Higher-Order Programming

Recall: a function is *higher order* if it takes a function as input, or returns a function as output. Otherwise, it is *first order*

```
map square [1,2,3,4] => [1,4,9,16]
map hd [[1,2],[3,4],[5,6]] => [1,3,5]
map tl [[1,2,3],[4,5,6]] => [[2,3],[5,6]]
```

Anonymous (unnamed) functions are like lambda functions in Scheme:

Scheme: `(lambda(x)(* x x))`

ML: `fn x => x*x`

```
map (fn x=>x*x)[1,2,3,4] => [1,4,9,16] :int list
map (fn x=>x+1)[1,2,3,4] => [2,3,4,5] :int list
map (fn l => hd(tl l)) [[1,2,3],[4,5,6]]
         =>[2,5] : int list
```


Higher-Order Programming (Cont'd)

Returning functions as values:

```
fun compose(F,G) = (fn x => F(G(x)));
compose(hd,tl) = (fn x => hd(tl(x)));
compose(hd,tl)[1,2,3,4] => 2
map(compose(hd,tl)) [[1,2,3],[4,5,6]] => [2,5]
```


Functions - Cont'd

ML takes functions of just one parameter(!!)

What about `fun add x (y:int) = x + y;`

val add = fn: add → int → int

ML transforms functions of several parameters to a composition of one-parameter functions, using *currying*:

```
fun add x y =
  fn y => (fn x => (x + y))
```

So, we don't have to provide all parameters:

```
add 3;
```

val it = fn : int → int

```
it 4;
```

val it = 7 : int

Functions (Cont'd)

We have seen curried functions or functions with tuple as a parameter. They are "almost" equivalent, but have different types:

```
fun exponent1(x,0) = 1.0
|   exponent1(x,y) = x * exponent1(x,y-1);
```

val exponent1 = fn: real*int → real

```
fun exponent2 x 0 = 1.0
|   exponent2 x y = x * exponent2 x (y-1);
```

val exponent2 = fn: real → int → real

Can easily convert between curried and uncurried notations:

```
fun curry F (x1,x2,x3) = F x1 x2 x3;
```

val curry = fn: ('a → 'b → 'c → 'd) → 'a * 'b * 'c → 'd

```
fun uncurry F x1 x2 x3 = F(x1,x2,x3);
```

val uncurry = fn : ('a * 'b * 'c → 'd) → 'a → 'b → 'c → 'd

Function `reduce`

How do we write a function to add all elements of the list?

```
fun sum_all [] = 0
|  sum_all(a::y) = a + sum_all(y);
```

And to multiply all elements of the list?

```
fun product_all [] = 1
|  product_all(a::y) = a * product_all(y);
```

We want to generalize such functions (REUSE!).

```
fun reduce f [] v = v
|  reduce f (a::y) v = f(a, reduce f y v);
```

Now,

```
fun add(x,y) :  int = x+y;
multiply(x,y) :  int = x*y;
reduce add [1,2,3,4,5] 0;
  val it = 15 : int
reduce multiply [1,2,3,4,5] 1;
  val it = 120 : int;
```


How does `reduce` work?

Winding Phase:

```
reduce add [1,2,3] 0 = add (1, reduce add [2,3] 0)
reduce add [2,3] 0 = add (2, reduce add [3] 0)
reduce add [3] 0 = add (3, reduce add [] 0)
```

Unwinding Phase: `reduce add [] 0 = 0`

```
reduce add [3] 0 = add 3 0 = 3
reduce add [2,3] 0 = add 2 3 = 5
reduce add [1,2,3] 0 = add 1 5 = 6
```

In general case, `reduce f` $[l_1, l_2, ...l_n]$ `v =`
`f` l_1 `(f` l_2 `( ...  (f` l_n `v) ...))`

Creative use of `reduce`

Can define a lot of functions using `reduce`, like `append`, `length`, etc.

How to do that:

1. write the original function, e.g., `length`
2. `v` is whatever the function returns on base case
3. The problem is to write `f`
   (a) Use definition of `reduce` and example to figure out what `f` is supposed to produce
   (b) Write a (named or unnamed) function `f`.

Writing `length` in terms of `reduce`

```
fun length [] = 0
|  length (_::y) = 1+length(y);
```

So, `v` is 0.

Now, try `length [1,2,3]` using `reduce`:
`f(1, f(2, f(3, 0)))`:
We want f(3, 0) to return 1
f(2, f(3, 0)) to return 2, and
f(3, f(2, f(3, 0))) to return 3.

So, function `f` should take 2 parameters and return second + 1:
`fn x y => y+1`

Therefore, `length` in terms of `reduce` is:
`length L = reduce (fn x y => y+1) L 0`

reduce_if even in terms of reduce

Function reduce_if even L should return [] when L is an empty list. So, v is [].

Now, try reduce_if even [2,5,3,4] using reduce:

f(2, f(5, f(3, f(4, [])))):

We want f(4, []) to return []

(since 4 is even)

f(3, f(4, [])) to return [3],

f(5, f(3, f(4, []))) to return [5,3], and

f(2, f(5, f(3, f(4, [])))) to return [5,3]

So, what is f? f takes an element and and a list, checks if the element is even, and if not, cons it to the list:

```
fun temp a y = if even(a) y else a::y;
```

reduce_if even in terms of reduce:

```
reduce_if even L = reduce temp L [];
```


Dealing with Operators

Operators + and * can be treated as the named functions op + and op *. These can be passed to other functions:

```
reduce op + [1,2,3,4,5] 0;
val it = 15 :  int

reduce op * [1,2,3,4,5] 1;
val it = 120 :  int
```


Functional Programming in ML Review

- Primitive types, records, lists, tuples
- Programming in ML:
  - Recursive functions
  - Pattern matching
- Type definition
  - Recursive datatypes
- String manipulation
- Type checking:
  - Type variables
  - Parametric polymorphism
  - Overloading
  - Eqtypes
- Higher-order programming
- Exception handling

ML - Review

Covered chapter 8 and 9.1-9.5 of Sethi

- ML is a strongly typed language which does static typechecking.
- An expression is not correct if ML cannot figure out its type.
- Differentiates between records (heterogeneous) and lists (homogeneous).
- Allows parametric polymorphism - ability to define functions to operate on data of any type and still be able to type check them.

ML - Review (cont'd)

- Uses structural equivalence.
- Has type variables.
- Overloading, or ad-hoc polymorphism, conflicts with parametric.
- Allows definition of new types, as enumerated constants or as recursive types.
- Has pattern matching in functions.
- Uses pattern matching to process user-defined types.

ML vs Scheme - Similarities

- Functional languages
- Interpreted, although compilers are also available
- Can evaluate expressions
- Can provide bindings
- Can define named and unnamed functions.
- Can operate on lists
- Does implicit memory management

But Scheme has no typechecking (Scheme fails "dynamically"):

```
(define (sum x) (cond (null?  x) 0)
  (else (+ (car x) (sum (cdr x)))))
(sum '(1 2 "de"))
```

ML users pay a price: user-defined types, type variables, parametric polymorphism, overloading...

food for slide eater

food for slide eater