14: Functions as Values

First class values

Racket is a *functional programming language*, primarily because Racket's functions are **first class values**.

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

1. *consumed* as function arguments
2. *produced* as function results
3. *bound* to identifiers
4. *stored* in lists and structures

Functions are first class values in the *Intermediate Student* (and above) versions of Racket.

> Change your language level to *Intermediate Student with Lambda*.

First class values in other languages

Functions as first-class values have 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, and Python.

Consuming functions

In *Intermediate Student* a function can consume another function as an argument:

```
(define (foo f x y) (f x y))

(foo + 2 3) ⇒ (+ 2 3) ⇒ 5

(foo * 2 3) ⇒ (* 2 3) ⇒ 6

(foo append (list 'a 'b 'c) (list 1 2 3))
⇒ (append (list 'a 'b 'c) (list 1 2 3))
⇒ (list 'a 'b 'c 1 2 3)
```

Is this useful?

Consider two similar functions, `eat-apples` and `keep-odds`.

> Example: Eating apples

Consider two similar functions, `eat-apples` and `keep-odds`.

```
(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))]))
```

> Example: Keeping odd numbers

Consider two similar functions, `eat-apples` and `keep-odds`.

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

> Example: Abstracting out differences

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.

Because functions are first class values, we can write one function to do both these tasks because we can supply the predicate to be used as an argument to that function.

> Abstracting `keep-odds` to `my-filter`

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

(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))]))

(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))
```

» Tracing `my-filter`

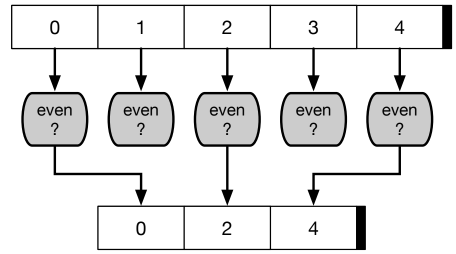
```
(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))

(my-filter even? (list 0 1 2 3 4))
⇒ (cond [(empty? (list 0 1 2 3 4)) empty]
        [(even? (first (list 0 1 2 3 4)))
         (cons (first (list 0 1 2 3 4))
               (my-filter even? (rest (list 0 1 2 3 4))))]
        [else (my-filter even? (rest (list 0 1 2 3 4)))]))
⇒ (cons 0 (my-filter even? (list 1 2 3 4)))
⇒ (cons 0 (my-filter even? (list 2 3 4)))
⇒* (cons 0 (cons 2 (cons 4 empty)))
```

`my-filter` performs the same actions as the built-in function `filter`.

`filter` is available beginning with Intermediate Student.

`filter` handles the general operation of selectively keeping items on a list.



`filter` is an example of a **higher order function**. Higher order functions either consume a function or produce a function (or both).

We'll see more higher order functions in the next lecture module.


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

(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))
```

`filter` and other higher order functions provided in Racket are used to apply common patterns of simple recursion.

We'll discuss how to write contracts for them shortly.

Ex. 1

Use `filter` to write a function that keeps all multiples of 3.

`(keep-multiples3 (list 1 2 3 4 5 6 7 8 9 10))` ⇒ `(list 3 6 9)`

Ex. 2

Use `filter` to write a function that keeps all multiples of 2 or 3.

`(keep-multiples23 (list 1 2 3 4 5 6 7 8 9 10))` ⇒ `(list 2 3 4 6 8 9 10)`

Ex. 3

Use `filter` to write a function that consumes a `(listof Num)` and keeps only values between 10 and 30, inclusive.

```
(check-expect (keep-inrange (list -5 10.1 12 7 30 3 19 6.5 42))
              (list 10.1 12 30 19))
```

Ex. 4

Use `filter` to write a function that consumes a `(listof Str)` and removes all strings of length greater than 6.

```
;; (keep-short lst) Keep all the values in lst of length at most 6.
;; Example:
(check-expect (keep-short (list "Strive" "not" "to" "be" "a" "success"
                                "but" "rather" "to" "be" "of" "value"))
              (list "Strive" "not" "to" "be" "a"
                    "but" "rather" "to" "be" "of" "value"))

;; keep-short: (listof Str) → (listof Str)
```

Ex. 5

Write a function `(sum-odds-or-evens lst)` that consumes a `(listof Int)`. If there are more evens than odds, the function returns the sum of the evens. Otherwise, it returns the sum of the odds.
Use **`local`**.

> Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as `filter`.
Advantages include:

1. Reducing code size.
2. Avoiding cut-and-paste.
3. Fixing bugs in one place instead of many.
4. Improving one functional abstraction improves many applications.

We will do more of this in the next lecture module.

