Lecture 07: Programming Languages [CMPU-235]
September 21, 2017

Sameer S Pradhan
Vassar College
Standard ML
Features learned so far

- functions
- tuples
- lists
- let expressions
- OPTIONs
- records
- datatypes
- case expressions
- type synonyms
- pattern matching
- exceptions
- type variables

Today

- First class functions
- Currying
- Higher order functions
Functions are values

- Can use them **anywhere** we use values
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, ...

- First-class citizens of language, afforded all the “rights” of any other values

- Functions can **take** functions as arguments
- Functions can **return** functions as results
  - ... so functions are *higher-order*

- This is not a new language feature; just a consequence of choice we made long ago when we said “functions are values”
fun map (f, xs) =
    case xs of
    [] => []
    | x::xs' => (f x)::(map(f, xs'))

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

Map is, without doubt, in the higher-order function hall-of-fame
- The name is standard (for any data structure)
- You use it all the time once you know it:
  - saves a little space
  - but more importantly, communicates what you are doing
- Similar predefined function: List.map
  - But it uses currying (later lecture)
fun filter (f, xs) = 
   case xs of 
       [] => [] 
     | x::xs => if f x 
              then x::(filter(f, rest)) 
              else filter(f, rest) 

filter : ('a -> bool) * 'a list -> 'a list

Filter is also in the hall-of-fame
- So use it whenever your computation is a filter
- Similar predefined function: List.filter
  - Also uses currying
Higher-order functions work over any type

Great for recursive traversals over your own data structures (datatype bindings)

- E.g.: Are all constants in an arithmetic expression even numbers?

```haskell
fun true_of_all_constants(f,e) = 
    case e of 
    Constant i => f i 
    | Add(e1,e2) => 
      true_of_all_constants(f, e1) 
      andalso true_of_all_constants(f, e2) 
    | ... (* other cases omitted here *)
fun all_even e = 
    true_of_all_constants(fn x => (x mod 2) = 0,e)
```
Returning Functions

- Since functions are first-class values, can return them from other functions

- Silly example:

  ```latex
  fun double_or_triple f = 
  if f 7
  then fn x => 2*x
  else fn x => 3*x
  ```

  Has type \((\text{int} \to \text{bool}) \to (\text{int} \to \text{int})\)

  But the REPL prints \((\text{int} \to \text{bool}) \to \text{int} \to \text{int}\)
  because REPL never prints unnecessary parentheses
  and because \(\to\) is right-associative, e.g.,
  \(\text{t1} \to \text{t2} \to \text{t3} \to \text{t4}\) means \(\text{t1} \to (\text{t2} \to (\text{t3} \to \text{t4}))\)
Revisiting—Why Functional Programming?
Why *semantics* and *idioms*?

This course focuses as much as it can on *semantics* (and *idioms*)

- Correct reasoning about programs, interfaces, and compilers *requires* a precise knowledge of *semantics*
  - Not “I feel that conditional expressions might work like this”
  - Not “I like curly braces more than parentheses”
  - Much of software development is designing precise interfaces; what a PL means is a really good example

- **Idioms** make you a better programmer
  - Best to see in multiple settings, including where they shine
  - You’ll understand (insert your favorite language here) better even if I never show you equivalent idioms in that language
Okay, so why is $y\%$ of course with functional language where:

- Mutation is discouraged
- One-of types via constructs like datatypes
- Higher-order functions are very convenient

Because:

1. These features are invaluable for correct, elegant, efficient software (great way to think about computation)
2. Functional languages have always been ahead of their time
3. Functional languages well-suited to where computing is going

Most of course is on (1), so a few minutes on (2) and (3) ...
Ahead of their time

All of these were dismissed as “beautiful, worthless, slow things PL professors make you learn in school”

- Garbage collection (Java didn’t exist in 1995, LISP did)
- Generics (**List<T>** in Java didn’t exist in 2003, ‘**a list**’ in ML [1990] did)
- Higher-order functions (C# 3.0 [2007], Java 8 [2013?], Lisp [1958])
- Recursion (a big fight in 1960 about this – I’m told 😊)
- Type inference (C# 3.0)
- MapReduce (everybody)

What will the next “discovery” be?
- Maybe pattern-matching?
- “To conquer” vs. “to assimilate”
Recent Surge

- F#, C# 3.0, LINQ (Microsoft)
- Scala (Twitter, LinkedIn, FourSquare)
- Java 8
- Haskell (dozens of small companies/teams)
- Erlang (distributed systems, Facebook chat)
- OCaml (JaneStreet)
- ...

Full disclosure: SML is showing its age, but OCaml and F# are very similar
Why a surge?

Some guesses:
- Concise, elegant, **productive** programming
- Javascript, Python, Ruby helped break the Java/C/C++ hegemony
- Avoiding mutation is *the* best way to make concurrent and parallel programming easier
Is this real programming?

- “Use of languages in this course seems silly”
  - Precisely because lecture and homework focus on interesting language constructs, not on writing big programs

- “Real” programming needs file I/O, string operations, floating-point, graphics, project managers, testing frameworks, threads, build systems, ...
  - Functional languages have all that and more

- Note: If we used Java as an example language instead of ML, you’d develop mistaken idea that Java is silly!
## Orthogonal Language Features

<table>
<thead>
<tr>
<th></th>
<th>Dynamically typed</th>
<th>Statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>—</td>
<td>SML</td>
</tr>
<tr>
<td>Object oriented</td>
<td>Python</td>
<td>Java</td>
</tr>
</tbody>
</table>

Could fill in missing cell with Racket or Scheme.
No such thing as a “best” PL

There are good general design principles for PLs

A good language is a relevant, crisp interface for writing software

Software leaders should know PL semantics and idioms

Learning PLs is not about syntactic tricks for small programs

Functional languages have been on the leading edge for decades
  – Ideas get absorbed by the mainstream, but very slowly
  – Meanwhile, use the ideas to be a better C/Java/PHP hacker
Infix Operators
Cons is *infix*, not *prefix*

```ocaml
datatype 'a list = nil | :: of 'a * 'a list

val lst : int list = 1::2::3
```

- Why do we get to write cons inside as `1::2::3` instead of outside as `::(1,::(2,::(3,nil)))`?
- Answer: **not** because infix is “baked into” ML. Actually because of one missing piece of list definition:

```ocaml
datatype 'a list = nil | :: of 'a * 'a list

infix ::
```

- Same for append operator `@`
Can make your own *infix* operators

```haskell
fun plus (x, y) = x + y
infix plus
val seven = 3 plus 4
```

(This is a silly example; we’ll see a really good one when we talk about function composition)
Type Constructors
List is not a type

[] : 'a list
:: : 'a * 'a list -> 'a list
map : ('a -> 'b) * 'a list -> 'b list

- Types: 'a list, int list, (int->int) list, etc.
- Not a type: list

- So what is list?
List of a type constructor

- *Type constructor:* makes a new type out of an old type
- Don’t confuse with datatype constructor!
  - Datatype constructors make new **values** out of old values

- Given a type \( t \), we can write \( t \text{ list} \), and \( t \text{ list} \) is a type.

- Similarly: **option** is a type constructor, not a type
  - Given a type \( t \), we can write \( t \text{ option} \), and \( t \text{ option} \) is a type.

