15-150 Spring 2012
Homework 05

Out: Tuesday, 14 February 2012
Due: Saturday, 25 February 2012 at 01:50 EST

1 Introduction

This homework will focus on applications of higher order functions, polymorphism, and user-defined datatypes.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

To submit your solutions, place your `hw05.pdf` and modified `hw05.sml` files in your handin directory on AFS:

```
/afs/andrew.cmu.edu/course/15/150/handin/<yourandrewid>/hw05/
```

Your files must be named exactly `hw05.pdf` and `hw05.sml`. After you place your files in this directory, run the check script located at

```
/afs/andrew.cmu.edu/course/15/150/bin/check/05/check.pl
```

then fix any and all errors it reports.

Remember that the check script is *not* a grading script—a timely submission that passes the check script will be graded, but will not necessarily receive full credit.

Also remember that your written solutions must be submitted in PDF format—we do not accept MS Word files.

Your `hw05.sml` file must contain all the code that you want to have graded for this assignment and compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Methodology

You must use the five step methodology for writing functions for every function you write on this assignment. In particular, you will lose points for omitting the purpose, examples, or tests even if the implementation of the function is correct.

1.4 Style

We will continue grading this assignment based on the style of your submission as well as its correctness. Please consult course staff, your previously graded homeworks, or the published style guide as questions about style arise.

1.5 Due Date

This assignment is due on Saturday, 25 February 2012 at 01:50 EST. Note that this is not the normal day or time! Remember that this deadline is final and that we do not accept late submissions.

1.6 Recursive Code

Unless otherwise stated, the functions you define for this assignment *must not be recursive.* Instead, you should rely on the higher order list functions to do the work of manipulating lists for you. While you are free to use others as you see fit, the following list functions will very likely be useful:

- `List.map`, which takes a function and a list and returns a list whose elements are the result of applying the given function to the corresponding element in the given list. For example,

```
- List.map (fn x => Int.toString (x*x*2)) [4,2,1];
val it = ["32","8","2"] : string list
```

  This is a curried version of the `map` we wrote in class.

- `ListPair.map`, which takes a function and a pair of lists and applies the function to the pairs of elements of the list in order. For example,

```
fun doctor (n : int, s : string) : string =
  case n of
    0 => ""
  | _ => s ^ (doctor (n - 1,s)
- ListPair.map (fn(i,s) => doctor (i*3,s))
           ([0,2,1],[‘‘hello’’,’’goodbye’’]);
val it = ["",‘‘goodbyegoodbyegoodbyegoodbyegoodbyegoodbye’’]
                                                  : string list
```

If this feels familiar, it's because `ListPair.map` can be expressed through a combination of the normal `List.map` and your `zip` function from Homework 3.

- `ListPair.zip`, which takes a pair of lists and returns a list of pairs whose length is the minimum of the lengths of the input lists. For example,

  ```
  - ListPair.zip([1,2,4,2], [true, false, false]);
  val it = [(1,true),(2,false),(4,false)] : (int * bool) list
  ```

  This is the same as the function `zip` from Homework 3.

- `ListPair.unzip`, takes a list of pairs and returns a pair of lists. For example,

  ```
  - ListPair.unzip [(1,"true"),(2,"false"),(4,"false")];
  val it = ([1,2,4],["true","false","false"])
                  : int list * string list
  ```

  which is similar to the `unzip` from Homework 3.

Be warned: solutions that are correct but are recursive instead of calling an appropriate higher order function will receive little or no credit.

1.7 Rational Numbers

In several of the tasks on this assignment, we will use an SML representation of the rational numbers. Rational numbers are represented by the type `rat` in `hw05.sml`. We also have the following functions (the first four are defined to be infix operators) for computing with rational numbers:[1]

```
val // : int * int -> rat

val ++ : rat * rat -> rat
val -- : rat * rat -> rat
val ** : rat * rat -> rat
val divide : rat * rat -> rat
val ~~ : rat -> rat
```

[1] The definitions of these functions are given in the files in the rationals directory. You can read these files if you choose, but you do not need to understand their contents to complete the tasks on this assignment.

The `//` function computes the rational number corresponding to the first `int` over the second. This is how you get values of type `rat` from values of type `int`.

The other functions compute the indicated operation (*i.e.,* addition, subtraction, multiplication, division, and unary negation) on values of type `rat`.

For example, the expression

```
val x = (5 // 9) ++ (12 // 10)
```

binds the identifier `x` to a value of type `rat` that represents

$$\frac{5}{9} + \frac{12}{10} = \frac{79}{45}$$

If you have an expression that has type `rat` and you would like to see a string representation of it, you can call the function `r2s` also provided in `hw05.sml`. For example, consider the following session with the SML REPL:

```
val r2s = fn : rat -> string
- 5 // 0;
uncaught exception Fail [Fail: denominator can’t be zero]
  raised at: rat.sml:31.42-31.74
- 5 // 1;
val it = - : Rational.t
- r2s (5 // 1);
val it = ‘‘5’’ : string
- r2s ( (5//1) ** (1221 // 51) ++ (2//90));
val it = ‘‘91592/765’’ : string
```

2 Proofs

2.1 Write And Prove

In this question you will write a simple function on lists and prove its correctness. This is an independent part of this assignment meant to give you practice coding and proving at the same time; you shouldn't worry if you don't use the function defined here in rest of the assignment.

You may not use `@`, any other built-in list functions, or any helper functions for the tasks in this section. If you do, you will receive little or no credit for the whole section. The implementations of the programming tasks in this section may be recursive.

2.1.1 Implementing Concat

Task 2.1 (5%). Write a function

```
concat : 'a list list -> 'a list
```

`concat` flattens a list of lists of any type into one list of that type while preserving the order. For example,

```
concat [] ≅ []
concat [[]] ≅ []
concat [[[]]] ≅ [[]]
concat [[1]] ≅ [1]
concat [[],["a","b"],[],[],[],["z"]] ≅ ["a","b","z"]
concat [[1,2],[5,6],[],[10,10]] ≅ [1,2,5,6,10,10]
```

2.1.2 A Theorem About Concat

Recall the definition of the append function we gave in Lecture 3:

```
fun ap (l1 : 'a list, l2 : 'a list) : 'a list =
  case l1
   of [] => l2
    | x::xs => x :: (ap (xs,l2))
```

We can write another implementation of the `concat` spec in terms of `ap` as

```
fun concatap (l : 'a list list) : 'a list =
  case l
   of [] => []
    | (x::xs) => ap(x, concatap xs)
```

Your task will be to prove that these two implementations are indistinguishable.

Task 2.2 (3%). First, prove Lemma 1 relating `ap` and `concatap`.[2]

Lemma 1. *For all values* $l1$: α `list` *and all values* $l2$: α `list list`

$$\texttt{ap}(\texttt{l1}, \texttt{concatap l2}) \cong \texttt{concatap}(\texttt{l1} :: \texttt{l2})$$

Task 2.3 (12%). Now, prove Theorem 1 by structural induction. You will almost certainly need to use Lemma 1 that you proved above in your proof here; you may also use Lemma 2 freely without proof. Remember to closely follow the proof templates we give in the lecture notes, argue carefully with equivalence, and cite the lemmas as you use them.[3]

Theorem 1. *For all values* l : α `list list`, `concat l` $\cong$ `concatap l`

Lemma 2. *For all values* l : α `list`, `ap([],l)` $\cong$ l

Solution 2.3

Proof. **WTS:** For all values $l1$: α `list` and all values $l2$: α `list list`

$$\texttt{ap}\ (\texttt{l1}, \texttt{concatap l2}) \cong \texttt{concatap}\ (\texttt{l1} :: \texttt{l2})$$

$$\begin{aligned}
& \texttt{concatap}\ (\texttt{l1} :: \texttt{l2}) & \\
\cong\ & \texttt{case}\ (\texttt{l1} :: \texttt{l2})\ \texttt{of}\ [\,] => [\,] & \textsc{step},\ \texttt{l1} :: \texttt{l2}\ \textsc{value} \\
& \mid (\texttt{x} :: \texttt{xs}) => \texttt{ap}\ (\texttt{x}, \texttt{concatap xs}) & \\
\cong\ & \texttt{ap}\ (\texttt{l1}, \texttt{concatap l2}) & \textsc{step}
\end{aligned}$$

□

[2] *Hint:* You do not need induction for this proof.

[3] It's interesting to note that we could have stated Theorem 1 a more concisely as

$$\texttt{concat} \cong \texttt{concatap}$$

which is a direct transcription of the intuition of the problem into a formal statement. The statement given is an immediate expansion of this, using the definition of contextual equivalence at a function type, so we don't lose anything by being a little bit more verbose.

Proof. **WTS:** For all values l : α `list list`, `concat l` $\cong$ `concatap l`.

Proceed by structural induction on l.

Case: `l` = [].

WTS: `concat` [] $\cong$ `concatap` [].

$$\begin{array}{lll} & \texttt{concat []} & \\ \cong & \texttt{case [] of [] => [] | ...} & \textsc{step}, \texttt{[]}\ \textsc{value} \\ \cong & \texttt{[]} & \textsc{step} \end{array}$$

$$\begin{array}{lll} & \texttt{concatap []} & \\ \cong & \texttt{case [] of [] => [] | ...} & \textsc{step}, \texttt{[]}\ \textsc{value} \\ \cong & \texttt{[]} & \textsc{step} \end{array}$$

The result follows by transitivity of $\cong$.

Case: `l` = `x :: xs`.

WTS: `concat (x :: xs)` $\cong$ `concatap (x :: xs)`.

IH1: `concat xs` $\cong$ `concatap xs`.

By induction on `x` : α `list`.

Subcase: `x` = [].

WTS: `concat ([] :: xs)` $\cong$ `concatap ([] :: xs)`

$$\begin{array}{lll} & \texttt{concat ([] :: xs)} & \\ \cong & \texttt{case ([] :: xs) of ... | [] :: tls => concat tls} & \textsc{step}, \texttt{[] :: xs}\ \textsc{value} \\ \cong & \texttt{concat xs} & \textsc{step} \\ \cong & \texttt{concatap xs} & \text{IH1} \end{array}$$

$$\begin{array}{lll} & \texttt{concatap ([] :: xs)} & \\ \cong & \texttt{case ([] :: xs) of ... | x :: xs => ap (x, concatap xs)} & \textsc{step}, \texttt{[] :: xs}\ \textsc{value} \\ \cong & \texttt{ap ([], concatap xs)} & \textsc{step} \\ \cong & \texttt{concatap xs} & \textsc{Lemma}\ 2 \end{array}$$

The result follows by transitivity of $\cong$.

Subcase: $\texttt{x} = \texttt{y} :: \texttt{ys}$.

WTS: $\texttt{concat}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs}) \cong \texttt{concatap}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs})$

IH2: $\texttt{concat}\ (\texttt{ys} :: \texttt{xs}) \cong \texttt{concatap}\ (\texttt{ys} :: \texttt{xs})$

$$
\begin{array}{lll}
 & \texttt{concat}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs}) & \\
\cong & \begin{array}{l}\texttt{case}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs})\ \texttt{of}\ \ldots \\ \mid (\texttt{y} :: \texttt{ys}) :: \texttt{xs} \Rightarrow \texttt{y} :: \texttt{concat}\ (\texttt{ys} :: \texttt{xs}) \\ \mid\ \ldots\end{array} & \textsc{step},\ (\texttt{y} :: \texttt{ys}) :: \texttt{xs}\ \textsc{value} \\
\cong & \texttt{y} :: \texttt{concat}\ (\texttt{ys} :: \texttt{xs}) & \textsc{step} \\
\cong & \texttt{y} :: \texttt{concatap}\ (\texttt{ys} :: \texttt{xs}) & \text{IH2}
\end{array}
$$

Note that both $\texttt{ap}, \texttt{concatap}$ are total because they match all possible constructors of their inputs and make recursive calls with strictly structurally smaller arguments.

$$
\begin{array}{lll}
 & \texttt{concatap}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs}) & \\
\cong & \begin{array}{l}\texttt{case}\ ((\texttt{y} :: \texttt{ys}) :: \texttt{xs})\ \texttt{of}\ \ldots \\ \mid \texttt{z} :: \texttt{zs} \Rightarrow \texttt{ap}\ (\texttt{z}, \texttt{concatap}\ \texttt{zs})\end{array} & \textsc{step},\ (\texttt{y} :: \texttt{ys}) :: \texttt{xs}\ \textsc{value} \\
\cong & \texttt{ap}\ ((\texttt{y} :: \texttt{ys}), \texttt{concatap}\ \texttt{xs}) & \textsc{step} \\
\cong & \begin{array}{l}\texttt{case}\ (\texttt{y} :: \texttt{ys})\ \texttt{of}\ \ldots \\ \mid \texttt{y} :: \texttt{ys} \Rightarrow \texttt{y} :: (\texttt{ap}\ (\texttt{ys}, \texttt{concatap}\ \texttt{xs}))\end{array} & \textsc{step}, \texttt{concatap}\ \textsc{total} \\
\cong & \texttt{y} :: (\texttt{ap}\ (\texttt{ys}, \texttt{concatap}\ \texttt{xs})) & \textsc{step} \\
\cong & \texttt{y} :: \texttt{concatap}\ (\texttt{ys} :: \texttt{xs}) & \texttt{y}\ \textsc{value}, \textsc{Lemma}\ 1
\end{array}
$$

The result follows by transitivity of $\cong$.

$\square$

2.2 Justifications

In Lecture 4, we had a function `raiseBy : int list * int -> int list` that added its `int` argument to each element of the `int list`. We proved that for all values `l : int list`, `a: int`, `b: int`,

$$\texttt{raiseBy}(\texttt{raiseBy}(\texttt{l},\texttt{a}),\texttt{b}) \cong \texttt{raiseBy}(\texttt{l},\texttt{a}+\texttt{b})$$

As it turns out, this is a special case of a property called *map fusion*. Mapping `f` over some list `l` and then mapping another function `g` over the result gives a list that is equivalent to the one you would get if you map `g o f` ("g composed with f") over the original `l`. In this task, you will prove that `map` has this *map fusion* property: `map (g o f)` $\cong$ `(map g) o (map f)`.

