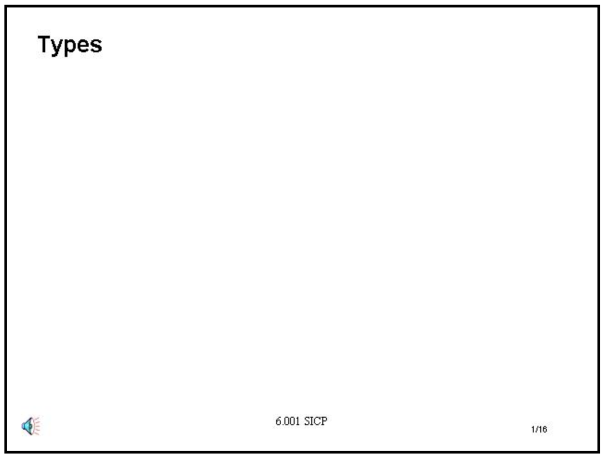
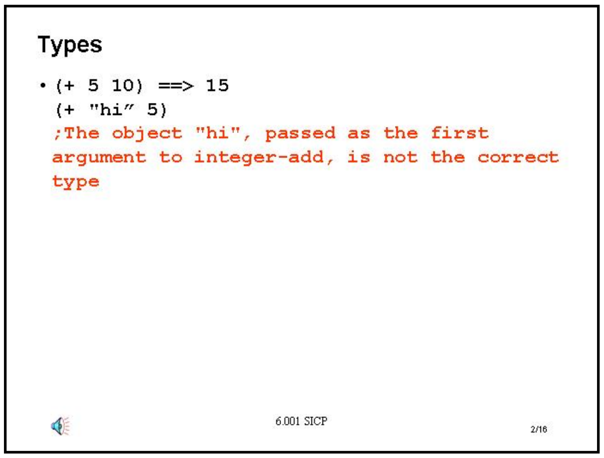
6.001 Notes: Section 6.1

Slide 6.1.1
When we first starting talking about Scheme expressions, you may recall we said that (almost) every Scheme expression had three components, a syntax (legal ways of gluing things together), a semantics (a way of deciding the meaning associated with an expression), and a type. We haven't said much about types yet. Today we are going to look in more detail at types, and especially how we can use types to design procedures with different kinds of behaviors.



Types

6.001 SICP 1/16



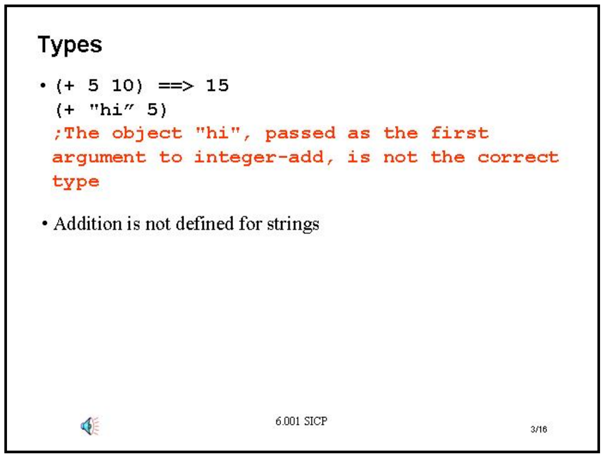
Types

```
• (+ 5 10) ==> 15
  (+ "hi" 5)
 ;The object "hi", passed as the first
 argument to integer-add, is not the correct
 type
```

6.001 SICP 2/16

Slide 6.1.2
So let's motivate the idea of a type. Consider the following two examples. The first expression we know will correctly add the two numbers to get 15. The second expression we expect to give an error, since while we may like to get "high fives", we know that we can't add a string to a number, as addition is only used on numbers.
Note what this implies. It suggests that the addition procedure has associated with it an expectation of what kinds of arguments it will get, in particular an expectation about the type of each argument, in this case, numbers.

Slide 6.1.3
And here, since we have given a string as an input, instead of an integer, the procedure should complain, stating that it can't perform its job since it was not given what it expected.

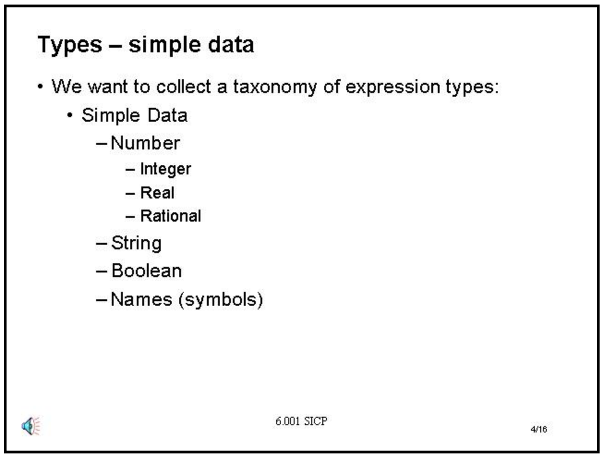


Types

```
• (+ 5 10) ==> 15
  (+ "hi" 5)
 ;The object "hi", passed as the first
 argument to integer-add, is not the correct
 type
```

• Addition is not defined for strings

6.001 SICP 3/16



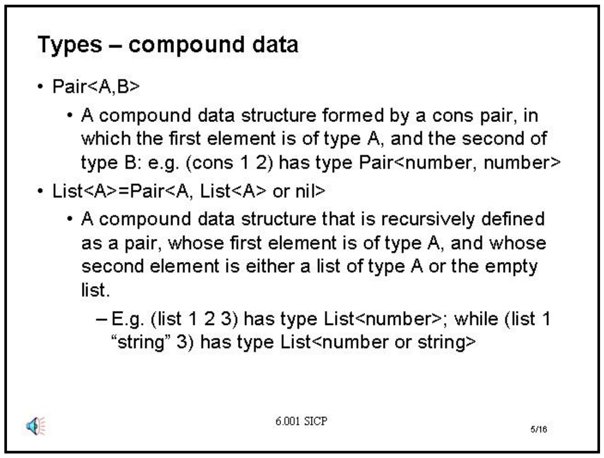
Types – simple data

- We want to collect a taxonomy of expression types:
  - Simple Data
    - Number
      - Integer
      - Real
      - Rational
    - String
    - Boolean
    - Names (symbols)

6.001 SICP 4/16

Slide 6.1.4
So now let's formalize this idea. We want to assign a type to virtually every kind of expression in our language.
We have already seen some basic primitives: numbers, which are a type; strings, another type; booleans, a third type; and symbols or names for things, to which we will be returning in a few lectures. For some things, like numbers, we could actually be more specific, for example distinguishing integers from real numbers.
As we saw with our motivating example, we typically want to know the type of a data structure in order to decide if some procedure is appropriate to apply to that type. These basic primitive data types will usually be sufficient to handle the cases we will see.