- You can define your own type constructors, too!
datatype 'a stack = S of 'a list
val empty_stack = S []
fun push (x, S s) = S (x::s)
fun top (S []) = raise Empty
  | top (S (x::_)) = x
fun pop (S []) = raise Empty
  | pop (S (_,::s)) = S s
Maps as lists

```
datatype (''k,'v) map = M of (''k * 'v) list

exception NotFound;

val empty_map = M []

fun put (k, v, M m) = M ((k,v)::m)

fun exists (k, M []) = false
  | exists (k, M ((k', v)::m)) = if k=k'
      then true
      else exists(k, M m)

fun get (k, M []) = raise NotFound
  | get (k, M ((k',v)::m)) = if k=k'
      then v
      else get(k, M m)
```
Function Composition
In math, \( f \circ g(x) = f(g(x)) \)

\[
\text{fun compose } (\text{f}, \text{g}) = \text{fn } x \Rightarrow f(g(x))
\]

- Type \((\text{'b} \to \text{'c}) \times (\text{'a} \to \text{'b}) \to (\text{'a} \to \text{'c})\) but the REPL prints something equivalent

- ML standard library provides this as infix operator \( \circ \)
  - That’s small-case letter \( \circ \), as in...
- Example (third version best):

\[
\text{fun sqrt_of_abs i} = \text{Math.sqrt(Real.fromInt(abs i))}
\]
\[
\text{fun sqrt_of_abs i} = (\text{Math.sqrt o Real.fromInt o abs}) i
\]
\[
\text{val sqrt_of_abs} = \text{Math.sqrt o Real.fromInt o abs}
\]
Left-to-Right vs Right-to-Left

val sqrt_of_abs = Math.sqrt o Real.fromInt o abs

In math, function composition is read right-to-left

- “take absolute value; convert to real; take square root”

But programmers often prefer reading left-to-right

- “Pipelines” of functions are common in functional programming
- *Pipeline operator* is very popular (and predefined) in F#
  - Can define ourselves in ML

infix |>  
fun x |> f = f x

fun sqrt_of_abs i =  
i |> abs |> Real.fromInt |> Math.sqrt
Fold
fun map (f,xs) =
  case xs of
    [] => []
  | x::xs' => (f x)::(map(f,xs'))

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

map (fn x => shirt_color(x), [ ])

bad style!

= [gold, blue, red]
fun map (f,xs) =
  case xs of
    [] => []
   | x::xs' => (f x)::(map(f,xs'))

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

map (shirt_color, [ ] )

  = [gold, blue, red]
fun filter (f, xs) =
    case xs of
        [] => []
    | x::xs => if f x
    then x::(filter(f, rest))
    else filter(f, rest)

filter : ('a -> bool) * 'a list -> 'a list

filter(is_vulcan, [ ])

= [ ]

(er, half vulcan)
Another famous function—**Fold**

`foldl`

- used to *iterate* over recursive data structures
- several synonyms/cousins: `reduce`, `inject`, etc.
- Accumulates an answer by repeatedly applying `f` to “answer so far”
  
  \[ \text{foldl}(f, \text{acc}, [x_1, x_2, x_3]) \text{ computes } f(x_3, f(x_2, f(x_1, \text{acc}))) \]

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

- This version “folds from the left”

  ```
  val foldl = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
  ```
val l = [
  
]

foldl(max_rank, Ensign, l) (* = Captain *)
foldr

- Folds "from the right"

\[
\text{foldr}(f, \text{acc}, [x_1, x_2, x_3]) \text{ computes } f(x_1, f(x_2, f(x_3, \text{acc})))
\]

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

val foldr = fn : ('a * 'b -> 'b) * 'b * 'a list -> 'b
Examples of Fold

These are useful and do not use “private data”

fun f1 xs = fold((fn (x,y) => x+y), 0, xs)
fun f2 xs = fold((fn (x,y) => x andalso y>=0),
                true, xs)

These are useful and do use “private data”

fun f3 (xs,hi,lo) =
    fold(fn (x,y) =>
         x + (if y >= lo andalso y <= hi
             then 1
             else 0)),
         0, xs)
fun f4 (g,xs) = fold(fn (x,y) => x andalso g y),
                    true, xs)
The benefits of iterators

- These “iterator-like” functions are not built-in to the language
  - Just a programming pattern
  - Though many languages have built-in support, which often allows stopping early without using exceptions

- This pattern separates recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures