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.
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.
We didn’t make a big deal of it then, but we’ve already seen functions that consume functions. Recall:

\[
\text{(map \textit{sq}r \text{(list 0 1 2 3 4)) ⇒ (list 0 1 4 9 16)}
\]
\[
\text{(filter even? (list 0 1 2 3 4)) ⇒ (list 0 2 4)}
\]
\[
\text{(foldr + 0 (list 0 1 2 3 4)) ⇒ 10}
\]

We can also write our own functions that consume another function as an argument:

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

\[
(\text{foo + 2 3) ⇒ (+ 2 3 3 2) ⇒ 10}
\]
\[
(\text{foo * 2 3) ⇒ (* 2 3 3 2) ⇒ 36}
\]
\[
(\text{foo string-append "abc" "123")}
\Rightarrow (\text{string-append "abc" "123" "123" "abc")}
\Rightarrow "abc123123abc"
\]

This example is not so useful. But remember how much we could do with \textit{foldr}!

Let’s consider carefully to see what powers this gives us.
Earlier we developed a recursive function to sort in non-decreasing order:

```
;; (sort lon) sorts the elements of lon in non-decreasing order
;; sort: (listof Num) -> (listof Num)
(define (sort lon)
  (cond [(empty? lon) empty]
        [else (insert (first lon) (sort (rest lon)))])))
```

```
;; (insert n slon) inserts the number n into the sorted list slon
;; so that the resulting list is also sorted.
;; insert: Num (listof Num) -> (listof Num)
;; requires: slon is sorted in non-decreasing order
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [([<= n (first slon)) (cons n slon)]
         [else (cons (first slon) (insert n (rest slon)))])))
```

To change it to sort in non-increasing order, the only change was to replace \(<=\) with \(>=\).
Let’s replace the $\leq$ with a parameter: a predicate that tells us if a pair of values is ordered.

Looking at the code, we only use $\leq$ in the `insert` function. So we don’t need to change `sort` at all... except it needs to pass this predicate on to each call to `insert`.

```
;; (isort lon pred?) sorts the elements of lon.
;; (pred? x y) tells us if x comes before y.

;; isort: (listof ...) ... -> (listof ...)
(define (isort lon pred?)
  (cond [(empty? lon) empty]
        [else (insert (first lon) (isort (rest lon) pred?)
                     pred?)])
)
```

We might say: “to sort a list, sort the rest of the list, then insert the first item into that list, following the rule named `pred?`.”
Then we need only make a small change to \texttt{insert}:

\begin{verbatim}
;; (insert n slst pred?) inserts the value n into the sorted list slst
;; so that the resulting list is also sorted.
;; (pred? x y) tells us if x comes before y.

;; insert: ... (listof ...) ... -> (listof ...)
;; requires: slst is already sorted according to pred?.
(define (insert n slst pred?)
  (cond [(empty? slst) (cons n empty)]
        [(pred? n (first slst)) (cons n slst)]
        [else (cons (first slst) (insert n (rest slst) pred?))]))
\end{verbatim}

Instead of writing <=, we simply write in the predicate. The only other change is to pass this value to the recursive call.
Here, by writing a function that consumes a predicate, we can sort any kind of value, in any kind of order.

Functions that consume functions can do \textit{a lot}.

**Exercise**

Write a predicate \texttt{wacky?} that allows you to use \texttt{isort} to sort a \texttt{(listof (anyof Num Str))} so all numbers appear at the front, in descending order, and strings appear at the back, in ascending order.

For example:

\begin{verbatim}
(check-expect (isort (list 900 "charlie" 42 "alpha" -8 "bravo") wacky?)
  (list -8 42 900 "charlie" "bravo" "alpha"))
\end{verbatim}

In the next module we’ll look again at \texttt{map}, \texttt{foldr}, \texttt{foldl}, and \texttt{filter}, and see further how we can write functions that consume 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) (+ n 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

\[
\text{((make-adder 3) 4)  \\
⇒ ((local ((define (f m) (+ 3 m))) f) 4)  \\
⇒ (define (f_1 m) (+ 3 m)) (f_1 4)  \\
⇒ (+ 3 4) ⇒ 7}
\]

Before
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: \((\text{make-adder 3} \ 4)\).
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 functions such as `filter` and `map`.

Recall also how we used `lambda` to produce functions that are only used once. We’ll take another look at it in the next module.
Write a function (make-divisible? n) that produces a predicate function. The predicate function consumes a Int, produces 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 \texttt{make-adder} can be bound to an identifier and then used repeatedly.

\begin{verbatim}
(define add2 (make-adder 2))
(define add3 (make-adder 3))

(add2 3) \Rightarrow 5
(add3 10) \Rightarrow 13
(add3 13) \Rightarrow 16
\end{verbatim}
How does this work?

