Functional Programming Techniques for Philosophy and Linguistics
Chris Barker and Jim Pryor, NASSLLI 2016
JP's Monday Handout

A. Imperatival/procedural model of computation

= sequence of directions
(store this in memory location so-and-so, remove the top element of stack such-and-such, …)

versus *Declarative/functional model of computation*

`2 + 7 ≤ 9` ⟹ True
`2 + 7` ⟹ 9
`2` ⟹ 2
{`2 + x ≤ 9`, `x > 5`, `x > y`} ==> assignments where `x` ↦ 6 or `x` ↦ 7, and `y` < `x`
/`ima[a-z]*ing`/ ==> {`imaging`, `imagining`}

versus other models …

B. Polymorphic Lambda Calculus (System F)

Before we could have type schemas, `id = λx : α. x`, but `id` must still be "monomorphic."
You couldn't say: `(\f. f 7 … f True) id`.

Now we'll instead use α as a full-fledged piece of syntax, a "type variable." Let `P, Q` be our metalanguage schemas for type expressions, and `M, N` schemas for value expressions.

	type constants	type vars	functional types	generic/polytypes
type expressions:	`E, Bool/t`	`α, β`	`P → Q`	**`(∀α. P)`**

	value constants	value vars	abstraction	application
value expressions:	`0, True, …`	`x, y`	`(λx : P. M)`	`M N`
			`(Λα. M)`	**`M [P]`**

Now we can have a polymorphic id function:

(λ id : (∀α. α → α). … id [Number] 7 … id [Bool] True …) **(Λα. (λx : α. x))**

C. Declaring datatypes in Functional Programming Languages

1. type Bool = True ( ) + False ( )
   Linguists call it `t`. This type is "inhabited by" exactly two values, and so is sometimes written as `2`.
   (2^A = functions `A → Bool`? Powerset of `A`s?)

2. type Boring = Only1 ( )
   This type is inhabited by exactly one value. The type is sometimes written as `1` or `Unit` or `()`.
   Why would this type be useful? (i) to output a "dummy" result, (ii) in patterns?, (iii) as a "dummy" input?

3. type TwoBools = Paired (Bool × Bool)
   Only one variant (no + like in #1), but every instance of this variant will contain two Bool values as components.

4. type PairOfBool α = Paired′ (α × Bool)
5. type Pair α β = Paired″ (α × β)
6. type Pair′ α = Paired″′ (α × α)

7. type PairOrTriple α = Paired″″ (α × α) + Tripled (α × α × α)

8. type ShortlistOfBool = NoElems ( ) + OneElem (Bool)
9. type Shortlist α = NoElems′ ( ) + OneElem′ (α)
   As with PairOrTriple, here we have two variants, but one has *no* components (there will be a single instance of this variant, the "empty" Bool or α Shortlist), and the other variant has *just one* component — but there may be several instances of that variant, one for each choice of α as its component, or each choice of Bool.
10. type Mediumlist α = NoElems′ ( ) + OneElem″ (α) + TwoElems (α × α)

11. type List α = NoElems″ ( ) + SomeElems (α × List α)
In instances of the SomeElems variant, the one component is called the "head" and the other is called the "rest" or "tail" of the list.
Notice that this datatype is specified *recursively*: this can't be done directly in System F, though it can be partially emulated there (including for this datatype).
All of 1-10 can be directly translated into System F type expressions, for instance, 10 corresponds to the System F type: $\forall\beta.\ \beta \to (\alpha \to \beta) \to (\alpha \to \alpha \to \beta) \to \beta$.

D. How do these datatypes look in real programming languages?

	Haskell	OCaml
1.	`data Bool = True | False`	`type bool = True | False`
2.	`type Unit = ( )`	`type unit = ( )`
	`data Unit = Only1`	
3.		`type boolpair = bool * bool`
5.	`type Pair a b = (a, b)`	`type ('a, 'b) pair = 'a * 'b`
	`data Pair a b = Paired a b`	
	`data Pair a b = Paired (a, b)`	
9.	`data Maybe a = Nothing | Just a`	`type ('a) option = None | Some of ('a)`
11.	`type List a = [ a ]`	
	`data List a = Null | Cons a (List a)`	`type ('a) list = Null | Cons of ('a * ('a) list)`

E. Notice the difference between Lists and Pairs (Triples, n-Tuples)

Type #11 (`List α`): (i) must be *homogeneous in the type* of its elements
(ii) values of *different length* (one having no elements, the other a head with a tail that has no elements, a third a head with a tail that itself has a head and a tail that has no elements, …) *can inhabit the same type*: these are all `List α`s.

Types like #5 (`Pair α β`): (i) can be heterogenous in the type of their elements (α and β may be different types), though they don't have to be (see types #3 and 6).
(ii) `Pair a a` and `Triple a a a` would be different types. Values of the one would be type-distinct from values of the other.

F. Idioms of functional programming

- datatypes
- lambdas and lets
- pattern matching
- recursive definitions

1. `let x = 3 in M`
`≈ (λx. M) 3`

2. multiple arguments to a function

"curried" style: `(λx. λy. N) 3 4  ≡  (λx y. N) 3 4`
type: `Number → Number → …`

n-tuple arguments: `(λ(x, y). N) (3, 4)`
type: `Number × Number → …`

3. `let (x, y) = (3, 4) in N`
pattern matching: structure `(  ,  )` that matched values are expected to have
variables `x  y` that get bound to components of the incoming structure
`≈ let pair = (3, 4) in let x = fst pair in let y = snd pair in N`
patterns can contain literal values as well as variables: `let (3, y) = (3, 4) in N`
even: `let (3, 4) = … in N`
that pattern will just match the value `(3, 4)`
sometimes the same symbols express patterns, other times values: just as in `(λx. … x …)` or `∀x. … x …`

4. other examples of pattern matching

case *bool_expression* of { True ⟶ *do_one_thing* | False ⟶ *do_another_thing* }
case *shortlist_expression* of { NoElems ⟶ *do_one_thing* | OneElem x ⟶ *do_something_using_x* }
case *list_expression* of { NoElems ⟶ *do_one_thing* | SomeElems x xs ⟶ *do_something_using_x_and_xs* }
case *list_expression* of { [ ] ⟶ *do_one_thing* | x◁xs ⟶ *do_something_using_x_and_xs* }

	Haskell	OCaml
a◁[] ≡ [a]	a:[]	a::[]
a◁[b] ≡ [a, b] ≡ [a]▷b	a:b:[] ≡ [a, b]	[a; b]
[a, b]◁▷[c, d] ≡ [a, b, c, d]	[a,b] <> [c, d] (or ++)	List.append [a; b] [c; d]

5. recursive definitions

let map = λf xs. case xs of { [ ] ⟶ [ ] | x◁xs ⟶ (f x) ◁ (map f xs) }
let map f xs = case xs of { [ ] ⟶ [ ] | x◁xs ⟶ f x ◁ map f xs }

map odd [0, 1, 2, 3] ⇝ odd 0 ◁ map odd [1, 2, 3] ⇝ False ◁ map odd [1, 2, 3] ⇝
    … ⇝ False ◁ (True ◁ (False ◁ (True ◁ map odd [ ]))) ⇝ [False, True, False, True]
map odd [0, 1, 2, 3] ⟹ ("evaluates to") [False, True, False, True]

filter odd [0, 1, 2, 3] ⟹ [1, 3]
let filter f xs = …*homework…*

map f xs ≅ { f x | x ∈ xs }
filter f xs ≅ { x | x ∈ xs ∧ f x }

6. functions like SomeElems/◁ are called "injections": given some arguments, here of types α and List α, creates an(other) value of type List α
   functions like head (where head [0, 1, 2, 3] ⟹ 0) and fst (where fst (3, 4) ⟹ 3) are called "projections"
   fst is a projection on a *tuple-style* pair of arguments (see point 2, above)
   const (where const 3 4 ⟹ 3) is a *curry-style* projection on its two arguments
   let const = λx y. x

7. g ∘ f is the composition of functions g and f, defined as λx. g (f x)
   map f is shorthand for λxs. map f xs (a "partial application" of map)
   map f ∘ filter g ≡ (λxs. map f xs) ∘ (λxs. filter g xs) ≡ λxs. map f (filter g xs)
   filter g ∘ map f ≡ λxs. filter g (map f xs)