15-150 Fall 2014
Homework 07

Out: Friday 24th October, 2014
Due: Tuesday 4th November, 2014 at 23:59 AST

1 Introduction

This assignment will get you to practice working with higher-order functions, continuations, and regular expressions.

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 "hw07", and then "Download handout". Uncompressing the downloaded file will create the directory `hw07-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/07` (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/07` directory will need to contain the file `hw07.pdf` with your written solutions and the files `fs.sml`, `basix.sml`, `minix.sml`, `cooking.sml`, `cooking2.sml`, `regex.sml`, `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!**

```
make
```

from the `hw/07` directory on any Unix machine (this include Linux and Mac). This will produce a file `hw07-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 `hw07-handin.tgz` file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the `hw/07` directory. That will create `hw07-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 7 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 `fs.sml`, `basix.sml`, `minix.sml`, `cooking.sml`, `cooking2.sml`, `regex.sml`, `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 4$^{\text{th}}$ November, 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.

1.4 Proof Structure

Remember that every proof by induction is structured as follows:

1. The specific *technique* being employed and on what.

2. The *structure* of the proof (number of cases and what they are).
3. For each base case:
   - The statement specialized to this case ("*To show*").
   - The proof of this case.
4. For each inductive case:
   - The statement specialized to this case ("*To show*").
   - The induction hypothesis or hypotheses (*IH*).
   - The proof of this case.

Following this methodology, students have historically submitted proofs that contained fewer errors and were more likely to be correct than otherwise.

1.5 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
```

1.6 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.

1.7 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!

1.8 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.

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

This often results in bizarre error messages. When that happens, it is safe to delete the `.cm` directory and compile again from scratch.

2 File Systems

From the point of view of a user, the file system on your computer consists of files that contain data, and of directories that contain files and other directories.[3] Both files and directories are also associated with *metadata* such as name, size and permissions. The trouble is that different operating systems use different metadata. We can capture the common parts of all file systems by means of the following types:

```
type data = string
datatype 'a fs = File of 'a * data        (* (metadata, file content) *)
               | Dir  of 'a * 'a fs list  (* (metadata, dir contents) *)
```

Here, the type `data` represents the contents of a file (for simplicity we take it to be a `string`). A directory is nothing more than a file system with its own metadata. The type `'a fs` is used to represent file systems with metadata of type `'a`.

We will manipulate file systems, both of the generic kind above as well as (simple) instances. In the following exercises, you don't have to use higher-order functions on lists if you don't want to, but doing so will make your code considerably shorter. If you decide to use them, functions such as `List.map`, `List.foldr`, `List.filter` and `List.tabulate` (defined int `List` standard basis library) will prove very useful.

2.1 Generic file systems

The signature `FS` in file `fs.sig` defines a few basic operations on generic file systems:

```
val map: ('a * string -> 'b * string) -> ('a -> 'b) -> 'a fs -> 'b fs
val reduce: ('a * string -> 'b) -> ('a * 'b list -> 'b) -> 'a fs -> 'b
val count: 'a fs -> int * int
```

They work as follows:

- `map ff fd sys` returns the file system `sys'` obtained by applying the function `ff` to the data and metadata of the files in file system `sys`, and the function `fd` to the metadata of its directories.
- `reduce ff fd sys` returns the value `v` obtained by applying the function `ff` to the files in file system `sys` and recursively combining these results by using the function `fs` over the contents of its directories.
- `count sys` returns the pair `(nf,nd)` where `nf` is the number of files in file system `sys` and `nd` is the number of directories in `sys`.

[3]Some operating systems use the word "folder" instead of directory. We will stick with "directory".

Task 2.1 (10 points) Implement these functions in the structure `GenericFs` in file `fs.sml`. The implementation of `map` and `reduce` may be recursive, but the code for `count` should not.

2.2 The BasiX file system

The BasiX^{TM} file system gives a name to each file and directory. It is defined as

```
type basix = string F.fs
```

where `F` is a structure that ascribtes to signature `FS`. The metadata associated with each file and directory is its name, a string. Each entry in a directory is required to have a different name.

The big advantage of the BasiX file system is that it allows to refer to each node in a file system using a *path*. A path to such a node is the list of the directory names that are traversed to access it. It is defined by the type

```
type path = string list
```

The signature `BASIX` in file `basix.sig` defines operations to manipulate a BasiX file system:

```
val mkfs: string -> basix
val cp: basix -> path -> basix -> basix
val mkdir: string -> path -> basix -> basix
val mkfile: string * string -> path -> basix -> basix
val mkdir_p: path -> basix -> basix
val exec: (basix -> 'a) -> path -> basix -> 'a option
val ls: path -> basix -> string list
val grep_r: (string -> bool) -> basix -> string list
val grep_rl: (string -> bool) -> basix -> path list
```

The signature `BASIX` also provides all the declarations made available by `FS` through the structure `F`. These operations work as follows:[4]

- `mkfs s` creates a file system whose root is called `s`.
- `cp e p sys` returns a file system `sys'` obtained by adding entry `e` to `sys` along path `p`. If `p` does not exist in `sys`, if it does not end in a directory, or if the name `d` already exists in `p`, this function returns `sys` unchanged.
- `mkdir d p sys` creates an empty directory called `d` in path `p` in `sys`. It is subject to the same constraints as `cp`.

[4]Most of these operations are simplified versions of Unix commands: `cp`, `mkdir` and `ls` work similarly to their Unix homonyms; `mkdir_p` is Unix's `mkdir -p`; finally, `grep_r` and `grep_rl` are extensions of `grep -r`.

- `mkfile (n,c) p sys` add the file `n` with content `c` in path `p` in `sys`. It is subject to the same constraints as `cp`.
- `mkdir_p p sys` creates the path `d` in file system `sys`. Differently from `mkdir`, it will create any missing directory along `p`. The last item in `d` is understood as a directory.
- `exec f p sys` follows the path `p` in `sys` and executes the function `f` to the resulting node. If `p` does not exist or does not end in a directory, it returns `NONE`.
- `ls p sys` follows the path `p` in `sys` and returns a list of the node names it finds there. Specifically, if `p` ends in a file, it returns the name of this file. If instead `p` ends in a directory, it returns the list of the names of the entries in this directory. If `p` does not exist or ends in a file, it returns the empty list.
- `grep_r f sys` returns the list of the file names in `sys` whose contents satisfies the function `f`.
- `grep_rl f sys` returns the list of the paths to the file in `sys` whose contents satisfies the function `f`.

Task 2.2 (30 points) Implement these functions in the functor `Basix` in file `basix.sml`. The implementation of `cp`, `exec` and `mkdir_p` may be recursive, but the code for the remaining functions should not. The operations imported from signature `FS` are automatically made available to you: no need to reimplement them.

2.3 The MiniX file system

MiniX^{TM} is a successor of the BasiX file system which store not only the file or directory name as metadata, but also the size of the node. Its type is given as

```
type meta = string * int
type minix = meta B.F.fs
```

Specifically, the size information associated with a file is simply the size (i.e., the number of characters) of its data. The size information associated with a directory is the fixed size 1024.

The signature `MINIX`, found in file `minix.sig`, contains a structure `B` that ascribes to signature `BASIX`. This signature also defines the following three functions

