15-150 Fall 2014
Homework 06

Out: Tuesday 30th September, 2014
Due: Tuesday 21st October, 2014 at 23:59 AST

1 Introduction

This assignment will get you acquainted with SML's module system and representation independence.

1.1 Getting the Homework Assignment

The starter files for the homework assignment have been distributed through Autolab at

`https://autolab.cs.cmu.edu`

Select the page for the course, click on "hw06", and then "Download handout". Uncompressing the downloaded file will create the directory `hw06-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/06` (see below): copy the starter files there.[1]

1.2 Submitting the Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

In preparation for submission, your `hw/06` directory will need to contain the file `hw06.pdf` with your written solutions and the files `util.sml`, `listvec.sml`, `sparsevec.sml`, `funvec.sml`, `matrix.sml` and `test-all.sml` with your code. The starter code for this assignment contains incomplete versions of these files. You will need to complete them with the solution to the various programming tasks in the homework.

To submit your solutions, run

[1] The download and working directory have different names so that you don't accidentally overwrite your work were the starter files to be updated. **Do not do your work in the download directory!**

© Carnegie Mellon University 2014

```
make
```

from the `hw/06` directory on any Unix machine (this include Linux and Mac). This will produce a file `hw06-handin.tgz`, containing the files to be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw06-handin.tgz` file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the `hw/06` directory. That will create `hw06-handin.tgz`, containing the files to be handed in for this homework assignment and directly submit this to Autolab.

All submission will go through Autolab's "autograder". The autograder simply runs a series of tests against the reference solution. Each module has an associated number of points equal to the number of functions you need to complete. For each such function, you get 1.0 points if your code passes all tests, and 0.0 if it fails at least one test. Click on the cumulative number for a module for details. Obtaining the maximum for a module does not guarantee full credit in a task, and neither does a 0.0 translate into no points for it. In fact, the course staff will be running additional tests, reading all code, and taking into account other aspects of the submitted code such as structured comments and tests (see below), style and elegance.

To promote good programming habits, your are limited to a maximum of 10 submissions for this homework. Use them judiciously! In particular, make sure your code compiles cleanly before submitting it. Also, make sure your own test suite is sufficiently broad.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

The SML files `util.sml`, `listvec.sml`, `sparsevec.sml`, `funvec.sml`, `matrix.sml` and `test-all.sml` must contain all the code that you want to have graded for this assignment, and must 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 Due Date

This assignment is due on Tuesday 21st October, 2014 at 23:59 AST. Remember that there are no late days for this course and you will get 2 bonus points for every 12 hours that you submit early.

Because of the Eid break, bonus points will be calculated a bit differently than usual. Briefly, it will be as if the week of October 3rd to 9th did not exist. Specifically:

- Submissions made between midnight (0:00) Tuesday 30th September, 2014 and Thursday 2nd October, 2014 at 23:59 AST will yield the normal number of bonus periods minus 14 (i.e., minus 7 days).

- Submissions made between midnight (0:00) Friday 3rd October, 2014 and Thursday 9th October, 2014 at 23:59 AST will earn 24 bonus periods (12 days).
- Bonus periods for submissions made between midnight (0:00) Friday 10th October, 2014 and the deadline (Tuesday 21st October, 2014 at 23:59 AST) will be computed as usual.

This is meant to encourage you to relax during the break. Said this, feel free to complete the homework then if that works for you.

1.4 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

You must use the following five step methodology for writing functions, for *every* function you write in this assignment:

1. In the first line of comments, write a call template of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function)
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  factorial (n) ==> res
 *  REQUIRES:  n >= 0
 *  ENSURES: res is  n!
 *)
fun factorial (0: int): int = 1
  | factorial n = n * factorial (n-1)

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

1.5 Testing Modules

Because modules encourage information hiding, the way to test SML structures and functors is a bit different from what you did in the past. In fact, outside of a module, you may have no way to view the values of an abstract type. This means you can't compare the result of an operation with the expected value because you have no way to construct this expected value.

So, how to test modular code? There are essentially two ways to proceed.

