Functional abstraction

Readings: HtDP, sections 19-24.

Language level: Intermediate Student With Lambda

Topics:

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

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.

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))]))
```

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))]))
```


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

In the Intermediate language, functions are values. In fact, they 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. 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

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

(foo + 2 3) ⇒ 5
(foo * 2 3) ⇒ 6

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))]))
```

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))

(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 structural recursion.

We'll 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 (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.

Here's what happens if we apply it 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

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.

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

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
```


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 '∗)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [(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've 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


```
;; newapply: Sym AltAExpList → Num
(define (newapply f aaxl)
  (cond [(and (empty? aaxl) (symbol=? f '*)) 1]
        [(and (empty? aaxl) (symbol=? f '+)) 0]
        [else ((lookup-al f trans-table)
               (eval (first aaxl)) (newapply f (rest aaxl)))]))
```

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

(+) ⇒ 0 and (*) ⇒ 1

```
;; newapply: Sym AltAExpList → Num
(define (newapply f aaxl)
  (local [(define op (lookup-al f trans-table))]
    (cond [(empty? aaxl) (op)]
          [else (op (eval (first aaxl))
                    (newapply f (rest aaxl)))])))
```

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

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 (Num Num $\rightarrow$ Bool), because that's 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
(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 (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 X, and use it in different places to indicate where the same type is needed.

The contract for filter

filter consumes a predicate with contract (X → 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 filter is thus:

```
;; 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.

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.

```
(define (my-make-posn x y)
  (local
    [(define (symbol-to-value s)
       (cond [(symbol=? s 'x) x]
             [(symbol=? s 'y) y]))]
    symbol-to-value))
```

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 have been substituted into the local definition.

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

This yields:

```
(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):

```
(p1 'x) ⇒ 3
```

We can define a few convenience functions to simulate posn-x and posn-y:

```
(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.

Anonymous functions

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

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.

Producing anonymous functions

```
(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.


```
(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 then take that value and deliver it 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.

Introducing lambda

```
(local [(define (name-used-once x1 ... xn) exp)]
    name-used-once)
```

can also be written

```
(lambda (x1 ... xn) exp)
```

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 filter.

We can then replace

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

with the following:

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


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

We’re jumping ahead to it because of its central importance in Racket, Lisp, and the history of computation in general.

The designers of the teaching languages could have renamed it as they did with other constructs, but chose not to out of respect.

The word lambda comes from the Greek letter, used as notation in the first formal model of computation.

We can use lambda to simplify make-adder. Instead of

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

we can write:

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


lambda also 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))
```

There is really only one kind of define, which binds a name to a value.

Internally,

```
(define (interest-earned amount)
  (* interest-rate amount))
```

is translated to

```
(define interest-earned
  (lambda (amount) (* interest-rate amount)))
```

which binds the name interest-earned to the value (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 step.

For example, here's make-adder rewritten using lambda.

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

What is ((make-adder 3) 4)?


```
(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, so it is substituted in place of make-adder.

Syntax and semantics of Intermed. Student w/ lambda

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).

We need a rule for evaluating applications where the function being applied is anonymous (a lambda expression.)

((lambda (x1 ... xn) exp) v1 ... vn) $\Rightarrow$ exp'

where exp' is exp with all occurrences of x1 replaced by v1, all occurrences of x2 replaced by v2, and so on.

As an example:

((lambda (x y) (* (+ y 4) x)) 5 6) $\Rightarrow$ (* (+ 6 4) 5)

Suppose during a computation, we want to specify some action to be performed one or more times in the future.

Before knowing about lambda, we might build a data structure to hold a description of that action, and a helper function to consume that data structure and perform the action.

Now, we can just describe the computation clearly using lambda.

Example: character translation in strings

We'd like a function, translate, that translates one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of 'a' to a 'b'. In another, we might translate lowercase characters to the equivalent uppercase character and digits to '*'.

```
(check-expect (translate "abracadabra" ...) "bbrbcbdbbrb")
(check-expect (translate "Testing 1-2-3" ...) "TESTING *-*-*")
```

We use ... to indicate that we still need to supply some arguments.

We could imagine translate containing a cond:

```
(cond [(char=? ch #\a) #\b]
      [(char-lower-case? ch) (char-upcase ch)]
      [(char-numeric? ch) #\*]
      ...)
```

But this fails for a number of reasons:

- The rules are "hard-coded"; we want to supply them when translate is applied.
- A lower case 'a' would always be translated to 'b'; never to 'B'

But the idea is inspiring...

Suppose we supplied translate with a list of question/answer pairs:

```
;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)
```

Like cond, we could work our way through the TranslateSpec with each character. If the Question produces true, then apply the Answer to the character. If the Question produces false, go on to the next Question/Answer pair.

What are the types for Question and Answer?

Functions as first class values can help us. Both Question and Answer are functions that consume a Char.

Question produces a Bool and Answer produces a character. This completes our data definition, above:

```
;; A Question is a Char → Bool
;; An Answer is a Char → Char
```

And a completed example:

```
(check-expect (translate "Testing 1-2-3"
                         (list (list char-lower-case? char-upcase)
                               (list char-numeric? (lambda (ch) #\*))))
              "TESTING *-*-*")
```


Translate: developing the code

translate consumes a string and produces a string but we need to operate on characters. This suggests a wrapper function:

```
;; A TranslateSpec is one of:
;; * empty
;; * (cons (list Question Answer) TranslateSpec)

;; (translate s spec) translates the string s according to the given specifi
;; translate: Str TranslateSpec → Str
(define (translate s spec)
  (list→string (trans-loc (string→list s) spec)))
```



```
;; trans-loc (listof Char) TranslateSpec → (listof Char)
(check-expect (trans-loc (list #\a #\9)
                         (list (list char-lower-case? char-upcase))) (list #
(define (trans-loc loc spec)
  (cond [(empty? loc) empty]
        [(cons? loc) (cons (trans-char (first loc) spec)
                           (trans-loc (rest loc) spec))]))

(define (trans-char ch spec)
  (cond [(empty? spec) ch]
        [((first (first spec)) ch) ((second (first spec)) ch)]
        [else (trans-char ch (rest spec))]))
```

```
(check-expect (translate "Testing 1-2-3"
                         (list (list char-lower-case? char-upcase)
                               (list char-numeric? (lambda (ch) #\*))))
              "TESTING *-*-*")
(check-expect (translate "abracadabra" (list (list (lambda (ch) (char=?
                                                                 (lambda (ch) #\b))))
              "bbrbcbdbbrb")
```

The repeated lambda expressions suggest some utility functions:

```
(define (is-char? c1) (lambda (c2) (char=? c1 c2)))
(define (always c1) (lambda (c2) c1))
```


Abstracting another set of examples

Here are two early list functions we wrote.

```
(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (- (first lst)) (negate-list (rest lst)))]))

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr→tr (first payroll))
                    (compute-taxes (rest payroll)))]))
```


We look for a difference that can't be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

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

Tracing my-map

(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))

my-map performs the general operation of transforming a list element-by-element into another list of the same length.

The application
(my-map f (list x1 x2 ... xn)) has the same effect as evaluating
(list (f x1) (f x2) ... (f xn)).

We can use my-map to give short definitions of a number of functions we have written to consume lists:

```
(define (negate-list lst) (my-map - lst))
(define (compute-taxes lst) (my-map sr→tr lst))
```

How can we use my-map to rewrite trans-loc?

The contract for my-map

my-map consumes a function and a list, and produces a list.

How can we be more precise about its contract, using parametric type variables?

Built-in abstract list functions

Intermediate Student also provides map as a built-in function, as well as many other abstract list functions. Check out the Help Desk (in DrRacket, Help → Help Desk → How to Design Programs Languages → 4.17 Higher-Order Functions)

The abstract list functions map and filter allow us to quickly describe functions to do something to all elements of a list, and to pick out selected elements of a list, respectively.

Abstracting another set of examples

The functions we have worked with so far consume and produce lists.

What about abstracting from functions such as count-symbols and sum-of-numbers, which consume lists and produce values?

Let's look at these, find common aspects, and then try to generalize from the template.


```
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum-of-numbers (rest lst)))]))

(define (prod-of-numbers lst)
  (cond [(empty? lst) 1]
        [else (* (first lst) (prod-of-numbers (rest lst)))]))

(define (count-symbols lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (count-symbols (rest lst)))]))
```

Note that each of these examples has a base case which is a value to be returned when the argument list is empty.

Each example is applying some function to combine (first lst) and the result of a recursive function application with argument (rest lst) .

This continues to be true when we look at the list template and generalize from that.


```
(define (list-template lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                   (list-template (rest lst)) ...)]))
```

We replace the first ellipsis by a base value.

We replace the rest of the ellipses by some function which combines (first lst) and the result of a recursive function application on (rest lst).

This suggests passing the base value and the combining function as parameters to an abstract list function.

The abstract list function foldr

```
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                       (my-foldr combine base (rest lst)))]))
```

foldr is also a built-in function in Intermediate Student With Lambda.

Tracing my-foldr

(my-foldr f 0 (list 3 6 5)) ⇒
(f 3 (my-foldr f 0 (list 6 5))) ⇒
(f 3 (f 6 (my-foldr f 0 (list 5))) ⇒
(f 3 (f 6 (f 5 (my-foldr f 0 empty))) ⇒
(f 3 (f 6 (f 5 0))) ⇒ . . .

Intuitively, the effect of the application
(foldr f b (list x1 x2 . . . xn)) is to compute the value of the expression
(f x1 (f x2 (. . . (f xn b) . . .))).

foldr is short for “fold right”.

The reason for the name is that it can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.

foldr can be used to implement map, filter, and other abstract list functions.

The contract for foldr

foldr consumes three arguments:

- a function which combines the first list item with the result of reducing the rest of the list;
- a base value;
- a list on which to operate.

What is the contract for foldr?

Using foldr

```
(define (sum-of-numbers lst) (foldr + 0 lst))
```

If lst is (list x1 x2 ... xn), then by our intuitive explanation of foldr, the expression (foldr + 0 lst) reduces to

(+ x1 (+ x2 (+ ... (+ xn 0) ...)))

Thus foldr does all the work of the template for processing lists, in the case of sum-of-numbers.

The function provided to foldr consumes two parameters: one is an element on the list which is an argument to foldr, and one is the result of reducing the rest of the list.

Sometimes one of those arguments should be ignored, as in the case of using foldr to compute count-symbols.

The important thing about the first argument to the function provided to foldr is that it contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to foldr in this case can ignore the value of the first parameter, and just add 1 to the reduction of the rest of the list.

```
(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))
```

The function provided to foldr, namely
(lambda (x rror) (add1 rror)),
ignores its first argument.

Its second argument is the **r**esult of **r**ecursing **o**n the **r**est (rror) of the list (in this case the length of the rest of the list, to which 1 must be added).

More examples

What do these functions do?

```
(define (bar lon)
  (foldr max (first lon) (rest lon)))

(bar '(1 5 23 3 99 2))

(define (foo los)
  (foldr (lambda (s rror) (+ (string-length s) rror)) 0 los))

(foo '("one" "two" "three"))
```


Using foldr to produce lists

So far, the functions we have been providing to foldr have produced numerical results, but they can also produce cons expressions.

foldr is an abstraction of structural recursion on lists, so we should be able to use it to implement negate-list from module 05.

We need to define a function (lambda (x rror) ...) where x is the first element of the list and rror is the result of the recursive function application.

negate-list takes this element, negates it, and conses it onto the result of the recursive function application.

The function we need is

```
(lambda (x rror) (cons (— x) rror))
```

Thus we can give a nonrecursive version of negate-list (that is, foldr does all the recursion).

```
(define (negate-list lst)
  (foldr (lambda (x rror) (cons (— x) rror)) empty lst))
```

Because we generalized negate-list to map, we should be able to use foldr to define map.

Let's look at the code for my-map.

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

Clearly empty is the base value, and the function provided to foldr is something involving cons and f.

In particular, the function provided to foldr must apply f to its first argument, then cons the result onto its second argument (the reduced rest of the list).

```
(define (my-map f lst)
  (foldr (lambda (x rror) (cons (f x) rror)) empty lst))
```

We can also implement my-filter using foldr.

Imperative languages, which tend to provide inadequate support for recursion, usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.

Anything that can be done with the list template can be done using foldr, without explicit recursion (unless it ends the recursion early, like insert).

Does that mean that the list template is obsolete?

No. Experienced Racket programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

Generalizing accumulative recursion

Let’s look at several past functions that use recursion on a list with one accumulator.

```
;; code from lecture module 12

(define (sum-list lon)
  (local [(define (sum-list/acc lst sum-so-far)
            (cond [(empty? lst) sum-so-far]
                  [else (sum-list/acc (rest lst)
                                      (+ (first lst) sum-so-far))]))]
    (sum-list/acc lon 0)))
```

```
;; code from lecture module 9 rewritten to use local

(define (my-reverse lst0)
  (local [(define (my-rev/acc lst list-so-far)
            (cond [(empty? lst) list-so-far]
                  [else (my-rev/acc (rest lst)
                                    (cons (first lst) list-so-far))]))]
    (my-rev/acc lst0 empty)))
```


The differences between these two functions are:

- the initial value of the accumulator;
- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.


```
(define (my-foldl combine base lst0)
  (local [(define (foldl/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl/acc (rest lst)
                                   (combine (first lst) acc))]))]
    (foldl/acc lst0 base)))

(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))
```

We noted earlier that intuitively, the effect of the application

```
(foldr f b (list x1 x2 . . . xn))
```

is to compute the value of the expression

```
(f x1 (f x2 (. . . (f xn b) . . .)))
```

What is the intuitive effect of the following application of foldl?

```
(foldl f b (list x1 . . . xn-1 xn))
```

The function foldl is provided in Intermediate Student.

What is the contract of foldl?

Higher-order functions

Functions that consume or produce functions like filter, map, and foldr are sometimes called **higher-order functions**.

Another example is the built-in build-list. This consumes a natural number n and a function f, and produces the list
(list (f 0) (f 1) . . . (f (sub1 n)))

(build-list 4 (lambda (x) x)) $\Rightarrow$ (list 0 1 2 3).

Clearly build-list abstracts the "count up" pattern, and it is easy to write our own version.


```
(define (my-build-list n f)
  (local
    [(define (list-from i)
       (cond [(>= i n) empty]
             [else (cons (f i) (list-from (add1 i)))]))]
    (list-from 0)))
```

Build-list examples

$\sum_{i=0}^{n-1} x_i$:

```
(define (sum n f)
  (foldr + 0 (build-list n f)))
(sum 4 sqr) ⇒ 14
```


We can now simplify mult-table even further.

```
(define (mult-table nr nc)
  (build-list nr
              (lambda (r)
                (build-list nc
                            (lambda (c)
                              (* r c))))))
```


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 using lambda, bound to identifiers, and placed in lists.

You should be familiar with the built-in abstract list functions provided by Racket, understand how they abstract common recursive patterns, and be able to use them to write code.

You should be able to write your own abstract list functions that implement other recursive patterns.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.