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

- Combinator programming – "parser combinators"
- Representing sets as higher-order functions
- Representing pairs as higher-order functions
- Building comparators using higher-order functions

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: easy of creating programs – programs are just expressions

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

Parser combinators

Def A parser is a function from token list -> (token list) option.

Idea is to define functions that build parsers, rather than building parsers "by hand."

E.g. 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';;
parsex ['x'];;
parsex['a'];;
```

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 parsexy = token 'x' ++ token 'y'
parsexy ['x', 'y']
parsexy ['x', 'z']
```

Parser combinators

```
let (||) p q = fun cl ->  match p cl with None -> q cl
                                         | Some cl' -> Some cl';;
let parsexyorz = parsexy || token ‘z’
parsexyorz[‘x’, ‘y’]
parsexyorz [‘z’]
```

Parser combinators

Put this together to define parser for grammar:

```
A  ->  aB  | b
B  ->  cB  |  A
```

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

```
parseA ['a';'c';'c';'a';'b‘]
```

Representing sets as higher-order functions

Def. 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.

let pair a b =

let fst p =

let snd p =

Building comparators using higher-order functions

Def A *comparator* is a function of type $\alpha * \alpha$ -> bool.

E.g. (>) is a comparator.

(=) is a comparator.

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 (<);;
```