Inside-the-box testing: You can't build values outside your module, but you can do so inside (typically). Then, what you would do is to put your normal tests inside the structure you are working on. As usual, if a test fails, a `binding non exhaustive` exception will be raised.

This is a bit trickier to do with functors, because you may not have a way to build values that depend on the functor's parameters. In this case, outside-the-box testing is your only option.

Outside-the-box testing: Many modules export a printing function and an equality function (conventionally called `toString`) and `eq`, respectively). You can then use the equality function to test that the value returned by a function is the value you expect. You can use the printing function to visualize returned values of hidden type.

When a module does not provide such functions, it exports operations that interact with each other, somehow. You can leverage these interactions for testing purposes. For example, a dictionary exports `insert` and `lookup` operations. You may test a module implementing dictionaries by populating a dictionary using `insert` and then use `lookup` to check that the expected entries are in it (and that unexpected entries are not).

Best of all, you want to use a combination of inside- and outside-the-box testing. Notice that inside-the-box testing is implementation-dependent, but outside-the-box is not.

1.6 Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

1.7 The Compilation Manager

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, we will use SML's *compilation manager*. The compilation manager (CM) is a system that keeps track of what files have been modified and runs just them (and the files that depend on them) through SML. If you have used `make` on a Unix system, the idea is very similar.

Using CM is simple. In fact, there are two ways to do so:

Go to the directory containing your work and run at the terminal prompt (written `#`):

```
# sml -m sources.cm
```

or

Launch SML from the directory containing your work, and then run at the SML prompt (written `-`):

```
- CM.make "sources.cm";
```

Both wil load all the files listed in `sources.cm`[2] and take you to the SML prompt. Do so whenever you change your code. No need to call `use` — in fact you may confuse CM. For large programs, CM offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code.

In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. Type either `sml -m sources.cm` at the terminal prompt or `CM.make "sources.cm";` at the SML prompt.
3. Fix errors and repeat.

`CM.make` creates a directory called `.cm` in the current working directory. It gets populated with metadata needed to work out compilation dependencies. The `.cm` directory can safely be deleted at the completion of this assignment (in fact, it can become quite large)

It's sometimes happens that the metadata in the `.cm` directory gets into an inconsistent state — if you run `CM.make` with different versions of SML in the same directory, for example. This often results in bizarre error messages. When that happens, it is safe to delete the `.cm` directory and compile again from scratch.

[2]The file `sources.cm` contains a list of the files tracked by CM. Feel free to take a peek if you are curious!

2 Warmup

Your code for the following functions should go in the structure called `Util` in file `util.sml`.

Task 2.1 (5 points) Write the function

```
chop: ('a -> bool) -> 'a list -> 'a list
```

such that the call `chop p l` returns a list identical to `l` except that all elements which satisfy `p` have been removed from the end of the list, stopping with the first element from the end that does not satisfy the predicate. Here are some examples:

```
chop (fn x => x mod 2 = 0) [1,2,3,4,5,6,8,10] ≅ [1,2,3,4,5]
chop (fn [] => true | _ => false) [[],[1],[1,2],[],[]] ≅ [[],[1],[1,2]]
chop (fn x => x > 7) [] ≅ []
```

Test your code in file `test-all.sml`.

Task 2.2 (5 points) Write the function

```
combine: ('a * 'a -> 'a) -> ('a list * 'a list) -> 'a list
```

such that the call `combine f (l1,l2)` takes a function `f` that combines pairs of the same type into a third value of that type and two lists `l1` and `l2`. It returns in output a list whose ith element is computed by by applying `f` to the ith elements of `l1` and `l2`. If the input lists do not have the same length, the unused suffix of the longer list appears unchanged as the suffix of the list of output.

More formally, let l_1, l_2 be lists with respective lengths n and m.

