CSE 341
Lecture 8

curried functions
Ullman 5.5

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Recall: helper ML files

- We are starting to accumulate lots of helper code
  - `map`, `filter`, `reduce`, `--`, etc.

- Let's put it into a helper file `utility.sml`
  - in our other programs, we can say:

```
use "utility.sml";
```

Curried functions (5.5)

- Recall that functions really have just 1 parameter:
  a *tuple* representing each value you want to pass
  - the parameters are dependent on each other;
    must all be passed at once, as a tuple

- **curried function**: Divides its arguments such that they can be partially supplied, producing intermediate functions that accept the remaining arguments.
  - A more powerful way to connect a function to parameters
  - can only be used with functions defined in a curried form

    (in "pure" functional langs, EVERY function can be curried)

Curried function syntax (5.5)

```
fun name param1 param2 ... paramN = expression;
```

- Example:

```
(* Computes x ^ y. Assumes y >= 0. *)
fun pow x 0 = 1
|   pow x y = x * pow x (y - 1);
```

Partial application

- If your function is written in curried form, you can supply values for some of its arguments to produce a function that accepts the remaining arguments.
  - That new *partial application* function can be useful to pass to `map`, `filter`, `reduce`, o, or use in a variety of ways.

- Example:

```
- val powerOfTwo = pow 2;
val powerOfTwo = fn : int -> int
- powerOfTwo 10;
val it = 1024 : int
```

How currying is applied

- Note the type of pow:

```
- fun pow x 0 = 1
= |   pow x y = x * pow x (y - 1);
```

val pow = fn : int -> int -> int

  - What does this type mean?

- Every application of curried functions is a composition:
  - `pow 2 10` creates an intermediate function `(pow 2)` and calls it, passing it the argument `10`

ML's "real" map function (5.6.3)

- ML includes a map function, but it is in curried form:

```
fun map F [] = []
|   map F (first::rest) = (F first) :: (map F rest);
```

  - What is the type of this `map` function?

- It is done this way so that you can partially apply `map`:

```
- val absAll = map abs;
val absAll = fn : int list -> int list

- absAll [2, ~4, ~6, 8, ~10];
val it = [2,4,6,8,10] : int list

- absAll [~1, ~9, 4, ~19];
val it = [1,9,4,19] : int list
```

ML's "real" filter function

- It's really called `List.filter`
  - or, at the top of your program, write: `open List;`
    and then you can just write `filter *`
  - it is in curried form: `filter` ***`P list`***

```
- open List;
- fun nonZero(x) = x <> 0;
- filter nonZero [8, 0, 0, 2, 0, 9, ~1, 0, 4];
val it = [8,9,~1,4] : int list
```

(using open is discouraged; it clutters the global namespace)*

ML's "real" reduce functions (5.6.4)

- It's really called `List.foldl` and `List.foldr`
  - but you can just write `foldl` or `foldr`
  - each takes an initial value to start the folding with
  - in curried form: `foldl` ***`F initialValue list`***
    - `foldl` applies from the left: F z (F y (... (F a initialValue)))
    - `foldr` goes from the right: F a (F b (... (F z initialValue)))

```
- foldl op+ 0 [8, 0, 0, 2, 0, 9, ~1, 0, 4];
val it = [22] : int
- foldr op^ "??" ["hi", "how", "are", "you"];
val it = "hihowareyou??" : string
- foldl op^ "??" ["hi", "how", "are", "you"];
val it = "youarehowhi??" : string
```

foldl, foldr exercise

- Define a function `len` that uses `foldl` or `foldr` to compute the length of a list.

- Solution:

```
fun len lst =
        foldr op+ 0 (map (fn x => 1) lst);
```

Currying operators

- The common infix binary operators such as op+ aren't in curried form. But we can fix that!

- The following `curry` function wraps an un-curried two-argument function into a curried form:

```
- fun curry f x y = f(x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- val doubled = curry op* 2;
val doubled = fn : int -> int
```

Curried operator exercise

- Define a function `numZeros` that accepts a list and produces the number of occurrences of 0 in the list.
  - Use curried functions, operators, composition, and map/filter/reduce.

- Solution:
  - ```
    use "utility.sml";
    open List;
    val numZeros = length o (filter (curry op= 0));
    ```
    ```

Operator precedence

- ML has the following descending levels of precedence:
  - **infix** 7 `* / mod div`
  - **infix** 6 `+ - ^`
  - **infixr** 5 `:: @`
  - **infix** 4 `= <> > >= < <=`
  - **infix** 3 `:= o`
  - **infix** 0 `before`

- When defining an operator, you can set its precedence:

```
infix 5 --;
```

Subtleties of precedence

- Binding of a function to a parameter has high precedence
  - `fun f x::xs = []` is interpreted as
    `fun (f x)::xs = []`
  - `fun f(x::xs) = []` is better!

  - `map curry op+ 1` is interpreted as
    `(map curry) op+ 1`
  - `map (curry op+ 2)` is better!

  - Adding parentheses is always okay to remove ambiguity.

Curry / higher-order exercise

- Recall our past exercise about Pascal's triangle:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

- Modify our function `triangle` to use curried functions, the "real" map function, composition, etc.
  - `triangle 6` produces `[[1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1], [1,5,10,10,5,1]]`

triangle curry solution

```
(* returns n choose k *)
fun combin n k =
    if k = 0 orelse k = n then 1
    else if k = 1 then n
    else combin (n-1) (k-1) + combin (n-1) k;

(* Returns the first n levels of Pascal's triangle.
   This version uses currying, real map,etc.*)
fun triangle n =
   map (fn(r) => map (combin r) (1--r)) (1--n);
```