# 15-150 Summer 2016
# Homework 05

Out: Sunday, 29 May 2016
Due: Wednesday, 1 June at 23:59 EST

## 1 Introduction

This homework will focus on applications of higher order functions, polymorphism, and user-defined datatypes.
**NOTE:** If you see `Warning: calling polyEqual` at any point on this assignment, that is not a cause for concern. Please disregard this warning.

### 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

Submissions will be handled through Autolab, at

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

In preparation for submission, your `hw/05` directory should contain a file named exactly `hw05.pdf` containing your written solutions to the homework.

To submit your solutions, run `make` from the `hw/05` directory (that contains a `code` folder and a file `hw05.pdf`). This should produce a file `hw05.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw05.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

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

Your `hw05.sml` file 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 Wednesday, 1 June at 23:59 EST. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

### 1.4 Methodology

You must use the four step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
2. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
3. Implement the function.
4. Provide testcases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

## 1.5 Style

In this and future homeworks, we will also grade your code for style. If your code for a task does not adhere to the style guidelines on the course website, you will temporarily receive a score of 0 for that task. You may then fix the problems with your code style and show it to a TA within one week of getting your graded assignment back. If you do this, you will get the number of points you would get if your original submission had proper style. If you do not do this, you will keep the grade of 0 for that task.

## 1.6 Style

For this assignment, we will be grading your submissions based on your coding style. There are several ways that you can learn what is good style and what isn't:

- Your returned and graded homework submissions have been graded for style so use the markups for a reference.
- We have published solution code for the previous assignments, labs, and lectures.
- We have published a style guide at

  `https://www.cs.cmu.edu/~15150/resources/handouts/style.pdf`.

  There is also a copy in the `docs` subdirectory of your git clone.
- You can ask your TAs about specific examples, or post on Piazza asking general questions.

Note that if any code you submit for a problem violates our style guidelines, you will receive a 0 for that problem. You will have one week from the time the homework is handed back to the class to fix your style and bring your corrected code and handed-back homework to a TA. If your code is satisfactory, you will then receive the grade your original code would have gotten had it satisfied the style guidelines.

# 2 Types and Polymorphism

In class we discussed typing rules. In particular:

- A function expression `fn x => e` has type `t -> t'` if and only if, by assuming that `x` has type `t`, we can show that `e` has type `t'`.
- An application $\mathtt{e}_1$ $\mathtt{e}_2$ has type `t'` if and only if there is a type `t` such that $\mathtt{e}_1$ has type `t -> t'` and $\mathtt{e}_2$ has type `t`.
- An expression can be used at any instance of its most general type.

**Task 2.1** (4 pts). Consider the following ML function declaration:

```
fun elephant (dumbo, ears) =
  case dumbo of
      [] => ears
    | oooo::TRUMPET => "BEEP" ^ elephant (TRUMPET, ears)
```

What is the most general type of `elephant`?

**Task 2.2** (3 pts). Now consider the following function expression:

```
fn x => (fn y => x)
```

What is its most general type?

**Task 2.3** (3 pts). Now look at this slightly different expression:

```
(fn x => (fn y => x)) []
```

What is the most general type of *this* expression?

# 3 Rationals

For the next few problems, we will introduce a user-defined type named `rat`, a representation of rational numbers with the following type definition:

```
type rat = int * int
```

A rational number (or "fraction") can be represented by a pair of integers $(a, b)$, where $b \neq 0$ and $a$ and $b$ have no common factors. For example, $(1, 2)$ is a valid representation for "one half", commonly written in math notation as $1/2$ or $\frac{1}{2}$. However, $(2, 4)$ is not a valid representation, because 2 and 4 have a common factor of 2. We define the rational number "zero" to be represented as $(0, 1)$. This is the *only* pair whose "numerator" is 0. [1]

A pair of integers $(a, b)$ is *valid* if and only if $b \neq 0$ and the greatest common divisor of $|a|$ and $|b|$ is 1. A pair $(a, b)$ represents the fraction $a/b$.

