More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Passing functions with private data to iterators: Done
- Combining functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming) (optional)
- Implementing an ADT with a record of functions (optional)

Combining functions

Canonical example is function composition:

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

```
fun sqrt_of_abs i = Math.sqrt o Real.fromInt o abs
```

As in math, function composition is "right to left"

- "take absolute value, convert to real, and take square root"
- "square root of the conversion to real of absolute value"

"Pipelines" of functions are common in functional programming and many programmers prefer left-to-right

- Can define our own infix operator
- This one is very popular (and predefined) in F#

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

Another example

- "Backup function"

```
fun backup1 (f, g) = fn x =>
  case f x of
  NONE => g x
| SOME y => y
```

As is often the case with higher-order functions, the types hint at what the function does

`('a -> 'b option) * ('a -> 'b) -> 'a -> 'b`
More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:
- Passing functions with private data to iterators: Done
- Combining functions (e.g., composition): Done
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming) (optional)
- Implementing an ADT with a record of functions (optional)

Currying

- Recall 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 famous logician Haskell Curry

Example

```haskell
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```

- Calling \((\text{sorted3 } 7)\) returns a closure with:
  - Code: \(\text{fn } y \Rightarrow \text{fn } z \Rightarrow z >= y \quad \text{andalso} \quad y >= x\)
  - Environment maps \(x\) to 7

- Calling that closure with 9 returns a closure with:
  - Code: \(\text{fn } z \Rightarrow z >= y \quad \text{andalso} \quad y >= x\)
  - Environment maps \(x\) to 7, \(y\) to 9

- Calling that closure with 11 returns \(true\)

Syntactic sugar, part 1

- In general, \(e1 \ e2 \ e3 \ e4 \ldots\), means \(\ldots(e1 \ e2) \ e3 \ e4\)
- So instead of \((\text{sorted3 } 7) \ 9 \ 11\), can just write \(\text{sorted3 } 7 \ 9 \ 11\)
- Callers can just think “multi-argument function with spaces instead of a tuple expression”
  - Different than tupling: caller and callee must use same technique

Syntactic sugar, part 2

- In general, \(\text{fun } f \ p1 \ p2 \ p3 \ldots \ = \ e\),
  means \(\text{fun } f \ p1 = \text{fn } p2 \Rightarrow \text{fn } p3 \Rightarrow \ldots \Rightarrow e\)
- So instead of \(\text{val sorted3 } = \text{fn } x \Rightarrow \text{fn } y \Rightarrow \text{fn } z \Rightarrow \ldots\)
  or \(\text{fun sorted3 } x = \text{fn } y \Rightarrow \text{fn } z \Rightarrow \ldots\)
  can just write \(\text{fun sorted3 } x \ y \ z = x \Rightarrow y \quad \text{andalso} \quad y >= x\)
- Callees can just think “multi-argument function with spaces instead of a tuple pattern”
  - Different than tupling; caller and callee must use same technique

Final version

As elegant syntactic sugar (even fewer characters than tupling) for:

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

```haskell
val sorted3 = fn x => fn y => fn z =>
  z >= y andalso y >= x
val t1 = ((sorted3 7) 9) 11
```
Curried fold

A more useful example and a call to it
  – Will improve call next

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

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

Note: foldl in ML standard-library has f take arguments in opposite order

“Too Few Arguments”

• Previously used currying to simulate multiple arguments
  – But if caller provides “too few” arguments, we get back a closure
    waiting for the remaining arguments
    – Called partial application
    – Convenient and useful
    – Can be done with any curried function
  • No new semantics here: a pleasant idiom

Example

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

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

As we already know, fold (fn (x,y) => x+y) 0 evaluates to a closure that given xs, evaluates the case-expression with f bound to fold (fn (x,y) => x+y) and acc bound to 0

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

Iterators

• Partial application is particularly nice for iterator-like functions
  – Example:
    ```
    fun exists predicate xs =
      case xs of
        [] => false
      | x::xs' => predicate x
        orelse exists predicate xs'

    val no = exists (fn x => x=?) [4,11,23]
    val hasZero = exists (fn x => x=0)
    ```
• For this reason, ML library functions of this form usually curried
  – Examples: List.map, List.filter, List.foldl

The Value Restriction Appears 😞

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
```

Efficiency

So which is faster: tupling or currying multiple-arguments?

• They are both constant-time operations, so it doesn’t matter in most of your code – “plenty fast”
  – Don’t program against an implementation until it matters!
• For the small (zero?) part where efficiency matters:
  – It turns out SML/NJ compiles tuples more efficiently
  – But many other functional-language implementations do better with currying (OCaml, F#, Haskell)
    • So currying is the “normal thing” and programmers read t1 -> t2 -> t3 -> t4 as a 3-argument function that also allows partial application

More idioms

• We know the rule for lexical scope and function closures
– Now what is it good for

A partial but wide-ranging list:
• Passing functions with private data to iterators: Done
• Combining functions (e.g., composition): Done
• Currying (multi-arg functions and partial application): Done
• Callbacks (e.g., in reactive programming) (optional)
• Implementing an ADT with a record of functions (optional)

ML has (separate) mutation

• Mutable data structures are okay in some situations
  – When “update to state of world” is appropriate model
  – But want most language constructs truly immutable
• ML does this with a separate construct: references

• Introducing now because will use them for next closure idiom
• Do not use references on your homework
  – You need practice with mutation-free programming
  – They will lead to less elegant solutions

References

• New types: t ref where t is a type
• New expressions:
  – ref e to create a reference with initial contents e
  – e1 := e2 to update contents
  – !e to retrieve contents (not negation)

References example

```ml
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(*) x + 1 does not type-check *)
```

• A variable bound to a reference (e.g., x) is still immutable: it will always refer to the same reference
• But the contents of the reference may change via :=
• And there may be aliases to the reference, which matter a lot
• References are first-class values
• Like a one-field mutable object, so := and 1 don’t specify the field
**Callbacks**

A common idiom: Library takes functions to apply later, when an event occurs – examples:
- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks
- Different callbacks may need different private data with different types
- Fortunately, a function's type does not include the types of bindings in its environment
- (In OOP, objects and private fields are used similarly, e.g., Java Swing’s event-listeners)

**Mutable state**

While it’s not absolutely necessary, mutable state is reasonably appropriate here
- We really do want the “callbacks registered” to change when a function to register a callback is called

**Example call-back library**

Library maintains mutable state for “what callbacks are there” and provides a function for accepting new ones
- A real library would also support removing them, etc.
- In example, callbacks have type `int -> unit`

So the entire public library interface would be the function for registering new callbacks:

```haskell
val onKeyEvent : (int -> unit) -> unit
```

(Because callbacks are executed for side-effect, they may also need mutable state)

**Library implementation**

```haskell
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =
  cbs := f :: (!cbs)

fun onEvent i =
  let fun loop fs
    =
      case fs
        of
          [] => ()
        | f :: fs' => (f i; loop fs')
    in
      loop (!cbs)
  end
```

**Clients**

Can only register an `int -> unit`, so if any other data is needed, must be in closure’s environment
- And if need to “remember” something, need mutable state

Examples:
```haskell
val timesPressed = ref 0

val _ = onKeyEvent (fn _ =>
  timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
  onKeyEvent (fn j =>
    if i=j
      then print ("pressed " ^ Int.toString i)
      else ()
  )
```

**More idioms**

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:
- Passing functions with private data to iterators: Done
- Combining functions (e.g., composition): Done
- Currying (multi-arg functions and partial application): Done
- Callbacks (e.g., in reactive programming): Done
- Implementing an ADT with a record of functions (optional)
Optional: Implementing an ADT

As our last idiom, closures can implement **abstract data types**
- Can put multiple functions in a record
- The functions can share the same private data
- Private data can be mutable or immutable
- Feels a lot like objects, emphasizing that OOP and functional programming have some deep similarities

See code for an implementation of immutable integer sets with operations `insert`, `member`, and `size`

The actual code is advanced/clever/tricky, but has no new features
- Combines lexical scope, datatypes, records, closures, etc.
- Client use is not so tricky