Producing functions

We saw in lecture module 14 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.

> Example: make-adder


```
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

What is `(make-adder 3)`?
We can answer this question with a trace.

```
(make-adder 3)
⇒ (local [(define (f m) (+ 3 m))] f)
⇒ (define (f_1 m) (+ 3 m)) f_1
```

`(make-adder 3)` is the renamed function `f_1`, 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.

> Example: make-adder applied immediately


```
((make-adder 3) 4)
⇒ ((local [(define (f m) (+ 3 m))] f) 4)
⇒ (define (f_1 m) (+ 3 m)) (f_1 4)
⇒ (+ 3 4) ⇒ 7
```

Before	**Now**
First position in an application must be a built-in or user-defined function.	First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments.
A function name had to follow an open parenthesis.	A function application can have two or more open parentheses in a row: `((make-adder 3) 4)`.

» A note on scope


```
(define (add3 m)
  (+ 3 m))
```

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
```

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

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

Using **`local`** to produce a function gives us a way to create semi-custom functions "on the spot" to use in expressions. This is particularly useful with higher order functions such as `filter`.

In the next lecture module we'll see an easier way to produce functions that are only used once – like `eat-apples`.

Ex. 6

Write a function `(make-divisible? n)` that produces a predicate function. The predicate function consumes a `Int`, returns `true` if its argument is divisible by `n`, and `false` otherwise.

You may test your function by having it produce a function for `filter`:

```
(check-expect (filter (make-divisible? 2) (list 0 1 2 3 4 5 6 7 8 9))
              (list 0 2 4 6 8))
(check-expect (filter (make-divisible? 3) (list 0 1 2 3 4 5 6 7 8 9))
              (list 0 3 6 9))
(check-expect (filter (make-divisible? 4) (list 0 1 2 3 4 5 6 7 8 9))
              (list 0 4 8))
```


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
```

» Tracing a bound identifier

How does this work?

```
(define add2 (make-adder 2))
⇒ (define add2 (local [(define (f m) (+ 2 m))] f))
⇒ (define (f_1 m) (+ 2 m)) ; rename and lift out f
  (define add2 f_1)

(add2 3)
⇒ (f_1 3)
⇒ (+ 2 3)
⇒ 5
```

Storing functions in lists & structures

Recall our code in lecture module 11 for evaluating arithmetic expressions (just + and * for simplicity):

```
(define-struct opnode (op args))
;; An OpNode is a (make-opnode (anyof '* '+) (listof AExp)).
;; An AExp is (anyof Num OpNode)

;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list ))) 1)

;; eval: AExp → Num
```

> Example: eval and apply from M11


```
;; eval: AExp → Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (my-apply (opnode-op exp) (opnode-args exp))]))

;; (my-apply op args) applies the arithmetic operator op to args.
;; my-apply: (anyof '+ '*) (listof AExp) → Num
(define (my-apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+) (+ (eval (first args))
                             (my-apply op (rest args)))]
        [(symbol=? op '*) (* (eval (first args))
                             (my-apply op (rest args)))]))
```

> Example: Evaluating expressions with functions

In `opnode` we can replace the symbol representing a function with the function itself:

```
(define-struct opnode (op args))
;; An opnode is a (make-opnode ??? (listof AExp))
;; An AExp is (anyof Num opnode)

(check-expect (eval 3) 3)
(check-expect (eval (make-opnode + (list 2 3 4))) 9)
(check-expect (eval (make-opnode + empty)) 0)
```

Some observations about Intermediate Student that will be handy:

`(+ 1 2)` ⇒ `3`	`(* 2 3)` ⇒ `6`
`(+ 1)` ⇒ `1`	`(* 2)` ⇒ `2`
`(+ )` ⇒ `0`	`(* )` ⇒ `1`

> Example: Evaluating expressions with functions

`eval` does not change. Here are the changes to `my-apply`:

Old:

```
(define (my-apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+) (+ (eval (first args))
                             (my-apply op (rest args)))]
        [(symbol=? op '*) (* (eval (first args))
                             (my-apply op (rest args)))]))
```

New:

```
(define (my-apply op args)
  (cond [(empty? args) (op )]
        [else  (op (eval (first args))
                   (my-apply op (rest args)))]))
```

> Example: Observations

This works for any binary function that is also defined for zero arguments.

Next steps:

We know that a structure with *n* fields can be replaced with an *n*-element list.

For example:
`(eval (list + 1 (list * 3 3 3)))`
vs.
`(eval (make-opnode + (list 1 (make-opnode * (list 3 3 3)))))`

Quoting is still another way to represent a list. Using that technique,
`(eval (list + 1 (list * 3 3 3)))` becomes `(eval '(+ 1 (* 3 3 3)))` – a very natural representation.

This seems like a ‘win’, but...

Aside: Quoting

Quote notation or **quoting** gives a super-compact notation for lists.

`cons` notation emphasizes a fundamental characteristic of a list – it has a first element and the rest of the elements. Elements of the list can be computed as the list is constructed. But writing out a list with `cons` notation is unwieldy.

`list` notation makes our lists more compact but loses the reminder about the first element and the rest. Like `cons`, elements of the list can be computed as the list is constructed.

Quote notation is even more compact but loses the ability to compute elements during construction. This implies that every quoted list is a literal value. A quoted list cannot (easily) contain a function.

Aside: Quoting (cont.)

Examples:

1. `'1` ⇒ `1`, `'"ABC"` ⇒ `"ABC"`, `'earth` ⇒ `'earth`
2. `'(1 2 3)` ⇒ `(list 1 2 3)`
3. `'(a b c)` ⇒ `(list 'a 'b 'c)`
4. `'(1 ("abc" earth) 2)` ⇒ `(list 1 (list "abc" 'earth) 2)`
5. `'(1 (+ 2 3))` ⇒ `(list 1 (list '+ 2 3))`
6. `'()` ⇒ `empty`
7. `'(1 2 (make-posn 3 4) 5)` ⇒ `(list 1 2 (list 'make-posn 3 4) 5)`

`'X` is an abbreviation for `(quote X)`. `quote` is a special form; it does not evaluate its arguments in the normal fashion.

> CS135 will use quoting to represent lists more compactly for this version of `eval` and `apply` and to represent graphs in M18. We will not use it elsewhere and it will not be tested.

Ex. 7

Convert each value into quote notation, and enter the quoted version into DrRacket. (Your solution should contain the quote symbol, `'`, but should not contain `cons` or `list`.) If your quoted code is correct, DrRacket will convert it back to the same code in list notation.

1. `(cons 4 (cons "Donkey" (cons 'ice-cream empty)))`
2. `(list 'paper 'pen "eraser" (list 32 'pencil (list "calculator")))`

Recap

We'd like to use quoted lists to make the input to `eval` more natural:

```
(eval '(+ 2 (* 3 4) (+ 5 6)))
```

However, quoting turns the `+` and `*` functions into symbols: `'+` and `'*`.

Can we implement `eval` and `apply` without resorting to another **`cond`** and lots of boilerplate code?

Yes: create a dictionary (association list) that maps each symbol to a function.

> Example: Functions in a table (1/2)


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

;; (lookup-al key al) finds the value in al corresponding to key
;; lookup-al: Sym AL → ???
(define (lookup-al key al)
  (cond [(empty? al) false]
        [(symbol=? key (first (first al))) (second (first al))]
        [else (lookup-al key (rest al))]))
```

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

```
((lookup-al '+ trans-table) 3 4 5) ⇒ 12
```

> Example: Functions in a table (2/2)


```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(cons? ex) (my-apply (lookup-al (first ex) trans-table)
                              (rest ex))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: ??? (listof AExp) → Num
(define (my-apply op args)
  (cond [(empty? args) (op )]
        [else  (op (eval (first args))
                   (my-apply op (rest args)))]))
```

> Functions in lists and structures (summary)

- We've stored functions in both a structure and a list.
- Using a function instead of a symbol got rid of a lot of boiler-plate code in `apply`.
- Using quote notation made our expressions much more succinct, but forced us to again deal with symbols to represent functions.
- Putting symbols and functions in an association list provided a clean solution.
- Adding a new binary function (that is also defined for 0 arguments) only requires a one line addition to `trans-table`.

Functions as first class values (summary)

As a first class value, we can do anything with a function that we can do with other values. We saw them all in the last example:

- **Consume:** `my-apply` consumes the operator
- **Produce:** `lookup-al` looks up a symbol, producing the corresponding function
- **Bind:** results of `lookup-al` to `op`
- **Store:** stored in `trans-table`

Contracts and types

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

Until now, the type of data has been constructed from building blocks consisting of basic (built-in) types, defined (struct) types, `anyof` types, and list types such as `(listof Sym)`.

What is the type of a function consumed or produced by another function?

Ex. 8 Using the code in the commentary and without looking at the video again, reproduce the logic to arrive at the contracts for `make-between`, `in-discontinuous-range`, and `make-in-discontinuous-range`.

> Contracts as types

We can use the contract for a function as its type.

For example, the type of `>` is `(Num Num → Bool)`, because that's the contract of that function.

We can then use type descriptions like this in contracts for functions which consume or produce other functions.