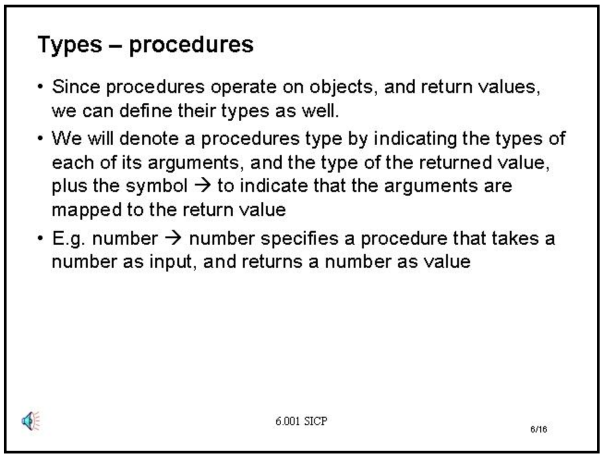
Slide 6.1.5
For compound data, we saw that the basic element was a pair, and we have a type notation to denote one. Note that we include in this notation a specification of the types of the elements within the pair.
We also saw that we could construct a list out of pairs. So we can create a type notation for that as well. In essence we are creating a formal definition for a list: we specify that this compound data structure is recursively defined as a pair, whose first element is of some type A, and whose second element is either another list or is the empty list. This exactly reflects the description we gave when we introduced lists.
Note that in our type specifications, we can include the possibility of alternative types of elements, as shown in the example.



Types – compound data

- Pair<A,B>
  - A compound data structure formed by a cons pair, in which the first element is of type A, and the second of type B: e.g. (cons 1 2) has type Pair<number, number>
- List<A>=Pair<A, List<A> or nil>
  - A compound data structure that is recursively defined as a pair, whose first element is of type A, and whose second element is either a list of type A or the empty list.
    - E.g. (list 1 2 3) has type List<number>; while (list 1 "string" 3) has type List<number or string>

6.001 SICP 5/16



Types – procedures

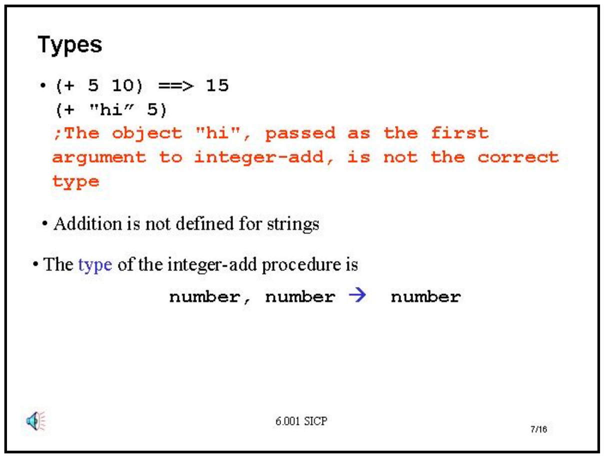
- Since procedures operate on objects, and return values, we can define their types as well.
- We will denote a procedures type by indicating the types of each of its arguments, and the type of the returned value, plus the symbol → to indicate that the arguments are mapped to the return value
- E.g. number → number specifies a procedure that takes a number as input, and returns a number as value

6.001 SICP 6/16

Slide 6.1.6
Now, we don't just have data structures in our language, we have procedures that operate on them. So we want to identify different types of procedures as well. We will do this by creating a notation that indicates the number of arguments to a procedure, the type of each, and the type of the value returned by the procedure. For example, number maps to number would indicate a procedure that takes a single number as input and returns a number as output.

Slide 6.1.7

We have now seen that for procedures we can also associate a type. In particular, we can specify for a procedure that it expects a particular number of arguments, and each argument is expected to be of a particular type. Moreover, the procedure's output is also of a particular type. And going back to our original example, we know that integer addition has a particular expectation on the types of arguments.

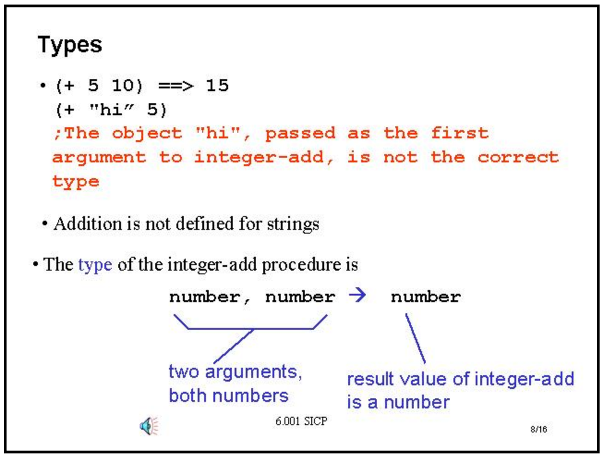


Types

- `(+ 5 10) ==> 15`
  `(+ "hi" 5)`
  `;The object "hi", passed as the first argument to integer-add, is not the correct type`
- Addition is not defined for strings
- The type of the integer-add procedure is

  `number, number → number`

6.001 SICP 7/16



Types

- `(+ 5 10) ==> 15`
  `(+ "hi" 5)`
  `;The object "hi", passed as the first argument to integer-add, is not the correct type`
- Addition is not defined for strings
- The type of the integer-add procedure is

  $\underbrace{\texttt{number, number}}_{\text{two arguments, both numbers}} \rightarrow \underbrace{\texttt{number}}_{\text{result value of integer-add is a number}}$

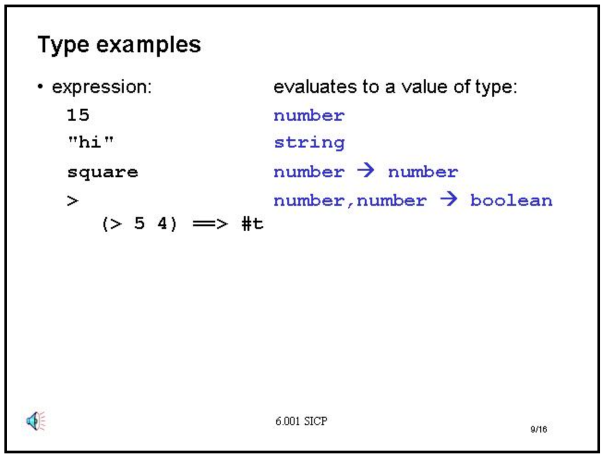
6.001 SICP 8/16

Slide 6.1.8

So we can denote this as shown, indicating the number and type of arguments, or inputs, and the type of output of a procedure. This type declaration states that this procedure maps two numbers to a number.

Slide 6.1.9

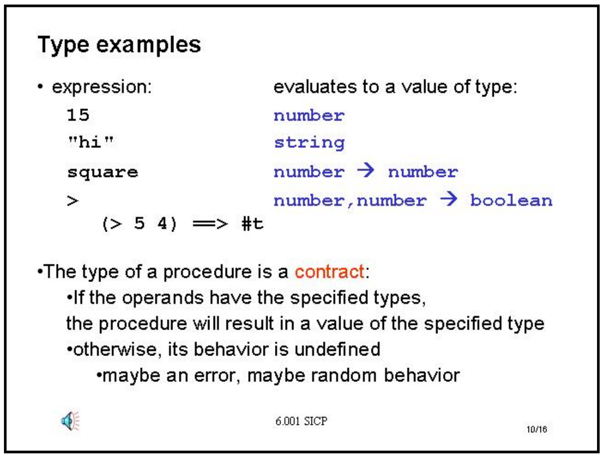
This means we can now talk about the types associated with simple expressions, and to procedures. Here are some examples. Note that square, which we defined earlier, is a procedure that takes one input, a number, and produces a number as output. The primitive > is a predicate that takes two numbers as input, and returns a true or false value, hence a boolean, as output (true if the first number is greater than the second one).



Type examples

• expression:	evaluates to a value of type:
`15`	`number`
`"hi"`	`string`
`square`	`number → number`
`>`	`number,number → boolean`
`(> 5 4) ==> #t`	

6.001 SICP 9/16



Type examples

• expression:	evaluates to a value of type:
`15`	`number`
`"hi"`	`string`
`square`	`number → number`
`>`	`number,number → boolean`
`(> 5 4) ==> #t`	

- The type of a procedure is a contract:
  - If the operands have the specified types, the procedure will result in a value of the specified type
  - otherwise, its behavior is undefined
    - maybe an error, maybe random behavior

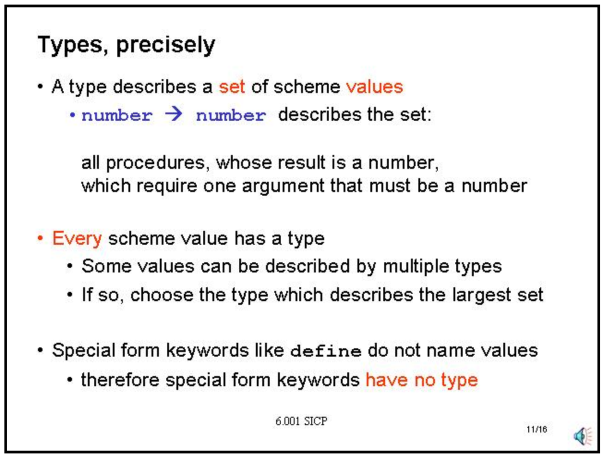
6.001 SICP 10/16

Slide 6.1.10

Thus, the type associated with a procedure specifies a kind of contract. It states the number and type of inputs, and basically guarantees that a particular type of output will be returned given those types of inputs, but says "all bets are off" if the types of the inputs are incorrect.

Note that it does not guarantee that the correct answer is returned, only that the correct type of answer will be returned. Many languages require that the programmer explicitly state this contract. In **strongly typed** languages, the programmer must specify at time of definition the types of inputs and the type of output. This allows the system to catch errors and help the programmer with debugging but restricts the flexibility of the procedures. Scheme is a **weakly typed** language, meaning that the programmer does not specify the types of inputs and outputs. Thus, errors will not be caught until run-time, which can make debugging harder, but the programmer tends to have more flexibility in creating procedures. Even in a weakly typed language, a good programmer will take advantage of the discipline of type analysis.

Slide 6.1.11
More formally, we see that a type actually describes many procedures, basically any procedure that maps the specified number and type of inputs to the specified type of output. We also see that not only does every Scheme **value** have a type, but that in some cases, a value may be described by several types (for example an integer is also a kind of number). When this happens, we will typically choose the type which describes the largest set of objects as our designated type. Thus, for example, addition maps two integers to an integer, but it also maps two numbers (including real numbers) to a number, so we would use this latter type as our specification for addition. Finally, note that some special forms do not name values, and hence do not have a type associated with them.


Types, precisely

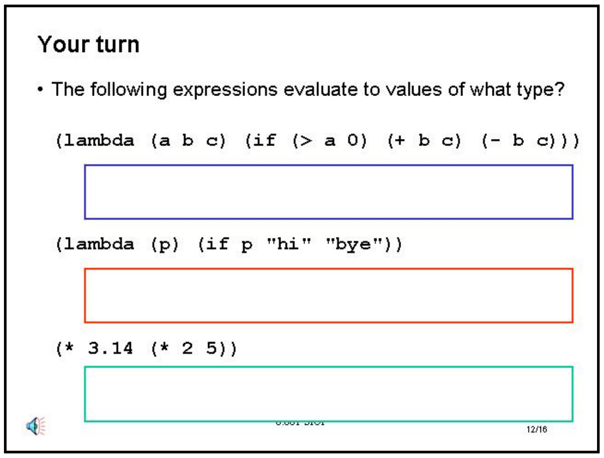
- A type describes a set of scheme values
  - `number → number` describes the set:

    all procedures, whose result is a number, which require one argument that must be a number

- Every scheme value has a type
  - Some values can be described by multiple types
  - If so, choose the type which describes the largest set

- Special form keywords like `define` do not name values
  - therefore special form keywords have no type

6.001 SICP 11/16


Your turn

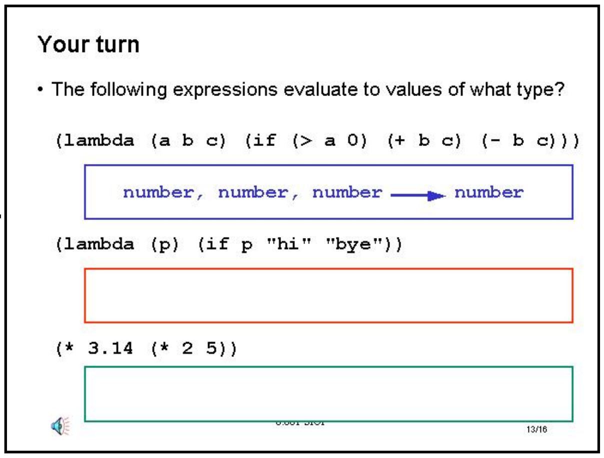
- The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
(lambda (p) (if p "hi" "bye"))
(* 3.14 (* 2 5))
```

12/16

Slide 6.1.12
Let's see if you are getting this. Here are three example expressions. If evaluate each of them, you will get some Scheme object, and you should be able to reason through to determine the type of that returned value. When you are ready to see the answers, go to the next slide.

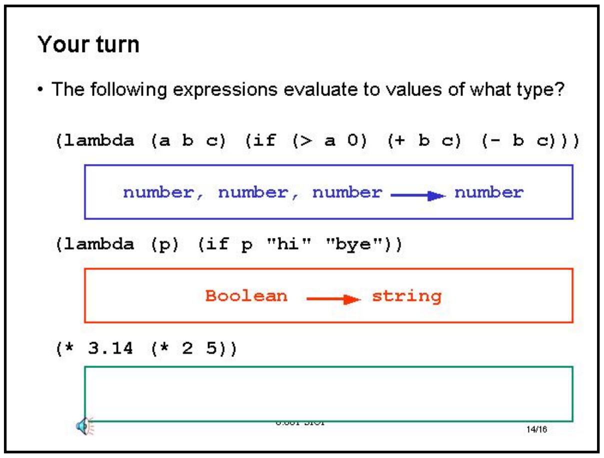
Slide 6.1.13
We know that the type of the first expression should be a procedure, since we know that **lambda** will create a procedure. We also know that the number of arguments for this procedure is 3. In terms of the types of the arguments, we can see that the first one must be a number, since **greater than** expects a number as argument. The other two arguments must also be numbers, since they are used with procedures that take numbers as input. Finally, the value returned by this procedure is a number, since both the consequent and the alternative clauses of the `if` expression return numbers. Hence, we have the type declaration shown here, a procedure that maps three numbers to a number.


Your turn

- The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
    number, number, number ——▶ number
(lambda (p) (if p "hi" "bye"))
(* 3.14 (* 2 5))
```

13/16



Your turn

- The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

number, number, number → number

```
(lambda (p) (if p "hi" "bye"))
```

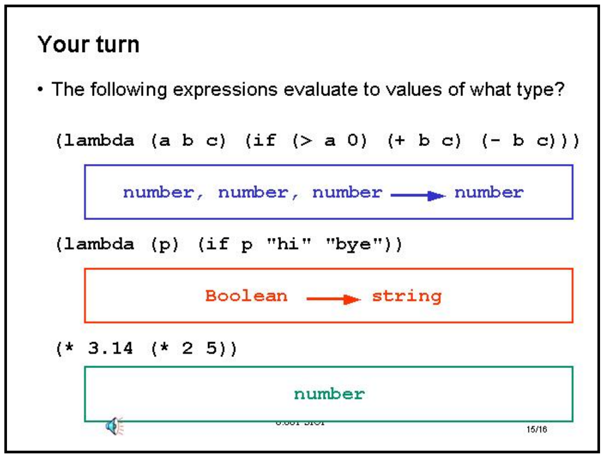
Boolean → string

```
(* 3.14 (* 2 5))
```

6.001 SICP 14/16

Slide 6.1.14
For the second example, we also know it is a procedure, due to the lambda. For the input argument, we can see that it must be an expression whose value is a boolean, since `if` expects such a value as the value of its predicate. The return value of this procedure must be a string, since both the consequent and alternative clauses return strings.

Slide 6.1.15
And of course the last expression is just an expression whose type is a number.



Your turn

- The following expressions evaluate to values of what type?

```
(lambda (a b c) (if (> a 0) (+ b c) (- b c)))
```

number, number, number → number

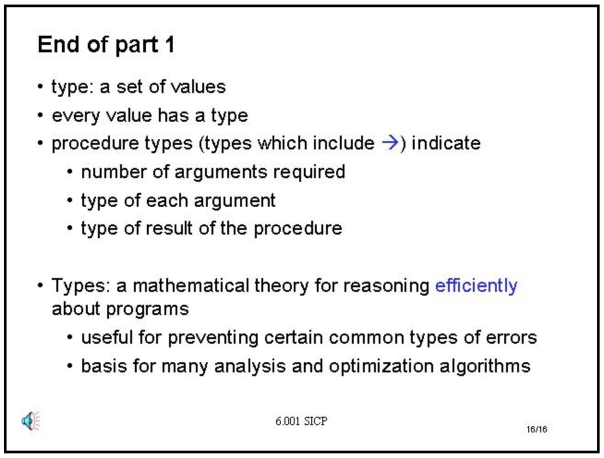
```
(lambda (p) (if p "hi" "bye"))
```

Boolean → string

```
(* 3.14 (* 2 5))
```

number

6.001 SICP 15/16



End of part 1

- type: a set of values
- every value has a type
- procedure types (types which include →) indicate
  - number of arguments required
  - type of each argument
  - type of result of the procedure
- Types: a mathematical theory for reasoning efficiently about programs
  - useful for preventing certain common types of errors
  - basis for many analysis and optimization algorithms

6.001 SICP 16/16

Slide 6.1.16
So to summarize: a type is a set of values that characterizes a set of things that belong together, due to common behavior. And virtually every expression in Scheme has a value that has a type associated with it.
Of particular interest to us are procedures, which we can also type, based on the number and type of arguments, and the type of the returned value.
As we are going to see, types provide a very nice mathematical framework for reasoning about programs. First, they can provide a basis for preventing common errors in our code: if we know that a procedure expects a particular type of argument we can use this to ensure that other procedures or expressions supply the correct kind of input. Second, we are about to see how types of procedures can serve as a key tool in helping us design other procedures, by providing a framework for understanding how values are passed between procedures. In the rest of this lecture, we are going to use this idea to explore how we can create complex, but powerful, procedures that manipulate a wide range of information beyond simple numbers.

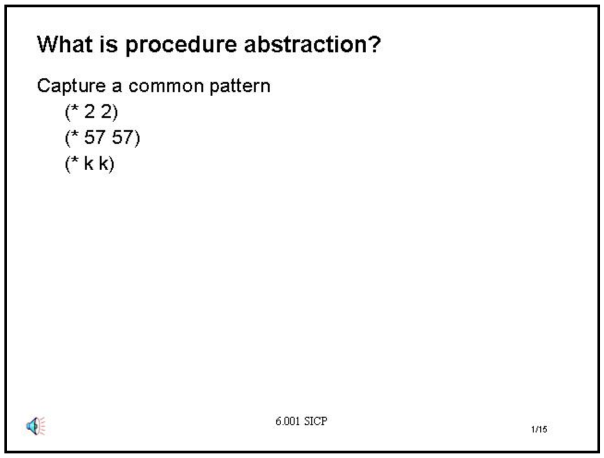
6.001 Notes: Section 6.2

Slide 6.2.1

So why bother with types? We are going to see that they are very useful in helping us reason about procedures, to catch errors and to deduce what kinds of inputs and outputs different procedures that we create must have.

In the rest of this lecture, we are going to explore how we can capture complex patterns in procedures, using types to help us reason things through.

To do this, let's go back to the basic idea of a procedure, which was to capture some common pattern of computation. For example, here are three expressions that have a common pattern, that of multiplying a value by itself. We know that this is a common pattern, and we would like to be able to refer to it using a name. To do this, we first need to capture this pattern of computation in a procedure abstraction.



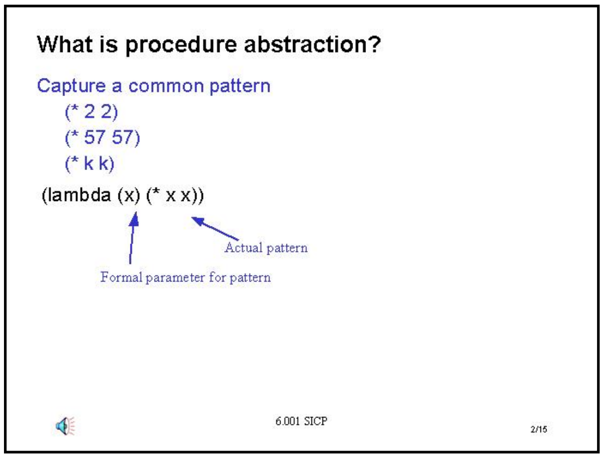
What is procedure abstraction?

Capture a common pattern
(* 2 2)
(* 57 57)
(* k k)

6.001 SICP 1/15

Slide 6.2.2

... and here is how we do that. We create a lambda, with a formal parameter to capture the part of the expression that can change, and with a body that actually specifies the common pattern, using the formal parameter as a place holder for input values. The key point is that this **lambda** expression captures that common pattern so that we can reuse it many times.



What is procedure abstraction?

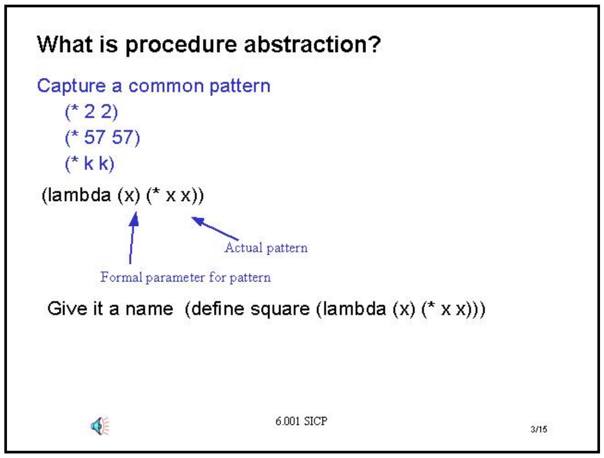
Capture a common pattern
(* 2 2)
(* 57 57)
(* k k)
(lambda (x) (* x x))

Actual pattern
Formal parameter for pattern

6.001 SICP 2/15

Slide 6.2.3

Once we have captured that common pattern, we want to be able to refer to it, so we give a name to the procedure.



What is procedure abstraction?

Capture a common pattern
(* 2 2)
(* 57 57)
(* k k)
(lambda (x) (* x x))

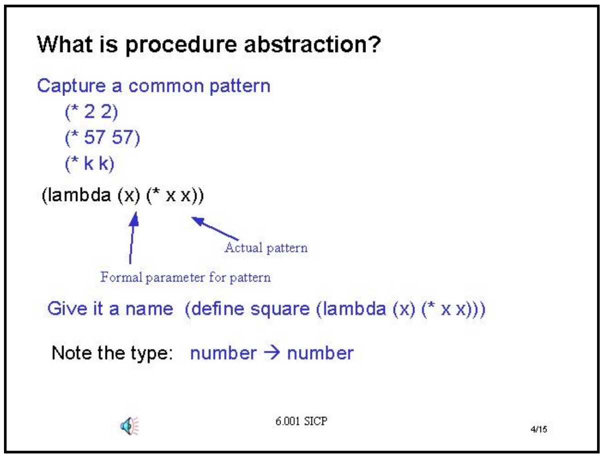
Actual pattern
Formal parameter for pattern

Give it a name (define square (lambda (x) (* x x)))

6.001 SICP 3/15

Slide 6.2.4

And note that now we can specify the type of this procedure, in this case a procedure that takes a single number as input, and returns a number as its value.



What is procedure abstraction?