If $n = m$:

$$\texttt{combinef}([\texttt{x}_1, \ldots, \texttt{x}_n], [\texttt{y}_1, \ldots, \texttt{y}_n]) \;\cong\; [\texttt{f}(\texttt{x}_1, \texttt{y}_1), \ldots, \texttt{f}(\texttt{x}_n, \texttt{y}_n)]$$

If $n > m$:

$$\texttt{combinef}([\texttt{x}_1, \ldots, \texttt{x}_n], [\texttt{y}_1, \ldots, \texttt{y}_m]) \;\cong\; [\texttt{f}(\texttt{x}_1, \texttt{y}_1), \ldots, \texttt{f}(\texttt{x}_m, \texttt{y}_m), \texttt{x}_{m+1}, \ldots, \texttt{x}_n]$$

If $n < m$:

$$\texttt{combinef}([\texttt{x}_1, \ldots, \texttt{x}_n], [\texttt{y}_1, \ldots, \texttt{y}_m]) \;\cong\; [\texttt{f}(\texttt{x}_1, \texttt{y}_1), \ldots, \texttt{f}(\texttt{x}_n, \texttt{y}_n), \texttt{y}_{n+1}, \ldots, \texttt{y}_m]$$

For example,

```
combine op+ ([1,2,3,4], [400, 300, 200, 100]) ≅ [401, 302, 203, 104]
combine op+ ([1,2], [400, 300, 200, 100]) ≅ [401, 302, 200, 100]
combine op+ ([1,2,3,4], [400, 300]) ≅ [401, 302, 3, 4]
```

Test your code in file `test-all.sml`.

3 Infinite Vectors

In this exercise, we consider vectors that may contain an arbitrary number of elements. We examine a few operations on them. We give three representations for these vectors in SML and three implementations of their operations.

3.1 Mathematical Definition

An *infinite-dimensional vector of compact support* is an infinite tuple of numbers, often written in column format as

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \end{pmatrix}$$

such that only finitely many of the entries v_i are nonzero. Formally, the tuple above is an infinite-dimensional vector of compact support if there exists an integer N such that for all $i \geq N$,$v_i = 0$. For example,

$$\begin{pmatrix} 1 \\ 3 \\ 6 \\ 9 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix}$$

is an infinite-dimensional vector of compact support since only the first four numbers in the tuple are nonzero. In contrast, if we take $v_i = 2i$, i.e.

$$\begin{pmatrix} 2 \\ 4 \\ 6 \\ 8 \\ 10 \\ 12 \\ \vdots \end{pmatrix}$$

we do not get an infinite-dimensional vector of compact support since there are infinitely many nonzero entries — you can prove that $2i = 0$ *if and only if* $i = 0$.

© Carnegie Mellon University 2014

3.2 Abstract Specifications

To represent this mathematical object, we first need to represent elements and then represent vectors of such elements.

3.2.1 Elements

The elements that we will be interested in are of any type that supports addition and multiplication and has a zero element. This is represented immediately by the signature

```
signature ELT =
sig
  type t
  val zero:     t
  val p:        t * t -> t
  val m:        t * t -> t
  val z:        t -> bool
  val toString: t -> string
  val eq:       t * t -> bool
end
```

Here,

- `t` is a type that supports the operations
- `p` represents addition on the type
- `m` represents multiplication on the type
- `zero` is the additive identity for the type with respect to `p`
- `z` is a characteristic function that tells you if a particular value of type `t` is the same as `zero`.
- `toString` returns a string representation of an element of type `t`.
- `eq` tests whether two elements of type `t` are equal.

This signature is provided in file `elt.sig` in the starter code.

The structure `IntElt` in starter file `intelt.sml` ascribes to this signature. It uses `int` as the type `t`. Feel free to use it in your code, or create structures for other types of elements.

3.2.2 Vectors

Given this representation of elements, we define a signature that describes vectors of elements and interesting operations on them as follows (see also starter file `infvec.sig`):

```
signature INFVEC =
sig
  structure E : ELT
  type infvec

  val toString:  infvec -> string
  val valid:     infvec -> bool
  val eq:        infvec * infvec -> bool

  val toVec:     E.t list -> infvec
  val toList:    infvec -> E.t list

  val iszero:    infvec -> bool
  val flatat:    infvec -> int
  val component: infvec * int -> E.t

  val add:       infvec * infvec -> infvec
  val scale:     E.t * infvec -> infvec
  val convolve:  infvec * infvec -> infvec

  val filter:    (E.t -> bool) -> infvec -> infvec
end (* signature INFVEC *)
```

This signature centers around the abstract type `infvec` that represents vectors of a particular type of elements. The detailed specifications for each value in the signature are given next.

In the description below, the specification of most of the operations in `INFVEC` include a pictorial example — these examples use vector with integer elements for convenience but your implementations should be generic.

- `toString v` returns a string representation of vector `v`. It is pre-implemented for you as part of the starter code for this homework.
- `valid v` checks whether the vector `v` satisfies the representation invariants of the chosen implementation. It should always return `true` for a correct implementation, but it is a useful debugging tool to get there.
- `eq (v1,v2)` $\cong$ `true` if and only if the vectors `v1` and `v2` are equal.
- `toVec l` computes a vector whose initial non-zero prefix is the elements of `l` in their

original order. For example,

$$\texttt{toVec [0, 0, 5, 2, 3]} \cong \begin{pmatrix} 0 \\ 0 \\ 5 \\ 2 \\ 3 \\ 0 \\ 0 \\ \vdots \end{pmatrix}$$

- `toList v` returns a list of elements that constitute the non-zero prefix of `v`. For instance, if `v` is the vector created in the previous example, then `toList(v)` $\cong$ `[0, 0, 5, 2, 3]`. The list returned should have no trailing zeroes.
- `iszero v` $\cong$ `true` if and only if `v` consists entirely of zeroes.
- `flatat v` returns the smallest index N such that v_i is zero for all $i \geq N$. Following the mathematical tradition of the object we're representing, indices start at 1. For example,

$$\texttt{flatat} \begin{pmatrix} 0 \\ 2 \\ 6 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \cong 7$$

- If $i \geq 1$, `component (v, i)` $\cong v_i$. For example,

$$\texttt{component(} \begin{pmatrix} 0 \\ 2 \\ 6 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \texttt{, 3)} \cong 6 \qquad \texttt{component} \begin{pmatrix} 0 \\ 2 \\ 6 \\ 0 \\ 0 \\ 8 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \texttt{4} \cong \texttt{0}$$

- `add(v, w)` adds the vectors `v` and `w` component-wise. More formally,

$$\texttt{add(}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \end{pmatrix}\texttt{,}\begin{pmatrix} w_1 \\ w_2 \\ \vdots \end{pmatrix}\texttt{)} \cong \begin{pmatrix} v_1 + w_1 \\ v_2 + w_2 \\ \vdots \end{pmatrix}$$

- `scale(x, v)` computes the vector that results from multiplying each component of `v` by `x`. More formally,

$$\texttt{scale(x,}\begin{pmatrix} v_1 \\ v_2 \\ \vdots \end{pmatrix}\texttt{)} \cong \begin{pmatrix} \texttt{x} * v_1 \\ \texttt{x} * v_2 \\ \vdots \end{pmatrix}$$

- `convolve(v, w)` returns the vector convolution of the vectors `v` and `w`. Formally, the n^{th} component of `convolve(v, w)` is the sum

$$\sum_{i=1}^{n} v_i * w_{n-i+1}$$

  Pictorially,

$$\texttt{convolve(}\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ \vdots \end{pmatrix}\texttt{,}\begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ \vdots \end{pmatrix}\texttt{)} \cong \begin{pmatrix} v_1 * w_1 \\ v_1 * w_2 \ + \ v_2 * w_1 \\ v_1 * w_3 \ + \ v_2 * w_2 \ + \ v_3 * w_1 \\ v_1 * w_4 \ + \ v_2 * w_3 \ + \ v_3 * w_2 \ + \ v_4 * w_1 \\ \vdots \end{pmatrix}$$

  For example,

$$\texttt{convolve(}\begin{pmatrix} 2 \\ 7 \\ 8 \\ 0 \\ 0 \\ \vdots \end{pmatrix}\texttt{,}\begin{pmatrix} 5 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix}\texttt{)} \cong \begin{pmatrix} 10 \\ 37 \\ 47 \\ 8 \\ 0 \\ \vdots \end{pmatrix}$$

- `filter p v` evaluates to a vector with those elements of `v` that satisfy `p` in their original locations and all elements that don't satisfy `p` replaced by zero.