> Contracts as types: Examples


```
;; my-apply: (Num Num -> Num) (listof AExp) -> Num
(define (my-apply op args) ...)
```

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

;; (lookup-al key al) finds the value in al corresponding to key
;; lookup-al: Sym (listof (list Sym (Num Num → Num))) →
;;       (anyof false (Num Num → Num))
(define (lookup-al key al)
  (cond [(empty? al) false]
        [(symbol=? key (first (first al))) (second (first al))]
        [else (lookup-al key (rest al))]))
```

> Contracts for higher order functions

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

We might be tempted to conclude that its contract is
`(Any → Bool) (listof Any) → (listof Any)`.

But this is not specific enough.

Consider the application `(filter odd? (list 1 2 3))`. This does not obey the contract (the contract for `odd?` is `Int → 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 of `(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 and the type of list.

To express this, we use a **type variable**, such as X, and use it in different places to indicate where the same type is needed.

> The contract for filter

`filter` consumes a list of type `(listof X)`.

That implies that the predicate must consume an `X`. The predicate must also produce a `Boolean`. It thus has a contract (and type!) of `(X → Bool)`.

`filter` produces a list of the same type it consumes.

Therefore the contract for `filter` is:

```
;; filter: (X → Bool) (listof X) → (listof X)
```

Here `X` stands for the unknown data type of the list.

We say `filter` is **polymorphic** or **generic**; it works on many different types of data.

> Type variables

We have used type variables in contracts for a long time. For example, `(listof X)`.

What is new is using the same variable multiple times in the same contract. This indicates a relationship between parts of the contract. For example, `filter`'s list and predicate are related.

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

Type variable vs. `Any`

Recall from M08 that `Any` is just an abbreviation for `(anyof Nat Int Num Sym Bool Str ...)` where ... is every other type in your program. Use a type variable unless the parameter can always take `(anyof Nat Int Num Symb Bool Str ...)`.

> Using contracts to understand

Many of the difficulties one encounters in using higher order 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.

Ex. 9

Write a version of insertion sort, `(isort pred? lst)`, which consumes a predicate and a `(listof X)` and produces `lst` in sorted order.

```
;; (isort pred? lst) sorts the elements of lst so that adjacent
;;    elements satisfy pred?.
;; Examples:
(check-expect (isort < (list 3 4 2 5 1))
              (list 1 2 3 4 5))
(check-expect (isort > (list 3 4 2 5 1))
              (list 5 4 3 2 1))
(check-expect (isort string<? (list "can" "ban" "fan"))
              (list "ban" "can" "fan"))
```

What is the contract for `isort`?

Ex. 10

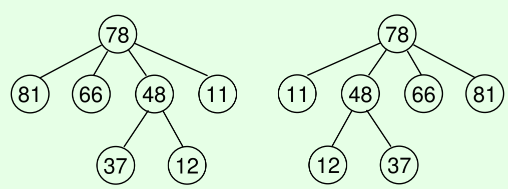
Consider `create-checker`, a function which consumes a function, `f`, and a list of numbers, `answers`. `f` consumes a string and produces a number. `create-checker` produces a function that consumes a string. It produces `true` if `f` applied to the argument produces a number that is in `answers`, and `false` otherwise. Write the contract for `create-checker`.

Ex. 11

Here is a definition of a generalized tree where any node can have many children:

```
(define-struct gnode (key children))
;; A GT (Generalized Tree) is a (make-gnode Nat (listof GT))
```

Write a function `tested-gt-sum` which consumes a predicate and a GT. The predicate consumes a Nat. The function `tested-gt-sum` produces the sum of all keys in the GT for which the predicate produces `true`.
For example, when called with the predicate `odd?` and one of the following GTs, the function produces 129.



Goals of this module

- You should understand the idea of functions as first-class values: how they can be supplied as arguments, produced as values, bound to identifiers, and placed in lists and structures.
- You should understand how a function's contract can be used as its type. You should be able to write contracts for functions that consume and/or produce functions.

The following functions and special forms have been introduced in this module:

`filter`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`* + - ... / < <= = > >= abs add1` **`and`** `append boolean? ceiling char-alphabetic? char-downcase char-lower-case? char-numeric? char-upcase char-upper-case? char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect check-within` **`cond`** `cons cons? cos` **`define define-struct`** `define/trace e eighth` **`else`** `empty? equal? error even? exp expt fifth filter first floor fourth integer? length list list->string list?` **`local`** `log max min modulo negative? not number->string number? odd?` **`or`** `pi positive? quotient remainder rest reverse round second seventh sgn sin sixth sqr sqrt string->list string-append string-downcase string-length string-lower-case? string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=? string>? string? sub1 substring symbol=? symbol? tan third zero?`