Capture a common pattern
(* 2 2)
(* 57 57)
(* k k)
(lambda (x) (* x x))

Actual pattern
Formal parameter for pattern

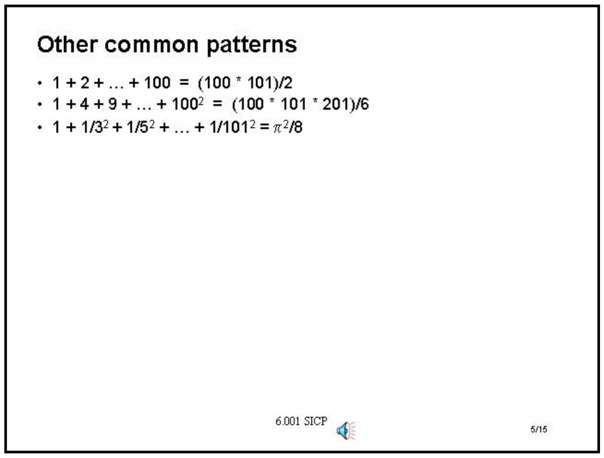
Give it a name (define square (lambda (x) (* x x)))

Note the type: number → number

6.001 SICP 4/15

Slide 6.2.5

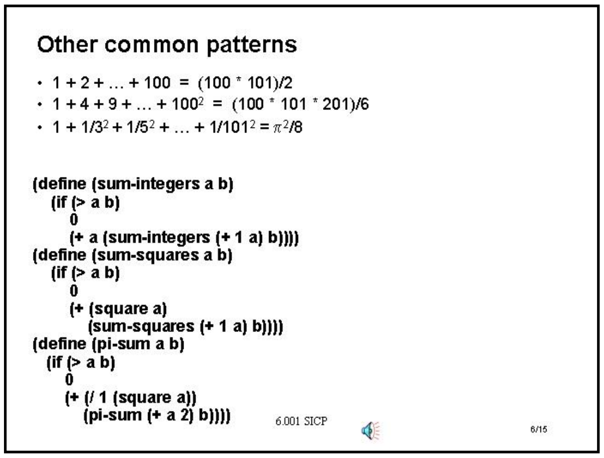
So the key idea in procedure abstraction is to capture a common pattern, give it a name, then use that name as if it were a primitive operation, without having to worry about the details of how that concept is computed. Thus, for example, we now have the notion of squaring, and we can use **square** as if it were a basic operation.
Let's take the idea of capturing common patterns and push on it. Here are three expressions from mathematics, adding up the integers, adding up the square of the integers (notice that we have used the concept of squaring here as a primitive) and adding up the recipricals of the odd squares. Note that the last expression is interesting, as it gives us a pretty good approximation to **pi**.



Other common patterns

- $1 + 2 + \ldots + 100 = (100 * 101)/2$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 201)/6$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8$

6.001 SICP 5/15



Other common patterns

- $1 + 2 + \ldots + 100 = (100 * 101)/2$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 201)/6$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8$

```
(define (sum-integers a b)
   (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))
(define (sum-squares a b)
   (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ 1 a) b))))
(define (pi-sum a b)
  (if (> a b)
     0
     (+ (/ 1 (square a))
        (pi-sum (+ a 2) b))))
```

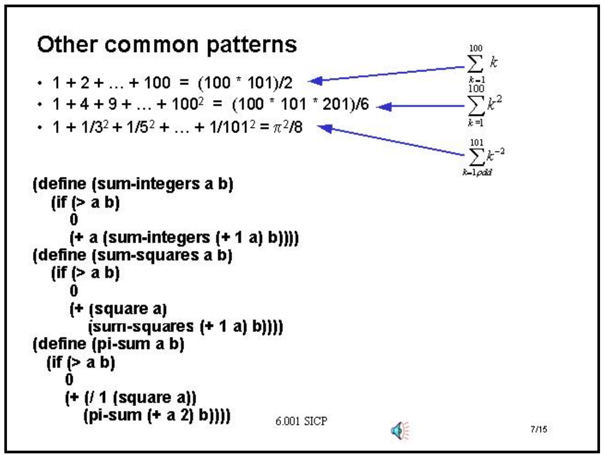
6.001 SICP 6/15

Slide 6.2.6

We can easily write procedures to compute these expressions. Each has the form we expect. For example, the first procedure has a nice recursive structure in which we add the value of the input parameter **a** to whatever we get by summing the rest of the integers from **a+1** to **b**. The other procedures similarly use the idea of reducing a problem to a simpler version of the same problem, plus a simple operation. Notice how we have already incorporated the abstraction of **square**, burying the details of how it is computed behind the procedural abstraction we just built.

Slide 6.2.7

If we stop here, however, we haven't quite captured all of the common pattern here. In particular, each of these three mathematical expressions is a **summation**, that is, a pattern in which we add up a series of values, starting from some specified point, ending at some specified point, with some method of incrementing our place, and with some particular thing that we "sum" at each stage. In fact, the mathematicians have notation for this, the summation notation shown.
Note that this is more than just a convenient notation. The idea of summation is a general pattern that we would like to capture, the same way the mathematicians have.

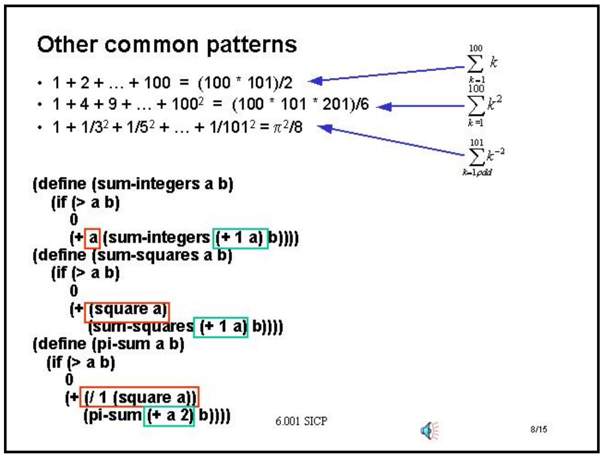


Other common patterns

- $1 + 2 + \ldots + 100 = (100 * 101)/2 \leftarrow \sum_{k=1}^{100} k$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 201)/6 \leftarrow \sum_{k=1}^{100} k^2$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8 \leftarrow \sum_{k=1, odd}^{101} k^{-2}$

```
(define (sum-integers a b)
   (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))
(define (sum-squares a b)
   (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ 1 a) b))))
(define (pi-sum a b)
  (if (> a b)
     0
     (+ (/ 1 (square a))
        (pi-sum (+ a 2) b))))
```

6.001 SICP 7/15



Other common patterns

- $1 + 2 + \ldots + 100 = (100 * 101)/2 \leftarrow \sum_{k=1}^{100} k$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 201)/6 \leftarrow \sum_{k=1}^{100} k^2$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8 \leftarrow \sum_{k=1, odd}^{101} k^{-2}$

```
(define (sum-integers a b)
   (if (> a b)
      0
      (+ a (sum-integers (+ 1 a) b))))
(define (sum-squares a b)
   (if (> a b)
      0
      (+ (square a)
         (sum-squares (+ 1 a) b))))
(define (pi-sum a b)
  (if (> a b)
     0
     (+ (/ 1 (square a))
        (pi-sum (+ a 2) b))))
```

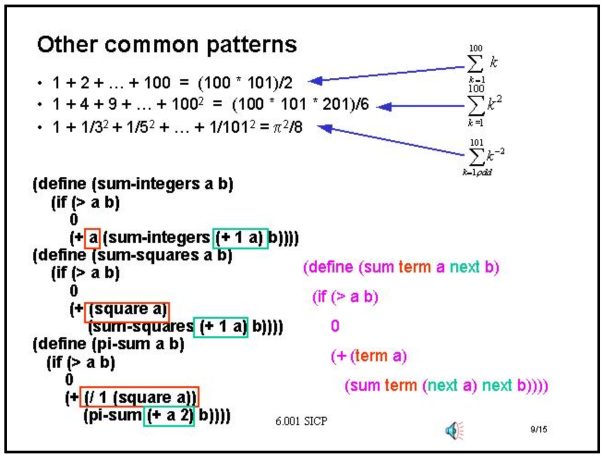
6.001 SICP 8/15

Slide 6.2.8

If we look at our procedures, we can see that there clearly is a common pattern to them. Here are highlights for the two things that are different in these procedures. There is the **term** that we add at each stage, shown in red, and there is the way in which we move to the next value in the sequence, shown in green.
Now, how did we go from (* 2 2) or (* a a) to the idea of squaring? We gave a name to each part of the pattern that was going to have to change, and we capture the pattern in a procedure using that parameter. Let's do the same thing here. We will need a parameter for the term to be added in, a parameter for the next stage in the procedure, and of course the two existing parameters, **a** and **b**. Let's now capture the commonality of this computation in a procedure that uses these parameters.

Slide 6.2.9

So here is the procedure, with those four parameters. All I have done is substitute the formal parameter for the right spot in the pattern of the computation, which forms the body of this procedure. But this looks a little different from things we have seen before so let's look more carefully at this in the next slide.



Other common patterns

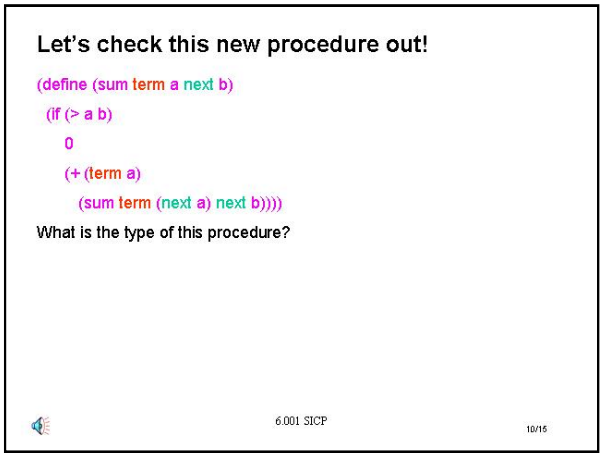
- $1 + 2 + \ldots + 100 = (100 * 101)/2$ ← $\sum_{k=1}^{100} k$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 201)/6$ ← $\sum_{k=1}^{100} k^2$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8$ ← $\sum_{k=1, odd}^{101} k^{-2}$

```
(define (sum-integers a b)
   (if (> a b)
       0
       (+ a (sum-integers (+ 1 a) b))))
(define (sum-squares a b)
   (if (> a b)
       0
       (+ (square a)
          (sum-squares (+ 1 a) b))))
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1 (square a))
         (pi-sum (+ a 2) b))))
```

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

6.001 SICP 9/15

Slide 6.2.10



Let's check this new procedure out!

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

What is the type of this procedure?

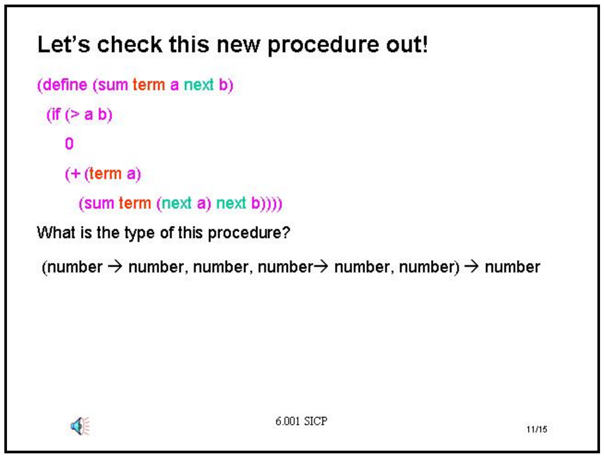
6.001 SICP 10/15

Up to this point, every procedure we have written has expected a primitive object as a parameter, typically a number. This procedure is different, however. We can see from where the parameters `term` and `next` appear in the body of the procedure that this procedure expects these parameters to be themselves **procedures!**

Does this really work? Well, what is the type of this procedure?

Slide 6.2.11

By using the same kind of reasoning we did earlier, we can see that this procedure requires four input parameters, and the output value must be a numbers, since both clauses of the `if` expression should return a number.



Let's check this new procedure out!

