Programming Languages

Function-Closure Idioms

Adapted from Dan Grossman's PL class, U. of Washington
More idioms

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

A partial but wide-ranging list:

• Pass functions with private data to iterators (map/filter): Done
• Combine functions (e.g., composition)
• Currying (multi-arg functions and partial application)
• Callbacks (e.g., in reactive programming)
• Implementing an ADT with a record of functions
Combine functions

Canonical example is function composition:

```
(define (compose f g) (lambda (x) (f (g x))))
```

- Creates a closure that “remembers” what \( f \) and \( g \) are bound to
- This function is built-in to Racket; but this definition is basically how it works.
- 3rd version is the best (clearest as to what it does):

```
(define (sqrt-of-abs i) (sqrt (abs i)))
(define (sqrt-of-abs i) ((compose sqrt abs) i))
(define sqrt-of-abs (compose sqrt abs))
```
Currying and Partial Application

• Currying is the idea of calling a function with an incomplete set of arguments.
• When you "curry" a function, you get a function back that accepts the remaining arguments.
• Named after Haskell Curry, who studied related ideas in logic.
• Useful in situations where you want to call/pass a function, but you don't know the values for all the arguments yet.
  – Ex: a function of two arguments, but coming from two separate places (scopes) in your program.
Motivation example

• We want to write code that takes a list of numbers and returns a list of the number 4 raised to the power of each number.
  – in: \((x_1 \ x_2 \ \ldots \ \ x_n)\)
  – out: \((4^{x_1} \ 4^{x_2} \ \ldots \ 4^{x_n})\)

• We could use a lambda expression:
  – \((\text{map (lambda (x) (expt 4 x)) lst})\)

• But this can get tedious to do over and over.

• What if the \texttt{expt} function were defined differently?
Currying and Partial Application

• We know \((\text{expt } x \ y)\) raises \(x\) to the \(y\)'th power.
• We could define a different version of \texttt{expt} like this:

\[
\begin{align*}
\text{(define (expt-curried x)} & \\
& \quad \text{(lambda (y) (expt x y))}
\end{align*}
\]

• We can call this function like this:

\[
((\text{expt-curried 4}) \ 2)
\]

• This is an incredibly flexible definition:
  – We can call with two arguments as normal (with extra parens)
  – Or call with one argument to get a function that accepts the remaining argument.

• This is critical in some other functional languages (albeit, not Racket or Scheme) where functions may have at most one argument.
Currying and Partial Application

• Currying is still useful in Racket with the curry function:
  – Turns a function \((f \ x_1 \ x_2 \ x_3 \ldots \ x_n)\)
    into a function \((((((f \ x_1) \ x_2) \ x_3) \ldots \ x_n)\)
  – curry takes a function \(f\) and some optional arguments
  – Returns a function that accumulates remaining arguments until
    \(f\) can be called (all arguments are present).
• \((\text{curry} \ \text{expt} \ 4) \equiv (\text{expt-curved} \ 4)\)
• \(((\text{curry} \ \text{expt} \ 4) \ 2) \equiv ((\text{expt-curved} \ 4) \ 2)\)
• These can be useful in definitions themselves:
  – \((\text{define} \ \text{double} \ x) \ (* \ 2 \ x))\)
  – \((\text{define} \ \text{double} \ (\text{curry} \ * \ 2))\)
Currying and Partial Application

- Currying is also useful to shorten longish lambda expressions:
- Old way: \( \text{map (lambda (x) (+ x 1)) '(1 2 3)} \)
- New way: \( \text{map (curry + 1) '(1 2 3)} \)

- Great for encapsulating private data: \textit{list-ref is the built-in get-nth.}

\[
\begin{align*}
\text{(define get-month} \\
\text{  (curry list-ref '}(\text{Jan Feb Mar Apr May Jun} \\
\text{  Jul Aug Sep Oct Nov Dec}))
\end{align*}
\]

- This example introduces a new datatype: symbol.
  - Symbols are similar to strings, except they don't have quotes around them (and you can't take them apart or add them together like strings).
Currying and Partial Application

- But this gives zero-based months:
  