We've *hidden* the implementation of rationals from your code in `hw05.sml` – they use pairs behind the scenes, but you may use only the functions provided to work with them. In order to enforce that all rationals are valid, the library code will not allow you to call just *any* pair of integers a rational number. Instead, we have provided an infix function `//` that takes two ints and returns a value of type rat. All rationals in your homework file should be defined this way. So in your homework file you should write:

```
val zero:rat = 0 // 1
val one:rat = 1 // 1
val half:rat = 1 // 2
```

The returned value will *not* look like a pair (and you won't be able to pattern-match against it like you would other pairs.)

To help with the next few problems, we have given you a few helper functions (**You do NOT need to implement these functions**):

### 3.0.1 Plus

```
++ : rat * rat -> rat
```

For all valid pairs of integers $(a, b)$ and $(c, d)$, `(a//b) ++ (c//d)` returns a valid pair $(p, q)$ such that $a/b + c/d = p/q$. This pair is called the (rational) sum of $(a, b)$ and $(c, d)$. Note: `++` is infix.

Examples:

```
half ++ half ≅ 1 // 1
one ++ one ≅ 2 // 1
```

[1] For a fraction written as $\frac{a}{b}$ we usually call $a$ the *numerator* and $b$ the *denominator*.

### 3.0.2 Times

```
** : rat * rat -> rat
```

For all valid pairs of integers $(a, b)$ and $(c, d)$, `(a//b) ** (c//d)` returns a valid rational $(p, q)$ such that $(a/b)(c/d) = p/q$. This pair is called the (rational) product of $(a, b)$ and $(c, d)$. Note: `**` is infix.

Examples:

```
one ** one ≅ 1 // 1
half ** half ≅ 1 // 4
```

## 3.1 More Rational Operations

You can subtract rationals with `--` (infix), and divide with `divide` (*not* infix). You can negate rationals with ∼∼ (prefix, like a normal SML function).

## 3.2 RatEq

Since you won't be able to pattern match against rationals, you should use `ratEq` to test rational values.

```
ratEq : rat * rat -> bool
```

For all valid rats $r1$ and $r2$, `ratEq`$(r1, r2)$ returns true if $r1 = r2$.

Examples:

```
ratEq(one, one) ≅ true
ratEq(half, one) ≅ false
ratEq(1//2, 2//4) ≅  true
```

# 4 Integration and Differentiation

We can represent a polynomial $c_0 + c_1 x + c_2 x^2 + \ldots$ 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`). Therefore, we have the following type definition for polynomials:

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

where `rat` is a rational as defined in the previous section. For instance, see:

```
val p : poly = fn x => case x of
                          0 => 1 // 1
                        | 1 => 1 // 2
                        | 2 => 7 // 1
                        | _ => 0 // 1
```

The function `p` would represent the polynomial $1 + \frac{1}{2}x + 7x^2$.

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

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

Also note the functions such as

```
polynomialEqual : poly * poly * int -> bool
```

in `lib.sml`. As with the rationals, you will not be able to pattern match against polynomials. Some of the lib functions may be useful in helping you test your code.

## 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 pts). Define the function

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

that computes the derivative of a polynomial using the definition above. 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. Because $C$ 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 pts). Define the function

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

that, given a polynomial, computes the family of polynomials corresponding to its integral. Note that `integrate` should *not* be recursive.

# 5 Concat Write-and-Prove

In this question you will write a simple function on lists and prove its correctness. **You may not use @, any other built-in list functions, or any helper functions for the tasks in this section.** If you do, the proof will be difficult and you will receive little or no credit for the whole section. The implementations of the programming tasks in this section may be recursive.

### Implementing Concat

**Task 5.1** (5 pts). 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]
```

### A Theorem About Concat

Recall the definition of `append` that combines two lists into a single list:

```
(* append : 'a list * 'a list -> 'a list *)
(* REQUIRES: true
 * ENSURES: append (l1, l2) => l1 @ l2
 *)
fun append ([] : 'a list, l2: 'a list) : 'a list = l2
  |  append (x::xs, l2) = x :: (append (xs, l2))
```

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

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

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

First, we give you Lemma 1 relating `append` and `concatap` and Lemma 2 about `append`.

**Lemma 1.** *For all types* `t`*, all expressions* `l1 : t list` *that evaluate to a value, and all expressions* `l2 : t list list` *that evaluate to a value,*

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

**Lemma 2.** *For all types* `t` *and all expressions* `l : t list` *that evaluate to a value,*

$$\texttt{append([], l)} \cong \texttt{l}$$

**Task 5.2** (15 pts). Now, prove Theorem 1 by structural induction. You will almost certainly need to use Lemma 1; you may also use Lemma 2 freely without proof. You may cite the totality of `concatap` and `append` without proof. If your implementation of `concat` is total, then you also may cite the totality of `concat` without proof. However, if your implementation of `concat` is not total and you justify your proof by stating that `concat` is total, you will receive few points. Remember to closely follow the proof templates we give in the lecture notes, argue carefully with equivalence, and cite the lemmas justifying the preconditions are satisfied as you use them. [2]

**Theorem 1.** *For all types* `t` *and all values* `l : t list list`*,*

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

Hint: Since `l` could be two-dimensional, you may consider using nested induction.

[2] 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 extensional equivalence at a function type, so we don't lose anything by being a little bit more verbose.

# 6 Brandon's Potluck Party

Brandon, the instructor of 15-150[3] , is planning to hold a potluck for the entire course staff. He wants to know what the best time to meet is, so he makes everyone send him their schedules, a list of tuples representing busy times. He has recruited you to help him with this endeavor, and has given you the following information: for each member of the course staff, their obligations are disjoint (i.e. no single schedule will have overlaps), all "schedules" are well-formatted (i.e. the first element of each tuple, representing the "start" time, will be strictly less than the second element of the tuple, the "end" time). Assume time (measured in some arbitrary unit) starts at 0 and ends at the largest number given in the input. Also assume TA's can instantly transport themselves back and forth from Brandon's house to their commitments, so they can make it for a potluck that starts right as a previous engagement ends.

Keep in mind that Brandon's computer is *very* old, and can only finish running functions with work of at most $O(n \log n)$ before the heat death of the universe (where $n$ is the total number of intervals he's given).

**Task 6.1** (11 pts). Write the function

```
all_available : (int * int) list list -> (int * int) list
```

that returns a single list of all intervals (in no particular order) when all members of the course staff are available. Note that `all_available` should *not* be recursive! You may write helper functions for this problem, but they should not be recursive either!

To help you out, here are some hints!

*Hint 1:* We have provided you a polymorphic sorting function with $O(n \log n)$ work. You will probably find this helpful.

*Hint 2:* You may find it useful to define your own datatype for this problem.

*Hint 3:* Consider the parenthesis matching problem. It's quite similar to this problem.

*Hint 4:* You already wrote a `concat` function for a previous task.

Here are some examples of what we're looking for (note that the ordering of the elements in the result is arbitrary):

```
all_available [] ==> []
all_available [[(1, 2), (4, 6), (8, 12)], [(1, 3), (5, 9)]] ==> [(0, 1),
                                                                 (3, 4)]
all_available [[(0, 1), (2, 3)], [(5,7)]] ==> [(1,2), (3,5)]
all_available [[(0, 1), (2, 3)]] ==> [(1,2)]
```

**Task 6.2** (2 pts). Informally justify why your function's work is in $O(n \log n)$.

[3]If you didn't know this, now you do!

**Task 6.3** (2 pts). Brandon wants to hold a *good* potluck, so he only wants intervals that are at least of a minimum length. Using another higher-order function and your `all_available`, write the function

```
all_good_available : (int * int) list list * int -> (int * int) list
```

such that `all_good_available (L, n)` only returns intervals that are of length at least $n$.

# 7 Higher-Order Functions

Recently 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 7.1** (15 pts). 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]]
```

Also, if the inner lists are all empty, the function should return the empty list. For example,

```
transpose [[], [], []] ==> []
```

```
transpose [[]] ==> []
```

```
transpose [] ==> []
```

Your function only needs to work if all the inner lists have the same length, so for example

```
transpose [[1,2],[3]]
```

can have whatever behavior you find most convenient. Your implementation of `transpose` may use recursion, but should use higher-order functions where possible.

