PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341
Section 4

Mutual Recursion, Modules, and Currying

Adapted from slides by Yuma Tou and Taylor Blau

Agenda

1. Mutual Recursion

2. ML module practice

3. Currying practice

Mutual Recursion

Say we want to write a function that takes a list and returns a bool which is true if and only if the list has alternating 0s and 1s.

- `is_alternating [0,1,0] = true`
- `is_alternating [1,0,1] = true`
- `Is_alternating [1,1,0] = false`

The idea

- `val zero = fn : int list -> bool`
- `val one = fn : int list -> bool`

Each function checks if the list begins with a zero or one, and then calls the other on the tail.

Let’s try it!

The problem

zero cannot call one, because one was defined after zero and so is not in the closure of zero.

If we reverse them, then one cannot call zero.

What do we do?

The solution

fun zero xs = …

and one xs = …

Now both functions can call each other!

Modules

Good for organization

- Can group bindings into separate modules

Good for maintaining invariants by hiding implementation details from client

- e.g. keeping rationals in lowest terms

Module practice

Remember: structure Foo :> BAR is allowed if Foo provides:

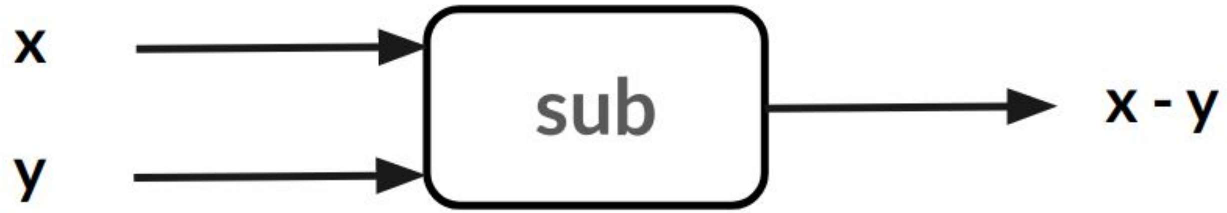
- every non-abstract type in BAR (as specified)
- every abstract type in BAR (in some way)
- every val-binding in BAR (can have more general types)
- every exception in BAR

Foo can also define things that are not defined in BAR!

Currying

Before Currying:

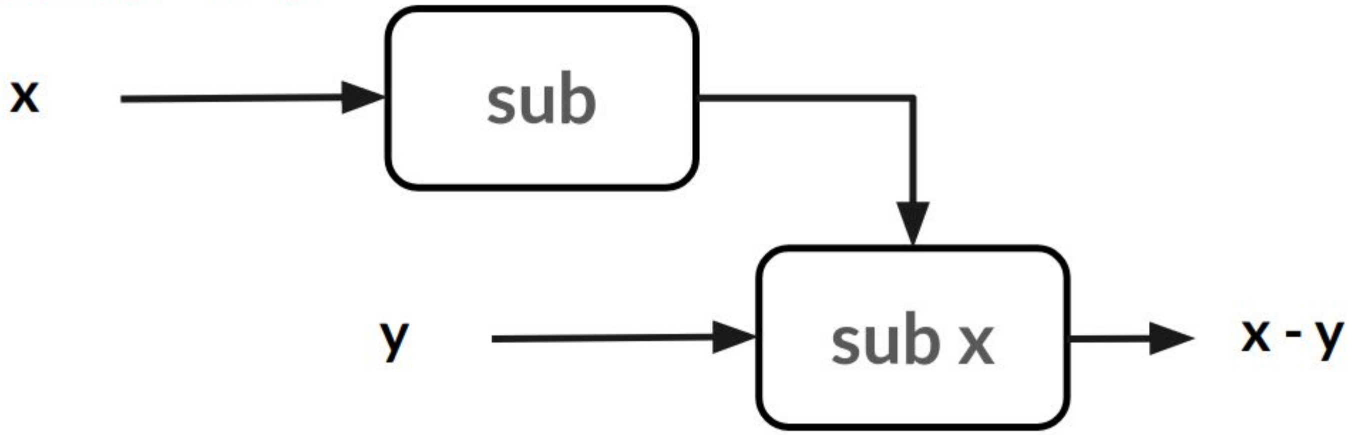
```
fun sub (x, y) = x - y
```



Currying

With Currying:

```
fun sub x y = x - y
```



Currying - Syntactic Sugar

Here are three ways to say the same thing

```
(* where e is some expression *)
fun f x y = e
val f = fn x => (fn y => e)
fun f x = fn y => e
```

Unnecessary function wrapping

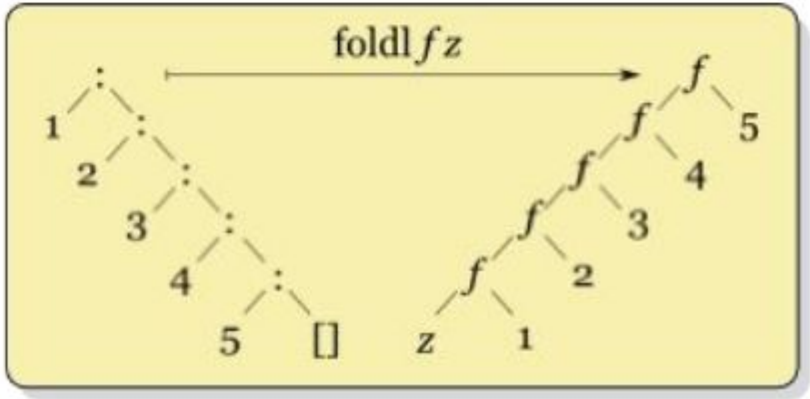
```
fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

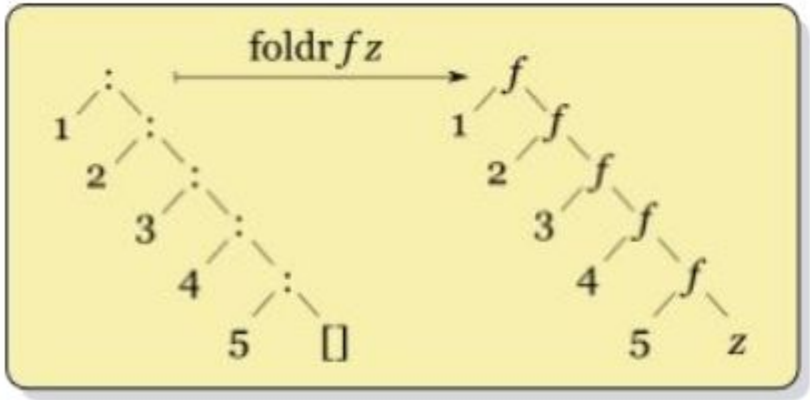
- Previously learned not to write `fun f x = g x` when we can write `val f = g`
- This is the same thing, with `fold (fn (x,y) => x+y) 0` in place of `g`

Tangent: foldr v foldl

```
List.foldl (fn (x, acc) => x+acc) 0 xs
List.foldr (fn (x, acc) => x+acc) 0 xs
```

```
0 [1, 2, 3, 4, 5]             [1, 2, 3, 4, 5] 0
```


foldl f z
:
1
2
3
4
5
[]
f
z
1
2
3
4
5


foldr f z
:
1
2
3
4
5
[]
f
z