Functional abstraction

Functions are first class values
Contracts and types
Anonymous functions
Syntax & semantics
Abstracting from examples
Higher-order functions

**Functional abstraction**
Readings

HtDP, sections 19 – 24
Language level

Intermediate Student With Lambda
What is abstraction?

Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

Example: writing a function.

The differences in parameter values are forgotten, and the similarity is captured in the function body.

We have seen many similarities between functions, and captured them in design recipes.

But some similarities still elude us.
Keeping odd numbers

(define (keep-odds lst)
  (cond
   [(empty? lst) empty]
   [(odd? (first lst))
    (cons (first lst) (keep-odds (rest lst)))]
   [else (keep-odds (rest lst))])))

(keep-odds '(1 2 3 4 5 6 7 8 9 10))
=> '(1 3 5 7 9)
Eating apples

(define (eat-apples lst)
  (cond
    [(empty? lst) empty]
    [(not (symbol=? (first lst) 'apple))
      (cons (first lst) (eat-apples (rest lst)))]
    [else (eat-apples (rest lst))]))

(eat-apples '(apple banana cherry apple dirt))
=> '(banana cherry dirt)
Abstracting from these examples

What these two functions have in common is their general structure.

Where they differ is in the *specific predicate* used to decide whether an item is removed from the answer or not.

We could write one function to do both these tasks if we could supply, as an argument to that function, the predicate to be used.

The Intermediate language permits this.
Functions as first class values
Functions as first-class values

In the Intermediate language, functions are values. In fact, they are *first-class values*.

Functions have the same status as the other values we have seen. They can be:

- consumed as function arguments,
- produced as function results,
- bound to identifiers, and
- put in structures and lists.
Functions as first-class values has historically been missing from languages that are not primarily functional.

The utility of functions-as-values is now widely recognized, and they are at least partially supported in many languages that are not primarily functional, including C++, C#, Java, Go, JavaScript, Python, and Ruby.

Functions-as-values provides a clean way to think about the concepts and issues involved in abstraction.

You can then worry about how to implement a high-level design in a given programming language.
Consuming functions

\[
\text{(define (foo func x y) (func x y))}
\]

\[
\text{(foo + 2 3) => 5}
\]

\[
\text{(foo * 2 3) => 6}
\]
Generalizing keep-odds: my-filter

(define (my-filter pred? lst)
  (cond
   [(empty? lst) empty]
   [(pred? (first lst)); keep the current value if it
     (cons ; satisfies some predicate
       (first lst)
       (my-filter pred? (rest lst)))]
   [else (my-filter pred? (rest lst))])))
Tracing my-filter

(my-filter odd? (list 5 6 7))
=> (cons 5 (my-filter odd? (list 6 7)))
=> (cons 5 (my-filter odd? (list 7)))
=> (cons 5 (cons 7 (my-filter odd? empty)))
=> (cons 5 (cons 7 empty))

my-filter is an abstract list function which handles the general operation of removing items from lists.
Using my-filter

(define (keep-odds lst) (my-filter odd? lst))

Using built-in predicate odd?
Using my-filter

(define (keep-oDDS lst) (my-filter odd? lst))

(define (not-an-apple? item)
  (not (symbol=? item 'apple)))

(define (eat-apples lst) (my-filter not-an-apple? lst))

Using custom predicate not-an-apple?
Using my-filter

```
(define (keep-odds lst) (my-filter odd? lst))

(define (not-an-apple? item)
  (not (symbol=? item 'apple)))

(define (eat-apples lst) (my-filter not-an-apple? lst))
```

The function \texttt{filter}, which behaves identically to our my-filter, is built into Intermediate Student and Full Racket.

\texttt{filter} and other abstract list functions provided in Racket are used to apply common patterns of structural recursion.

We will discuss how to write contracts for them shortly.
Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as filter.

• It reduces code size.
• It avoids cut-and-paste.
• Bugs can be fixed in one place instead of many.
• Improving one functional abstraction improves many applications.
Producing functions

We saw in lecture module 09 how `local` could be used to create functions during a computation, to be used in evaluating the body of the `local`.

But now, because functions are values, the body of the `local` can produce such a function as a value.

Though it is not apparent at first, this is enormously useful. We illustrate with a very small example.
(define (make-adder n)
  (local
   [(define (func m) (+ n m))]
   func))

What is (make-adder 3)?

We can answer this question with a trace.
(make-adder 3)
=> (local
   [(define (func m) (+ 3 m))]
   f)
=> (define (func_42 m) (+ 3 m))
func_42

(make-adder 3) is the renamed function func_42, which is a function that adds 3 to its argument.

We can apply this function immediately, or we can use it in another expression, or we can put it in a data structure.
Here is what happens if we apply it immediately.

```lisp
((make-adder 3) 4)
=> ((local
    [(define (func m) (+ 3 m))]
    func)
   4)
=> (define (func_42 m) (+ 3 m)) ; lifts out func
   (func_42 4) ; and renames it
=> (+ 3 4)
=> 7
```
A note on scope

\[
\begin{align*}
\text{(define (add3 m)} & \quad \text{(define (make-adder n)} \\
\text{ (+ 3 m))} & \quad \text{(local} \\
& \qquad [(\text{define (f m) (+ n m))] \\
& \qquad f))
\end{align*}
\]

In \text{add3} the parameter \text{m} is of no consequence after \text{add3} is applied. Once \text{add3} produces its value, \text{m} can be safely forgotten.

However, our earlier trace of \text{make-adder} shows that after it is applied, the parameter \text{n} does have a consequence. It is embedded into the result, \text{f}, where it is “remembered” and used again, potentially many times.
Binding functions to identifiers

The result of `make-adder` can be bound to an identifier and then used repeatedly.

```
(define add2 (make-adder 2))
(define add3 (make-adder 3))
```

```
(add2 3) => 5
(add3 10) => 13
(add3 13) => 16
```
(define add2 (make-adder 2))
=> (define add2
    (local
      [(define (func m) (+ 2 m))]
      func))
=> (define (func_43 m) (+ 2 m)); lifts out func
(define add2 func_43); and renames it

(add2 3)
=> (func_43 3)
=> (+ 2 3)
=> 5
Putting functions in lists

Recall our code in lecture Module 08 for evaluating alternate arithmetic expressions such as '(+ (* 3 4) 2).

;; eval: AltAExp -> Num
(define (eval aax)
  (cond
    [(number? aax) aax]
    [else (my-apply (first aax) (rest aax))])))
;;; my-apply: Sym AltAExpList -> Num
(define (my-apply f aaxl)
  (cond
   [(and (empty? aaxl) (symbol=? f '+)) 0]
   [(and (empty? aaxl) (symbol=? f '*)) 1]
   [(symbol=? f '+)
     (+ (eval (first aaxl)) (my-apply f (rest aaxl)))]
   [(symbol=? f '*)
     (* (eval (first aaxl)) (my-apply f (rest aaxl))))
   )]

Note the similar-looking code.
Much of the code is concerned with translating the symbol `'+' into the function `+`, and the same for `'∗' and `∗'.

If we want to add more functions to the evaluator, we have to write more code which is very similar to what we have already written.

We can use an association list to store the above correspondence, and use the function `lookup-al` we saw in lecture Module 06 to look up symbols.
(define trans-table (list (list '+ +)
               (list '* *)))

Now (lookup-al '+ trans-table) produces the function +.

    ((lookup-al '+ trans-table) 3 4 5)
=> 12
;; apply: Sym AltAExpList -> Num
(define (apply f aaxl)
  (cond
   [(and (empty? aaxl) (symbol=? f '+)) 0]
   [(and (empty? aaxl) (symbol=? f '∗)) 1]
   [else ((lookup-al f trans-table)
            (eval (first aaxl))
            (apply f (rest aaxl))))]))

We can simplify this even further, because in Intermediate Student, + and ∗ allow zero arguments:

• (+) => 0
• (∗) => 1
Now, to add a new binary function (that is also defined for 0 arguments), we need only add one line to trans-table.
Contracts and types
Contracts and types

Our contracts describe the type of data consumed by and produced by a function.

Until now, the type of data was either a basic (built-in) type, a defined (struct) type, an anyof type, or a list type, such as List-of-Symbols, which we then called (listof Sym).

Now we need to talk about the type of a function consumed or produced by a function.
We can use the contract for a function as its type.

For example, the type of \( > \) is \((\text{Num} \ \text{Num} \rightarrow \text{Bool})\), because that is the contract of that function.