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

```scheme
(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 '* empty)) 1)

;; eval: AExp -> Num
```
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (my-apply (opnode-op exp) (opnode-args exp))])))
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 (anyof ...) (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
(+ 1) ⇒ 1
(+ ) ⇒ 0
(* 2 3) ⇒ 6
(* 2) ⇒ 2
(* ) ⇒ 1
```
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)))]))
```

We rely upon the fact that (+ ) ⇒ 0 and (*) ⇒ 1. Any other function we use this with needs to do something similar.
Now we have a way to evaluate an expression structured using `make-opnode`, where the `op` field stores the actual function to evaluate, rather than a symbol that represents it.

Does this mean we never need to “translate” a symbol or other value into a function?

No!

Imaging if you wanted to write a Racket interpreter. You get a `Str` such as

```
(expt (+ 2 3) e)
```

What does "expt" mean? What does "+" mean? "e"?

Or consider writing a spreadsheet program, such as Libreoffice Calc. Each cell may contain a formula, and is stored in an XML file like so:

```
<table:table-cell table:formula="of:=AVERAGE([.A1:.A2])"
```

The computer needs to be able to know what "AVERAGE(...)" means.

So we need to be able to conveniently convert a value into a function.
(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
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp -> Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(opnode? ex) (my-apply (lookup-al (opnode-op ex) trans-table)
                               (opnode-args ex))]
        [else #f]))

;; (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 `my-apply`.
- 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`.
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?
Exercise

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

For example, the type of > is \((\text{Num\ Num} \rightarrow \text{Bool})\), 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.
(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))])))
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.
An application of \((\text{filter } \text{pred? } \text{lst})\) 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 \textbf{type variable}, such as \(x\), and use it in different places to indicate where the same type is needed.
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.
We have used type variables in contracts for a long time. For example, \((\text{listof } X)\).

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, \textit{filter}'s list and predicate are related.

We will soon see examples where more than one type variable is needed in a contract.
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.
Consider the function \((isort \ lst \ pred?)\) that we wrote earlier. It consumes a \((listof \ X)\) and a predicate, and produces \(lst\) in sorted order.

\[
\begin{align*}
\text{(check-expect)} \ & (isort \ (list \ 3 \ 4 \ 2 \ 5 \ 1) \ <=) \ (list \ 1 \ 2 \ 3 \ 4 \ 5)) \\
\text{(check-expect)} \ & (isort \ (list \ "charlie" \ "delta" \ "bravo" \ "alpha") \ string<?) \\
& \quad (list \ "alpha" \ "bravo" \ "charlie" \ "delta") \\
\text{(check-expect)} \ & (isort \ (list \ #\t \ #\b \ #\e \ #\a) \ char<?) \ (list \ #\a \ #\b \ #\e \ #\t))
\end{align*}
\]

What is the contract for \(isort\)?
We can use the ideas of producing and binding functions to simulate structures. Consider a structure representing a point:

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

This can be simulated with a function:

```scheme
;; (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))
```

(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_1 s)
    (cond [(symbol=? s 'x) 3]
          [(symbol=? s 'y) 4]])
    (define p1 symbol-to-value_1)

p1 is now a function with the x and y values we supplied to mk-point coded in.
To get out the $x$ value, we can use $(p1 \ 'x)$:

$$(p1 \ 'x) \Rightarrow (\text{symbol-to-value}_1 \ 'x) \Rightarrow \ldots \Rightarrow 3$$

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

```scheme
(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_2`.

We have just seen how to implement structures without using lists.
Our trace made it clear that the result of a particular application, say \( (\text{mk-point} \ 3 \ 4) \), is a “copy” of \text{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 \text{mk-point}.

This is possible because the “copy” of \text{symbol-to-value}, even though it was defined in a \textit{local} definition, survives after the evaluation of the \textit{local} is finished.
Write a function with contract: `(listof (list X Y)) -> (X -> (anyof Y false))`

Does it feel like there are many such functions? There are very few!

Discuss this in the discussion board and figure out what the function needs to do. Carefully think about some examples. Consider the following `(listof (list X Y))`:

```
(list (list 1 "one")
     (list 2 "two")
     (list 3 "three"))
```
Anonymous functions