```
val basix2minix: B.basix -> minix
val minix2basix: minix -> B.basix
val du: minix -> int
```

which operate as follows:

- `basix2minix sys` upgrades the BasiX file system `sys` to the corresponding `MiniX` file system by adding the appropriate size information in each node.
- `minix2basix sys` downgrades the `Minix` file system `sys` to the corresponding `BasiX` file system.
- `du sys` returns the overall size of the file system `sys` by adding up the sizes found at each node.

Task 2.3 (15 points) Implement these functions in the functor `Minix` in file `minix.sml`. Your code should not be recursive.

The signature `MINIX` also declares `MiniX` variants of all the `BasiX` operations defined in signature `MINIX` — not that their type is different however.

Task 2.4 (10 points) Implement these functions in the functor `Minix` in file `minix.sml`. Your code should not be recursive.

2.4 Testing

Test your code for the all the tasks in this exercise in file `test-all.sml`.

Task 2.5 (5 bonus points) Also in file `test-all.sml`, give a non-recursive implementation of the functions `map` using `reduce`. You defined these functions in Task 2.1.

3 Cooking with SML

Welcome to the SML kitchen! Here you will be cooking up delicious recipes in style (by which we mean *continuation-passing* style).

When cooking, we rely on a cookbook that provides us with many recipes, where each recipe names a dish and lists it ingredients. However, being skilled chefs, we know that some ingredients could themselves be prepared on the basis of recipes. For example, suppose we wish to cook lasagna and we have every ingredient except for pasta. However, we have flour and eggs, and a recipe to make pasta from scratch using flour and eggs. Then, we can still cook lasagna even though we are missing pasta.

Thus, one goal is to determine, given a list of ingredients and a cookbook, if we can cook a dish. Another is to make a shopping list for the missing ingredients of any dish we are craving. We represent our kitchen with the following types:

```
type ingr     = string
type dish     = string
type recipe   = dish * dish list
type cookBook = recipe list
type fridge   = ingr list
```

Here is an example of a cookbook and two fridges, given to you in the starter file `cooking-test.sml`.

```
(* the recipes *)
val lasagna   = ("lasagna",   ["pasta", "bechamel", "meat",
                                "bolognese", "parmesan"])
val pasta     = ("pasta",     ["flour", "egg"])
val bechamel  = ("bechamel",  ["flour", "butter", "milk", "pepper"])
val bolognese = ("bolognese", ["tomato", "onion", "pepper", "secret"])
val secret    = ("secret",    ["balsamic", "parsley", "mozzarella"])

(* The cookbook *)
val italianCookbook = [lasagna, pasta, bechamel, bolognese, secret]

(* Two different fridges *)
val bigFridge   = ["flour", "egg", "milk", "butter", "pepper", "tomato",
                   "onion", "balsamic", "parsley", "meat", "parmesan"]
val smallFridge = ["pasta", "bolognese", "meat", "parmesan", "flour",
                   "milk", "butter", "pepper"]
```

You may assume that we bought enough of each ingredient so that we may use it in multiple recipes without running out.

You have been provided the following helper functions in the starter file `cooking.sml`:

- `lookupRecipe: dish -> cookBook -> dish list option`. This function checks a cookbook for a recipe to cook the input dish. If one exists, it returns `SOME` of the list of ingredients to cook it.
- `checkFridge: ingr -> fridge -> bool`. This function checks your fridge to see if it contains a given ingredient.

3.1 Can we cook that?

Before we get started, we need to introduce two new SML concepts.

1. The simplest type in SML is `unit`: it has a single value, the unit value, which is written `()`. Below, we will use it when we need to trigger the execution of a function (specifically of a continuation), but we don't have anything interesting to pass to it.
2. A function cannot be called before it is declared in SML. If we want to make two functions that call each other (such functions are called *mutually recursive*), we must introduce the second one with the keyword `and` (instead of `fun`).

In the next task you will write two mutually recursive helper functions that are needed by the function `canCook`, which tell us whether we can cook a dish given the contents of our fridge and the recipes in our cookbook. Here's the code for `canCook` (also in `cooking.sml`):

```
fun canCook (d: dish) (F: fridge) (B: cookBook): bool =
      canMakeDish d F B (fn () => true)
```

As you can see, it makes use of the function `canMakeDish`, whose last argument is a continuation whose argument is of type `unit`.

Task 3.1 (10 points) Write the following mutually recursive functions in continuation-passing style:

```
canMakeDish: dish -> fridge -> cookBook -> (unit -> bool) -> bool
canFindIngr: ingr list -> fridge -> cookBook -> (unit -> bool) -> bool
```

Here is what they should do:

- `canMakeDish d F B k` $\cong$ `true` iff you can make dish `d` using cookbook `B` and the items in fridge `F`, *and `k ()` evaluates to `true`.*
- `canFindIngr ds F B k` $\cong$ `true` iff you have or can make all the ingredients in list `ds` given fridge `F` and cookbook `B`, *and `k ()` evaluates to `true`.*

In both functions, the last argument is a *success continuation*, whose type is `unit -> bool`. A success continuation is a stack of tasks to be done later. You add a task to the success continuation when making a recursive call. The success continuation is called once you are completely done with the current task (you have carried it out successfully). Calling it retrieves the next task to be carried out.

To receive credit, your implementation should use the continuation to store any extra work that needs to be done when the function calls itself recursively.

When you are finished, you can test your code with the function `canCook`.

3.2 What are we missing?

If you are inspired to cook a dish but you do not have the necessary ingredients, you will go buy the missing ingredients so that you may make dinner! However, first you need to determine *which* ingredients you are missing. The function `marketRun`, given in the starter file `cooking.sml`, does precisely that, by calling the helper function `missingIngr`.

Task 3.2 (10 points) Write the following mutually recursive helper functions:

```
missingIngr: dish -> fridge -> cookBook -> (ingr list -> ingr list)
             -> ingr list
missingIngrList: ingr list -> fridge -> cookBook -> (ingr list -> ingr list)
             -> ingr list
```

Here is what they should do:

- `missingIngr d F B k` $\cong$ `l` iff the missing ingredients for cooking dish `d` given fridge `F` and cookbook `B` form some list `l'`, *and `k l'` evaluates to `l`.*

- `missingIngrList ds F B k` $\cong$ `l` if the missing ingredients among subdishes `ds` given fridge `F` and cookbook `B` is some list `l'`, *and `k l'` evaluates to `l`.*

Here, the last argument for both functions is also a success continuation, this time of type `ingr list -> ingr list`. This continuation takes as input the list of missing ingredients computed so far, and returns the overall list of missing ingredients by carrying out the tasks you had stored in the continuation. You should use the continuation function to store work for later when one of your functions makes multiple recursive calls.

When you are finished, you can test your code with the function `marketRun`, which calls your functions to determine the list of missing ingredients for a given dish.

3.3 What can we cook?

Some days, you just don't feel like going shopping. On days like these, it would be nice to just have a list of everything you can cook with the ingredients you have.

Task 3.3 (10 points) Define the SML function

```
canMake: cookBook -> fridge -> dish list
```

in continuation-passing style, with the help of the auxiliary function:

```
feasible: dish list -> cookBook -> fridge -> (dish list -> dish list)
          -> dish list
```

such that `canMake B F` ≅ `ds` where `ds` is the list of dishes from cookbook `B` that you can cook given what's in fridge `F`. Note that the last argument of `feasible` is a continuation of type `dish list -> dish list`. This continuation function takes as input the list the dishes you can cook you have computed so far and returns the complete list of which dishes you can cook. Use your continuation function to store all work done in the recursive case.

3.4 Cooking with limited ingredients

In the previous task, we assumed that "we bought enough of each ingredient so that we may use it in multiple recipes without running out". That's not realistic: if you have one egg in your fridge and the recipe for lasagna calls for two, you won't be able to prepare it. In fact, recipes use ingredients in set amounts, e.g., two eggs, 0.250 Kg of flour, and so on. Finally, what if you are really hungry and want to double your recipe for lasagna?

We will now accommodate all these possibilities. To do so, we need to change the declarations for the types `ingr` and `dish` as follows:

```
type ingr = real * string
type dish = real * string
```

where, when writing an ingredient as `(m,s)`, `s` is the name of the ingredient as before and `m` is the quantity of this ingredient we have or need. Similarly for a dish `(m,s)`: `s` is the name of the dish and `m` is the number of servings. For example, our cookbook from above could now be written as follows:

```
(* the recipes *)
val lasagna   = ((4.0, "lasagna"),  [(0.25,"pasta"), (0.5,"bechamel"),
                                     (0.4,"meat"), (0.2,"bolognese"),
                                     (0.1,"parmesan")])
val pasta     = ((1.0, "pasta"),    [(0.5,"flour"), (3.0,"egg")])
val bechamel  = ((0.2,"bechamel"),  [(0.1,"flour"), (0.5,"butter"),
                                     (0.5,"milk"), (0.01,"pepper")])
val bolognese = ((2.0,"bolognese"), [(3.0,"tomato"), (1.0,"onion"),
                                     (0.01,"pepper"), (1.0, "secret")])
val secret    = ((1.0,"secret"),    [(0.1,"balsamic"), (1.0,"parsley"),
                                     (0.5,"mozzarella")])
```

```
(* The cookbook *)
val italianCookbook = [lasagna, pasta, bechamel, bolognese, secret]

(* Two different fridges *)
val bigFridge   = [(2.3,"flour"), (6.0,"egg"), (1.2,"milk"), (2.0,"butter"),
                   (0.2,"pepper"), (4.0,"tomato"), (5.0,"onion"),
                   (0.8,"balsamic"), (1.0, "parsley"), (0.7,"meat"),
                   (0.3,"parmesan")]
val smallFridge = [(1.5,"pasta"), (0.8,"bolognese"), (0.6,"meat"),
                   (0.2,"parmesan"), (1.1,"flour"), (1.0,"milk"),
                   (2.0,"butter"), (0.1,"pepper")]
```

In this example, to make four servings of lasagna, we need a quarter Kg of pasta, half a cup of bechamel, 0.4 Kg of meat, a fifth of a can of bolognese, and 100 g of Parmesan. In our small fridge, we have 1.5 Kg of pasta, three fifth of a can of bolognese, etc.[5]

You may assume as an invariant that all quantities are positive and (for simplicity) that ingredient/dish appears at most once in a recipe, fridge and cookbook.

Task 3.4 (15 bonus points) Reimplement the functions defined in the first part of this exercise to account for this updated definition of ingredients and dishes. In particular, you need to be sure you have enough of each ingredient to be able to prepare a dish (in the right amount). Similarly, when writing your shopping list, you need to account for ingredients you have, but now in the quantity required. You can find some initial code in the starter file `cooking2.sml`. The datatype `status` replaces options in functions like `checkFridge`: use the value `Pos (m,s)` to say that you have `m` units of `s` in your fridge, and the value `Neg (m,s)` to indicate that you are missing `m` units of `s` — i.e., if you had an extra `m` units of `s` you would have the quantity you need.

3.5 Testing

Test your code for all the tasks in this exercise in the file `test-all.sml`.

[5]The units are for illustration purpose only: for simplicity we will assume that there are no units associated with our quantities.

4 Regular Expressions

In class, we introduced five different operators to describe regular expressions over some alphabet Σ:

- $\mathbf{1}$, which describes the language consisting of just the empty string;
- c, which describes the language consisting of the single character c in Σ;
- $r_1 r_2$, the language obtained by concatenating strings in $L(r_2)$ to strings in $L(r_1)$;
- $r_1 + r_2$, the language of the strings in either $L(r_1)$ or $L(r_2)$;
- r^*, the language obtained by concatenating any number of strings in $L(r)$.

Other operators are available to construct regular expressions. Here are two of them that are particularly useful:

- $\_$ (pronounced "wildcard"), which describes the language of the strings consisting of any single character from the alphabet:

$$L(\_) = \{c \mid c \in \Sigma\}$$

- $r_1 \cap r_2$ (pronounced "r_1 intersection r_2"), which describes the language of the strings that are *both* in $L(r_1)$ *and* in $L(r_2)$:

$$L(r_1 \cap r_2) = \{s \mid s \in L(r_1) \text{ and } s \in L(r_2)\}$$

The regular expression matcher `match` from class is given in the starter file `regex.sml`. We have extended the datatype definition of `regex` to include the new constructors `Wild` and `Both`, which correspond to $\_$ and $\cap$, respectively. Your job is to extend `match` to deal with these new constructors, and prove parts of the correctness of your implementation.

As we saw in class, the *correctness* statement for `match` has two parts, *soundness* and *completeness.* Then, we stated and proved these two parts separately. For variety, we will consider a combined statement.

Theorem 1 (Correctness of `match`).
For all `r: regex`, for all `cs: char list` and for all `k: char list -> bool`:

(Soundness) *If `match r cs k` $\cong$ `true`, then there exist `p,s: char list` such that*

- *`p @ s` $\cong$ `cs`,*
- *`p` $\in L($`r`$)$ and*
- *`k s` $\cong$ `true`.*

(Completeness) *If there exist `p,s: char list` such that*

- *`p @ s` $\cong$ `cs`,*

- *`p` $\in L($`r`$)$ and*
- *`k s` $\cong$ `true`,*

then `match r cs k` $\cong$ `true`.

Recall that the proof proceeds by lexicographic induction on `r` and `cs`: it distinguishes cases on `r` and only when needed on `cs`. Furthermore, either of the following two situations must be met for us to be allowed to use an induction hypothesis:

- Either the property refers to a regular expression that is strictly smaller than `r` (and to an arbitrary string `cs`),
- or, the property refers to `r` itself but to a string that is strictly smaller than `cs`.

You can find a detailed proof of the cases we saw in class on the course wiki.

Let's now implement the clauses of `match` for `Wild` and `Both`, and prove that the resulting code is sound. In each of the following coding tasks, we strongly recommend that you think through the correctness spec when you are writing the code. If you're stuck on the implementation, try doing the proof of soundness and/or completeness — this will guide you to the answer.

Test your code for all the programming tasks in this exercise in file `test-all.sml`.

In your proofs, you may assume the following lemma, but must cite it when you use it:

Lemma 1. *For all `cs: char list`, if `cs` $\cong$ `p@s`, `cs` $\cong$ `p'@s'` and `s` $\cong$ `s'`, then `p` $\cong$ `p'`.*

Task 4.1 (5 points) In the file `regex.sml`, complete the clause of function `match` corresponding to `Wild`.

Task 4.2 (10 points) Complete the soundness proof case for `Wild` in the prof of correctness of `match` — *you do not need to prove completeness for this task.*:

Case `r` $\cong$ `Wild`:

To show:

(soundness) For all `cs: char list` and `k: char list -> bool`,
if `match Wild cs k` $\cong$ `true`, then there are `p,s: char list` such that `p@s` $\cong$ `cs` and `p` $\in L(\_)$ and `k s` $\cong$ `true`.

(completeness) For all `cs: char list` and `k: char list -> bool`,
if there are `p,s: char list` such that `p@s` $\cong$ `cs` and `p` $\in L(\_)$ and `k s` $\cong$ `true`, then `match Wild cs k` $\cong$ `true`.
*You do **not** need to prove completeness.*

Task 4.3 (15 points) In the file `regex.sml`, complete the clause of function `match` corresponding to `Both(r1,r2)`.

Task 4.4 (20 points) Complete the soundness proof case for `Both(r1,r2)` in the proof of correctness of `match` — *you do not need to prove completeness for this task.*

Case `r` $\cong$ `Both`(r1, r2):

To show:

(soundness) For all `cs: char list` and `k: char list -> bool`, if `match (Both(r1,r2)) cs k` $\cong$ `true`, then there are `p,s: char list` such that `p@s` $\cong$ `cs` and `p` $\in L($`r1` $\cap$ `r2`$)$ and `k s` $\cong$ `true`.

(completeness) For all `cs: char list` and `k: char list -> bool`, if there are `p,s: char list` such that `p@s` $\cong$ `cs` and `p` $\in L($`r1` $\cap$ `r2`$)$ and `k s` $\cong$ `true`, then `match (Both(r1,r2)) cs k` $\cong$ `true`.
*You do **not** need to prove completeness.*

Note: Do these two tasks carefully! There is a plausible-looking, but incorrect, implementation of `Both`; this case of the proof will fail if your code has this bug.

Task 4.5 (20 bonus points) Why stop here? Carry out the completeness proof cases for `Wild` and `Both`.