We’ve covered a lot so far

- Core ML features
  - Expressions, Values, Types
- How to build complex values
  - Tuples, records, ...
- How to create, use complex types
  - Lists, trees, expressions,...
  - Pattern-matching
- How to write recursive functions

Today: Functions taking, returning functions

Recap: Functions

Two ways of writing function expressions:

1. Anonymous functions:

   Parameter (formal)  Body Exp
   \[ \text{val } \text{name } = \text{fn } x \mapsto e \]

2. Named functions:

   Parameter (formal)  Body Exp
   \[ \text{fun } \text{name } x = e \]

Recap: Function Application

Function value is: \(<\text{code, Env at definition}>\)

- \(\langle\text{formal, body-exp}, \text{Env}\rangle\)
- Called a “closure” \(f_{\text{fn}} x \mapsto e \)

“apply” the argument \(a_2\) to the function \(a_1\)

Evaluation:

1. Evaluate \(a_1\) in current \(\text{env}\) to get (function) \(\text{fn}\)
2. Evaluate \(a_2\) in current \(\text{env}\) to get (argument) \(\text{v}_2\)
3. Evaluate body \(e\) in \(\text{env} E\) extended by binding \(x\) to \(v_2\)

Example

\[ \text{fun } f \ g = \]
\[ \text{let } \]
\[ \text{val } x = 0 \]
\[ \text{in } g \ 2 \]
\[ \text{end; } \]
\[ \text{val } x = 100; \]
\[ \text{fun } h \ y = x + y; \]
\[ f \ h; \]
\[ \text{102} \]
\[ \to \ f \ y \Rightarrow 100 + y \]
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Creating, (Returning) Functions

Returning functions

val \texttt{lt} = (\texttt{fn} \ x \Rightarrow \texttt{fn} \ y \Rightarrow x < y)?

fun \texttt{lt} \ x \ y = x < y;

In general, these two are (mostly) equivalent:

\begin{itemize}
  \item val \texttt{f} = \texttt{fn} \ x1 \Rightarrow \ldots \Rightarrow \texttt{fn} \ xn \Rightarrow \texttt{e}
  \item fun \texttt{f} \ x1 \ldots xn = \texttt{e}
\end{itemize}

Parameterized “tester”

- Create many similar testers
- Where is this useful?

Remember this?

fun sort (\texttt{lt},l) =
  case \texttt{l} of
  [] => []
  | (h::t) =
    let
      \texttt{val} \ (l,r) = partition (\texttt{lt} \ h),t
    in
      \texttt{sort} (\texttt{lt},l)::h::\texttt{sort} (\texttt{lt},r)
    end;

- Use “tester” to partition list
  - Tester parameterized by \texttt{qsort “pivot” h}

Function Currying

Multiple argument functions by returning a function that takes the next argument

- Named after a person (Haskell Curry)

fun \texttt{lt} \ x \ y = x < y;

Curried version: \texttt{fun} \texttt{f} \ x1 \ldots xn = \texttt{e}

Could have done: \texttt{fun} \texttt{lt} \ (x,y) = x<y;

- But then no “testers” possible
- Must pick good order of arguments
Using parameterized testers

```haskell
fun sort (lt, l) = 
  case l of
    [] => []
  | (h::t) = 
    let
      val (l, r) = partition (h, lt)
      in
        sort (lt, l) @ h::sort (l, r)
      end;

partition
  - Takes a tester (and a list) as argument
  - Returns a pair: (list passing test, list failing test)
  - Can be called with any tester!
```

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?
  Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions
Using, (Taking) Functions

Taking functions

```haskell
fun filter (_ , []) = []
  | filter (f, h::t) = 
    if f h
      then h::filter (f, t)
    else (filter (f, t));

fun neg f = fn x => not (f x)
fun partition (f, l) = (filter(f, l), filter(neg f, l))

filter, neg, partition: higher-order functions
  - Take a any tester as argument!
```

Higher-order functions: map

```haskell
fun map f [] = []
  | map f (h::t) = (f h)::(map f t)

(map f t) == map f t
(map f t) (at 'a list) == 'b list

Type says it all!
  - Applies “f” to each element in input list
  - Makes a list of the results

(map f, t)
(map f, t)
(map f, t)
(map f, t)
```

Map examples

```haskell
fun square x = x*x

fun sq_list = map square

fun toUpper s = implode(map Char.toUpper (explode s))
```

Another pattern: Accumulation

```haskell
fun max (x,y) = if x > y then x else y;
fun listMax l = 
  let
    fun help (cur, []) = cur
    | help (cur, h::t) = help(max(cur, h), t)
  in
    help (0, l)
  end;

fun concat l = 
  let
    fun help (cur, []) = cur
    | help (cur, h::t) = help(cur, t)
  in
    help (0, l)
  end;
```

```

```
What's the pattern?

```
fold f cur l = 
  case l of
    [] => cur
    h:t => fold f (f h t)

fold f b [v,y,v_2,v_3,v_4,v_5] 
  f a f (b,v) [_______]  
  f a f (v,y) [_______]  
  f a f (v_2,v_3) [_______]  
  f a f (v_4,v_5) [_______] 
```

Examples of fold

```
val listMax = fold max 0 

val concat = fold (ops) "" 

val multiplier = fold (ops) 1 
```

Examples of fold

```
fun f l = fold (op::) [] l 
```

Funcs taking/returning funcs

Identify common computation “patterns”

- **Iterate** a function over a set, list, tree ...
- **Accumulate** some value over a collection fold

Pull out (factor) “common” code:
- **Re-use** in many different situations

Funcs taking/returning funcs

Higher-order funcs enable **modular** code

- Each part only needs **local** information

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client sort</td>
<td>list</td>
</tr>
</tbody>
</table>

- Uses meta-functions: map, fold, filter
- With locally-dependent funs (lt h), square etc.
- Without requiring implement. details of data structure

- Provides meta-functions: map, fold, filter to traverse, accumulate over lists, trees etc.
- Meta-functions don’t need client info (tester? accumulator?)

Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

- Creating, (Returning) Functions
- Using, (Taking) Functions

- Iterator, Accumul, Reuse computation pattern w/o exposing local info
**Funcs taking/returning funcs**

- Higher-order funcs enable modular code
- Each part only needs local information
- Higher-order funcs enable code composition
  - Join simpler functions to get complex functions

```haskell
fun compose (f,g) = fn x => f (g x)
fold compose (fn x => x) [f1,f2,...,fn]
```

**Functions are “first-class” values**

- Arguments, return values, bindings ...
- What are the benefits?

**Parameterized, similar functions**
  (e.g. Testers)

**Creating, (Returning) Functions**

**Iterator, Accumul, Reuse computation, pattern w/o exposing local info**

**Compose Functions:**
- Flexible way to build Complex functions from primitives.

---

**What is the deal with ‘a’?**

These meta-functions have strange types:

```
map: ('a -> 'b) -> 'a list -> 'b list
filter: ('a -> bool) -> 'a list -> 'a list
```

Why?

---

**Polymorphism**

- Poly = many, morph = kind
- ‘a’ and ‘b’ are type variables!
- For-all types: For all ‘a’, ‘b’: ‘a * ‘b -> ‘b * ‘a’
- ‘a’, ‘b’ can be “instantiated” with any type:
  w/ int,string : int * string -> string * int
  w/ char,int list : char * int list -> int list * char
  w/ int -> int,bool : (int -> int) * bool -> bool * (int -> int)

---

**Instantiation at use**

```
map: ('a -> 'b) -> 'a list -> 'b list
```

```haskell
fun f x = x + 10;
val fm = map f;
```

```haskell
fun f x = x**2 like";
val fm = map f1 ["cat", "dog", "burrito"];
```

---

**Polymorphic ML types**

- Poly = many, morph = kind
- Possible ML types:
  - tv = ‘a | ‘b | ‘c | ...
  - T = int | bool | string | char | ...
    - \( T_1 \ast T_2 \ast \ldots \ast T_n \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T \)
  - Implicit for-all at the “left” of all types
    - Never printed out, or specified
Polymorphism enables Reuse

- Can reuse generic functions:
  ```
  map : 'a * 'b → 'b * 'a
  filter : ('a → bool * 'a list) → 'a list
  rev : 'a list → 'a list
  length : 'a list → int
  swap : 'a * 'b → 'b * 'a
  ```

- If function (algorithm) is “independent” of type, can reuse code for all types!

Genericity is everywhere...

- What’s the type of this function?
  ```
  fun sort (lt, l) =  
  case l of  
  [ ] ⇒ []  
  | (h::t) ⇒  
  let  
  val (l, r) = partition ((lt h), t)  
  in  
  (sort (lt, l)) @ h @ (sort (lt, r))  
  end;
  ```

Polymorphism enables Reuse

- Can reuse generic functions:
  ```
  map : 'a * 'b → 'b * 'a
  filter : ('a → bool * 'a list) → 'a list
  rev : 'a list → 'a list
  length : 'a list → int
  swap : 'a * 'b → 'b * 'a
  sort : ('a → 'a → bool * 'a list) → 'a list
  fold : ...
  ```

- Type is instantiated for each use:

Not just functions ...

- Data types are also polymorphic!
  ```
  datatype 'a list =  
  Nil  
  | Cons of ('a * 'a list)
  ```

Datatypes with many type variables

- Multiple type variables
  ```
  datatype ('a,'b) tree =  
  Leaf of ('a * 'b)  
  | Node of ('a,'b) tree * ('a,'b) tree
  ```

- Type is instantiated for each use:

Polymorphic Data Structures

- “Container” data structures independent of type!
- Appropriate type is instantiated at each use:
  ```
  'a list
  ('a , 'b) Tree
  ('a , 'b) Hashtbl ...
  ```

- Appropriate type instantiated at use
  - No “casting” as in C++/Java
  - Static type checking catches errors early
  - Cannot add “int” key to “string” hashtable

- Generics: feature of next versions of Java,C#,.VB
Polymorphic Types
- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

Remember: ML is statically typed
- Expressions (Syntax) → Values (Semantics)
- Compile-time “Static” → Types
  1. Programmer enters expression
  2. ML checks if expression is “well-typed”
     - Using a precise set of rules, ML tries to find a unique
type for the expression meaningful type for the expr
  3. ML evaluates expression to compute value
     - Of the same “type” found in step 2

Polymorphic Type Inference
- Computing the types of all expressions
  - At compile time: Statically Typed
  - Each binding is processed in order
  - Types are computed for each binding
    - For expression and variable bound to
  - Types used for subsequent bindings

Polymorphic Type Inference
- Can have no idea what a function does,
  but still know its exact type
  - A function may never (or sometimes terminate),
    but will still have a valid type
  - Every expression accepted by ML must have
    a valid inferred type

Example 1
```
val x = 2 + 3
val y = Int.toString x
```

Example 2
```
val x = 2 + 3
val y = Int.toString x
fun inc y = x + y
```
fun foo x = 
   let 
   val (y,z) = x 
   in 
   (-y) + z 
   end;

fun cat [] = "" 
   | cat (h::t) = h^(cat t)

fun cat [] = cat [] 
   | cat (h::t) = h^(cat t)

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

fun fold f b [] = b 
   | fold f (h::t) = fold f (f (h,b)) t