CSE 341
Lecture 6

exceptions;
higher order functions; `map`, `filter`, `reduce`

Ullman 5.2, 5.4

slides created by Marty Stepp

http://www.cs.washington.edu/341/

Exceptions (5.2)

- **exception**: An object representing an error.
  - can be generated ("raised") and repaired ("handled")
  - an elegant way to provide non-local error recovery

- exceptions can be used in ML for many reasons:
  - to bail out of a function whose preconditions are violated
  - "partial" functions that don't map entire domain to range
  - when a function doesn't know how to handle a particular problem (e.g., file not found; empty list; etc.)
  - ...

Raising an exception

```
(* defining an exception type *)
exception name [of parameterTypes];

(* raising ("throwing") an exception *)
raise exceptionName (parameterValues);
```

- ML includes many pre-defined exception types, but you can (and often should) define your own

Raising exception example

```
exception Negative;

(* Computes n!, or 1*2*3*...*n-1*n. *)
fun factorial(0) = 1
|   factorial(n) =
        if n < 0 then raise Negative
        else n * factorial(n - 1);


- factorial(~4);
```

uncaught exception Negative
raised at: stdIn:6.29-6.37

Exception with parameter

```
exception Negative of string;

(* Computes n!, or 1*2*3*...*n-1*n. *)
fun factorial(0) = 1
|   factorial(n) =
      if n < 0
      then raise Negative("Can't pass a    ¬
                    negative number for n!")
      else n * factorial(n - 1);
```

Handling an exception (5.2.3)

`expression1 handle exception => expression2`

- *handling* an exception stops it from going all the way up the call stack and stopping the program with an error
- the above code tries to compute ***expression1***, but ...
  - if that computation raises an exception of type ***exception***, then expression1 will be replaced by ***expression2***.

  - The ***`exception => expression2`*** syntax is an example of a *match*, which we'll see more later.

Handling exception example

```
(* Returns 2 * n!.  A silly function.
   If factorial fails, produces 0. *)
fun example(n) =
      2 * factorial(n) handle Negative => 0;

- example(4);
val it = 48 : int

- example(~3);
val it = 0 : int
```

Operators as functions

- Every operator in ML is really a function defined with op:

```
- op +;
```
*val it = fn : int * int -> int*

```
- op +(2, 5);
```
val it = 7 : int

```
- op *(op +(2, 5), op ~(4));    (* (2+5)*~4 *)
```
val it = ~28 : int

```
- op ^("hello", "world");
```
val it = "helloworld" : string

Defining an operator

```
(* if defining a binary operator *)
infix operator;

fun op operator = expression;
```

- The operator can call itself recursively in its own expression as: `op` ***`operator`***`(`***`parameters`***`)`

Defining operator example

```
(* Exponentiation operator, computes x^y.
   Not tail-recursive.  Fails for y<0. *)
infix ^^;
fun op ^^(x, 0) = 1
|   op ^^(x, y) = x * op ^^(x, y - 1);

- 2 ^^ 10;
val it = 1024 : int
```

- *Exercise:* Define an operator `--` such that `a--b` will create a list of the integers [a, a+1, a+2, ..., b-1, b].

Defining operator solution

```
(* x--y Produces [x,x+1,...,y-1,y].
   Not tail-recursive. *)
infix --;
fun op --(x, y) =
    if x > y then []
    else x :: op --(x+1, y);

(* alternate version *)
fun x--y =
    if x > y then []
    else x :: (x+1)--y;
```

Functions as values

- in ML, a variable is just a symbol in the environment that maps from a name to a value
- a *function* is actually just a symbol as well; it maps from a function name to a piece of code to execute
- you can assign a variable (`val`) to refer to a function:

```
- val xyz = factorial;
```
val xyz = fn : int -> int

```
- xyz(5);
```
val it = 120 : int

Functions as parameters

- Since functions are values,
  we can pass them as parameters to other functions!

```
- fun callAndAdd1(f) = f(4) + 1;
```
val callAndAdd1 = fn : (int -> int) -> int

```
- callAndAdd1(factorial);
```
val it = 25 : int

Higher-order functions (5.4)

- **higher-order function**: A function that accepts another function as input and/or produces a function as output.
  - `callAndAdd1` is higher-order, as is `apply` below.

```
- fun apply(f, x) = f(x);
```

*val apply = fn : ('a -> 'b) * 'a -> 'b*

```
- apply(round, 3.54);
```

val it = 4 : int

Common higher-order functions

Many functional languages provide the following functions:

- `map(`***`F, list`***`)`: Applies *F* to each element of the list [*a*, *b*, *c*, ...] and produces a new list [*F*(*a*), *F*(*b*), *F*(*c*), ...].
- `filter(`***`P, list`***`)`: Applies a boolean function ("predicate") *P* to each element, and produces a new list of every element *k* from the list where P(*k*) was true.
- `reduce(`***`F, list`***`)`: Collapses *list* into a single value by applying F to pairs of elements. *F* is assumed to accept two of *list*'s values and produce a single value each time.

Implementing map

- ML includes `map`, but let's think about how it might be implemented by writing our own version.

```
- fun map(F, []) = []
  |   map(F, first::rest) =
                    F(first) :: map(F, rest);
```
*val map = fn : ('a -> 'b) * 'a list -> 'b list*

```
- map(abs, [2, ~7, 19, ~1, ~95, 6]);
```
val it = [2,7,19,1,95,6] : int list

map exercise

- Use map to convert a list of `ints` into their square roots.
  - Example: turn `[4, 9, 1, 2, 16]` into `[2.0, 3.0, 1.0, 1.41421356237, 4.0]`

- Solution:

```
- val L = [4, 9, 1, 2, 16];
- map(Math.sqrt, map(real, L));
```

Implementing filter

- ML includes `List.filter`, but let's think about how it might be implemented by writing our own version.

```
- fun filter(P, []) = []
  |   filter(P, first::rest) =
    if P(first) then first :: filter(P, rest)
    else filter(P, rest);
```

*val map = fn : ('a -> 'b) * 'a list -> 'b list*

```
- fun positive(x) = x > 0;
- filter(positive, [2, ~7, 19, ~1, ~95, 6]);
```

val it = [2,19,6] : int list

Filter exercise

- Define a function `removeAll` that accepts a list *L* and a value *k* and produces a new list containing *L*'s elements with all occurrences of *k* removed. (Use `filter`.)
  - Example: `removeAll([2, 9, 2, 2, 7, ~8, 2, 4], 2)` produces `[9, 7, ~8, 4]`

- Solution:

```
fun removeAll(L, k) =
    let fun equalsK(x) = x = k
    in  filter(equalsK, L)
    end;
```

Implementing reduce

- ML includes `List.foldl` and `List.foldr`, but let's think about how reduce might be implemented by writing our own version:

```
exception EmptyList;
fun reduce(F, []) = raise EmptyList
|   reduce(F, [value]) = value
|   reduce(F, first::rest) =
        F(first, reduce(F, rest));
```

*val reduce = fn : ('a * 'a -> 'a) * 'a list -> 'a*

reduce exercises

- Use `reduce` to compute the sum and product of a list.

```
val L = [2, 5, ~1, 8];
reduce(op +, L);
```

- Use `reduce` to square the elements of a list.
  - Example: turn `[2, 5, ~1, 8]` into `[4, 25, 1, 64]`

```
fun square(x) = x*x;
reduce(square, L);
```