# Functions as First Class Values

**Readings:** HtDP, sections 19-20.

**Language level:** Intermediate Student

**Topics:**

- Consuming functions
- Producing functions
- Binding functions
- Storing functions
- Contracts and types
- Simulating structures

# 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.

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

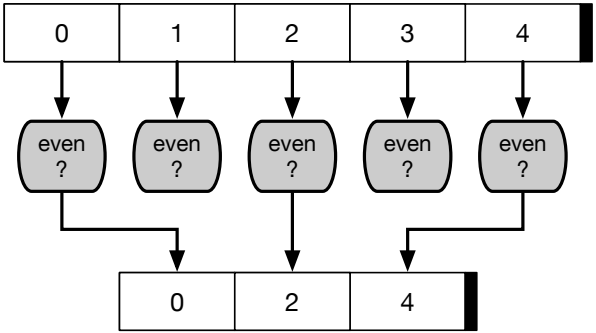
```
(my-filter even? (list 0 1 2 3 4))
⇒ (cons 0 (my-filter even? (list 1 2 3 4)))
⇒ (cons 0 (my-filter even? (list 2 3 4)))
⇒ (cons 0 (cons 2 (my-filter even? (list 3 4))))
⇒ (cons 0 (cons 2 (my-filter even? (list 4))))
⇒ (cons 0 (cons 2 (cons 4 (my-filter even? empty))))
⇒ (cons 0 (cons 2 (cons 4 empty)))
```

`my-filter` handles the general operation of removing items from lists.

Functions such as `my-filter` that consume a `(listof X)` and a function to generalize it are called **abstract list functions** (abbreviated **ALFs**) or **higher order functions**.

We'll see more ALFs in the next lecture module.

# » `my-filter`, visually



# > Using my-filter

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

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

The function `filter`, which behaves identically to our `my-filter`, is built into Intermediate Student and full Racket.

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

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

# Exercise 1

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

```
(keep-inrange '(-5 10.1 12 7 30 3 19 6.5 42)) ⇒ '(10.1 12 30 19)
```

# Exercise 2

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

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

# Exercise 3

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

`(keep-multiples3 '(1 2 3 4 5 6 7 8 9 10))` $\Rightarrow$ `'(3 6 9)`

# Exercise 4

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

`(keep-multiples23 '(1 2 3 4 5 6 7 8 9 10))` $\Rightarrow$ `'(2 3 4 6 8 9 10)`

# > Advantages of functional abstraction

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

- Reducing code size.
- Avoiding cut-and-paste.
- Fixing bugs in one place instead of many.
- 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 12 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_42 m) (+ 3 m)) f_42
```

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

# > Example: make-adder applied immediately

```
((make-adder 3) 4)
⇒ ((local [(define (f m) (+ 3 m))] f) 4)
⇒ (define (f_42 m) (+ 3 m)) (f_42 4)
⇒ (+ 3 4) ⇒ 7
```

**Before**

First position in an application must be a built-in or user-defined function.

A function name had to follow an open parenthesis.

**Now**

First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments.

A function application can have two or more open parentheses in a row: `((make-adder 3) 4)`.

```
(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.

In the next lecture module we'll see an easier way to produce functions. That will allow us to produce functions "on the spot" to be consumed by functions such as `filter`. This is very useful.

# 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
```

# » Tracing a bound identifier

How does this work?

```
(define add2 (make-adder 2))
⇒ (define add2 (local [(define (f m) (+ 2 m))] f))
⇒ (define (f_43 m) (+ 2 m)) ; rename and lift out f
  (define add2 f_43)

(add2 3)
⇒ (f_43 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 ainode (op args))
;; An AINode is a (make-ainode (anyof '+ '*) (listof AExp)))
;; An AExp is (anyof Num AINode)

;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(check-expect (eval 3) 3)
(check-expect (eval (make-ainode '+ '(2 3 4))) 9)
(check-expect (eval (make-ainode '+ '())) 0)
```

# > Example: `eval` and `apply` from M11

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(ainode? ex) (my-apply (ainode-op ex) (ainode-args ex))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: op (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 `ainode` we could replace the symbol that represents a function with the function itself:

```
(define-struct ainode (op args))

(check-expect (eval 3) 3)
(check-expect (eval (make-ainode + '(2 3 4))) 9)
(check-expect (eval (make-ainode + '())) 0)
```

Some observations about Intermediate Student that will be handy:

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

`(* 2 3)` ⇒ `6`
`(* 2)` ⇒ `2`
`(* )` ⇒ `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)))]))
```

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.

Quoting gives a really compact notation that may be easier to read.

For example:

```
(eval '(+ 1 (* 3 3 3)))
```

vs.

```
(eval (make-ainode + (list 1 (make-ainode * (list 3 3 3)))))
```

Seems like a ‘win’, but now our operator is a symbol again!

# > 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: op (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 was either a basic (built-in) type, a defined (struct) type, an `anyof` type, or a list type such as `(listof Sym)`.

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

# > 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: Example

```
(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 abstract list functions

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

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

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.

# > 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.

# Example: Simulating structures

We can use the ideas of producing and binding functions to simulate structures. Consider a structure representing a point:

```
(define-struct point (x y))
;; A Point is a (make-point Num Num)
```

This can be simulated with a function:

```
;; (mk-point x y) produces a "structure" representing (x,y).
;; mk-point: Num Num → ____________
(define (mk-point x y)
  (local [(define (symbol-to-value s)
            (cond [(symbol=? s 'x) x]
                  [(symbol=? s 'y) y]))]
    symbol-to-value))
```

# > Tracing `mk-point` (1/2)

```
(define p1 (mk-point 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 have been substituted into the **`local`** definition.

We now rename `symbol-to-value` and lift it out.

```
⇒ (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 `mk-point` coded in.

# > Tracing `mk-point` (2/2)

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

`(p1 'x)` ⇒ `(symbol_to_value_38 'x)` ⇒ `...` ⇒ `3`

We can define a few convenience functions to simulate the structure accessor functions `point-x` and `point-y`:

```
(define (point-x p) (p 'x))
(define (point-y p) (p 'y))
```

If we apply `mk-point` 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.

# > Simulating structures summary

Our trace made it clear that the result of a particular application, say `(mk-point 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 `mk-point`.

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.

# 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.