Currying and Partial Application

and other tasty closure recipes
More idioms for closures

- Function composition
- Currying and partial application
- Callbacks (e.g., reactive programming, later)
- Functions as data representation (later)
Function composition

fun compose (f, g) = fn x => f (g x)

Closure “remembers” f and g

: ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)

REPL prints something equivalent

ML standard library provides infix operator \(\circ\)

fun sqrt_of_abs i = Math.sqrt(Real.fromInt(abs i))
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs

Right to left.
Pipelines (left-to-right composition)

“Pipelines” of functions are common in functional programming.

```fsharp
infix |> 
fun x |> f = f x

fun sqrt_of_abs i = 
i |> abs |> Real.fromInt |> Math.sqrt
```

(F#, Microsoft's ML flavor, defines this by default)
Currying

• Every ML function takes exactly one argument

• Previously encoded $n$ arguments via one $n$-tuple

• Another way: Take one argument and return a function that takes another argument and...
  – Called “currying” after logician Haskell Curry
Example

```plaintext
val sorted3 = fn x => fn y => fn z =>
    z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- Calling `(sorted3 7)` returns a closure with:
  - Code `fn y => fn z => z >= y andalso y >= x`
  - Environment binds `x` to 7
- Calling `that` closure on 9 returns a closure with:
  - Code `fn z => z >= y andalso y >= x`
  - Environment binds `x` to 7, `y` to 9
- Calling `that` closure on 11 returns `true`
Function application is left-associative

```
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

```
e1 e2 e3 e4
means
(((e1 e2) e3) e4)
```

```
val t1 = sorted3 7 9 11
```

Callers can just think
“multi-argument function with spaces instead of a tuple expression”
Does not interchange with tupled version.
Function definitions are sugar (again)

\[
\text{val sorted3} = \text{fn x} = \text{fn y} = \text{fn z} =
\text{z} \geq \text{y andalso y} \geq \text{x}
\]

\[
\text{val t1} = ((\text{sorted3 7}) \ 9) \ 11
\]

\[
\text{fun f } p1 \ p2 \ p3 \ ... = e
\]

desugars to

\[
\text{fun f } p1 = \text{fn p2} = \text{fn p3} = \ ... = \ e
\]

\[
\text{fun sorted3 } x \ y \ z = \ z \geq \ y \ andalso \ y \geq \ x
\]

Callees can just think

“multi-argument function with spaces instead of a tuple pattern”

Does not interchange with tupled version.
As elegant syntactic sugar (fewer characters than tupling) for:

```haskell
fun sorted3 x y z = z >= y andalso y >= x
val t1 = sorted3 7 9 11
```

Function application is left-associative.

Types are right-associative:

```
sorted3 : int -> int -> int -> bool
```
means
```
sorted3 : int -> (int -> (int -> bool))
```
Curried foldl

```haskell
fun foldl f acc xs = 
case xs of
    [] => acc
| x::xs' => foldl f (f(x,acc)) xs'

fun sum xs = foldl (fn (x,y) => x+y) 0 xs
```
Partial Application

fun foldl f acc xs =
    case xs of
        [] => acc
    | x::xs' => foldl f (f(acc,x)) xs'

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

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

foldl (fn (x,y) => x+y) 0
evaluates to a closure that, when called with a list xs, evaluates
the case-expression with:
    f bound to the result of foldl (fn (x,y) => x+y)
    acc bound to 0
Unnecessary function wrapping

```plaintext
fun f x = g x (* bad *)
val f = g (* good *)

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

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

(* best? *)
val sum = fold (op+) 0
```

Treat infix operator as normal function.
Iterators and partial application

fun exists predicate xs =
  case xs of
    [] => false
  | x::xs' => predicate x
      orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val hasZero = exists (fn x => x=0)

For this reason, ML library functions of this form are usually curried

- List.map, List.filter, List.foldl, ...
The Value Restriction 😞

If you use partial application to create a polymorphic function, it may not work due to the value restriction

- Warning about “type vars not generalized”
  - And won’t let you call the function

- This should surprise you; you did nothing wrong 😊 but you still must change your code.

- See the code for workarounds

- Can discuss a bit more when discussing type inference
More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function’s arguments are in the wrong order for the partial application you want?

Naturally, it is easy to write higher-order wrapper functions
- And their types are neat logical formulas

```ml
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```