CS 421 Lecture 18: More examples of higher-order functions

- Lecture outline
  - Combinator programming
  - Representing sets as higher-order functions
  - Representing pairs as higher-order functions
  - Building comparators using higher-order functions
  - Environment/closure model

Review: combinator-style programming

- Can write complex programs by defining a library of higher-order functions and applying them to one another (and to first-order or built-in functions).
- Advantage: ease of creating programs – programs are just expressions

- Example: build a parser by writing "parser combinators."

Review: parser combinators

- Define a parser to be a function from token list -> (token list) option.
  - Idea is to define functions that build parsers, rather than building parsers "by hand."
- Parser to recognize a single token:

```
let token s = fun cl -> if cl=[] then None
                  else if s=hd cl then Some (tl cl)
                  else None;;


let parsex = token 'x';;
```

Review: parser combinators

- "Combinators" to combine parsers into larger parsers:

```
let (++) p q = fun cl -> match p cl with None -> None
                                       | Some cl' -> q cl';;
```

```
let (||) p q = fun cl -> match p cl with None -> q cl
                                       | Some cl' -> Some cl';;
```

```
let rec parseA cl = ((token 'a' ++ parseB) || token 'b') cl
   and parseB cl = ((token 'c' ++ parseB) || parseA) cl;;
```

Representing sets as higher-order functions

- <u>Def</u>. A *set* is a function from values to bool.

```
type intset = int -> bool
```

- E.g.:

```
{2} = fun x -> (x=2)
{2,3} = fun x -> (x=2) or (x=3)
```

- Set operations:

```
(* member: int -> intset -> bool *)
let member n s =



(* emptyset: intset *)
let emptyset =
```

Representing sets as higher-order functions

```
(* add: int -> intset -> intset *)
let add n s =

(* union: intset -> intset -> intset *)
let union s1 s2 =

(* intersection: intset -> intset -> intset *)
let intersection s1 s2 =


(* remove: int -> intset -> intset *)
let remove n s =
```

Representing sets as higher-order functions

```
(* complement: intset -> intset *)
let complement s =



(* intsAbove: int -> intset *)
let intsAbove n =
```

- Note: cannot list elements

Representing pairs as higher-order functions

- Def. A *pair* is a value p with a constructor pair: α -> β -> pair, and functions fst: pair -> α and snd: pair -> β such that fst(pair a b) = a and snd(pair a b) = b.
- Pair operations:

```
let pair a b =
```

```
let fst p =
```

```
let snd p =
```

Building comparators using higher-order functions

- Def. A *comparator* is a function of type α * α -> bool.
  - E.g., (>) and (=) are comparators

- Can build specific comparators, e.g.:

```
fun lexorder2 (x,y) (x',y') = x<x' or (x=x' & y<y');;

lexorder2 ('a','b') ('a','c')

lexorder2 ('a','z') ('b','a')

lexorder2 ('b','b') ('a','c')
```

Building comparators using higher-order functions

- But it's more fun to build them using higher-order functions:

```
let or_comp comp1 comp2 = fun (x, y) ->
        (comp1 (x, y)) or (comp2 (x, y))


let lte = or_comp (<) (=)


let and_comp comp1 comp2 = fun (x, y) ->
        (comp1 (x, y)) & (comp2 (x, y))
```

Building comparators using higher-order functions

```
let lex_comp comp1 comp2 =
  fun (x,y) (x',y') -> comp1 (x, x')
                       or (x=x' & comp2 (y, y'))


let lexorder2 = lex_comp (<) (<);;
```

Building comparators using higher-order functions

```
let lex_comp_list comp =
  let rec aux lis1 lis2 = match (lis1, lis2) with
      ([], _) -> true
    | (_, []) -> false
    | ((x::x'), (y::y')) -> comp x y
                            or (x=y & aux x' y')
  in aux;;



let alphalex = lex_comp_list (<);;
```

Evaluating expressions

- Substitution model
  - Substitute “free” occurrences of a variable with the value of the formal parameter
- Environment model
  - Pass environment as an extra argument

Environment

- Record what value is associated with a given variable
  - It is a function var -> value
- Central to the semantics and implementation of a language
- Its maintenance depends on the language
  - Lexical vs. dynamic scoping
- Notation:

$$\rho \; \{x_1 \rightarrow v_1,\; x_2 \rightarrow v_2,\; \ldots,\; x_n \rightarrow v_n\}$$

$$x_i = x_j \;\Leftrightarrow\; i = j$$

Environment

- Example

```
let ρ be {x -> 3, z -> "hi", w -> []}
ρ(x) = 3
ρ(z) = "hi"
ρ(k) = undefined / error
```

- Environment update

```
ρ[x -> 10] = {x -> 10, z -> "hi", w -> []}
ρ[k -> true] = {x -> 10, z -> "hi", w -> [], k -> true}
ρ[k -> true](k) = true
```

Building the environment

```
  ρ
let x = e
  ρ[x->v_e] (v_e is the value e evaluates to in ρ)
```

```
  ρ
let x = e1
  ρ[x->v_e1]
in e2      (evaluate e2 in ρ[x->v_e1])
  ρ
```

Example

```
let x = 5

let y = x + 6

let x = x + y

let x = 3

in x+y
```

Functions are values, too

- What do we store in the environment for a function variable?
  - A “closure”: triple <x, e, ρ>

- Function application f(e’) in environment ρ’
  - Evaluate f in ρ’ to a closure <x, e, ρ>
  - Evaluate e’ in ρ’ to a value v
  - Evaluate e in ρ[x->v]

Example

```
let x = 5

let f y = x + y

let x = 8

f x
```