6: Functional Abstraction

What is abstraction?

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

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.

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 templates and design recipes.

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 templates and design recipes.

We can do more abstraction.

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (sqr (first lst))
                       (sqr-all (rest lst)))]))
```

(sqr-all (list 2 -4 3)) $\Rightarrow^*$ (list 4 16 9)

A familiar function and an unfamiliar one

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (sqr (first lst))
                       (sqr-all (rest lst)))]))
```

(sqr-all (list 2 -4 3)) $\Rightarrow^*$ (list 4 16 9)

```
(define (increment-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (add1 (first lst))
                       (increment-all (rest lst)))]))
```

(increment-all (list 2 -4 3)) $\Rightarrow^*$ (list 3 -3 4)

Abstracting from these examples

What `sqr-all` and `increment-all` have in common
is their general structure.

What `sqr-all` and `increment-all` have in common
is their general structure.

Where they differ is in the specific function applied to each element
of the argument list (`sqr` for the first and `add1` for the second).

What `sqr-all` and `increment-all` have in common
is their general structure.

Where they differ is in the specific function applied to each element of the argument list (`sqr` for the first and `add1` for the second).

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

What `sqr-all` and `increment-all` have in common
is their general structure.

Where they differ is in the specific function applied to each element of the argument list (`sqr` for the first and `add1` for the second).

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

This is not permitted in Beginning Student with List Abbreviations.

Abstracting from these examples

What `sqr-all` and `increment-all` have in common
is their general structure.

Where they differ is in the specific function applied to each element of the argument list (`sqr` for the first and `add1` for the second).

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

This is not permitted in Beginning Student with List Abbreviations.

However, it is permitted in the Intermediate Student Language (ISL).

```
(define (sqr-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (sqr (first lst))
                       (sqr-all (rest lst)))]))
(define (increment-all lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst) (cons (add1 (first lst))
                       (increment-all (rest lst)))]))

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

`map` is a built-in function in ISL.

Using the map function

We can use the `map` function to give a concise definition of `sqr-all`.

Using the map function

We can use the `map` function to give a concise definition of `sqr-all`.

```
(define (sqr-all lst) (map sqr lst))
```

Or we could simply replace all uses of `sqr-all`.

We can use the `map` function to give a concise definition of `sqr-all`.

```
(define (sqr-all lst) (map sqr lst))
```

Or we could simply replace all uses of `sqr-all`.

For example, `(sqr-all mylist)` becomes `(map sqr mylist)`.

We can use the `map` function to give a concise definition of `sqr-all`.

```
(define (sqr-all lst) (map sqr lst))
```

Or we could simply replace all uses of `sqr-all`.

For example, `(sqr-all mylist)` becomes `(map sqr mylist)`.

`map` is an example of an **abstract list function**. We will soon see others.

We can use the `map` function to give a concise definition of `sqr-all`.

```
(define (sqr-all lst) (map sqr lst))
```

Or we could simply replace all uses of `sqr-all`.

For example, `(sqr-all mylist)` becomes `(map sqr mylist)`.

`map` is an example of an **abstract list function**. We will soon see others.

More generally, `map` is an example of a **higher-order function**
(i.e., a function that consumes and/or produces functions).

What is the contract for `map`?

What is the contract for `map`?

First, we need to figure out how to write, in a contract, the type of an argument that is itself a function.

What is the contract for `map`?

First, we need to figure out how to write, in a contract, the type of an argument that is itself a function.

What we can do is use the contract of that function as its type.

A contract for map

What is the contract for `map`?

First, we need to figure out how to write, in a contract, the type of an argument that is itself a function.

What we can do is use the contract of that function as its type.

We might try this contract:

```
; map: (Any -> Any) (Listof Any) -> (Listof Any)
```

A contract for map

What is the contract for `map`?

First, we need to figure out how to write, in a contract, the type of an argument that is itself a function.

What we can do is use the contract of that function as its type.

We might try this contract:

```
; map: (Any -> Any) (Listof Any) -> (Listof Any)
```

But this does not accurately reflect the relationships among the various `Any` types.

What is the contract for `map`?

First, we need to figure out how to write, in a contract, the type of an argument that is itself a function.

What we can do is use the contract of that function as its type.

We might try this contract:

```
; map: (Any -> Any) (Listof Any) -> (Listof Any)
```

But this does not accurately reflect the relationships among the various `Any` types.

`(map sqr (list "bad" "data"))` is not a valid use of `map`, because `sqr` cannot be applied to a string. The contract should take this into account.

We introduce the idea of **type variables**, which can stand in for an unknown, arbitrary type the way an algebraic variable does for an unknown, arbitrary value.

We introduce the idea of **type variables**, which can stand in for an unknown, arbitrary type the way an algebraic variable does for an unknown, arbitrary value.

If we use the type variable `X` more than once in a contract, we mean that both those uses refer to the same type.

We introduce the idea of **type variables**, which can stand in for an unknown, arbitrary type the way an algebraic variable does for an unknown, arbitrary value.

If we use the type variable `X` more than once in a contract, we mean that both those uses refer to the same type.

Let's refine the contract of `map` using this idea.

```
; map: (Any -> Any) (Listof Any) -> (Listof Any)

; map: (X -> Any) (Listof X) -> (Listof Any)

; map: (X -> Y) (Listof X) -> (Listof Y)
```

We introduce the idea of **type variables**, which can stand in for an unknown, arbitrary type the way an algebraic variable does for an unknown, arbitrary value.

If we use the type variable `X` more than once in a contract, we mean that both those uses refer to the same type.

Let's refine the contract of `map` using this idea.

```
; map: (Any -> Any) (Listof Any) -> (Listof Any)

; map: (X -> Any) (Listof X) -> (Listof Any)

; map: (X -> Y) (Listof X) -> (Listof Y)
```

This is the most accurate contract for `map`, and it provides good guidance for the use of the function.

We saw this function in the previous lecture module.

```
(define (pos-elts lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
      (cond
        [(positive? (first lst))
          (cons (first lst) (pos-elts (rest lst)))]
        [else (pos-elts (rest lst))])]))
```

Here is a similar one.

```
(define (keep-odds lst)
  (cond
    [(empty? lst) empty]
    [(cons? lst)
      (cond
         [(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.

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.

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

Once again, ISL permits this.

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

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

`filter` is also a built-in function in ISL.

Simplifying filter

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

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

Exercise

What is the contract for `filter`?

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

Exercise

What is the contract for `filter`?

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

Exercise Solution

```
; filter: (X -> Boolean) (Listof X) -> (Listof X)
```

Producing functions

If functions are to be first-class values, we should be able to produce them while the program is being run.

If functions are to be first-class values, we should be able to produce them while the program is being run.

As an analogy, consider the expression `(* (+ 3 4) 5)`.

Evaluating this expression produces the intermediate value `7`, and the final value `35`, neither of which appear in the expression, and neither of which have a name or identifier bound to them.

If functions are to be first-class values, we should be able to produce them while the program is being run.

As an analogy, consider the expression `(* (+ 3 4) 5)`.

Evaluating this expression produces the intermediate value `7`, and the final value `35`, neither of which appear in the expression, and neither of which have a name or identifier bound to them.

We need a way of creating function values in a similar fashion.

The way to create function values is to use `lambda`.

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

A `lambda` expression resembles a function definition, but the keyword `define` is replaced with `lambda`, and there is no function name (the function is *anonymous*).

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

A `lambda` expression resembles a function definition, but the keyword `define` is replaced with `lambda`, and there is no function name (the function is *anonymous*).

A definition is not an expression, and it can only appear at the top level of a program.

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

A `lambda` expression resembles a function definition, but the keyword `define` is replaced with `lambda`, and there is no function name (the function is *anonymous*).

A definition is not an expression, and it can only appear at the top level of a program. A `lambda` expression can appear anywhere that an expression is expected.

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

A `lambda` expression resembles a function definition, but the keyword `define` is replaced with `lambda`, and there is no function name (the function is *anonymous*).

A definition is not an expression, and it can only appear at the top level of a program. A `lambda` expression can appear anywhere that an expression is expected.

A `lambda` expression is a value.

The way to create function values is to use `lambda`.

`(lambda (x) (* x x))` is the function which consumes a single argument and produces its square. It is behaviourally equivalent to the built-in function `sqr`.

A `lambda` expression resembles a function definition, but the keyword `define` is replaced with `lambda`, and there is no function name (the function is *anonymous*).

A definition is not an expression, and it can only appear at the top level of a program. A `lambda` expression can appear anywhere that an expression is expected.

A `lambda` expression is a value.

`lambda` is not available in ISL. But it is available in the next language level, Intermediate Student with Lambda (ISL+).

Why call it lambda?

It seems strange to use a Greek letter for this language feature.

Why call it lambda?

It seems strange to use a Greek letter for this language feature.

The name comes from the **lambda calculus**, which was the first general model of computation.

Why call it lambda?

It seems strange to use a Greek letter for this language feature.

The name comes from the **lambda calculus**, which was the first general model of computation.

The lambda calculus was defined by the logician Alonzo Church in the 1930's, and used by him to demonstrate the first uncomputable problem.

It seems strange to use a Greek letter for this language feature.

The name comes from the **lambda calculus**, which was the first general model of computation.

The lambda calculus was defined by the logician Alonzo Church in the 1930's, and used by him to demonstrate the first uncomputable problem.

The lambda calculus has only lambda (function creation) and function application. It has nothing else — no numbers, Booleans, strings, conditionals, or structures.

It seems strange to use a Greek letter for this language feature.

The name comes from the **lambda calculus**, which was the first general model of computation.

The lambda calculus was defined by the logician Alonzo Church in the 1930's, and used by him to demonstrate the first uncomputable problem.

The lambda calculus has only lambda (function creation) and function application. It has nothing else — no numbers, Booleans, strings, conditionals, or structures.

Yet it can express any computation that Racket can.

It seems strange to use a Greek letter for this language feature.

The name comes from the **lambda calculus**, which was the first general model of computation.

The lambda calculus was defined by the logician Alonzo Church in the 1930's, and used by him to demonstrate the first uncomputable problem.

The lambda calculus has only lambda (function creation) and function application. It has nothing else — no numbers, Booleans, strings, conditionals, or structures.

Yet it can express any computation that Racket can.

Many functional programming languages (including Racket) can be viewed as the lambda calculus with features added to make it easier to express computation (without adding any theoretical power).

An immediate use of `lambda` is in creating function arguments for applications of `map` or `filter`.

Using lambda

An immediate use of `lambda` is in creating function arguments for applications of `map` or `filter`.

Example

`(map (lambda (n) (* n n n)) mylist)` produces a list of the cube of every number in `mylist`.

An immediate use of `lambda` is in creating function arguments for applications of `map` or `filter`.

Example

`(map (lambda (n) (* n n n)) mylist)` produces a list of the cube of every number in `mylist`.

Example

`(lambda (x) (not (equal? x 'apple)))` is the function that produces `false` if it is applied to the symbol `'apple`, and produces `true` otherwise.

An immediate use of `lambda` is in creating function arguments for applications of `map` or `filter`.

Example

`(map (lambda (n) (* n n n)) mylist)` produces a list of the cube of every number in `mylist`.

Example

`(lambda (x) (not (equal? x 'apple)))` is the function that produces `false` if it is applied to the symbol `'apple`, and produces `true` otherwise.

Example

`(filter (lambda (x) (not (equal? x 'apple))) mylist)` is an expression that "eats apples" from `mylist`; it produces a list that has all values in `mylist` that are not `'apple`.

We don’t have to make many changes to our earlier syntax and semantics.

We don't have to make many changes to our earlier syntax and semantics.

First, we introduce a grammar rule for `lambda` expressions.

```
expr = (lambda (id id ...) expr)
```

We don't have to make many changes to our earlier syntax and semantics.

First, we introduce a grammar rule for `lambda` expressions.

```
expr = (lambda (id id ...) expr)
```

Before, the first position in a function application had to be the name of a built-in or user-defined function.

```
expr = (id expr expr ...)
```

Syntax and semantics of ISL+

We don't have to make many changes to our earlier syntax and semantics.

First, we introduce a grammar rule for `lambda` expressions.

```
expr = (lambda (id id ...) expr)
```

Before, the first position in a function application had to be the name of a built-in or user-defined function.

```
expr = (id expr expr ...)
```

The first position in an application is now an expression (computing the function to be applied).

```
expr = (expr expr expr ...)
```

We don't have to make many changes to our earlier syntax and semantics.

First, we introduce a grammar rule for `lambda` expressions.

```
expr = (lambda (id id ...) expr)
```

Before, the first position in a function application had to be the name of a built-in or user-defined function.

```
expr = (id expr expr ...)
```

The first position in an application is now an expression (computing the function to be applied).

```
expr = (expr expr expr ...)
```

It must be evaluated, just like the other arguments.

The rule for rewriting the application of a `lambda` value to arguments resembles the rule for application of a user-defined function.

The rule for rewriting the application of a `lambda` value to arguments resembles the rule for application of a user-defined function.

```
((lambda (x1 ... xn) exp) v1 ... vn) => modexp
```

Rewriting applications of lambda

The rule for rewriting the application of a `lambda` value to arguments resembles the rule for application of a user-defined function.

```
((lambda (x1 ... xn) exp) v1 ... vn) => modexp
```

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

The rule for rewriting the application of a `lambda` value to arguments resembles the rule for application of a user-defined function.

```
((lambda (x1 ... xn) exp) v1 ... vn) => modexp
```

where `modexp` 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) => (* (+ 6 4) 5)
```

The rule for rewriting the application of a `lambda` value to arguments resembles the rule for application of a user-defined function.

```
((lambda (x1 ... xn) exp) v1 ... vn) => modexp
```

where `modexp` 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) => (* (+ 6 4) 5)
```

We do not rewrite expressions in the body of a `lambda`, just as we previously did not rewrite expressions in the body of function definitions.

Before, there were two kinds of definitions:

```
(define interest-rate 3/100)
(define (interest-earned amount)
  (* interest-rate amount))
```

Now, there is only one kind of definition, the first kind, which binds a name to a value.

The second definition is rewritten to be like the first kind.

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

We can now remove the rule for rewriting the application of a user-defined function. The rule we just added for application of a `lambda` expression suffices.

Previously:

```
(interest-earned 200)
⇒  (* interest-rate 200)
⇒  (* 3/100 200)
⇒  6
```

Now:

```
(interest-earned 200)
⇒  ((lambda (amount) (* interest-rate amount)) 200)
⇒  (* interest-rate 200)
⇒  (* 3/100 200)
⇒  6
```

The Stepper in ISL+ shows this. But in our condensed traces, sometimes we will use the old style of tracing, because it is a little clearer.

Exercise

Which of the following defines a function `recip` which takes one parameter and returns its reciprocal?

1. `(define (recip x) (lambda (x) (/ 1 x)))`
2. `(define (recip) (lambda (x) (/ 1 x)))`
3. `(define recip (lambda (x) (/ 1 x)))`
4. `(lambda (recip x) (/ 1 x))`
5. None of the above

Exercise

Which of the following defines a function `recip` which takes one parameter and returns its reciprocal?

1. `(define (recip x) (lambda (x) (/ 1 x)))`
2. `(define (recip) (lambda (x) (/ 1 x)))`
3. `(define recip (lambda (x) (/ 1 x)))`
4. `(lambda (recip x) (/ 1 x))`
5. None of the above

Exercise Solution

The correct answer is (3). `(define recip (lambda (x) (/ 1 x)))`

`lambda` has uses far beyond what we have seen so far.

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

`lambda` has uses far beyond what we have seen so far.

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. To actually perform the action later on, we would need a helper function to consume that data structure and perform the action it described.

`lambda` has uses far beyond what we have seen so far.

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. To actually perform the action later on, we would need a helper function to consume that data structure and perform the action it described.

Now, we can just describe the computation clearly using `lambda`, and simply apply the resulting function when needed in future.

`lambda` has uses far beyond what we have seen so far.

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. To actually perform the action later on, we would need a helper function to consume that data structure and perform the action it described.

Now, we can just describe the computation clearly using `lambda`, and simply apply the resulting function when needed in future.

Example: the make-adder function

The make-adder function consumes a number and produces a function that adds that number to its argument.

```
(define (make-adder n)(lambda (x) (+ n x)))
⇒ (define make-adder (lambda (n) (lambda (x) (+ n x))))
```

The make-adder function consumes a number and produces a function that adds that number to its argument.

```
(define (make-adder n)(lambda (x) (+ n x)))
⇒ (define make-adder (lambda (n) (lambda (x) (+ n x))))
```

```
(define p3 (make-adder 3))
⇒ (define p3 ((lambda (n) (lambda (x) (+ n x))) 3))
⇒ (define p3 (lambda (x) (+ 3 x)))
```

The make-adder function consumes a number and produces a function that adds that number to its argument.

```
(define (make-adder n)(lambda (x) (+ n x)))
⇒  (define make-adder (lambda (n) (lambda (x) (+ n x))))
```

```
(define p3 (make-adder 3))
⇒  (define p3 ((lambda (n) (lambda (x) (+ n x))) 3))
⇒  (define p3 (lambda (x) (+ 3 x)))
```

```
(p3 4) ⇒  ((lambda (x) (+ 3 x)) 4) ⇒  (+ 3 4) ⇒  7
```

The make-adder function consumes a number and produces a function that adds that number to its argument.

```
(define (make-adder n)(lambda (x) (+ n x)))
⇒ (define make-adder (lambda (n) (lambda (x) (+ n x))))
```

```
(define p3 (make-adder 3))
⇒ (define p3 ((lambda (n) (lambda (x) (+ n x))) 3))
⇒ (define p3 (lambda (x) (+ 3 x)))
```

```
(p3 4) ⇒ ((lambda (x) (+ 3 x)) 4) ⇒ (+ 3 4) ⇒ 7
```

What is the contract of `make-adder`?

The make-adder function consumes a number and produces a function that adds that number to its argument.

```
(define (make-adder n)(lambda (x) (+ n x)))
⇒ (define make-adder (lambda (n) (lambda (x) (+ n x))))
```

```
(define p3 (make-adder 3))
⇒ (define p3 ((lambda (n) (lambda (x) (+ n x))) 3))
⇒ (define p3 (lambda (x) (+ 3 x)))
```

```
(p3 4) ⇒ ((lambda (x) (+ 3 x)) 4) ⇒ (+ 3 4) ⇒ 7
```

What is the contract of `make-adder`?

```
; make-adder: number -> (number -> number)
```

Racket provides the function `string->list` to convert a string to a list of characters.

Racket provides the function `string->list` to convert a string to a list of characters.

This is the most effective way to work with strings, though typically structural recursion on these lists is not effective, and generative recursion needs to be used.

Racket provides the function `string->list` to convert a string to a list of characters.

This is the most effective way to work with strings, though typically structural recursion on these lists is not effective, and generative recursion needs to be used.

In the example we are about to discuss, structural recursion works.

Racket provides the function `string->list` to convert a string to a list of characters.

This is the most effective way to work with strings, though typically structural recursion on these lists is not effective, and generative recursion needs to be used.

In the example we are about to discuss, structural recursion works.

The function `list->string` converts a list of characters to a string.

Racket's notation for the character 'a' is `#\a`.

The result of evaluating `(string->list "test")` is the list `'(#\t #\e #\s #\t)`.

Racket provides the function `string->list` to convert a string to a list of characters.

This is the most effective way to work with strings, though typically structural recursion on these lists is not effective, and generative recursion needs to be used.

In the example we are about to discuss, structural recursion works.

The function `list->string` converts a list of characters to a string.

Racket's notation for the character 'a' is `#\a`.

The result of evaluating `(string->list "test")` is the list `'(#\t #\e #\s #\t)`.

This is unfortunately ugly, but the `#` notation is part of a more general way of specifying values in Racket. We have already seen `#true` and `#false`.

For example, we might want to convert every ‘a’ in a string to a ‘b’. The string `"abracadabra"` becomes `"bbrbcbdbbrb"`.

For example, we might want to convert every ‘a’ in a string to a ‘b’. The string `"abracadabra"` becomes `"bbrbcbdbbrb"`.

This doesn’t require functional abstraction. If you’d known about characters in the previous lecture module, you could have written a function that does this.

For example, we might want to convert every ‘a’ in a string to a ‘b’. The string "abracadabra" becomes "bbrbcbdbbrb".

This doesn’t require functional abstraction. If you’d known about characters in the previous lecture module, you could have written a function that does this.

```
; a->b: String -> String
(define (a->b str)
  (list->string (ab-helper (string->list str))))

; ab-helper: (Listof Char) -> (Listof Char)
(define (ab-helper loc)
  (cond
    [(empty? lst) empty]
    [(char=? (first loc) #\a) (cons #\b (ab-helper (rest loc)))]
    [else (cons (first loc) (ab-helper (rest loc)))]))
```

The function `ab-helper` works through a list of characters, applying a predicate (“equals a?”) to each character, and applying an action (“make it b”) to characters that satisfied the predicate.

The function `ab-helper` works through a list of characters, applying a predicate (“equals a?”) to each character, and applying an action (“make it b”) to characters that satisfied the predicate.

We define a **translation** to be a pair (a list of length two) consisting of a predicate and an action.

The function `ab-helper` works through a list of characters, applying a predicate (“equals a?”) to each character, and applying an action (“make it b”) to characters that satisfied the predicate.

We define a **translation** to be a pair (a list of length two) consisting of a predicate and an action.

We might want to apply several translations to a string. We can describe the translation in our example like this:

```
(list (lambda (c) (char=? #\a c))
      (lambda (c) #\b))
```

The function `ab-helper` works through a list of characters, applying a predicate (“equals a?”) to each character, and applying an action (“make it b”) to characters that satisfied the predicate.

We define a **translation** to be a pair (a list of length two) consisting of a predicate and an action.

We might want to apply several translations to a string. We can describe the translation in our example like this:

```
(list (lambda (c) (char=? #\a c))
      (lambda (c) #\b))
```

Since these are likely to be common sorts of functions, we can write helper functions to create them.

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

(list (is-char? #\a) (always #\b))
```

A general character translation function

Our `translate` function will consume a list of translations and a string to be translated.

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create an result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

A general character translation function

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create an result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

If no predicate is satisfied by `c`, the result character is `c`.

A general character translation function

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create an result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

If no predicate is satisfied by `c`, the result character is `c`.

An example of its use: suppose we have a string `s`, and we want a version of it where all letters are capitalized, and all numbers are "censored" by replacing them with asterisks.

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create an result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

If no predicate is satisfied by `c`, the result character is `c`.

An example of its use: suppose we have a string `s`, and we want a version of it where all letters are capitalized, and all numbers are "censored" by replacing them with asterisks.

`char-alphabetic?`, `char-upcase`, and `char-numeric?` are built-in functions we can make use of.

Our `translate` function will consume a list of translations and a string to be translated.

For each character `c` in the string, it will create an result character by applying the action of the first translation on the list whose predicate is satisfied by `c`.

If no predicate is satisfied by `c`, the result character is `c`.

An example of its use: suppose we have a string `s`, and we want a version of it where all letters are capitalized, and all numbers are "censored" by replacing them with asterisks.

`char-alphabetic?`, `char-upcase`, and `char-numeric?` are built-in functions we can make use of.

```
(define s "Testing 1-2-3.")
(translate (list (list char-alphabetic? char-upcase)
                 (list char-numeric? (always #\*)))
           s)
⇒* "TESTING *-*-*."
```

```
(define (translate lot str)
  (list->string (trans-loc lot (string->list str))))

(define (trans-loc lot loc)
  (cond
    [(empty? loc) empty]
    [else (cons (trans-char lot (first loc))
                (trans-loc lot (rest loc)))]))

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

```
(define (trans-loc lot loc)
  (cond
    [(empty? loc) empty]
    [else (cons (trans-char lot (first loc))
                (trans-loc lot (rest loc)))]))
```

Exercise

What is the contract of `trans-loc`?

```
(define (trans-loc lot loc)
  (cond
    [(empty? loc) empty]
    [else (cons (trans-char lot (first loc))
                (trans-loc lot (rest loc)))]))
```

Exercise

What is the contract of `trans-loc`?

Exercise Solution

```
; (Listof (list (Char -> Boolean) (Char -> Char)))
; (Listof Char)
;    -> (Listof Char).
```

Lambda complicates scope

Previously, we had two notions of scope: global and local.

```
(define x 7)
(define (f x) (* x x))
(f 4) ⇒* 16
```

Lambda complicates scope

Previously, we had two notions of scope: global and local.

```
(define x 7)
(define (f x) (* x x))
(f 4) ⇒* 16
```

A name bound by a top-level definition (as in the first line above) is in global scope, visible to code below.

It can be shadowed by a use of the same name as a parameter, as in the second line. This introduces a new local scope for the the name, that is, the body of the function.

Previously, we had two notions of scope: global and local.

```
(define x 7)
(define (f x) (* x x))
(f 4) ⇒* 16
```

A name bound by a top-level definition (as in the first line above) is in global scope, visible to code below.

It can be shadowed by a use of the same name as a parameter, as in the second line. This introduces a new local scope for the the name, that is, the body of the function.

A use of `lambda` does something similar. But because `lambda` can occur anywhere an expression is expected, the situation is more complicated.

Lambda complicates scope

Each use of `lambda` introduces a new local scope.

Each use of `lambda` introduces a new local scope.

Lambdas may be nested, and an inner `lambda` may reuse a parameter name that is used by an outer `lambda`.

```
(lambda (x) (lambda (x) (* x x)))
```

Each use of `lambda` introduces a new local scope.

Lambdas may be nested, and an inner `lambda` may reuse a parameter name that is used by an outer `lambda`.

```
(lambda (x) (lambda (x) (* x x)))
```

In this expression, the `x` in `(* x x)` refers to the parameter of the inner `lambda`.

This expression creates a function that, when applied to an argument, ignores that argument and produces a function that squares its argument.

Terminology for use with scope

A use of `lambda` introduces new **binding occurrences** of the names of the parameters.

A use of `lambda` introduces new **binding occurrences** of the names of the parameters.

The body of the `lambda` may contain **bound occurrences** of those names, referring to the **binding occurrences** of the parameters.

Terminology for use with scope

A use of `lambda` introduces new **binding occurrences** of the names of the parameters.

The body of the `lambda` may contain **bound occurrences** of those names, referring to the **binding occurrences** of the parameters.

Each binding occurrence may have several bound occurrences, but each bound occurrence corresponds to exactly one binding occurrence.

A use of `lambda` introduces new **binding occurrences** of the names of the parameters.

The body of the `lambda` may contain **bound occurrences** of those names, referring to the **binding occurrences** of the parameters.

Each binding occurrence may have several bound occurrences, but each bound occurrence corresponds to exactly one binding occurrence.

The **scope** of a binding occurrence is the body of the `lambda`, except for places where the name is **shadowed** by a reuse.

A use of `lambda` introduces new **binding occurrences** of the names of the parameters.

The body of the `lambda` may contain **bound occurrences** of those names, referring to the **binding occurrences** of the parameters.

Each binding occurrence may have several bound occurrences, but each bound occurrence corresponds to exactly one binding occurrence.

The **scope** of a binding occurrence is the body of the `lambda`, except for places where the name is **shadowed** by a reuse.

We extend these ideas to top-level definitions
(including old-style function definitions).

Racket also provides other binding constructs.

To illustrate the usefulness of local scope, and to introduce Racket's constructs for local binding, we will discuss several implementations of a simple mathematical formula.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

To illustrate the usefulness of local scope, and to introduce Racket's constructs for local binding, we will discuss several implementations of a simple mathematical formula.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

```
(define (t-area a b c)
  (sqrt
    (* (s a b c)
       (- (s a b c) a)
       (- (s a b c) b)
       (- (s a b c) c))))

(define (s a b c)
  (/ (+ a b c) 2))
```

This is awkward.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Better: Compute `s` only once.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Better: Compute `s` only once.

```
(define (t-helper a b c s)
  (sqrt (* s
           (- s a)
           (- s b)
           (- s c))))

(define (t-area2 a b c)
  (t-helper a b c (/ (+ a b c) 2)))
```

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Better: But there is no need for a named helper function.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Better: But there is no need for a named helper function.

```
(define (t-area3 a b c)
  ((lambda (s)
     (sqrt (* s
              (- s a)
              (- s b)
              (- s c))))
   (/ (+ a b c) 2)))
```

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Best: Racket provides the `let` construct to make this use of `lambda` more readable.

Heron's formula says that the area of a triangle with sides a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$, where $s = (a+b+c)/2$.

Best: Racket provides the `let` construct to make this use of `lambda` more readable.

```
(define (t-area4 a b c)
  (let
    ([s (/ (+ a b c) 2)])
    (sqrt (* s
             (- s a)
             (- s b)
             (- s c)))))
```

Grammar rule:

```
expr = (let ([id expr] ...) expr)
```

Reduction rule:

```
(let ([x1 e1] ... [xn en]) exp)
⇒  ((lambda (x1 ... xn) exp) e1 ... en)
```

Grammar rule:

```
expr = (let ([id expr] ...) expr)
```

Reduction rule:

```
(let ([x1 e1] ... [xn en]) exp)
⇒  ((lambda (x1 ... xn) exp) e1 ... en)
```

In full Racket, `let` is implemented by a **macro** that specifies this rewriting rule.

```
(define-syntax-rule
  (let ([x e] ...) body)
     ((lambda (x ...) body) e ...))
```

Macros allow the programmer to create new syntax in a program. This makes Racket a laboratory for language design and implementation.

The `let` construct allows one to define several bindings, but none of the names bound can be used in any of the right-hand-side expressions.

This makes defining local recursive functions difficult.

The `let` construct allows one to define several bindings, but none of the names bound can be used in any of the right-hand-side expressions.

This makes defining local recursive functions difficult.

The more general `local` construct allows an arbitrary number of local definitions whose scope is the body expression.

Example

```
(local [
 (define (fact n)
   (cond
     [(zero? n) 1]
     [else (* n (fact (sub1 n)))]))]
 (fact 5))
```

Grammar rule:

```
expr = (local [defn ...] expr)
```

Reduction rule:

```
(local [(define x1 e1) ...] body) ⇒ ???
```

Grammar rule:

```
expr = (local [defn ...] expr)
```

Reduction rule:

```
(local [(define x1 e1) ...] body) ⇒ ???
```

The identifier `x1` is replaced by a *fresh* identifier `x1new` everywhere in the `local` expression.

This is repeated with the rest of the definitions.

The rewritten definitions are lifted out to the top level.

Grammar rule:

```
expr = (local [defn ...] expr)
```

Reduction rule:

```
(local [(define x1 e1) ...] body) ⇒ ???
```

The identifier `x1` is replaced by a *fresh* identifier `x1new` everywhere in the `local` expression.

This is repeated with the rest of the definitions.

The rewritten definitions are lifted out to the top level.

```
(local [] body) ⇒ body
```

The complete description of the reduction rule for `local` is the most complicated semantic rule we will see.

An example of rewriting a local expression

```
(local [
 (define (fact n)
   (cond
     [(zero? n) 1]
     [else (* n (fact (sub1 n)))]))]
 (fact 5))
⇒  (define (factnew n)
          (cond
            [(zero? n) 1]
            [else (* n (factnew (sub1 n)))]))
(local [] (factnew 5))
⇒  (define (factnew n)
          (cond
            [(zero? n) 1]
            [else (* n (factnew (sub1 n)))]))
(factnew 5)
⇒  ...
```

Let's try to generalize from the template for structural recursion on a list.

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

Let's try to generalize from the template for structural recursion on a list.

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

We replace the first ellipsis by a base value.

Let's try to generalize from the template for structural recursion on a list.

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

We replace the first ellipsis by a base value.

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

Let's try to generalize from the template for structural recursion on a list.

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

We replace the first ellipsis by a base value.

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

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

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

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

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

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

`foldr` is a built-in function in ISL+.

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

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

`foldr` is a built-in function in ISL+.

`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.

Tracing foldr

A generic trace of `foldr` might look something like this:

```
(foldr f 0 (list 3 6 5)) ⇒*
(f 3 (foldr f 0 (list 6 5))) ⇒*
(f 3 (f 6 (foldr f 0 (list 5)))) ⇒*
(f 3 (f 6 (f 5 (foldr f 0 empty)))) ⇒*
(f 3 (f 6 (f 5 0))) ⇒* ...
```

A real trace would substitute for `f` first.

A generic trace of `foldr` might look something like this:

```
(foldr f 0 (list 3 6 5)) ⇒*
(f 3 (foldr f 0 (list 6 5))) ⇒*
(f 3 (f 6 (foldr f 0 (list 5)))) ⇒*
(f 3 (f 6 (f 5 (foldr f 0 empty)))) ⇒*
(f 3 (f 6 (f 5 0))) ⇒* ...
```

A real trace would substitute for `f` first.

Intuitively, `(foldr f b (list x1 x2 ... xn))` computes `(f x1 (f x2 (... (f xn b) ...)))`.

Tracing foldr

A generic trace of `foldr` might look something like this:

```
(foldr f 0 (list 3 6 5)) ⇒*
(f 3 (foldr f 0 (list 6 5))) ⇒*
(f 3 (f 6 (foldr f 0 (list 5)))) ⇒*
(f 3 (f 6 (f 5 (foldr f 0 empty)))) ⇒*
(f 3 (f 6 (f 5 0))) ⇒* ...
```

A real trace would substitute for `f` first.

Intuitively, `(foldr f b (list x1 x2 ... xn))` computes `(f x1 (f x2 (... (f xn b) ...)))`.

`(foldr + 0 (list x1 x2 ... xn))` computes `(+ x1 (+ x2 (... (+ xn 0) ...)))`.

A generic trace of `foldr` might look something like this:

```
(foldr f 0 (list 3 6 5)) ⇒*
(f 3 (foldr f 0 (list 6 5))) ⇒*
(f 3 (f 6 (foldr f 0 (list 5)))) ⇒*
(f 3 (f 6 (f 5 (foldr f 0 empty)))) ⇒*
(f 3 (f 6 (f 5 0))) ⇒* ...
```

A real trace would substitute for `f` first.

Intuitively, `(foldr f b (list x1 x2 ... xn))` computes `(f x1 (f x2 (... (f xn b) ...)))`.

`(foldr + 0 (list x1 x2 ... xn))` computes `(+ x1 (+ x2 (... (+ xn 0) ...)))`.

```
(define (sum-list lst) (foldr + 0 lst))
```

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

Exercise

What is the contract for `foldr`?

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

Exercise

What is the contract for `foldr`?

Exercise Solution

```
; foldr: (??? -> ?) ?    ?        -> ?
; foldr: (??? -> ?) ? (Listof X) -> ?
; foldr: (X ? -> ?) ? (Listof X) -> ?
; foldr: (X ? -> ?) Y (Listof X) -> Y
; foldr: (X Y -> Y) Y (Listof X) -> Y
```

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.

Using foldr

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 the length of a list.

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 the length of a list.

```
(define (len lst) (foldr (lambda (x y) (add1 y)) 0 lst))
```

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 the length of a list.

```
(define (len lst) (foldr (lambda (x y) (add1 y)) 0 lst))
```

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

Its second argument represents the reduction of the rest of the list (in this case the length of the rest of the list, to which 1 must be added).

Exercise

What is the value of the following expression?

```
(foldr (lambda (x y) (+ x y y)) 1 '(3 4 5))
```

1. 24
2. 25
3. 31
4. 38
5. None of the above

Exercise

What is the value of the following expression?

```
(foldr (lambda (x y) (+ x y y)) 1 '(3 4 5))
```

1. 24
2. 25
3. 31
4. 38
5. None of the above

Exercise Solution

5. None of the above. The value is 39.

Using foldr to produce lists

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

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

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

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

`sqr-all` takes this element, squares it, and `cons`es it onto the result of the recursive application.

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

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

`sqr-all` takes this element, squares it, and `cons`es it onto the result of the recursive application.

The function we need is

```
(lambda (x y) (cons (sqr x) y))
```

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

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

`sqr-all` takes this element, squares it, and `cons`es it onto the result of the recursive application.

The function we need is

```
(lambda (x y) (cons (sqr x) y))
```

```
(define (sqr-all lst)
  (foldr (lambda (x y) (cons (sqr x) y)) empty lst))
```

Since `foldr` is an abstraction of structural recursion on lists, we should be able to use it to carry out the same computation as `sqr-all`.

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

`sqr-all` takes this element, squares it, and `cons`es it onto the result of the recursive application.

The function we need is

```
(lambda (x y) (cons (sqr x) y))
```

```
(define (sqr-all lst)
  (foldr (lambda (x y) (cons (sqr x) y)) empty lst))
```

Since we generalized `sqr-all` to `map`, we should be able to use `foldr` to define `map`.

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

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

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

```
(define (map f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (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 (map f lst)
  (foldr (lambda (x y) (cons (f x) y)) empty lst))
```

```
(define (map f lst)
  (cond
    [(empty? lst) empty]
    [else (cons (f (first lst))
                (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 (map f lst)
  (foldr (lambda (x y) (cons (f x) y)) empty lst))
```

Exercise: Implement `filter` using `foldr`.

Exercise

What is `(foldr cons empty mylist)`?

Exercise

What is `(foldr cons empty mylist)`?

Exercise Solution

It is just `mylist`.

Exercise

What is `(foldr cons empty mylist)`?

Exercise Solution

It is just `mylist`.

Exercise

What is `(foldr cons mylist1 mylist2)`?

Exercise

What is `(foldr cons empty mylist)`?

Exercise Solution

It is just `mylist`.

Exercise

What is `(foldr cons mylist1 mylist2)`?

Exercise Solution

It is `(append mylist1 mylist2)`.

`foldr` is universal for structural recursion on a single list parameter.

Anything that can be done with the list template can be done using `foldr`, without explicit recursion.

`foldr` is universal for structural recursion on a single list parameter.

Anything that can be done with the list template can be done using `foldr`, without explicit recursion.

Does that mean that the list template is obsolete?

`foldr` is universal for structural recursion on a single list parameter.

Anything that can be done with the list template can be done using `foldr`, without explicit recursion.

Does that mean that the list template is obsolete?

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

`foldr` is universal for structural recursion on a single list parameter.

Anything that can be done with the list template can be done using `foldr`, without explicit recursion.

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.

In practice, `map` and `filter` are used much more often.
`foldr` is used mostly in relatively short, simple expressions.

Exercise

The following partially completed Racket function should convert a string to uppercase:

```
(define (string-upcase s)
  (local [(define old-list (string->list s))
          (define new-list ( ... HERE ...))
          (define new-string (list->string new-list))]
    new-string))
```

Which would be the most appropriate abstract list function to combine with `char-upcase` to fill in the code at `HERE`?

Choices: `filter`, `map`, `foldr`, `lambda`, `append`

Exercise

The following partially completed Racket function should convert a string to uppercase:

```
(define (string-upcase s)
  (local [(define old-list (string->list s))
          (define new-list ( ... HERE ...))
          (define new-string (list->string new-list))]
    new-string))
```

Which would be the most appropriate abstract list function to combine with `char-upcase` to fill in the code at `HERE`?

Choices: `filter`, `map`, `foldr`, `lambda`, `append`

Exercise Solution

`map`. The full expression is `(map char-upcase old-list)`.

You should understand the idea of functions as first-class values: how they can be supplied as arguments, and how they can be produced as values using `lambda`.

You should understand the idea of functions as first-class values: how they can be supplied as arguments, and how they can be produced as values using `lambda`.

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

You should understand the idea of functions as first-class values: how they can be supplied as arguments, and how they can be produced as values using `lambda`.

You should be familiar with the built-in 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 the idea of functions as first-class values: how they can be supplied as arguments, and how they can be produced as values using `lambda`.

You should be familiar with the built-in 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 ISL+ that make use of functions as values.

You should understand the idea of functions as first-class values: how they can be supplied as arguments, and how they can be produced as values using `lambda`.

You should be familiar with the built-in 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 ISL+ that make use of functions as values.