# 15-150 Summer 2016
# Lab 7

7 June 2016

## 1 Introduction

The goal for this lab is to make you more familiar with exceptions, continuations, and regular expressions (as covered in lecture) in Standard ML.

Please take advantage of this opportunity to practice writing functions and proofs with the assistance of the TAs and your classmates. You are, as always, encouraged to collaborate with your classmates and to ask the TAs for help.

### 1.1 Getting Started

Update your clone of the `git` repository to get the files for this week's lab as usual by running

```
git pull
```

from the top level directory (probably named `15150`).

### 1.2 Methodology

You must use the four step methodology for writing functions for every function you write on this assignment. In particular, every function you write should have a REQUIRES and ENSURES clause and tests.

# 2 Regular Expressions

In class, we introduced six different operators to describe regular expressions:

- `Zero` (corresponding to the empty language)
- `One` (corresponding to the language consisting of just the empty string)
- `Char(c)` (corresponding to the language consisting of the one-character string `"c"`)
- `Times` (for concatenation)
- `Plus` (for alternation)
- `Star` (for repetition)

Here, we extend the set of regular expressions to include one more: `Wild`, the character wildcard symbol _, which represents the language consisting of any one character:

$$\mathcal{L}(\_) = \{\texttt{"c"} \mid \mathbf{c} \text{ is a character } \}$$

The datatype for regular expressions is similarly extended to include `Wild`:

```
datatype regexp = Zero
                | One
                | Char of char
                | Wild
                | Plus of regexp * regexp
                | Times of regexp * regexp
                | Star of regexp
```

**Task 2.1** Give a value `rml` of type `regexp` such that the language of `rml` is the set of all character lists ending with the characters in the string `".sml"`

**Task 2.2** Give a value `rname` of type `regexp` such that the language of `rname` is the set of all character lists beginning with any single character and ending with the first three characters in your last name. There are no characters between the two.

**Task 2.3** Match is a function

```
match : regexp -> char list -> (char list -> bool) -> bool
```

that uses continuation passing to determine if the given `char list` matches the given regular expression. Extend the function definition for `match` to cover all regular expressions built using `Wild`. Use the function template in `lab08.sml`.

**Task 2.4** Use your value from Task 4.2 to write at least two tests for the `Wild` case of `match`. Justify your choice of initial continuation.

**Task 2.5** Write a function

```
accept : regexp -> string -> bool
```

such that `accept` returns true iff the given string is matched by the regular expression. **Hint:** The function `String.explode : string -> char list` which converts a string into a list of chars may be useful.

**Task 2.6** Write an ML function

```
grep_ml : string list -> string list
```

such that `grep_ml L` returns a list of all strings in L that end with the substring `".sml"`. You can use any function or regexp defined in lab as well as any standard list-manipulating functions from class (in particular `List.filter` might be useful).

**Task 2.7** Define an ML function

```
nub : regexp -> regexp
```

such that for all values `R` of type `regexp`, `nub R` returns a value `R'` with $\mathcal{L}(\texttt{R'}) = \mathcal{L}(\texttt{R}) - \{\texttt{""}\}$. That is, `nub R` returns a `regexp` that's equivalent to `R`, except that it doesn't match the empty string. [1]

[1] For more information on this problem, check out this paper:

www.cs.cmu.edu/~rwh/papers/regexp/jfp.ps

# 3 Zero-Simplification

It happens that there are infinitely many ways of writing a regular expression that accepts nothing. For example, all of the following are regular expressions that accept nothing:

```
Zero, Plus(Zero, Zero), Plus(Zero, Times(Zero, One)), Times(Char #"a", Zero)
```

Write a function `zeroout R` that returns `Zero` if `R` accepts nothing and otherwise evaluates to a regular expression `R'` that is zero-simple and has the same language as `R`. `R` is a zero-simple regular expression if there are no instances of the constructor `Zero` within `R`.

# 4 Unlimited Type-Moon Works

Vincent-senpai is hard at work on his project (ha ha, as if) when he realises that what he really wants is the the ability to create a key-value map that stores values of differing types (not a map that's polymorphic over the type of values, but rather one that can, say, store both `int`s and `string`s at the same time). Basically, Vincent-senpai wants to be able to have a Python-style data structure in SML! Unfortunately, Vincent-senpai's advisor, Carl Krary, will hunt Vincent-senpai down and flay him alive if he even so much as thinks "I wish I could use Python..." in his presence.
To save his own skin, Vincent-senpai thinks hard. One idea he hits on is to declare a new datatype of the form

```
datatype mapval = Int of int | String of string
```

and making the map (say an association list in this case) be of type

```
(string * mapval) list
```

This suffers from the issue that it is not possible to dynamically "extend" the map to accept values of more types at runtime (imagine if we wanted to store values in the map based on user input, or something). Vincent-senpai, secretly a Haskell fan, is horrendously lazy and does not want to expend this effort.
Vincent-senpai thinks harder, back to when he was a little ichinensei. He recalls Erdmann-sensei telling him that "`exn` is an extensible type!" Will this single piece of knowledge save Vincent-senpai? It turns out that, knowing this, it is possible to store in a type-safe way values of *any* type in such a map! However, Vincent-senpai notoriously hates programming. Help Vincent-senpai implement his ideas so he won't be flayed and can hold office hours next week!

## 4.1 A universal type

A *universal type* is a type into which all other types can be embedded. It turns out that, for deep reasons, the `exn` type in SML (which is an *extensible* type, after all), is an *open variant*, meaning datatype constructors can be added to `exn` at any point in the execution of a program, not just statically at compile time. As such, exceptions can be used as our desired universal type!

**Task 4.1** Help Vincent-senpai implement a function

```
embed : unit -> ('a -> exn) * (exn -> 'a option)
```

that, when applied, produces the tuple `(inject, project)` such that `project(inject x) --> SOME x`. `inject` produces a value of the universal type `exn` that contains a value of type `'a`, and `project` takes a value created by `inject`[2] and returns the contained value of type `'a`.

[2]You'll see how to maintain such an invariant later in the class, but for now just don't call `project` on any value of type `exn` not produced by `inject`.

## 4.2 "Dynamically-typed" association lists!

Armed with the universal type from the previous section, you are now ready to implement association lists that store values of any type!
We have defined

```
type alist = ((string * exn) list * (exn -> string)
```

for you. Think of this as a list holding elements of universal type, and a function that can stringify elements of any type ("actual" type) the list contains.
An `alist` maintains the invariant that any element added to it can be stringified by its `toString` function.

The functions you will be implementing are `add_type`, `print_alist`, and `add_to_alist`. Here's an example to illustrate what these funcitons do:

```
val empty : alist = ([], fn _ => raise Fail "nooo")
val (al, inject, _) = add_type empty (fn x => x)
val (al', inject', _) = add_type al Int.toString
val al'' = add_to_alist al' ("15", inject "150")
val al''' = add_to_alist al'' ("150", inject' 15)
print_alist al'''
```

`al'''` is a map which maps the string "15" to the string "150", and the string "150" to the int 15. This code should print `[(15, 150), (150, 15)]` (with potential formatting differences compared to your code).

**Task 4.2** So, implement a function

```
add_type : alist -> ('a -> string) -> (alist * ('a -> exn) *
  (exn -> 'a option))
```

that, given an alist and a stringify function for a type, returns an alist augmented to accept elements of that type.

**Task 4.3** Now, implement

```
print_alist : alist -> unit
```

that prints out all the elements of an alist (preferably formatted somewhat nicely). You may find the Basis functions `TextIO.print` and `List.app` useful.

**Task 4.4** Now, implement

```
add_to_alist : alist -> (string * exn) -> alist
```

that adds an element (already injected into the universal type) into an alist.

**That was... *exceptional*. Have the TAs check you in.**