CS 421 Lecture 17 – Functional programming

- Using fold_right and fold_left
- Expression evaluation
  - Substitution model
  - Scope of definitions
- “Simple” examples
- Combinator programming

fold_right

$$\texttt{fold\_right f } [x_1; x_2; \ldots x_n] \texttt{ x}$$

$$= \texttt{f } x_1 \ (\texttt{f } x_2 \ (\ldots(\texttt{f } x_n \ \texttt{z})\ldots))$$

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

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

```
fold_right ______________________ lis ____
```

fold_left (corrected def)

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

`fold_left f z` $[x_1; x_2; ...x_n]$

= $f(\ldots (f\ (f\ z\ x_1)\ x2) \ldots)\ x_n$

Use `fold_left` **to compute the length of lis**

`fold_left` ___________________________ _____

Use `fold_left` **to compute map f lis**

`fold_left` ___________________________ _____

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)
```

Define `fold_left`:

Evaluation of expressions

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

- No expressions with free variables evaluated
- Expressions: constants, function definitions (fun x -> e), application of built-in functions, if, application of user-defined functions
- let expressions syntactic sugar for function applic; top-level definitions implicitly in let
- Will handle recursive functions after break; also will discuss closure model after break

Evaluation of expressions

Evaluate expression without free variables:

- 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

- 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)
```

Free variables

In rule for applications, substitute v for *free occurrences* of x in e'. Need to define "free occurrence."

Def. Free occurrences of x in e are those marked with an overbar after applying free to x and e:

free x e = match e with

Example of free occurrences

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

Scope rules

- Programs introduce names via "declarations", then refer to those names in "uses." A given name can be introduced in more than one declaration, but every use corresponds to a particular declaration. The question is: which one?
- The *scope of a declaration* of a name x is the parts of the program in which a use of x refers to this declaration
- A use of a name is *in the scope of a declaration* if that use is in the scope of that declaration
- N.B. the scope of a declaration can have holes, where the declaration is covered up by another declaration of the same name.

E.g. Scope rules in Java

```
class C {
   int y
   void f (x) { … x … f … y … g … }
   void g () { … }
}

class D extends C {
   int z
   void f (x) { … x … f … y … g … }
}
```

E.g. Scope rules in OCaml

1. ```
   let x = 2
    in let f = fun x -> x+x
       in f x
   ```
2. ```
   let x = 2
   in let y = x
      in let f z = let x=3 in y+z
         in f x
   ```
3. ```
   let x = 2
   in let add = fun x -> fun y -> x+y
      in let addx = add x
         in let x = 3 in  addx 1
   ```
   ```

Scope rules in OCaml

Scope rules are implied by expression evaluation rules.

Declarations are just function definitions fun x ->e

Scope of this declaration of x is exactly the free occurrences of x in e.

(Put differently, a use of a variable x is in the scope of the closest enclosing function definition for which x is the formal parameter.)

This is called *static scope*, or *lexical scope*, because the declaration corresponding to any use is known statically (before run time).

The scope rule of Lisp

- In Lisp, the declaration associated with a use of a variable x is determined as follows: at run-time, the most recent function application that has x as formal parameter (and which is still on the stack) gives the declaration of x.
- Lisp vs. Ocaml:

```
let h f = let x = 3 in f x
let f x = let g y = x + y in h g
f 5 => ?
```

The scope rule of Lisp

- In Lisp, the declaration associated with a use of a variable x is determined as follows: at run-time, the most recent function application that has x as formal parameter (and which is still on the stack) gives the declaration of x.
- Lisp vs. Ocaml:

```
let h f = let x = 3 in f x
let f x = let g y = x + y in h g
f 5
```

=> ?

“Simple” examples

Currying

“Simple” examples

Reversing arguments

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

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

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;;

parseA ['a';'c';'c';'a';'b‘]
```