**Task 7.2** (5 pts). 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` $\cong$ `true`, then `extract(p,l)` evaluates to `SOME(x,l')`, where `l'` is `l` without the first such `x` but unchanged otherwise.

2. If for every element `x` of `l`, `p x` $\cong$ `false` then `extract(p,l)` evaluates to `NONE`.

If there is more than one element satisfying the predicate in a particular argument list, you should return the first.

For example:

```
extract(oddP , [2,3,4]) ≅ SOME (3, [2,4])
extract(oddP , [2,4,6]) ≅ NONE
extract(oddP , [0,1,2,3,4,5]) ≅ SOME (1, [0,2,3,4,5])
extract(oddP , []) ≅ NONE
extract(fn x => String.size x < 2 , ["aaa","b","bca"])
                         ≅ SOME ("b", ["aaa", "bca"])
```

`extract` should be recursive. Make sure to test your implementation of `extract` thoroughly because you will use this function when you implement Blocks World in the next section.

# 8 Blocks World

In artificial intelligence, *planning* is the task of figuring what an agent (a robot, Siri, James Bond, 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:

```
---    ---    ---
|X|    |Y|    |Z|
---    ---    ---
------------------------
```

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

```
/|\
---
|Z|
---
       ---    ---
       |X|    |Y|
       ---    ---
------------------------
```

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

```
---
|Z|
---
---    ---
|X|    |Y|
---    ---
------------------------
```

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 `Y` and then put it on `X`. A planning problem would be something like "starting with the blocks on the table, make the tower `YZX`".

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 1. 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

---   ---   ---
|X|   |Y|   |Z|
---   ---   ---
-------------------------
Next move: pickup Z from table

/|\
---
|Z|
---
      ---   ---
      |X|   |Y|
      ---   ---
-------------------------
Next move: put Z on X

---
|Z|
---
---   ---
|X|   |Y|
---   ---
-------------------------
```

Figure 1: Sample Blocks World Interaction

## 8.1 Ontology

We will model Blocks World as follows:

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

  ```
  pickup <y> from table
  put <y> on table
  pickup <x> from <y>
  put <x> on <y>
  ```

  These moves act as follows:

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

  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.

## 8.2 Tasks

**Task 8.1** (5 pts). 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` from task 5.2.

**Task 8.2** (8 pts). 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`).

In addition, your datatype constructors must only describe the blocks, moves, and facts described in the ontology. In fact, your datatypes should capture exactly the states described in the ontology. You may not create constructors that allow for invalid states (i.e. states that the ontology does not cover).

Hint: the design of datatypes follows from the explanation of the ontology. To get a better understanding of the modeling, you may want to read through the remaining tasks.

**Task 8.3** (2 pts). 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 $X$,$Y$,$Z$ is on the table, and each of $X$,$Y$,$Z$ is free.

**Task 8.4** (3 pts). 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 (in the multi-set sense). 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 8.5** (7 pts). 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.

As explained in the ontology, for every move, we have a list of "before" facts that hold before the move is performed, and another list of "after" facts that hold after the move is performed.

For example, if we perform the move of picking up block $X$ from the table, the state that holds before the move should be a list of facts that represents at least the following: $X$ is free, $X$ is on the table, and the hand is empty. After the move is performed, those facts above do not hold anymore, and we have a new list of facts that represents at least the following: the hand holds $X$.

**Task 8.6** Optional: 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 7, 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`.

**Task 8.7 EXTREMELY OPTIONAL CHALLENGE TASK:**
If you're really really bored, here's something fun you can try.

The text-based interface we made for blocks world works but is kind of bland. Download a graphics library for SML and use it to implement a fancier interface for blocks world. You'll almost certainly have to make a custom .cm file, so don't modify the one for this assignment. Make a new one and when you're done submit it along with the rest of the homework.

If you want to do 2D graphics you can learn about SDL::ML at `http://www.hardcoreprocessing.com/Freeware/SDLML.html` and if you want to do 3D graphics you can learn about SML3D at `http://sml3d.cs.uchicago.edu/`.