CS 421 Lecture 16: Functional Programming

- Midterm post-mortem
- Lecture outline
  - Functional programming
  - Higher-order functions

Midterm post-mortem

- No grades yet
  - Probably Monday or Tuesday
- Each problem has been solved correctly
  - By at least two students from random sample of 5-6 exams
- Very few people finished all of the problems
  - Time was a factor
- Not many common mistakes

Midterm post-mortem

- Binary tree traversal
  - "In an in-order traversal, the left child is considered first, then the node itself, then the right child."

```
let rec iot t = match t with
    Empty -> []
  | Node(n, l, r) -> (iot l) @ [n] @ (iot r)
```

Midterm post-mortem

- Consider the following grammar:

```
A -> int | '(' B ')'
B -> A C
C -> '+' A C | ""
```

Midterm post-mortem

- Extra credit problem:
  - Functional programming in SCHEME

```
(define (append_lists lst1 lst2)
    (cond (equal lst1 nil)
        lst2
        (cons (car lst1)
              (append_lists (cdr lst1) lst2))))
```

Midterm post-mortem

- Questions?

History of functional languages

- LISP, APL (1960)
- ML (1976) – Milner, “A theory of type polymorphism in programming”
- SASL (1976) – lazy evaluation
- SCHEME (1975) – Guy Steele – dialect of LISP with higher-order functions
- Standard ML, CAML (1980’s)
- Erlang (1987) – Ericsson
- Haskell (1990) – lazy evaluation
- Python, …

Functional languages

- Definition:
  - Expressions (rather than statements)
  - Absence of side effects
  - “Large values”
- Essential:
  - Dynamic memory allocation
  - Recursion
- Optional
  - Static type checking with polymorphic types (ML, Haskell)
  - **Higher-order functions**, a.k.a. “functions as values” (Scheme, ML, Haskell, …)
  - Lazy evaluation (Haskell)

Higher-order functions

- Functions are a type of value (“first-class functions”)
  - Define anonymously
  - Pass as arguments
  - Bind to names
  - Assign to variables
  - Return from functions

Example

- Composition (math function)

  `(g ∘ f)(x) = g(f(x))`

Anonymous functions in Ocaml

- Notation:
  - "`fun x -> e`" – Ocaml expression whose value is a function
  - "`let f = fun x -> e`" is equivalent to "`let f x = e`"
- Examples:

```
(fun x -> x + x) 4;;
```

```
let f x y = x + y
```

```
let f (x,y) = x + y
```

Passing functions as arguments: map

- Higher-order functions in List module:

```
map : (α->β) -> α list -> β list
map f [x1;x2;…;xn] = [f x1;f x2;…;f xn]
```

(i.e. map f $[x_1;x_2;\ldots;x_n]$ = $[f\ x_1;f\ x_2;\ldots;f\ x_n]$)

E.g.,

```
let lis = [1;2;3;4]
let incr x = x + 1 (let incr = fun x -> x + 1)
map incr lis
=> [2;3;4;5]
```

or equivalently:

```
map (fun x -> x + 1) lis
```

"Correspondence principle" – doesn't matter if value is named or not

Passing functions as arguments: fold_right

```
fold_right : (α->β->β) -> (α list) -> β -> β
```

fold_right f [x_1;x_2;…;x_n] z = f x_1 (f x_2 (… (f x_n z)…))

```
fold_right (fun x y -> x + y) lis 0
[1;2;3;4] => 10
```

Note: can use "(+)" for function argument:

```
(+) x y ≡ x + y
```

Fold_right

```
fold_right (fun x -> fun y -> x :: y) lis []
=> lis

fold_right (fun x -> fun y -> x :: y) lis lis
=> lis @ lis

fold_right (fun x -> fun y -> (x + (hd y))::y) lis [0]
[1;2;3;4] => [10;9;7;4;0]

fold_right (fun x -> fun (y::ys) -> (x + y)::ys) lis
  [0]
=> ??
```

Map as fold_right

- Map is a special case of fold_right:

```
map f lis = fold_right
            (fun x -> fun y -> f x::y) lis []
```

Example

- Define `f`, `z` such that `fold_right f lis z` = the pair of lists `(l1,l2)` where `l1` contains the elements of `lis` that are < 0 and `l2` contains the rest

```
f = fun x ->
      fun (l1, l2) ->
        if x < 0
        then (x::l1, l2)
        else (l1, x::l2)

z = ([],[])
```

Fold_left

```
fold_left (α->β->α) -> α -> β list -> α
```

fold_left f [x_1;x_2;…;x_n] z = f(…(f (f z x_1) x2)…) x_n

```
fold_left (+) lis 0 => sum of lis
```

Example

- Define `mapplusone` $[x_1;x_2;\ldots;x_n] = [x_1+1;x_2+1;\ldots;x_n+1]$

```
let rec mapplusone lis =
   if lis = [] then []
   else (hd lis)+1 :: mapplusone (tl lis)

let rec map f lis =
   if lis = [] then []
   else f (hd lis) :: map f (tl lis)
```

Defining higher-order functions

```
let rec map f lis =
   if lis = [] then []
   else f (hd lis) :: map f (tl lis)

let rec fold_right f lis z =
   if lis = [] then z
   else f (hd lis) (fold_right f (tl lis) z)
```

Defining higher-order functions

```
map : (α->β) -> α list -> β list

let mapincr = map incr;;

mapincr : int list -> int list
```

Understanding higher-order functions

- Two approaches: substitution, or environment/closure model

- Consider: `let addone = map (fun x -> x+1)`
- Returns:

```
fun lis -> if lis = [] then []
           else f (hd lis)::map f (tl lis)
```

- But this has “f” as a *free variable*.

- Question: when `addone` is applied, where does the value of f come from?

Substitution model

- Replace free variable with its value

```
map (fun x -> x+1) =
fun lis -> if lis = [] then []
           else (fun x -> x+1)(hd lis)
                :: map (fun x -> x+1) (tl lis)
```

- Note: no free variables anymore

Environment/closure model

- Put free variables in a data structure called an *environment*:

```
{f → fun x -> x + 1}
```

- Keep expression and environment together in a pair:

```
(fun lis -> if lis = [] then []
            else f (hd lis)::map f (tl lis),
    {f → fun x -> x + 1})
```

- This pair is called a *closure*.
  - After applying map to function, the value is always kept in the form of the closure, never just the expression.

Next lecture

- More map & folding examples
- Expression evaluation