CS 421 Lecture 16

- Functional programming
- Higher-order functions

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 ("lexical scope")
- Standard ML, CAML (1980's)
- Erlang (1987) – Ericsson
- Haskell (1990) – lazy evaluation
- Python, ...

Functional languages

- Expressions (rather than statements)
  - Absence of side effects
  - "Large values"
- Dynamic memory allocation
- Recursion
- Static type checking with polymorphic types (ML, Haskell)
- **Higher-order functions**, aka "functions as values" (Scheme, ML, Haskell, Python, ...)
- Lazy evaluation (Haskell)

Higher-order functions

- Functions are a type of value ("first-class functions")
  - Define anonymously
  - Bind to names
  - Pass as arguments
  - Place in lists
  - Return from functions

Anonymous functions in Ocaml

- Notation: "fun x -> e" – Ocaml expression whose values is a function.
- "let f = fun x -> e" is equivalent to "let f x = e"
- *As with any other name, after defining, it is interchangeable with its value.*

```
# (fun a -> a+1) 4;;
5
# let incr = fun a -> a+1 ;;
# incr 4;;
5
```

↑ `# let incr a = a+1;;`

```
(fun a -> fun b ->
    a+b) 4 5;;

(fun (a,b) -> a+b))
              (4,5);;
```



```
let add = fun a -> fun b -> a+b;;
  = let add a b = a+b
```

equiv

```
let adduc (a,b) = a+b;;   adduc (4,5)..
    let adduc = fun (a,b) -> a+b
```

equiv

Function types

- Type of "fun x -> e" is the same as the type of f in "let f x = e"

```
# let add (a,b) = a+b ;;
fun = add : int * int -> int

# let add = fun (a,b) -> a+b ;;
fun = add : int * int -> int
```

Passing functions as arguments

Higher-order functions in List module:

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

applies a function to each element of a list –

map f [x_1;x_2;...x_n] = [f x_1;f x_2;...f x_n]

E.g.
```
let lis = [1;2;3;4]
let incr = fun x -> x + 1
map incr lis => [2;3;4;5]
```

or equivalently

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

Passing functions as arguments

fold = "reduce"

```
fold_right f [x1;x2;...xn] z
    = f x1 (f x2 (...(f xn z)...))
fold_right : (α->β->β)->(α list)->β->β

fold_right (fun x y -> x+y) lis 0 => 9
```

lis = [2; 3; 4]

α = int, β = int

add 2 (add 3 (add 4 0)) = 9

fold_right (fun x y → (string_of_int x)^y) lis "" ⇒ "234"

α = int, β = string

cat 2 (cat 3 (cat 4 "")) — "4", "34", "234"

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

Passing functions as arguments

```
fold_right (fun x -> fun y -> x :: y) lis []
  => lis
fold_right (fun x -> fun y -> x :: y) lis lis
  => lis @ lis
fold_right : (α → β → β) → α list → β → β
 (fun x -> fun y -> (x + (hd y))::y) lis [0]
  [1;2;3;4]   => [10;9;7;4;0]
fold_right :   α = int      β = int list
  (fun x -> fun (y::ys) -> (x + y)::ys) lis [0]
```

Passing functions as arguments

- 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 = []

Passing functions as arguments

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

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

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

Defining higher-order functions

Easiest way to define higher-order function: write a specific example explicitly, then abstract away function or functions (i.e. give function a name and pass it as an argument).

To write map: First, define function to convert a list of ints to a list of strings:

```
let rec mapinttostring x = match x with
    [] -> []
  | x::xs -> string_of_int x ::
                (mapinttostring xs)
```

Abstract away "string_of_int":

```
let rec map string_of_int x = ...
  ... same, except  map string_of_int xs
```

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)
```

Partial application of functions

When functions are <u>curried</u>, can apply to first argument only, yielding a function.

```
# let add = fun x -> fun y -> x+y ;;
# (add 3) 4;;
  7
# let add3 = add 3;;
# add3 4;;
  7
# let mapinttostring = map string_of_int ;;
```

let adduc = fun (x,y) -> x+y;;

~~adduc (3,)~~

add returns a function as its value

Understanding higher-order functions

Two approaches: Substitution, or environment/closure model.

Basic question: What happens when a function is applied?

- Evaluate body of function, but where do values of ~~arguments~~ variables come from?
  - Substitute actual parameters into the body of the function (ie. create a new function body), or
  - Create a table (called an "environment") mapping formal parameters to actuals.

Substitution is simpler "mathematically;" environment approach is more realistic.

Understanding higher-order functions

Consider: let add1 = 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 add1 is applied, where does the value of f come from?

Substitution model

Replace occurrences of formal parameter with value of actual parameter:

```
   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 any more.)

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 the function, the value is always kept in the form of the closure, never as just the expression.