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

Recall from predicate logic (with restricted quantification):
1. `∀x<7. 2 + x ≤ 9`

Or with a different relation in the restrictor:
2. `∀x=7. 2 + x ≤ 9  ⇒` ("evaluates to") `True`, given standard interpretations of ≤, =, +, numerical constants

Let's also use ⇒ to specify values of *subsentential* expressions:
3. `2 + 7  ⇒ 9`
4. `2 + x` $\Rightarrow_{\{x \mapsto 7\}}$ `9`

In #2, the restrictor (=) is syntactically guaranteed to have at most satisfier. Let's express such forms using the syntax:
5. `let x = 7 in 2 + x ≤ 9  ⇒ True` (as in #2)

Let's partially specify assignments for *subsentential* expressions, too:
6. `let x = 7 in 2 + x  ⇒ 9` (as in #4)
More-or-less equivalent to:
7. `(λx. 2 + x) 7  ⇒ 9` (as in #6)

Let's embed several `let`s to specify assignments more fully:
8. `let x = 3 in (let y = 4 in 2 + x + y)  ⇒ 9`
9. `let x = 3; y = 4 in 2 + x + y  ⇒ 9`

"Curried style" application of λ to several arguments:
10. `(λx y. 2 + x + y) 3 4  ⇒ 9`
"N-tuple style," mathematically more familiar:
11. `(λ(x, y). 2 + x + y) (3, 4)  ⇒ 9`
#11 uses "pattern matching" to bind `x` and `y` to each element of its pair argument. As does:
12. `let (x, y) = (3, 4) in 2 + x + y  ⇒ 9`

Our language has *simple* values (numbers, truth-values, functions, etc) as well as *container* values (lists, pairs, etc):
13. `[2, 3, 4]  ⇒ [2, 3, 4]`
14. `let x = 3; y = 4 in [2, x, y]  ⇒ [2, 3, 4]`
15. `let xs = [2, 3, 4] in head xs  ⇒ 2`
16. `let xs = [2, 3, 4] in tail xs  ⇒ [3, 4]`
17. `cons 2 [3, 4]  ⇒ [2, 3, 4]`
18. `2 ◁ [3, 4]  ⇒ [2, 3, 4]`
19. `let xs = [2, 3, 4]; ys = cons 1 (tail xs) in ys  ⇒ [1, 3, 4]`

We've seen pre-defined functions like `+`, `≤`, `head`, `cons`. We express our own functions with λ-terms:
20. `λx y. 2 + x + y`
This applies to values as exhibited in #10, #11. Recall also #9 ≈ #10. Now we bind variables to our λ–terms:
21. `let g = λx y. 2 + x + y in g 3 4  ⇒ 9` (as in #9 and #10)
22. `let g x y = 2 + x + y in g 3 4  ⇒ 9`
23. `let g = λx y. 2 + x + y; x = 3 in g x 4  ⇒ 9`

"Big Step" Operational Semantics for some of the constructions we've seen

(a) $$\frac{g \text{ includes } \{\mathtt{x} \mapsto v\}}{\mathtt{x} \Rightarrow_g v}$$

(b) $$\frac{\mathtt{A} \Rightarrow_g v_a \quad \mathtt{B} \Rightarrow_{g\,\{\mathtt{x} \mapsto v_a\}} v_b}{\mathtt{let\ x\ =\ A\ in\ B} \Rightarrow_g v_b}$$

(c) $$\frac{\mathtt{A} \Rightarrow_g \text{NoElems} \quad \mathtt{B} \Rightarrow_g v_b}{\mathtt{case\ A\ of\ \{\ NoElems} \longrightarrow \mathtt{B} \mid \mathtt{OneElem\ x} \longrightarrow \mathtt{C\ \}} \Rightarrow_g v_b}$$

(d) $$\frac{\mathtt{A} \Rightarrow_g \text{OneElem } v_a \quad \mathtt{C} \Rightarrow_{g\,\{\mathtt{x} \mapsto v_a\}} v_c}{\mathtt{case\ A\ of\ \{\ NoElems} \longrightarrow \mathtt{B} \mid \mathtt{OneElem\ x} \longrightarrow \mathtt{C\ \}} \Rightarrow_g v_c}$$

(e) $$\frac{}{\lambda\, x.\ B \Rightarrow_g \text{the "closure" } (\lambda x.\ B,\ g)}$$

(f) $$\frac{\mathtt{A} \Rightarrow_g v_a \quad \mathtt{H} \Rightarrow_g (\lambda x.\ B,\ g0) \quad B \Rightarrow_{g0\,\{\mathtt{x} \mapsto v_a\}} v_b}{\mathtt{H\ A} \Rightarrow_g v_b}$$

Call-by-value (or "eager," weak-head normal form) evaluation order

1. first, evaluate the outermost "head" subexpression using these rules, until the result is not an application
2. if the head isn't a λ-term (or primitive function), then stop
3. if the head isn't followed by an argument, then stop (we don't evaluate "under lambdas," but wait until they're applied)
4. evaluate the argument term using these rules, until the result is not an application
5. "perform the application," by evaluating the body of the head λ-term with its abstracted variable bound to the result of step 4

When could it matter what evaluation order we use?

(a) Infinitely recursing evaluations
"Thunks" (unapplied functions, usually expecting arguments of type Boring) are useful because in these languages we don't evaluate inside unapplied λ-terms (rule #3 in the call-by-value evaluation scheme)

(b) printing

```
pay_taxes (let _ = print "paying 100" in 90)
```

(c) mutation (rest of today's discussion)

Remember in predicate logic you can say:
24. `∀x. (∃ x. x = 9) ∨ … x …`
So too can we say:
25. `let x = 0; y = 0 in (let x = 3; y = 4 in 2 + x + y)  ⇒ 9`
Programmers describe this as the innermost let introducing "new local bindings" for `x`, `y` that "shadow" their more global bindings. The more local bindings are only in place until the closing "`)`".

Contrast:
26. `let x = 0; y = 0 in (let x = 3; y = 4 in (2 +)) (x + y)  ⇒ 2`
Where `(2 +)` is a "partial application" of +. Here the local bindings are in place only for the evaluation of that partial sum (where they make no difference). When we evaluate the further argument `(x + y)` for that sum, we use the more global bindings {`x ↦ 0`, `y ↦ 0`}.

λ-terms can contain variables that are bound outside them (variables that are "locally free"):
27. `let y = 4; f = λx. 2 + x + `**`y`**` in [f 3, f 13]  ⇒ [9, 19]`
How should we interpret variable occurrences like the `y` inside our λ-term when `y` later gets a different binding, shadowing the one in place where the λ-term was expressed?
28. `let y = 4; f = λx. 2 + x + `**`y`**`; y = 0 in [y, f 3, f 13, y]  ⇒ [0, ?, ?, 0]`
What programmers call "lexical" or "static" binding keeps interpreting that `y` inside the λ-term with its original binding to `4`. So the result is `[0, 9, 19, 0]`. This is the dominant strategy in contemporary programming. The other strategy (which programmers happen to call "dynamic," but that's misleading in a context like ours) interprets #28 with `y` using its new binding to `0`, yielding the result `[0, 5, 15, 0]`. We'll ignore that strategy in our discussion.

Contrast #28 (evaluating to `[0, 9, 19, 0]`) with:
29. `let y = 4; f = λx. (let y = 0 in 2 + x + y) in [y, f 3, f 13, y]  ⇒ [4, 5, 15, 4]`
Here the more local binding for `y` is inside the definition of `f`, so it *does* affect the result we get when applying `f`. Also, that binding is in effect only *inside* the definition of `f`, so it *doesn't* affect the interpretation of `y` at the start and end of the final list expression.

Everything so far has been "purely declarative." Now we'll introduce mutation, which comes in two varieties. "Explicit"-style mutation gives us *mutable container values*. "Implicit"-style mutation gives us *mutable variable bindings*. We'll introduce them in that order.
30. `let xs = [2, 3, 4]`$_{\texttt{mut}}$`; _ = set-head! xs 1 in xs  ⇒ [1, 3, 4]`
31. `let xs = [2, 3, 4]`$_{\texttt{mut}}$`; ( ) = set-head! xs 1 in xs  ⇒ [1, 3, 4]`
Notice the different syntax on the list expression, to indicate that the container is mutable. Also, contrast #30/#31 to #19, here repeated:
19. `let xs = [2, 3, 4]; ys = cons 1 (tail xs) in ys  ⇒ [1, 3, 4]`
In #19, `ys  ⇒ [1, 3, 4]`, but `xs` would still `⇒ [2, 3, 4]`. In #30/#31, on the other hand, `xs` itself now `⇒ [1, 3, 4]`. We achieve this not by constructing a new list, partially modeled on the one we originally bound `xs` to. Instead we *mutate* that original list, using the operation `set-head!` (We follow a convention of naming functions that have effects with a trailing "`!`") The usefulness of `set-head!` is in the *effect* you get from applying it, not from its return value, so in #30 we ignore it. I assume here that its return value is just the dummy, boring value `()` (or "unit"). (You might instead have chosen to make `set-head!` return the original, pre-mutated head.)

In introducing mutable lists and operations like `set-head!` into our language, we've abandoned what programmers call "referential transparency." Consider:
32.
```
let f = (let ys = [0]mut
            in λx. (let y = head ys; _ = set-head! ys (1+y) in 2 + x + y) )
   in [f 3, f 3, f 3]  ⇒ [5, 6, 7]
```
What result you will get from this depends on what order the "`f 7`"s in the final list expression are evaluated. I've assumed here it goes from left-to-right, but nothing mandates that choice. At each application of function `f` to a constant argument `3`, we end up adding different head-values from the mutating list `ys`. So the result of applying `f` is not determined by the values supplied to it as arguments.

An interesting aside:

33. `let y = 4 in y equal 3  ⇒ False`

34. `let ys = [2, 3, 4] in ys equal/egal [1, 3, 4]  ⇒ False`

35. `let xs = [2, 3, 4]`$_{\texttt{mut}}$`; ys = cons 1 (tail xs); zs = xs; _ = set-head! xs 1 in …`

(a) `… xs  ⇒ [1, 3, 4]`

(b) `… zs  ⇒ [1, 3, 4]`

(c) `… xs equal ys  ⇒ True`, because these lists contain the same elements

(d) `… xs egal ys  ⇒ False`, because they're distinct, independently mutable containers (unlike `xs` and `zs`)

So far, we've seen mutable list *values*, but the way *variable bindings* work hasn't changed. Once bound to a value, a variable stays so bound throughout the syntactic scope of that binding — though perhaps it may be "shadowed" by more local bindings of the same variable symbol. In such cases, though, the old binding would persist in the background, and would again be visible after the scope of the more local binding expires.

Some languages, more radically, permit *variable bindings themselves to be mutated* (not merely "shadowed").

36. `let y = 4; f = λx. (let _ = set! y 0 in 2 + x + y) in [y, f 3, f 13, y]  ⇒ [4, 5, 15, 0]`

Contrast #29, here repeated:

29. `let y = 4; f = λx. (let y = 0 in 2 + x + y) in [y, f 3, f 13, y]  ⇒ [4, 5, 15, 4]`

In #29 we only "shadow" the outermost binding of y with a new binding. In #36 we instead *mutate* the outermost binding, itself. So then even after the closing "**)**", after `f` has been applied to any argument, that new mutated binding will still be visible. That's why #36's result ends with `0` rather than `4`.

Some languages (e.g., Scheme) have *both* mutable container values *and also* mutable variable bindings. But commonly languages have at most one of them (mainstream, imperatival languages generally have mutable variable bindings). Here's an example of both together:

37. `let xs = [2, 3, 4]`$_{\texttt{mut}}$`;`

```
       zs = xs;
       _ = set-head! xs 1;
       _ = set! xs [0, 0, 0]mut
    in …
```

(a) `xs  ⇒ [0, 0, 0]`

(b) `zs  ⇒ [1, 3, 4]`

In the operation `set-head! xs 1`, we use the variable `xs` to mutate the *single list value* that both `xs` and `zs` are bound to. So when we later evaluate `zs`, that change is visible (as it also was in #35b). Here, though, we go on to do something we didn't do in #35. We go on to *mutate the variable* `xs` to become bound to a new list. `zs` still stays bound to the old list.

"Big Step" Operational Semantics for a language with mutable container values

BEFORE: $A \Rightarrow_{g} \quad v_a$

NOW: $\quad A \Rightarrow_{g,m} v_a, m'$

(a) $$\frac{\texttt{A} \Rightarrow_{g,m} v_a, m_a \qquad \texttt{B} \Rightarrow_{g\,\{\texttt{x} \mapsto v_a\},m_a} v_b, m_b}{\texttt{let x = A in B} \Rightarrow_{g,m} v_b, m_b}$$ (will respect dynamic effects in evaluation of A or B)

(b) $$\frac{\texttt{A} \Rightarrow_{g,m} v_a, m_a}{\texttt{newbox A} \Rightarrow_{g,m} \text{new location in } m_a,\ m_a \text{ with } v_a \text{ in that new location}}$$ $(\text{len } m_a,\ m_a \triangleright v_a)$

(c) $$\frac{\texttt{A} \Rightarrow_{g,m} \text{location } a, m_a \qquad m_a \text{ has } v_a \text{ in location } a}{\texttt{get A} \Rightarrow_{g,m} v_a, m_a}$$

(d) $$\frac{\texttt{A} \Rightarrow_{g,m} \text{location } a, m_a \qquad \texttt{B} \Rightarrow_{g,m_a} v_b, m_b}{\texttt{put B into A} \Rightarrow_{g,m} (\,), m_b \text{ with } v_b \text{ in location } a}$$

(e) `let x = newbox 0 in get x` $\Rightarrow_{g,m}$ 0, m with 0 in a new location

DERIVATION:

$$\frac{\dfrac{\dfrac{}{\texttt{0} \Rightarrow_{g,m} 0, m}}{\texttt{newbox 0} \Rightarrow_{g,m} \text{len } m, m \triangleright 0} \qquad \dfrac{\dfrac{}{\texttt{x} \Rightarrow_{g\,\{\texttt{x} \mapsto \text{len } m\},m \triangleright 0} \text{len } m, m \triangleright 0} \qquad m \triangleright 0 \text{ has 0 in location (len } m)}{\texttt{get x} \Rightarrow_{g\,\{\texttt{x} \mapsto \text{len } m\},m \triangleright 0} 0, m \triangleright 0}}{\texttt{let x = newbox 0 in get x} \Rightarrow_{g,m} 0, m \triangleright 0}$$

(f) `let x = newbox 0 in (let _ = put 1 into x in get x)` $\Rightarrow_{g,m}$ 1, m with 1 in a new location