  ```scheme
  (define get-month
    (curry list-ref
      '(Jan Feb Mar Apr May Jun
        Jul Aug Sep Oct Nov Dec)))
  ```

- Let's subtract one from the argument first:
  
  ```scheme
  (define get-month
    (compose
      (curry list-ref
        '(Jan Feb Mar Apr May Jun
          Jul Aug Sep Oct Nov Dec))
      (curryr - 1)))
  ```

- `curryr` curries from right to left, rather than left to right.
Currying and Partial Application

• Another example:

```scheme
(define (eval-polynomial coeff x)
  (if (null? coeff) 0
      (+ (* (car coeff) (expt x (- (length coeff) 1)))
          (eval-polynomial (cdr coeff) x))))
```

```scheme
(define (make-polynomial coeff)
  (lambda (x) (eval-polynomial coeff x)))
```

```scheme
(define make-polynomial (curry eval-polynomial))
```
Currying and Partial Application

- A few more examples:
  
- \((\text{map} \ (\text{compose} \ (\text{curry} \ + \ 2) \ (\text{curry} \ * \ 4)) \ '(1 \ 2 \ 3))\)
  - quadruples then adds two to the list '(1 2 3)

- \((\text{filter} \ (\text{curry} < \ 10) \ '(6 \ 8 \ 10 \ 12))\)
  - Careful! \text{curry} works from the left, so \((\text{curry} < \ 10)\) is equivalent to \((\text{lambda} \ (x) \ (< \ 10 \ x))\) so this filter keeps numbers that are greater than 10.

- Probably clearer to do:

  \((\text{filter} \ (\text{curryr} > \ 10) \ '(6 \ 8 \ 10 \ 12))\)

- (In this case, the confusion is because we are used to "<" being an infix operator).
Currying becomes really powerful when you curry higher-order functions.

Recall \((\mathit{foldr~} f~ \mathit{init}~ (x_1~ x_2~ \ldots~ x_n))\) returns
\((f~ x_1~ (f~ x_2~ \ldots~ (f~ x_{n-2}~ (f~ x_{n-1}~ (f~ x_n~ \mathit{init})))))\)

\[
\begin{align*}
\text{(define (sum-list-ok lst) (foldr + 0 lst))} \\
\text{(define sum-list-super-cool (curry foldr + 0))}
\end{align*}
\]
Another example

• Scheme and Racket have **andmap** and **ormap**.
• \((\text{andmap } f \ (x_1 \ x_2\ldots))\) returns \((\text{and } (f \ x_1) \ (f \ x_2) \ldots)\)
• \((\text{ormap } f \ (x_1 \ x_2\ldots))\) returns \((\text{or } (f \ x_1) \ (f \ x_2) \ldots)\)

\[
\begin{align*}
\text{(andmap (curryr > 7) '(8 9 10)) } & \Rightarrow \text{ #t} \\
\text{(ormap (curryr > 7) '(4 5 6 7 8)) } & \Rightarrow \text{ #t} \\
\text{(ormap (curryr > 7) '(4 5 6)) } & \Rightarrow \text{ #f} \\
\text{(define contains7 (curry ormap (curry = 7)))} \\
\text{(define all-are7 (curry andmap (curry = 7)))}
\end{align*}
\]
Another example

Currying and partial application can be convenient even without higher-order functions.

*Note:* \((\text{range } a \ b)\) returns a list of integers from \(a\) to \(b-1\), inclusive.

\[
\begin{align*}
\text{(define (zip lst1 lst2)} & \text{)} \\
& \text{(if (null? lst1) '())} \\
& \text{(cons (list (car lst1) (car lst2))} \\
& \text{(zip (cdr lst1) (cdr lst2))))})
\end{align*}
\]

\[
\begin{align*}
\text{(define countup (\textbf{curry} range 1))} \\
\text{(define (add-numbers lst)} & \text{)} \\
& \text{(zip (countup (length lst)) lst))}
\end{align*}
\]
When to use currying

- When you write a lambda function of the form
  - `(lambda (y1 y2 ...) (f x1 x2 ... y1 y2...))`
- You can replace that with
  - `(curry f x1 x2 ...)`

- Similarly, replace
  - `(lambda (y1 y2 ...) (f y1 y2 ... x1 x2...))`
- with
  - `(curryr f x1 x2 ...)`
When to use currying

• Try these:
  – Assuming \texttt{lst} is a list of numbers, write a call to \texttt{filter} that keeps all numbers greater than 4.
  – Assuming \texttt{lst} is a list of lists of numbers, write a call to \texttt{map} that adds a 1 to the front of each sublist.
  – Assuming \texttt{lst} is a list of numbers, write a call to \texttt{map} that turns each number (in \texttt{lst}) into the list (1 number).
  – Assuming \texttt{lst} is a list of numbers, write a call to \texttt{map} that squares each number (you should curry \texttt{expt}).
  – Define a function \texttt{dist-from-origin} in terms of currying a function \texttt{(dist x1 y1 x2 y2)} [assume \texttt{dist} is already defined elsewhere]
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
- (Can accomplish this in C++ with objects that contain private fields.)
Mutable state

While it’s not absolutely necessary, mutable state is reasonably appropriate here

- We really do want the “callbacks registered” and “events that have been delivered” to change due to function calls

In "pure" functional programming, there is no mutation.

- Therefore, it is guaranteed that calling a function with certain arguments will always return the same value, no matter how many times it's called.
- Not guaranteed once mutation is introduced.
- This is why global variables are considered "bad" in languages like C or C++ (global constants OK).
Mutable state: Example in C++

```cpp
int times_called = 0

int function() {
    times_called++;
    return times_called;
}
```

This is useful, but can cause big problems if somebody else modifies `times_called` from elsewhere in the program.
Mutable state

• Scheme and Racket's variables are mutable.
• The name of any function which does mutation contains a "!"
• Mutate a variable with set!:
  – Only works after the variable has been placed into an environment with define, let, or as an argument to a function.
  – set! does not return a value.

(define times-called 0)
(define (function)
  (set! times-called (+ 1 times-called))
times-called)
• Notice that functions that have side-effects or use mutation are the only functions that need to have bodies with more than one expression in them.
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 support removing them, etc.

```
(define callbacks '())
(define (add-callback func)
  (set! callbacks (cons func callbacks)))

(define (key-press which-key)
  (for-each
    (lambda (func) (func which-key)) callbacks))
```
(define (print-if-pressed key message)
  (add-callback
    (lambda (which-key)
      (if (string=? key which-key)
          (begin (display message) (newline)) #f))))

(define count-presses 0)
(add-callback
  (lambda (key)
    (set! count-presses (+ 1 count-presses))
    (display "total presses = ")
    (display count-presses)
    (newline)))
Improvement on the client side

- Why clutter up the global environment with count-presses?
- We don't want some other function mucking with that variable.
- Let's hide it inside a let that only our callback can access.

```
(let ((count-presses 0))
  (add-callback
   (lambda (key)
     (set! count-presses (+ 1 count-presses))
     (display "total presses = ")
     (display count-presses)
     (newline)))
```
Implementing an ADT

As our last pattern, closures can implement abstract data types

– They can share the same private data
– Private data can be mutable or immutable
– Feels quite a bit like objects, emphasizing that OOP and functional programming have similarities

The actual code is advanced/clever/tricky, but has no new features

– Combines lexical scope, closures, and higher-level functions
– Client use is not so tricky
(define (new-stack)
    (let ((the-stack '()))
        (define (dispatch method-name)
            (cond ((eq? method-name 'empty?) empty?)
                ((eq? method-name 'push) push)
                ((eq? method-name 'pop) pop)
                (#t (error "Bad method name")))
        (define (empty?) (null? the-stack))
        (define (push item) (set! the-stack (cons item the-stack)))
        (define (pop)
            (if (null? the-stack) (error "Can't pop an empty stack")
                (let ((top-item (car the-stack)))
                    (set! the-stack (cdr the-stack))
                    top-item)))
        dispatch)) ; this last line is the return value
                ; of the let statement at the top.