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

<u>Def</u> A parser is a function from $\boxed{\text{token list -> (token list) option}}$.

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

type α option =
Some α | None

E.g. Parser to recognize a single token:

token : char → $\underbrace{\text{(token list → (token list) option)}}_{\text{parser}}$

```
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'];;
```

⇒ Some []

⇒ None

Parser combinators

"Combinators" to combine parsers into larger parsers:

++ : parser → parser → parser

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

```
let parsexy = token 'x' ++ token 'y'
parsexy ['x', 'y']   ⇒ Some []
parsexy ['x', 'z']   ⇒ None
```

double (token 'x')
['x', 'x'] ⇒ Some []

let double p = fun cl → match p cl with None → None
| Some cl' → p cl';;

or

= p ++ p

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

{n | n > 10} = fun x -> x>10

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 = s n ;;
(* emptyset: intset *)
let emptyset = fun x -> false ;;
```

Representing sets as higher-order functions

(* add: int -> intset -> intset *)

let add n s = fun x -> s x or x = n ;;

(* union: intset -> intset -> intset *)

let union s1 s2 = fun x -> s1 x or s2 x ;;

(* intersection: intset -> intset -> intset *)

let intersection s1 s2 = fun x -> s1 x and s2 x ;;

(* remove: int -> intset -> intset *)

let remove n s = fun x -> s x && x ≠ n

add 5 (add 3 $\underbrace{\text{emptyset}}_{\text{fun } x \to \text{false}}$)

fun x -> (fun x -> false) x or $\cancel{x}$=3

= (fun x -> x = 3)

fun x -> (fun z -> z = 3) x or x = 5

fun x -> x = 3 or x = 5

Representing sets as higher-order functions

(* complement: intset -> intset *)

let complement s = fun x -> not (s x);;

(* intsAbove: int -> intset *)

let intsAbove n = fun x -> x > n

Advantages: - Don't need recursion to define set op's

- Infinite sets

Disadvantage: - Efficiency?

Consider: add 3 (add 3 (add 3 (add 2 empty)))

[Note: cannot list elements]

Representing pairs as higher-order functions

<u>Def</u> 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 = fun f → f a b

let fst p = p (fun x → fun y → x)

let snd p = p (fun x → fun y → y)

~~pair 3 'a'~~

(3, 'a') ✓

p = pair 2 3 = fun f → f 2 3

fst p = (fun f → (f 2) 3)(fun x → fun y → x)

= ((fun x → fun y → x) 2) 3

= (fun y → 2) 3 = 2

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:

or-comp : $(\alpha * \alpha \to bool) \to (\alpha * \alpha \to bool) \to (\alpha * \alpha \to bool)$

```
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 (<);;
```

Compare pairs $(x, y) < (x', y')$ if x, x' are lists and $x < x'$ lex.

let pairlex p1 p2 = alphalex (fst p1) (fst p2)

Preview of next lecture

Function objects in o-o languages = an object with one operation, usually called apply.

```
class PlusOne {
    int apply (int i) { return i+1; }
}

(new PlusOne().apply(3)) => 4
```

```
class Plus {
    int x;
    public Plus (int x) { this.x = x; }
    public int apply (int y) { return x+y; }
}

Plus add3 = new Plus(3);
        {
          add3.apply(4)     => 7
          {
```