt{let map f m = (* Version 2 *)}
    \texttt{let (r,c) = dimensions m in}
    \texttt{make r c (fun i j -> f (valOf(m i j)))}
\]

\texttt{toList}

In the functional representation, implementing \texttt{toList} requires making a list of lists of the correct dimensions in which the function \( f \) supplied to \texttt{make} is applied to all coordinates \((i,j)\) that are in bounds. The most straightforward way to do this is via nested maps on the coordinates from the two dimensions:

\[
\texttt{let toLists m = (* Version 1 *)}
    \texttt{let (rows, cols) = dimensions m in}
    \texttt{map (fun i ->}
        \texttt{map (fun j -> valOf (m i j))}
        \texttt{(range 1 cols))}
    \texttt{(range 1 rows)}
\]
The above version ends up calculating the underlying function \( f \) twice on the elements in the first row and the first column (due to \texttt{dimensions}). Although this is not a big deal, it is avoided in the following definition, which uses \texttt{ana} to calculate the elements and the dimensions simultaneously:

\[
\begin{align*}
\text{let rowToList } & \ i \ m = \\
& \text{match } m \ i \ 1 \text{ with} \\
& \quad \text{None } \to \text{ None} \\
& \quad | \text{ Some } v \to \text{ Some } (v :: (\text{ana } (\text{fun } j \to \text{match } m \ i \ j \text{ with} \\
& \quad \quad \text{None } \to \text{ None} \\
& \quad \quad | \text{ Some } x \to \text{ Some } (x, j+1)) 2))
\end{align*}
\]

\[
\begin{align*}
\text{let toLists } & \ m = \\
& \text{ana } (\text{fun } i \to \text{match } \text{rowToList } i \ m \text{ with} \\
& \quad \text{None } \to \text{ None} \\
& \quad | \text{ Some } r \to \text{ Some } (r, i+1))
\end{align*}
\]

### Group Problems

**Group Problem 1 [25]: OperationTreeSet**

**member**

The \texttt{member} function walks over the operation tree to determine if \( x \) is an element in the set – i.e., it has been inserted into the set but not deleted from it:

\[
\begin{align*}
\text{let rec } & \ 	ext{member } x \ s = \\
& \text{match } s \text{ with} \\
& \quad \text{Empty } \to \text{ false} \\
& \quad | \text{ Insert}(y,s') \to \text{ (x = y) || (member } x \ s') \\
& \quad | \text{ Delete}(y,s') \to \text{ (not (x = y)) && (member } x \ s') \\
& \quad | \text{ Union}(s1,s2) \to \text{ (member } x \ s1) || (member } x \ s2) \\
& \quad | \text{ Intersection}(s1,s2) \to \text{ (member } x \ s1) && (member } x \ s2) \\
& \quad | \text{ Difference}(s1,s2) \to \text{ (member } x \ s1) && \text{ (not (member } x \ s2))
\end{align*}
\]

Note that the clauses for \texttt{Union}, \texttt{Intersection}, and \texttt{Difference} are reminiscent of the \texttt{union}, \texttt{difference}, and \texttt{intersection} functions for the \texttt{PredSet} implementation in Problem Set 3.

**toList**

The \texttt{toList} function defers most of its work to the auxiliary functions in the \texttt{ListSetUtils} (LSU) module:

\[
\begin{align*}
\text{let rec } & \ 	ext{toList } s = \\
& \text{match } s \text{ with} \\
& \quad \text{Empty } \to \text{ []} \\
& \quad | \text{ Insert}(y,s') \to \text{ LSU.insert } y \ (\text{toList } s') \\
& \quad | \text{ Delete}(y,s') \to \text{ LSU.delete } y \ (\text{toList } s') \\
& \quad | \text{ Union}(s1,s2) \to \text{ LSU.union } (\text{toList } s1) \ (\text{toList } s2) \\
& \quad | \text{ Intersection}(s1,s2) \to \text{ LSU.intersection } (\text{toList } s1) \ (\text{toList } s2) \\
& \quad | \text{ Difference}(s1,s2) \to \text{ LSU.difference } (\text{toList } s1) \ (\text{toList } s2)
\end{align*}
\]

**fromList**

To maintain tree balance, the \texttt{fromList} function uses the \texttt{alts} function from Problem Set 2 to split a list into two lists of about the same size:
let alts xs = ListUtils.foldr (fun x (odds, evens) -> (x::evens, odds)) ([], []) xs

let rec fromList xs =
    match xs with
    | [] -> Empty
    | [x] -> singleton x
    | _ -> let (ys, zs) = alts xs in Union(fromList ys, fromList zs)

toSexp

The toSexp function recursively builds an s-expression representation of the tree. Note that according to the problem specification, empty should not appear in parens — it is an s-expression symbol, not an s-expression list.

let rec toSexp eltToSexp s =
    match s with
    | Empty -> Sexp.Sym "empty"
    | Insert(y, s') -> Sexp.Seq [Sexp.Sym "insert";
      eltToSexp y;
      toSexp eltToSexp s']
    | Delete(y, s') -> Sexp.Seq [Sexp.Sym "delete";
      eltToSexp y;
      toSexp eltToSexp s']
    | Union(s1, s2) -> Sexp.Seq [Sexp.Sym "union";
      toSexp eltToSexp s1;
      toSexp eltToSexp s2]
    | Difference(s1, s2) -> Sexp.Seq [Sexp.Sym "difference";
      toSexp eltToSexp s1;
      toSexp eltToSexp s2]
    | Intersection(s1, s2) -> Sexp.Seq [Sexp.Sym "intersection";
      toSexp eltToSexp s1;
      toSexp eltToSexp s2]

fromSexp

The fromSexp function is the inverse of toSexp:

let rec fromSexp eltFromSexp sexp =
    match sexp with
    | Sexp.Sym "empty" -> Empty
    | Sexp.Seq [Sexp.Sym "insert"; yx; sx] ->
      Insert(eltFromSexp yx, fromSexp eltFromSexp sx)
    | Sexp.Seq [Sexp.Sym "delete"; yx; sx] ->
      Delete(eltFromSexp yx, fromSexp eltFromSexp sx)
    | Sexp.Seq [Sexp.Sym "union"; s1x; s2x] ->
      Union(fromSexp eltFromSexp s1x, fromSexp eltFromSexp s2x)
    | Sexp.Seq [Sexp.Sym "intersection"; s1x; s2x] ->
      Intersection(fromSexp eltFromSexp s1x, fromSexp eltFromSexp s2x)
    | Sexp.Seq [Sexp.Sym "difference"; s1x; s2x] ->
      Difference(fromSexp eltFromSexp s1x, fromSexp eltFromSexp s2x)
    | _ -> raise (Failure ("TreeSet.fromExp -- can't handle sexp:
      ^ (Sexp.sexpToString sexp)\n"))
Group Problem 2 [75]: A Postfix Interpreter

a. [15]: Parsing and Unparsing Postfix

Fig. 1 shows the functions for parsing Postfix programs and commands from s-expressions.

```ml
let rec sexpToPgm sexp =  
match sexp with  
  Sexp.Seq(Sexp.Sym("postfix") :: Sexp.Int(n) :: comSexps) ->  
    Pgm(n, ListUtils.map sexpToCom comSexps)  
| _ -> raise (SyntaxError ("Ill-formed program: "  
    "  " (Sexp.sexpToString sexp)))

and sexpToCom sexp =  
match sexp with  
  Sexp.Int i -> Int i  
| Sexp.Str s -> Str s  
| Sexp.Seq comSexps -> Seq (ListUtils.map sexpToCom comSexps)  
| Sexp.Sym "add" -> Add  
| Sexp.Sym "sub" -> Sub  
| Sexp.Sym "mul" -> Mul  
| Sexp.Sym "div" -> Div  
| Sexp.Sym "rem" -> Rem  
| Sexp.Sym "lt" -> LT  
| Sexp.Sym "le" -> LE  
| Sexp.Sym "eq" -> EQ  
| Sexp.Sym "ne" -> NE  
| Sexp.Sym "ge" -> GE  
| Sexp.Sym "gt" -> GT  
| Sexp.Sym "pop" -> Pop  
| Sexp.Sym "swap" -> Swap  
| Sexp.Sym "sel" -> Sel  
| Sexp.Sym "get" -> Get  
| Sexp.Sym "put" -> Put  
| Sexp.Sym "prs" -> Prs  
| Sexp.Sym "pri" -> Pri  
| Sexp.Sym "exec" -> Exec  
| _ -> raise (SyntaxError ("Unknown command: "  
    "  " (Sexp.sexpToString sexp)))
```

Figure 1: Functions for parsing Postfix programs from s-expressions.

To avoid name clashes on the constructors Int, Str, and Seq (which are defined in both the Sexp and PostFix modules), it is necessary to use explicit qualification on the Sexp names Sexp.Int, Sexp.Str, and Sexp.Seq, regardless of whether the Sexp module is opened.

The s-expression representation of a Postfix program is

\[(\text{program } \text{numargs } \text{com}_1 \ldots \text{com}_n)\]

To allow for any number of commands, the pattern matching this must be

\[\text{Sexp.Seq(Sexp.Sym("postfix") :: Sexp.Int(n) :: comSexps)}\]

since this allows comSexps to be an sexp list of any length. It would be incorrect to use

\[\text{Sexp.Seq [Sexp.Sym("postfix"); Sexp.Int(n); comSexps]}\]
since this forces \texttt{comSexps} to be a single command s-expression.

Note how \texttt{ListUtils.map sexpToCom} is used both in \texttt{sexpToPgm} and \texttt{sexpToCom} (in parsing the \texttt{Seq} command) to parse a list of command s-expressions.

Fig. 2 shows the functions for unparsing POSTFIX programs and commands to s-expressions. Note how \texttt{ListUtils.map comToSexp} is used both in \texttt{pgmToSexp} and \texttt{comToSexp} (in unparsing the \texttt{Seq} command) to unparse a list of commands.

```ocaml
let rec pgmToSexp pgm =  
  match pgm with  
  Pgm(n,coms) -> Sexp.Seq((Sexp.Sym "postfix")  
                        :: (Sexp.Int n)  
                        :: (ListUtils.map comToSexp coms))

and comToSexp com =  
  match com with  
  Int i -> Sexp.Int i  
| Str s -> Sexp.Str s  
| Seq cs -> Sexp.Seq (ListUtils.map comToSexp cs)  
| Pop -> Sexp.Sym "pop"  
| Swap -> Sexp.Sym "swap"  
| Sel -> Sexp.Sym "sel"  
| Get -> Sexp.Sym "get"  
| Put -> Sexp.Sym "put"  
| Prs -> Sexp.Sym "prs"  
| Pri -> Sexp.Sym "pri"  
| Exec -> Sexp.Sym "exec"  
| Add -> Sexp.Sym "add"  
| Sub -> Sexp.Sym "sub"  
| Mul -> Sexp.Sym "mul"  
| Div -> Sexp.Sym "div"  
| Rem -> Sexp.Sym "rem"  
| LT -> Sexp.Sym "lt"  
| LE -> Sexp.Sym "le"  
| EQ -> Sexp.Sym "eq"  
| NE -> Sexp.Sym "ne"  
| GE -> Sexp.Sym "ge"  
| GT -> Sexp.Sym "gt"
```

Figure 2: Functions for unparsing POSTFIX programs to s-expressions.

b. [50]: Interpreting POSTFIX

Here is the \texttt{run} function
(* run : pgm -> int list -> ans *)
let run pgm args =
  match pgm with
  (Pgm(n,coms)) ->
  let len = List.length(args) in
  if n != len then
    ErrAns("program expected " ^ (string_of_int n)
    ^ " arguments but got " ^ (string_of_int len))
  else
    exec (coms, List.map (fun x -> IntVal x) args)

It first checks if the number of supplied arguments matches the number of expected arguments. If not, an error answer is returned. If so, it calls exec on an initial program state — i.e., a pair of commands and stack values. Not that each int argument must be converted to a stack value via the IntVal constructor.

The exec function and its associated helper functions are presented in Fig. 3. The exec function has a single base case (an empty command list) with three subcases:

1. The stack is empty (an error)
2. The stack has an integer at the top (the answer)
3. The stack has a non-integer at the top (an error)

The remaining clauses of exec handle the commands one by one. Pattern matching features are used extensively to make the function very concise. Here are the highlights:

- Integer, string, and executable sequence commands are just pushed onto the stack, but must first be coerced¹ to stack values via IntVal, StrVal, and SeqVal, respectively.
- The exec command is executed by appending the commands from the executable sequence at the top of the stack to the current command list.
- There are two clauses for sel. The first uses the pattern IntVal 0 to match the false case and the second uses IntVal _ to match the true case (since POSTFIX treats any non-zero integer as true).
- For processing the get command, the when guard guarantees that the integer index i is in the range [1, n], where n is the number of stack values below the index. List.nth is used to reference the stack element at index i. Because List.nth is 0-indexed and POSTFIX is 1-indexed, it is necessary to call List.nth on (i-1) rather than i.
- The put command is handled similarly to get. The updateNth auxiliary function is an analog of List.nth for updating the value at the given index in a list. For consistency with List.nth, updateNth is 0-indexed, so it is necessary to subtract 1 from the POSTFIX index i.
- All the arithmetic commands (Add, Sub, Mul, Div, Rem) can be handled in a single clause using OCAML’s or patterns (separated by vertical bars) and when guard (to prevent division and remainder by zero). The stack pattern (IntVal i1)::(IntVal i2)::vs guarantees that there are two integers values on the stack. Note that The arithop auxiliary function specifies which arithmetic operation to perform. Note that i2 is the first argument and i1 is the second argument of the arithmetic operation in accordance with the POSTFIX

¹How’s that for a rhyme!
(* exec : state -> ans *)
let rec exec s =
  match s with
| ([], []) -> ErrAns "final stack is empty"
| ([], (IntVal i)::_) -> IntAns i
| ([], vs) -> ErrAns("non-int at top of final stack: " ^ (stkToString vs))
| ((Int i)::cs, vs) -> exec(cs, (IntVal i)::vs)
| ((Str s)::cs, vs) -> exec(cs, (StrVal s)::vs)
| ((Seq coms)::cs, vs) -> exec(cs, (SeqVal coms)::vs)
| (Exec::cs, (SeqVal coms)::vs) -> exec(coms@cs, vs)
| (Pop::cs, _::vs) -> exec(cs, vs)
| (Swap::cs, v1::v2::vs) -> exec(cs, v2::v1::vs)
| (Sel::cs, v1::v2::(IntVal 0)::vs) -> exec(cs, v1::vs)
| (Sel::cs, v1::v2::(IntVal _)::vs) -> exec(cs, v2::vs) (* non-zero test is "true" *)
| (Get::cs, (IntVal i)::vs) when (1 <= i) && i <= List.length(vs) ->
  exec(cs, (List.nth vs (i-1))::vs)
| (Put::cs, (IntVal i)::v::vs) when (1 <= i) && i <= List.length(vs) ->
  exec(cs, (updateNth vs (i-1) v))
| (Prs::cs, (StrVal s)::vs) -> (StringUtils.print s; exec(cs, vs))
| (Pri::cs, (IntVal i)::vs) -> (StringUtils.print (string_of_int i); exec(cs,vs))
| (((Add | Sub | Mul | Div | Rem) as aop)::cs, (IntVal i1)::(IntVal i2)::vs) when not (((aop = Div) || (aop = Rem)) && (i1 = 0)) ->
  exec(cs, IntVal(arithop aop i2 i1)::vs)
| (((LT | LE | EQ | NE | GE | GT) as rop)::cs, (IntVal i1)::(IntVal i2)::vs) ->
  exec(cs, IntVal(boolToInt(relop rop i2 i1))::vs)
| ((c::cs, vs) ->
  ErrAns("invalid stack for " ^ comToString(c) ^ ": " ^ (stkToString vs))
and arithop aop x y =
  match aop with
| Add -> x + y
| Sub -> x - y
| Mul -> x * y
| Div -> x / y
| Rem -> x mod y
| _ -> raise (Failure "can’t happen") (* make type checker happy *)

and relop rop x y =
  match rop with
| LT -> x < y
| LE -> x <= y
| EQ -> x = y
| NE -> x != y
| GE -> x >= y
| GT -> x > y
| _ -> raise (Failure "can’t happen") (* make type checker happy *)

and boolToInt b = match b with true -> 1 | false -> 0

and updateNth xs n v = match xs with
| [] -> raise (Failure ("updateNth"))
| (x::xs') ->
  if n = 0 then (v::xs')
  else x::(updateNth xs' (n - 1) v)

Figure 3: The exec function and its helper functions.
specification for these commands (a crucial fact for the non-commutative commands sub, div, and rem).

- In a similar fashion, all the relational commands (LT, LE, EQ, NE, GE, GT) are handled in a single clause. The relop auxiliary function specifies the comparison to perform, and boolToInt converts the resulting boolean to an integer (1 for true and 0 for false) as required by Postfix.

- Because every valid command application is handled before the last match clause of exec, all command errors can be handled by a single clause. For uniformity, this error message for this clause mentions the command name and the contents of the offending stack.

c. [10]: GCD

You were asked to translate the following tail-recursive OCAML gcd function into Postfix:

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

There are many possible translations. Here we describe one approach.

In any approach we need to specify the convention for the stack locations of the integers a and b and the code gcd. Let’s assume that stack (from top down) looks like:

```
a  b  <gcd commands>
```

The `<gcd commands>` sequence is a sequence of Postfix commands that implements the if expression of the OCAML gcd function. This has the form:

```
(2 get ; copy b to the top of stack
  0 eq ; test if (b = 0)
  <true case commands>
  <false case commands>
  sel exec) ; execute the appropriate branch
```

The true case must return a. Since a is already at the top of the stack in our convention, `<true case commands>` can simply be the empty command sequence (). The false case must implement the OCAML invocation `gcd b (a mod b)`. This can be expressed as:

```
(2 get ; copy b to the top of the stack
  rem ; calculate a mod b
  swap ; top of stack is now b followed by (a mod b)
  3 get ; copy gcd command sequence to top of stack
  exec) ; execute the gcd command sequence on b and (a mod b)
```

So `<gcd commands>` is
Initially, the stack looks like

\[ \text{a} \]
\[ \text{b} \]

so some work must be done to transform it into

\[ \text{a} \]
\[ \text{b} \]
\[ \text{<gcd commands>} \]

This can be done as follows:

\[
\text{(postfix 2 swap 2 get (2 get 0 eq () (2 get rem swap 3 get exec) sel exec) 2 put 3 get exec)}
\]

So the final form of our Postfix gcd program is:

\[
\text{(postfix 2 swap 2 get \text{<gcd commands>} 3 put; Stack is now a b <gcd commands> 3 get exec); execute gcd on initial stack)}
\]

If we choose other conventions for stack locations, we will get different programs. For example, if the stack layout is

\[ \text{b} \]
\[ \text{a} \]
\[ \text{<gcd commands>} \]

then a corresponding translation is:

\[
\text{(postfix 2 swap (1 get 0 eq (pop) (swap 2 get rem 3 get exec) sel exec) 3 get swap 3 put swap 3 get exec)}
\]

All of the solutions discussed above use constant stack space to calculate gcd. There are other solutions that push stack frames containing the arguments to gcd without popping them. Although these give the correct answer, these solutions are not “in the spirit” of an iterative solution because they could potentially run out of stack space for a large enough problem.

The gcd function has the property that \( \text{gcd} \ x \ y \) is the same as \( \text{gcd} \ y \ x \) for all \( x \) and \( y \). A consequence is that there are Postfix programs in which careful attention is not paid to argument order, but the correct answer is nevertheless computed (though in a roundabout way).