Task 2.4 (15%). Fill in the blanks in the following proof. The goal is for you to practice careful justification of statements; **pay particular attention to valuability and totality**. Most lines will be fairly short (with answers similar to 'by the IH', or 'step, because e valuable', and so on), save for the line marked with *** which requires slightly more explanation. Your handin should be a list

1. justification for (1)
2. justification for (2)

   ⋮

18 justification for (18)

Assume the following lemma:

Lemma 3. *For all types `a`, `b` and values `f : a->b`, if `f` is total then `map f` is valuable and total.*

Theorem 2. *For all types `a`, `b`, `c`, all values `f : a -> b` and `g : b -> c`, if `f` and `g` are total, then*

$$\texttt{(map g) o (map f)} \cong \texttt{map (g o f) : a list -> c list}$$

Proof. Assume values `f`, `g` such that `f` and `g` are total.

First, observe that various functions are valuable and total; use these facts below.

`map f is valuable and total`	(1) by the above lemma; `f` is total by assumption
`map g is valuable and total`	(2) by the above lemma; `g` is total by assumption
`g o f is valuable`	(3) it steps to a value in one step
`g o f is total`	(4) it equals `fn x => g (f x)`, and totality of `f` and `g`
`map (g o f) is valuable and total`	(5) by the above lemma and (4)

Next, by definition of equality for functions, it suffices to show that

for all values `l : a list`, `(map g o map f) l` ≅ `map (g o f) l : c list`

Observe that:

```
   (map g o map f) l
≅  (fn x => map g (map f x)) l       (6) step, map g, map f valuable
≅  map g (map f l)                   (7) step, l value
```

It therefore suffices to show that

`(map g (map f l))` ≅ `map (g o f) l`

Proceed by induction on the structure of `l`.

Case for `[]`:

WTS: (8) `(map g (map f []))` ≅ `map (g o f) []`

```
    (map g (map f []))
≅   (map g [])                       (9) step, [] value
≅   []                               (10) step, [] value
≅   (map (g o f) [])                 (11) rstep, [], value, g o f valuable
```

Case for `x::xs`, where `x` and `xs` are values:

IH: (12) `(map g (map f xs))` ≅ `map (g o f) xs`

WTS: (13) `(map g (map f (x :: xs)))` ≅ `map (g o f) (x :: xs)`

```
    (map g (map f (x::xs)))
≅   (map g (f x :: map f xs)        (14) step, f, x::xs values
≅   g (f x) :: map g (map f xs)     (15) ***
≅   (g o f) x :: map g (map f xs)   (16) rstep, x value, f value, g value
≅   (g o f) x :: map (g o f) xs     (17) by the IH
≅   map (g o f) (x::xs)             (18) rstep, x::xs value; g o f valuable
```

The line denoted *** was able to step because:

- `f x` is valuable because `f` is total and `x` is a value
- `map f xs` is valuable because `map f` is total (Lemma 3) and `xs` is a value
- `g` is a value

□

3 Polymorphism, Higher-order functions, Options

Last week, we introduced several new language features: polymorphism, option types, and higher-order functions. In this problem, you will write some simple functions using these new tools. It is very likely that these functions will be helpful later on in the assignment.

Task 3.1 (7%). Write the function

```
allpairs : 'a list * 'b list -> ('a * 'b) list list
```

`allpairs(l1,l2)` returns all of the possible pairings of elements from `l1` with elements from `l2`, in their original order. For example,

```
allpairs([1,2,3],["a","b","c"]) == [[(1,"a"),(1,"b"),(1,"c")],
                                    [(2,"a"),(2,"b"),(2,"c")],
                                    [(3,"a"),(3,"b"),(3,"c")]]
allpairs([1,2,3],[]) == [[],[],[]]
allpairs([],[1,2,3]) == []
```

You may not define this function recursively.

Task 3.2 (15%). Write the function

```
transpose : 'a list list -> 'a list list
```

that interchanges the rows and columns of a list of lists. For example,

```
transpose [[1,2]] ==> [[1],
                        [2]]


transpose [[1,2], ==> [[1,3],
           [3,4]]      [2,4]]


transpose [[1,2],
           [3,4], ==> [[1,3,5],
           [5,6]]      [2,4,6]]
```

More formally, we will say that a value `m : 'a list list` is a *valid* `'a`*-matrix* iff the following conditions hold:

1. `length m` > 0
2. There exists some `n` > 0 such that for every element `l` of `m`, `length l` $\cong$ `n`

That is, `m` is non-empty, each element of `m` is non-empty, and `m` is rectangular.

Transpose must meet the following spec:

> If `m : 'a list list` is a valid `'a`-matrix, then `transpose m` evaluates to a list of length n, whose i^{th} element is the list of elements of elements of `m` that occur at position i of some inner list, in their original order.

Your implementation of `transpose` may be recursive, but will also need to use higher order functions.

Task 3.3 (8%). Write a function

```
fun extract (p : 'a -> bool, l : 'a list) : ('a * 'a list) option =
```

such that

1. If there is some element `x` of `l` for which `p x == true`, then `extract(p,l)` evaluates to `SOME(x,l')`, where `l'` is `l` without that particular `x` but unchanged otherwise.
2. If for every element `x` of `l`, `p x ==> false` then `extract(p,l)` evaluates to `NONE`.

If there is more than one element satisfying the predicate in a particular argument list, it is your choice which to return.

For example:

```
extract(oddP , [2,3,4]) == SOME (3,[2,4])
extract(oddP , [2,4,6]) == NONE
extract(fn x => String.size x < 2 , ["aaa","b","bca"])
                        == SOME ("b", ["aaa", "bca"])
```

`extract` should be recursive. You should use this function when you implement Blocks World below.

4 Polynomials

In lecture on Thursday, we will discuss that we can represent a polynomial $c_0+c_1x+c_2x^2+\dots$ as a function that maps a natural number, i, to the coefficient c_i of x^i: For these tasks we will take the coefficients to be rational numbers (of type `rat`) (rather than integers, as in lecture). Therefore, we have the following type definition for polynomials:

```
type poly = int -> rat
```

As an example of how to define operations on polynomials defined in this manner, recall the following definition of addition of polynomials from lecture:

```
fun plus (p1 : poly, p2 : poly) : poly = fn e => p1 e ++ p2 e
```

4.1 Differentiation

Recall from calculus that differentiation of polynomials is defined as follows:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=0}^{n} c_i x^i = \sum_{i=1}^{n} i c_i x^{i-1}$$

Task 4.1 (5%). Define the function

```
differentiate : poly -> poly
```

that computes the derivative of a polynomial represented in normal form. Note that `differentiate` should *not* be recursive.

4.2 Integration

Recall from calculus that integration of polynomials is defined as follows:

$$\int \sum_{i=0}^{n} c_i x^i \,\mathrm{d}x = C + \sum_{i=0}^{n} \frac{c_i}{i+1} x^{i+1}$$

where C is an arbitrary constant known as the constant of integration. As C can be any number, the result of integration is a family of polynomials, one for each choice of C. Therefore, we will represent the result of integration as a function of type `rat -> poly`.

Task 4.2 (5%). Define the function

```
integrate : poly -> (rat -> poly)
```

that computes the family of polynomials corresponding to the integral of the argument polynomial. Note that `integrate` should *not* be recursive.

5 Matrices

5.1 Representation

Lists of lists provide a simple representation of matrices in SML: each list in the nested list is a row of the matrix. If we want to consider matrices of rational numbers, we can use the `rat` type discussed above and define

```
type matrix = rat list list
```

For example, we encode the matrix

$$\begin{bmatrix} 10 & 0 & -3 \\ 8 & 1 & 3 \end{bmatrix}$$

in SML with the list of lists of `rat`s

```
[[10//1, 0//1, ~3//1],[8//1, 1//1, 3//1]]
```

This representation makes it very natural to exploit some of the higher order functions defined in the SML Basis library to implement familiar matrix operations in an elegant and hands-off way.

Before doing this, we need to define some terminology to describe matrices:

- We will say that a particular value `m` of type `rat list list` is a *valid matrix* if and only if the following conditions hold:
    1. `length m` > 0
    2. There exists some `n` > 0 such that for every element `l` of `m`, `length l` $\cong$ `n`
- The *height* of a valid matrix `m` is `length m`—which is to say the number of rows.
- If `m` is a valid matrix and `r` is any element of `m`, the *width* of `m` is `length r`—which is to say the number of columns.
- If a matrix has height `h` and width `w`, then we say it has dimensions `h` $\times$ `w`, and that it is an `h` $\times$ `w` matrix.

5.1.1 Assumption on Input

For all the questions in the section, you may assume that your inputs of type `matrix` are valid.

5.2 Starter Code

In addition to the higher order list functions, we have given you a few functions that will be useful as you complete this task. You should look at how we implemented these functions to get some ideas about how to implement your functions in the coming sections.

- `toString : matrix -> string` prints a decent representation of a matrix.
- `zed : int * int -> matrix` returns a matrix with every entry equal to zero. The first argument is the number of rows, the second argument is the number of columns.
- `width : matrix -> int` returns the number of columns in a matrix.
- `height : matrix -> int` returns the number of rows in a matrix.
- `summat : matrix list -> matrix`

  If `ms` is a list of matrices that all have dimension $n \times m$ with $n > 0$ and $m > 0$, then `summat(ms)` returns a matrix that is the sum of all the matrices in `ms`. This function is defined using your implementation of `plus`, so it won't work until you've completed that task.
- `mateq : matrix * matrix -> bool` evaluates to true if and only if both argument matrices are equal. Note that you should *only* use this in writing tests for your code.

5.3 Matrix Addition

Task 5.1 (7%). Write the function

```
plus : matrix * matrix -> matrix
```

If `A` and `B` are valid matrices with the same dimensions, `plus(A,B)` performs addition on the matrices A and B element-wise. For example,

$$\begin{bmatrix}1 & 3\\1 & 0\\1 & 2\end{bmatrix} + \begin{bmatrix}0 & 0\\7 & 5\\2 & 1\end{bmatrix} = \begin{bmatrix}1+0 & 3+0\\1+7 & 0+5\\1+2 & 2+1\end{bmatrix} = \begin{bmatrix}1 & 3\\8 & 5\\3 & 3\end{bmatrix}$$

`plus` should not be a recursive function.

5.4 Matrix Multiplication

5.4.1 Familiar Definition

We will now implement matrix multiplication through a short series of helper functions.[4] Recall that if A is an $(n \times x)$ matrix and B is an $(x \times m)$ matrix, their product (AB) is an

[4]The details in this section are adapted from several sources. Primarily, we used the fourth edition of Gilbert Strang's excellent book "Linear Algebra and Its Applications". Some of the diagrams and LaTeXwere obtained from http://en.wikipedia.org/wiki/Matrix_multiplication and http://mathworld.wolfram.com/MatrixMultiplication.html

$(n \times m)$ matrix whose entries are

$$(AB)_{i,j} = \sum_{k=1}^{x} A_{i,k} B_{k,j}$$

For example,

$$\begin{bmatrix} 8 & 9 & 9 \\ 5 & -1 & 20 \end{bmatrix} \begin{bmatrix} 10 & 2 \\ 40 & 8 \\ 9 & 0 \end{bmatrix} = \begin{bmatrix} 521 & 88 \\ 190 & 2 \end{bmatrix}$$

This sum can be represented visually, as shown in Figure 1.

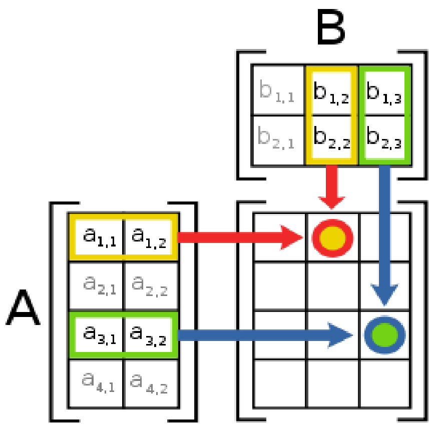

Figure 1: Matrix Multiplication

5.4.2 Alternative Statement

This familiar definition of multiplication is correct, but does not mesh well with our representation of matrices: we don't have constant time access to the individual elements in a matrix, so implementing this directly would be very wasteful. Instead, we'll use an equivalent definition of multiplication that's more suited to our list-of-rows representation.

Before we can give the alternative statement, we need to define a few new terms.

Def. If M is a matrix, the *transpose* of M is a matrix M^T where the columns of M^T are the rows of M. For example

$$\begin{bmatrix} 1 & 2 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Def. A *column vector* is a matrix with dimension $(n \times 1)$ for some n, as in

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

Def. If V and E are column vectors, the *outer product*, written $V \otimes E$, is the matrix product VE^T. Since V and E are column vectors, this has the much simpler form

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} \begin{bmatrix} e_1 & e_2 & \cdots & e_n \end{bmatrix} = \begin{bmatrix} v_1e_1 & v_1e_2 & \cdots & v_1e_n \\ v_2e_1 & v_2e_2 & \cdots & v_2e_n \\ \vdots & \vdots & \ddots & \vdots \\ v_me_1 & v_me_2 & \cdots & v_me_n \end{bmatrix}$$

where v_i and e_i denote the i^{th} elements of V and E respectively.

Def. If M is any matrix, we denote its i^{th} row as M_{ri}.

Def. If M is any matrix, we denote its i^{th} column as M_{ci}.

Def. If M is any matrix, we denote the column vector of all its rows M_r. The i^{th} element of M_r is M_{ri}. For example, if we have some matrix M with r-many rows and c-many columns,

$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,c} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,c} \\ \vdots & \vdots & \ddots & \vdots \\ m_{r,1} & m_{r,2} & \cdots & m_{r,c} \end{bmatrix}$$

then

$$M_r = \begin{bmatrix} \color{red}{M_{r1}} \\ \color{green}{M_{r2}} \\ \vdots \\ \color{blue}{M_{rr}} \end{bmatrix} = \begin{bmatrix} [\color{red}{m_{1,1}} & \color{red}{m_{1,2}} & \cdots & \color{red}{m_{1,c}}] \\ [\color{green}{m_{2,1}} & \color{green}{m_{2,2}} & \cdots & \color{green}{m_{2,c}}] \\ \vdots & \vdots & \ddots & \vdots \\ [\color{blue}{m_{r,1}} & \color{blue}{m_{r,2}} & \cdots & \color{blue}{m_{r,c}}] \end{bmatrix}$$

Def. If M is any matrix, we denote the column vector of all its columns M_c. The i^{th} element of M_c is M_{ci}. For example, if we have some matrix M with r-many rows and c-many columns,

$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,c} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,c} \\ \vdots & \vdots & \ddots & \vdots \\ m_{r,1} & m_{r,2} & \cdots & m_{r,c} \end{bmatrix}$$

then

$$M_c = \begin{bmatrix} M_{c1} \\ M_{c2} \\ \vdots \\ M_{cc} \end{bmatrix} = \begin{bmatrix} [m_{1,1} & m_{2,1} & \cdots & m_{r,1}] \\ [m_{1,2} & m_{2,2} & \cdots & m_{r,2}] \\ \vdots & \vdots & \ddots & \vdots \\ [m_{1,c} & m_{2,c} & \cdots & m_{r,c}] \end{bmatrix} = \left(M^T\right)_r$$

Using this notation, if A is a $(n \times x)$ matrix and B is a $(x \times m)$ matrix, the product (AB) is[5]

$$\begin{aligned} (AB) &= (A_c)^T B_r \\ &= \begin{bmatrix} A_{c1} & A_{c2} & \cdots & A_{cn} \end{bmatrix} \begin{bmatrix} B_{r1} \\ B_{r2} \\ \vdots \\ B_{rn} \end{bmatrix} \\ &= A_{c1} \otimes B_{r1} + A_{c2} \otimes B_{r2} + \cdots + A_{cn} \otimes B_{rn} \\ &= \sum_{i=1}^{n} A_{ci} \otimes B_{ri} \end{aligned}$$

Task 5.2 (7%). Write the function

```
val outerprod : rat list * rat list -> matrix
```

For `V1,V2 : rat list` representing vectors, `outerprod(V1, V2)` computes the outer product of `V1` and `V2`. `outerprod` should not be a recursive function.

Task 5.3 (7%). Write the function

```
times : matrix * matrix -> matrix
```

If `A` is a valid matrix with dimension $n \times x$ and `B` is a valid matrix with dimension $x \times m$, `times(A,B)` computes the product AB using the outer product definition. `times` should not be a recursive function.

Hint: If you make appropriate use of higher-order functions, your solutions to the tasks in this problem will be quite short and elegant. For example, our solutions to `plus`, `outerprod`, and `times` consist of 1 line of code each. Correct but more-verbose solutions will receive credit, but if you find yourself writing lines and lines of code, you should stop and reconsider.

[5] If you've taken Matrix or Linear algebra, this should be eerily familiar. The definition is an inner product where the elements in the column vectors are vectors, so instead of multiplying them, we take their outer product. It should be relatively easy to see why this is an equivalent definition to the familiar one. It happens to be defined over the vector space of vectors, rather than the vector space of rationals, which is why we use matrix addition and matrix outer product rather than rational addition and rational multiplication.

6 Blocks World

In artificial intelligence, *planning* is the task of figuring what an agent (a robot, that paperclip in Microsoft Word, your roommate, etc.) should do. One way to solve planning problems is to simulate the circumstances of the agent, so that you can simulate plans, and then search through potential plans for good ones.

A simple planning problem, which is often used to illustrate this idea, is *blocks world*. The idea is that there are a bunch of blocks on a table:

```
---   ---   ---
|A|   |B|   |C|
---   ---   ---
-------------------------
```

and a robotic hand. You can pick one block up with the hand:

```
/|\
---
|C|
---
      ---   ---
      |A|   |B|
      ---   ---
-------------------------
```

and place it back on the table or on another block:

```
---
|C|
---
---   ---
|A|   |B|
---   ---
-------------------------
```

Of course, you can't put a block on one that already has something on it, so in the next two moves we can't pick up `B` and then put it on `A`. A planning problem would be something like "starting with the blocks on the table, make the tower `BCA`".

In this problem, you will represent blocks world in ML, so that you can simulate plans (we won't ask you to search for plans that achieve specific goals).

At the end of the problem, you'll be able to interact with Blocks World as in Figure 2. We've written all the input/output code for you, so you just need to do the interesting bits.

```
- playBlocks ();

Possible moves:
  pickup <block> from table
  put <block> on table
  pickup <block> from <block>
  put <block> on <block>
  quit

---   ---   ---
|A|   |B|   |C|
---   ---   ---
-------------------------
Next move: pickup C from table

/|\
---
|C|
---
      ---   ---
      |A|   |B|
      ---   ---
-------------------------
Next move: put C on A

---
|C|
---
---   ---
|A|   |B|
---   ---
-------------------------
```

Figure 2: Sample Blocks World Interaction

6.1 Ontology

We will model Blocks World as follows:

- There are three blocks, A, B, C.
- We will represent the state of the world as a list of facts. There are five kinds of facts:
  - Block b is free (available to be picked up)
  - Block a is on block b
  - Block a is on the table
  - The hand is empty
  - The hand is holding block b
- At each step, there are four possible moves:

  ```
  pickup <b> from table
  put <b> on table
  pickup <a> from <b>
  put <a> on <b>
  ```

  These moves act as follows:

  - `pickup <a> from table`
    Before: a is free, and a is on the table, and the hand is empty.
    After: the hand holds a.
  - `put <b> on table`
    Before: the hand holds a.
    After: the hand is empty, and a is on the table, and a is free.
  - `pickup <a> from <b>`
    Before: a is free, and a is on b, and the hand is empty.
    After: b is free, and the hand is holding a.
  - `put <a> on <b>`
    Before: the hand holds a, and and b free.
    After: a is free, the hand is empty, and a is on b.

  In these descriptions, the "before" facts must hold about the world for the move to be executed; after executing the move, the "before" facts no longer hold (e.g. after picking up a block, the hand is no longer empty), and the "after" facts holds.

6.2 Tasks

Task 6.1 (5%). First, we will need a function to extract many elements from a list. Write a function

```
    extractMany : ('a * 'a -> bool * 'a list * 'a list) -> ('a list) option
```

`extractMany` is polymorphic in the list's element type, but it needs to test whether two list elements are equal. For this reason, `extractMany` takes an argument function `eq:'a * 'a -> bool` that can be used to test whether two values of type `'a` are equal.

`extractMany (eq,toExtract,from)` "subtracts" the elements of `toExtract` from `from`, checking that all the elements of `toExtract` are present in `from`. More formally, if `toExtract` is a sub-multi-set (according to the definition given in the subset-sum problem on HW 3, but using `eq` to determine when an element "appears") of `from`, then `extractMany(eq,toExtract,from)` returns `SOME xs`, where `xs` is `from` with every element of `toExtract` removed. If `toExtract` is not a sub-multi-set of `from`, then `extractMany(eq,toExtract,from)` returns `NONE`.

This means that the number of times an element occurs matters, but order does not:

```
extractMany(inteq, [2,1,2], [1,2,3,3,2,4,2]) == SOME [3,3,4,2]
extractMany(inteq, [2,2], [2]) == NONE
```

You may define this recursively, and should use `extract`.

Task 6.2 (8%). Define datatypes representing blocks, moves, and facts, according to the above ontology:

```
  datatype block = ...
  datatype move = ...
  datatype fact = ...
```

Observe the convention that datatype constructors start with an upper-case letter (e.g. `Node` and `Empty`).

Task 6.3 (2%). Define a `state` of the world to be a list of facts:

```
type state = fact list
```

Fill in

```
  val initial : state = ...
```

to represent the following state: the hand is empty, each of A,B,C is on the table, and each of A,B,C is free.

Task 6.4 (3%). Define a short helper function

```
  consumeAndAdd : (state * fact list * fact list) -> state option
```

`consumeAndAdd(s,before,after)` subtracts `before` from `s` and adds `after` to the result, checking that every fact in `before` occurs. More formally, if `before` is a sub-multi-set of `s`, then `consumeAndAdd(s, before, after)` returns `SOME s'`, where `s'` is `s` with `before` removed and `after` added. If `before` is not a sub-multi-set, `consumeAndAdd(s, before, after)` returns `NONE`.

You will need to use the provided function `extractManyFacts`, which instantiates your `extractMany` with an equality operation derived from the `fact` datatype.

`consumeAndAdd` should not be recursive.

Task 6.5 (6%). Implement a function

```
  step : (move * state) -> state option
```

If the "before" facts of `m` hold in `s`, then `step(m,s)` must return `SOME s'`, where `s'` is the collection of facts resulting from performing the move `m`. It should return `NONE` if the move cannot be applied in that state. This function should not be recursive.

Optional Task: In the file `blocks_world.sml`, fill in your datatype constructors at the spots indicated. You will then be able to play Blocks World interactively as follows:

```
- use "hw05.sml";
- use "blocks_world.sml";
- playBlocks();
```

Note that the support-code uses your `transpose` function from Section 3, so if the output seems wonky, you should check your `transpose` implementation for bugs. This task is optional; do not hand in `blocks_word.sml`.