```scheme
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
(make-adder 3)
```

The result of evaluating this expression is a function.

What is its name? It is **anonymous** (has no name).

This is sufficiently valuable that there is a special mechanism for it.
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))

This is a little unsatisfying, because not-symbol-apple? is such a small and relatively useless function.

It is unlikely to be needed elsewhere.

We can avoid cluttering the top level with such definitions by putting them in local expressions.
Producing anonymous functions

(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
              (not (symbol=? item 'apple)))]
    (filter not-symbol-apple? lst)))

This is as far as we would go based on our experience with local. But now that we can use functions as values, the value produced by the local expression can be the function not-symbol-apple?.

We can give that value as an argument to filter.
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                  (not (symbol=? item 'apple)))]
            not-symbol-apple?)
    lst))

But this is still unsatisfying. Why should we have to name `not-symbol-apple?` at all? In the expression `(* (+ 2 3) 4)`, we didn’t have to name the intermediate value 5.

Racket provides a mechanism for constructing a nameless function which can then be used as an argument.
Recalling \texttt{lambda}

\begin{verbatim}
(local [(define (name-used-once x_1 ... x_n) exp)]
    name-used-once)
\end{verbatim}

can also be written
\begin{verbatim}
(lambda (x_1 ... x_n) exp)
\end{verbatim}

\texttt{lambda} can be thought of as “make-function”.

It can be used to create a function which we can then use as a value – for example, as the value of the first argument of \texttt{filter}.
We can use lambda to replace

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

with the following:

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

But how could this work? As usual, we’ll approach it with a trace.
> Example: define eat-apples with **lambda**

We will image that **filter** is the following function, which accomplishes the same task:

```scheme
;; my-filter: (X -> Bool) (listof X) -> (listof X)
(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))]))
```
(define (eat-apples lst)
  (my-filter (lambda (item) (not (symbol=? item 'apple))) lst))

(eat-apples (list 'pear 'apple))
⇒ (my-filter (lambda (item) (not (symbol=? item 'apple))) (list pear apple))
⇒ (cond [[(empty? (list 'pear 'apple)) empty]
           [((lambda (item) (not (symbol=? item 'apple))))
              (first (list 'pear 'apple))]
            (cons (first (list 'pear 'apple))
              (my-filter (lambda (item) (not (symbol=? item 'apple)))
                         (rest (list 'pear 'apple))))]
           [else (my-filter (lambda (item) (not (symbol=? item 'apple)))
                         (rest (list 'pear 'apple)))]])

What does the underlined expression mean?
The double parentheses indicates that we need to compute the function (the inner expression) to apply to the arguments (the outer expression). In this case, “compute” means create the function using `lambda`.

Lambda expressions are already in the simplest form, so the next step in the trace is to reduce the arguments to values:

\[
\Rightarrow ((\lambda (item) (\text{not} (\text{symbol=? item 'apple)))) (\text{first} (\text{list} 'pear 'apple)))
\]

Finally, each argument is matched with the corresponding parameter and then substituted into the function’s body expression each place that parameter appears. The entire expression is replaced with the rewritten body expression.

\[
\Rightarrow (\text{not} (\text{symbol=? 'pear 'apple}))
\]
We can use `lambda` to simplify `make-adder`. Instead of

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

we can write:

```scheme
(define (make-adder n)
  (lambda (m) (+ n m)))
```
Introducing lambda

lambda is available in Intermediate Student with lambda, and discussed in section 24 of the textbook.

Lambda is the name of the Greek letter $\lambda$, which was used as notation in the first formal model of computation.

We’ll learn more about its central importance in the history of computation in the last lecture module.
When we first encountered \(((\text{make-adder~}3)~4)\), we noted the differences in function application:

**Before Module 13**
First position in an application must be a built-in or user-defined function.

A function name had to follow an open parenthesis.

**Module 13 and later**
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: \(((\text{make-adder~}3)~4)\) or \(((\text{lambda~}(x~y)~(+~x~y~x))~1~2)\).

These observations are also true of using \text{lambda}. 


Example: Tracing with lambda (2)

(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))

(define make-adder (lambda (x) (lambda (y) (+ x y))))

((make-adder 3) 4) ⇒ ;; substitute the lambda expression
(((lambda (x) (lambda (y) (+ x y))) 3) 4) ⇒
(((lambda (y) (+ 3 y)) 4) ⇒
  (+ 3 4) ⇒ 7

make-adder is defined as a constant using lambda. Like any other constant, make-adder is replaced by its value (the lambda expression).
lambda underlies the definition of functions.

Until now, we have had two different types of definitions.

;; a definition of a numerical constant
(define interest-rate 3/100)

;; a definition of a function to compute interest
(define (interest-earned amount) (* interest-rate amount))

But there is really only one kind of define, which binds a name to a value.
Internally,
\[
\text{(define (interest-earned amount) (* interest-rate amount))}
\]
is translated to
\[
\text{(define interest-earned (lambda (amount) (* interest-rate amount)))}
\]
which binds the name \text{interest-earned} to the function value
\[
\text{(lambda (amount) (* interest-rate amount))}.
\]
We should change our semantics for function definition to represent this rewriting.

But doing so would make traces much harder to understand.

As long as the value of defined constants (now including functions) cannot be changed, we can leave their names unsubstituted in our traces for clarity.

In stepper questions, if a function is defined using function syntax, you can skip the lambda substitution step. If a function is defined as a constant using lambda, you must include the lambda substitution step.
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.
In this module we added the following to our toolbox:

* filter

These are the functions and special forms currently in our toolbox: