CS 421 Lecture 17 – Higher-order functions

- Using fold_right
- Expression evaluation via substitution
- Short examples
- Combinator-style programming

fold_right

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

Use `fold_right` to remove all negative elements from a list:

```
fold_right ____________________ lis ____
```

Defining higher-order functions

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

Evaluation of expressions

Use substitution model: *in function calls, substitute actual parameter for formal parameter in body of function.*

- Details on following slides:
  - Expressions: constants, function definitions (fun x -> e), application of built-in functions, if, application of user-defined functions
  - let expressions syntactic sugar for function application; top-level definitions implicitly in let
  - Will handle recursive functions after break; also will discuss closure model after break
  - Key feature of substitution model: never evaluate expressions that have "free" variables; e.g. when evaluating e1+e2, e1 and e2 will consist solely of constants; when evaluating fun x -> e, the only "free" variable in e is x.

Evaluation of expressions

Evaluate expression:

- Constant n (int, bool, string, list, ..) $\Rightarrow$ n
- Abstraction fun x -> e
- Application of built-in operator: e1 + e2
- if e1 then e2 else e3

Evaluation of expressions (cont.)

- Application of user-defined function: e1 e2

Example of evaluation

```
(fun x -> fun y -> x+y) 1 2
```

Example of evaluation

(fun x -> fun y -> x y) (fun y -> y 4) (fun z -> z+1)

Short examples - Currying

- Can define two-argument functions in two ways:
  - Curried: let f x y = … x … y …
    (or, let f = fun x y -> … x … y …
    or, let f = fun x -> fun y -> … x … y …)
  - Uncurried: let f (x,y) = … x … y …
    (or, let f = fun (x,y) -> … x … y …
    or, let f = fun p -> … (fst p) … (snd p) …)

Sometimes want to use the "same" function both ways…

Short examples - Currying

- Can use higher-order function to turn curried function to uncurried form, and vice versa

Short examples – reversing arguments

Given f: $\alpha \to \beta \to \gamma$, produce f_R: $\beta \to \alpha \to \gamma$, s.t.

f_R x y = f y x

Short examples – applying function twice

Given f: $\alpha \to \alpha \to \alpha$, want ff: $\alpha \to \alpha \to \alpha$ such that
ff x = f (f x)

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

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."

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']

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‘]
```