We can then use type descriptions of this sort in contracts for functions which consume or produce other functions.
An example:

```
(define trans-table (list (list '+ +)
                        (list '* *)))
```

;; (lookup-al k alst) finds the value in alst
;; corresponding to key k
;; lookup-al:
;; Sym (listof (list Sym (Num Num -> Num))) ->
;;      (anyof false (Num Num -> Num))

```
define (lookup-al k alst)
    (cond
        [(empty? alst) false]
        [(equal? k (first (first alst))
            (second (first alst))]
        [else (lookup-al k (rest alst))])
```
Contracts for abstract list functions

`filter` consumes a function and a list, and produces a list.

We might be tempted to conclude that its contract is

\[(\text{Any} \rightarrow \text{Bool}) \ (\text{listof Any}) \rightarrow (\text{listof Any}).\]

But this is not specific enough.

Consider the application \(\text{filter} \ \text{odd?} \ '(1 \ 2 \ 3)\). This does not obey the contract (the contract for `odd?` is \(\text{Int} \rightarrow \text{Bool}\)) but still works as desired.

The problem: there is a relationship among the two arguments to `filter` and the result of `filter` that we need to capture in the contract.
Parametric types

An application (\texttt{filter\ pred?\ lst}), can work on any type of list, but the predicate provided should consume elements of that type of list.

In other words, we have a dependency between the type of the predicate (which is the contract of the predicate) and the type of list.

To express this, we use a type variable, such as \texttt{X}, and use it in different places to indicate where the same type is needed.
The contract for \texttt{filter}

\texttt{filter} consumes a predicate with contract \((X \rightarrow \text{Bool})\), where \(X\) is the base type of the list that it also consumes.

It produces a list of the same type it consumes. The contract for \texttt{filter} is thus:

\[
\texttt{;; filter: (X \rightarrow \text{Bool}) (listof X) \rightarrow (listof X)}
\]

Here \(X\) stands for the unknown data type of the list.

We say \texttt{filter} is \textit{polymorphic or generic}; it works on many different types of data.
The contract for `filter` has three occurrences of a type variable `X`.

Since a type variable is used to indicate a relationship, it needs to be used at least twice in any given contract.

A type variable used only once can probably be replaced with `Any`.

We will soon see examples where more than one type variable is needed in a contract.
Using contracts to understand

Many of the difficulties one encounters in using abstract list functions can be overcome by careful attention to contracts.

For example, the contract for the function provided as an argument to `filter` says that it consumes one argument and produces a Boolean value.

This means we must take care to never use `filter` with an argument that is a function that consumes two variables, or that produces a number.
Simulating structures

We can use the ideas of producing and binding functions to simulate structures.

\[
\text{(define (my-make-posn x y)}
\text{(local)}
\text{[[(define (symbol-to-value s)}
\text{[[(cond)}
\text{[[(symbol=? s 'x) x]}
\text{[[(symbol=? s 'y) y])))]
\text{symbol-to-value))]
\text{A trace demonstrates how this function works.}}
\]
(define p1 (my-make-posn 3 4))
=> (define p1
  (local
   [(define (symbol-to-value s)
      (cond
       [(symbol=? s 'x) 3]
       [(symbol=? s 'y) 4]])
    symbol-to-value))

Notice how the parameters 3 and 4 have been substituted into the local definition.

We now rename symbol-to-value and lift it out.
This yields:

```lisp
=> (define (symbol-to-value_38 s)
   (cond
     [(symbol=? s 'x) 3]
     [(symbol=? s 'y) 4])
   (define p1 symbol-to-value_38)
)
```

p1 is now a function with the x and y values we supplied to my-make-posn coded in.

To get out the x-value, we can use (p1 'x):

```lisp
   (p1 'x)
=> 3
```
We can define a few convenience functions to simulate posn-x and posn-y:

```scheme
(define (my-posn-x p) (p 'x))
(define (my-posn-y p) (p 'y))
```

If we apply my-make-posn again with different values, it will produce a different rewritten and lifted version of symbol-to-value, say `symbol-to-value_39`.

We have just seen how to implement structures without using lists.
Our trace made it clear that the result of a particular application, say (my-make-posn 3 4), is a “copy” of symbol-to-value with 3 and 4 substituted for x and y, respectively.

That “copy” can be used much later, to retrieve the value of x or y that was supplied to my-make-posn.

This is possible because the “copy” of symbol-to-value, even though it was defined in a local definition, survives after the evaluation of the local is finished.