The functions `toString`, `valid` and `eq` will be particularly useful for testing your code.

3.3 Implementations

Your task will be to implement this signature with three different underlying concrete representations. Document each implementation by stating clearly the abstraction function and representation invariants (listed below) and by giving specifications for each function as usual.

In your implementations below, *do not abuse the types.* Do not, for instance, covert a vector to list form, perform the operations there, and then convert it back. Submissions of this nature will receive little or no credit. Each functor below should be independent from the others both in spirit and letter.

3.3.1 Full List Representation

In your first implementation you will represent infinite-dimensional vectors of compact support using lists — specifically,

```
type infvec = E.t list
```

This implementation is defined follows.

Abstraction Function: A value `[`v_1`, `v_2`, ..., `v_n`]: E.t list` represents the infinite-dimensional vector of compact support

$$\begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \\ \texttt{E.zero} \\ \texttt{E.zero} \\ \vdots \end{pmatrix}$$

Representation Invariants: A value `l: E.t list` is a valid representation of an infinite-dimensional vector of compact support only if the following condition is satisfied:

1. The last number in `l` is not zero.

Task 3.1 (20 points) Complete the implementation of the functor `ListVec` found in starter file `listvec.sml`. Your code should be fully documented and satisfy the above representation invariants. Test your code in file `test-all.sml`.

3.3.2 Sparse List Representation

Your next representation will also use lists to represent the component numbers of a vector, but you will *only* store *nonzero* numbers. This is commonly known as a *sparse representation*. It is useful when vectors have only a small-number of nonzero entries, as is often the case in physical systems and graph theory.

For example, we would represent the following infinite-dimensional vector of compact support

$$\begin{pmatrix} 7.1 \\ 0.0 \\ 0.0 \\ 3.8 \\ 0.0 \\ 6.7 \\ 9.4 \\ 0.0 \\ 0.0 \\ \vdots \end{pmatrix}$$

using the list

```
[(1, 7.1), (4, 3.8), (6, 6.7), (7, 9.4)].
```

This idea is codified in making the following choice for the abstract type in the signature:

```
type infvec = (int * E.t) list
```

Here the indices become explicit in the representation, where they were implicit before, so that we can determine how many zero-valued entries are between each non-zero entry.

This implementation is defined as follows.

Abstraction Function: A value `l: (int * E.t) list` represents the infinite-dimensional vector of compact support

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \end{pmatrix}$$

where v_i is given by `(`i`,` v_i`)` if it appears in the list `l` and is zero otherwise.

Representation Invariants: A value `[(`i_1`,` v_{i_1}`), (`i_2`,` v_{i_2}`), ..., (`i_n`,` v_{i_n}`)]: (int * E.t) list` is a valid representation of an infinite-dimensional vector of compact support only if the following conditions are satisfied:

1. None of the v_i are zero.

2. An index i appears at most once as the first component of any pair in the list.
3. The list is is sorted in increasing order of the indices, that is, $i_1 < i_2 < \ldots < i_n$.
4. All the indices are strictly positive integers.

Task 3.2 (20 points) Complete the implementation of the functor `SparseVec` found in starter file `sparsevec.sml`. Your code should be fully documented and satisfy the above representation invariants. Test your code in file `test-all.sml`.

3.3.3 Functional Representation

The third implementation of infinite-dimensional vectors of compact support uses a function from indices to items as its implementation. That is to say,

```
type infvec = int * (int -> E.t)
```

This representation is defined as follows.

Abstraction Function: The value `(n, f): infvec` represents the infinite-dimensional vector of compact support

$$\begin{pmatrix} \texttt{f 1} \\ \texttt{f 2} \\ \texttt{f 3} \\ \vdots \\ \texttt{f (n-1)} \\ \texttt{E.zero} \\ \texttt{E.zero} \\ \vdots \end{pmatrix}$$

Representation Invariants: The value `(n, f): infvec` is a valid representation of an infinite-dimensional vector of compact support only if the following three conditions are satisfied:

1. `n` ≥ 1.
2. For all $x \geq 1$, `f x` returns a value.
3. `f (n-1)` $\ncong$ `E.zero`.

The number `n` is therefore the first index where the vector consists of only zeros.

Task 3.3 (20 points) Complete the implementation of the functor `FunVec` found in starter file `funvec.sml`. Your code should be fully documented and satisfy the above representation invariants. Test your code in file `test-all.sml`.

3.4 Analysis

Task 3.4 (15 points) Analyze the work of each of your three implementations of `convolve`. Your analysis should reflect the structure of your particular implementation. In your analysis, you may assume that the atomic operations on elements take constant time on all input.

4 Infinite Matrices

Matrices are familiar two-dimensional tables. An *infinite-dimensional matrix of compact support* is a doubly-infinite table,

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

such that only finitely many of the a_{ij} are nonzero. That is to say, there exists an index N such that $a_{ij} = 0$ whenever $i \geq N$ or $j \geq N$. The matrix table is often expressed in shorthand form by the expression (a_{ij}). The j^{th} *column* of the matrix (a_{ij}) is the infinite-dimensional vector of compact support

$$a_j = \begin{pmatrix} a_{1j} \\ a_{2j} \\ a_{3j} \\ \vdots \end{pmatrix}$$

4.1 Mathematical Definition

The two matrix operations we will focus on are matrix addition and matrix multiplication, written mathematically as $A + B$ and AB. You will shortly implement a representation of matrices and these two operations. The code for matrix addition will be similar to the code you wrote for vector addition. The code for matrix multiplication will be slightly different. It will in fact be based on a different kind of product, namely the multiplication of a matrix A by a vector v, written mathematically as $A\,v$.

Specifically, if A is the matrix (a_{ij}), and v is an infinite-dimensional vector of compact support

$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \end{pmatrix}$$

then the product $A\,v$ is another vector, given by the following expression:

$$A\,v \;=\; v_1 \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \end{pmatrix} \;+\; v_2 \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \\ \vdots \end{pmatrix} \;+\; v_3 \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \\ \vdots \end{pmatrix} \;+\; \cdots$$

In other words, the product $A\,v$ is a vector computed by adding up the columns of A, scaled by the components of v.

© Carnegie Mellon University 2014

By way of example, suppose A is matrix containing rainfall data by city and day, and v is the vector consisting of the number 1 for its first four components and 0 elsewhere, then $A\,v$ is simply a vector that tells us the total rainfall in each city for the first four hours of measuring time:

$$
\begin{aligned}
A\,v &= \begin{pmatrix} 5 & 8 & 9 & 5 & 0 & \cdots \\ 0 & 0 & 1 & 1 & 0 & \cdots \\ 15 & 3 & 3 & 3 & 0 & \cdots \\ 0 & 3 & 13 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ \vdots \end{pmatrix} \\
&= 1 \begin{pmatrix} 5 \\ 0 \\ 15 \\ 0 \\ 0 \\ \vdots \end{pmatrix} + 1 \begin{pmatrix} 8 \\ 0 \\ 3 \\ 3 \\ 0 \\ \vdots \end{pmatrix} + 1 \begin{pmatrix} 9 \\ 1 \\ 3 \\ 13 \\ 0 \\ \vdots \end{pmatrix} + 1 \begin{pmatrix} 5 \\ 1 \\ 3 \\ 0 \\ 0 \\ \vdots \end{pmatrix} + 0 \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{pmatrix} + \cdots \\
&= \begin{pmatrix} 27 \\ 2 \\ 24 \\ 16 \\ 0 \\ \vdots \end{pmatrix}.
\end{aligned}
$$

So, for example, the total rainfall in the first city in the first four hours was 27 millimeters.

One approach for computing the product AB of two matrices A and B is to observe that the columns of AB are the matrix-vector products of A with the columns of B. Specifically, suppose we write $B = (b_1\ b_2\ \ldots)$, where b_j is an infinite-dimensional vector of compact support representing the jth column of B. Then $AB = (Ab_1\ Ab_2\ \ldots)$, that is, the jth column of AB is the vector Ab_j formed by multiplying A and b_j. For example,

$$
\begin{pmatrix} 1 & 3 & 2 & 0 & \cdots \\ 0 & 1 & 10 & 0 & \cdots \\ 7 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} 2 & 5 & 0 & \cdots \\ -1 & 1 & 0 & \cdots \\ 10 & 4 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} = \begin{pmatrix} 19 & 16 & 0 & \cdots \\ 99 & 41 & 0 & \cdots \\ 14 & 35 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}
$$

since

$$\begin{pmatrix} 1 & 3 & 2 & 0 & \cdots \\ 0 & 1 & 10 & 0 & \cdots \\ 7 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 10 \\ 0 \\ \vdots \end{pmatrix} = \begin{pmatrix} 19 \\ 99 \\ 14 \\ 0 \\ \vdots \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 3 & 2 & 0 & \cdots \\ 0 & 1 & 10 & 0 & \cdots \\ 7 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} 5 \\ 1 \\ 4 \\ 0 \\ \vdots \end{pmatrix} = \begin{pmatrix} 16 \\ 41 \\ 35 \\ 0 \\ \vdots \end{pmatrix}.$$

where the other columns are all zero because they are zero in B.

4.2 Abstraction

The following signature captures the abstraction of a matrix based on infinite-dimension vectors of compact support containing elements of a certain type

```
signature MATRIX =
sig
  structure E: ELT
  structure Vec: INFVEC where E = E
  type matrix

  val toString: matrix -> string
  val eq:       matrix * matrix -> bool

  val toMat:    E.t list list -> matrix
  val toLists:  matrix -> E.t list list

  val vProd:    matrix * Vec.infvec -> Vec.infvec
  val add:      matrix * matrix -> matrix
  val mult:     matrix * matrix -> matrix
end
```

- `matrix` is the abstract type used to represent infinite-dimensional matrices of compact support.
- `toString m` returns a string representation of matrix `m`.
- `eq (m1,m2)` $\cong$ `true` if and only if the matrices `m1` and `m2` are equal.
- `toMat` takes a list of lists of elements representing the columns of a matrix in left-to-right order and produces the corresponding matrix.
- `toLists` expects a matrix and returns a list of list of elements representing its columns in left-to-right order.

- `vProd` multiplies a matrix and a vector, as described on page 16.
- `add` adds two matrices component-wise. Formally,

$$\texttt{add}((a_{ij}), (b_{ij})) \;\cong\; s(a_{ij} + b_{ij}).$$

- `mult` multiplies two matrices (see pages 17–18). Formally,

$$\texttt{mult}((a_{ij}), (b_{ij})) \;\cong\; s(c_{ij}),$$

  where $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + \cdots$, for all $i, j \geq 1$

4.3 Representation

Task 4.1 (20 points) Implement a functor `Matrix` that takes a structure that ascribes to `INFVEC` and produces a structure that ascribes opaquely to `MATRIX`. Your implementation should be an infinite-dimensional vector of compact support of infinite-dimensional vectors of compact support.

Notice that, unlike the three functors above, we have not explicitly given you the concrete representation of the abstract type `matrix`, abstraction function, or representation invariants. Part of your job, therefore, is to provide those things. Here is what you should do:

- Within the structure `Matrix` define a concrete representation of `matrix`. It must be based on the idea stated above, but you need to figure out how to write it specifically.
- Clearly document your abstraction function.
- Clearly document your representation invariant.

Test your code in file `test-all.sml`.

Task 4.2 (10 bonus points) Analyze the work and span of your matrix multiplication.