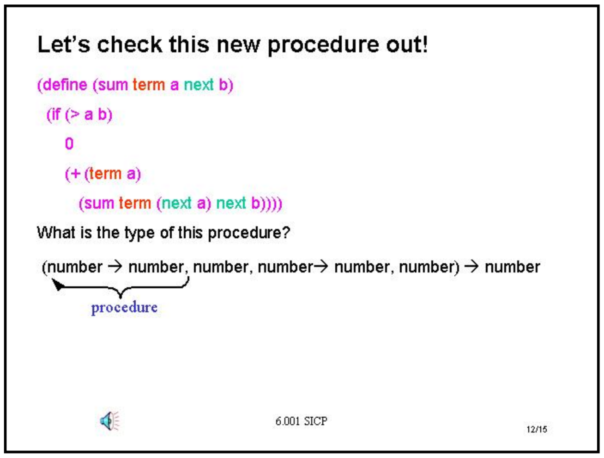
```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

What is the type of this procedure?

(number → number, number, number→ number, number) → number

6.001 SICP 11/15

Slide 6.2.12



Let's check this new procedure out!

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

What is the type of this procedure?

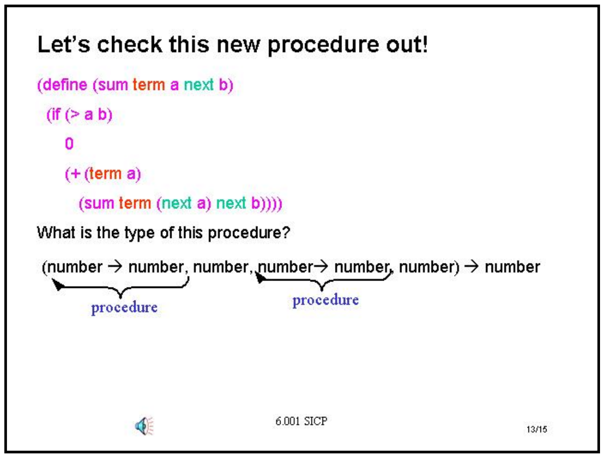
$\underbrace{\text{(number} \rightarrow \text{number,}}_{\text{procedure}}$ number, number→ number, number) → number

6.001 SICP 12/15

And what are the inputs? Well, the first parameter is used in a place that expects a procedure. That procedure has one input that we will shortly see is a number, and it must produce a number as output, as that value will be used by the addition operation. The second parameter we can see must be a number, since we are going to use the greater than operation on it. Thus, we see the formalism for the first two inputs to `sum`, a procedure mapping numbers to numbers, and a number.

Slide 6.2.13

The same reasoning helps us deduce the form for the third parameter (a procedure mapping numbers to numbers) for the fourth parameter (a number).



Let's check this new procedure out!

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
      (sum term (next a) next b))))
```

What is the type of this procedure?

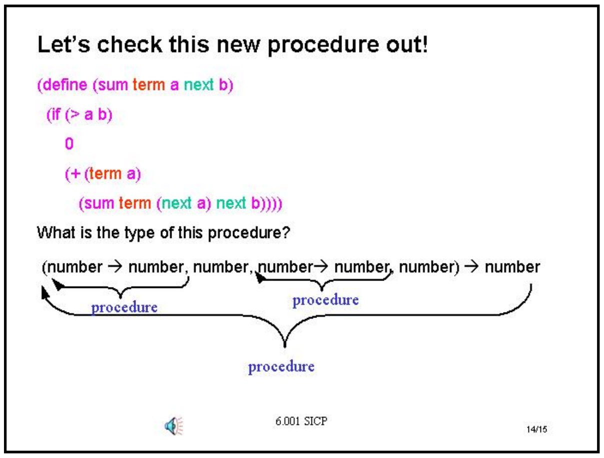
$\underbrace{(\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}, \underbrace{\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}) \rightarrow \text{number}$

6.001 SICP 13/15

Slide 6.2.14

... and overall this procedure, `sum`, will return a number, since both clauses of the `if` expression that forms the body of the procedure must return a number.

Thus we can characterize the type of this new procedure, `sum`. It just happens that some of the parameters for this procedure are themselves procedures!



Let's check this new procedure out!

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a)
      (sum term (next a) next b))))
```

What is the type of this procedure?

$\underbrace{\underbrace{(\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}, \underbrace{\text{number} \rightarrow \text{number}}_{\text{procedure}}, \text{number}) \rightarrow \text{number}}_{\text{procedure}}$

6.001 SICP 14/15

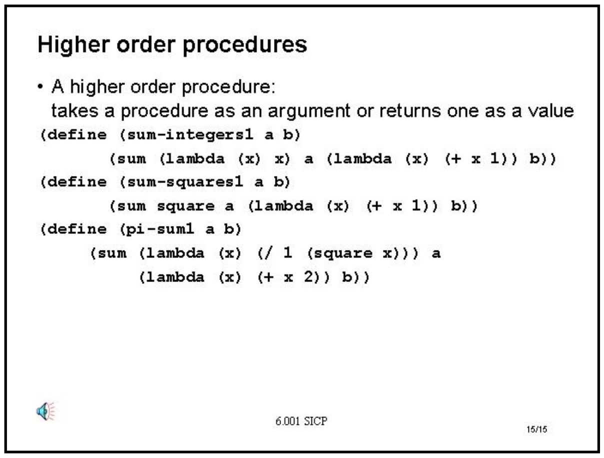
Slide 6.2.15

So this is a new kind of beast, which we call a **higher order procedure**. It is a procedure that takes as inputs other procedures, and it may, as we will see, also return a procedure as its returned value.

Note that those input procedures may be named procedures, like `square`, or they may be pure `lambda` expressions. In either case, we know that the value of the expression will be a procedure object that can then be substituted into the body of **sum**.

And now we see that we have captured the common pattern of **summation**, and that our three earlier expressions are just special versions of this general computation.

Does this really work? Well, just use the substitution model to check that procedures get substituted in the right places in the body of **sum** and that as a consequence the three expressions shown here will compute the appropriate series of terms.



Higher order procedures

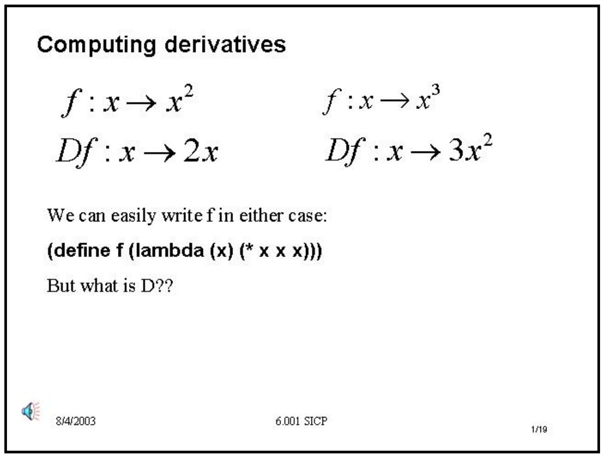
- A higher order procedure:
  takes a procedure as an argument or returns one as a value

```
(define (sum-integers1 a b)
        (sum (lambda (x) x) a (lambda (x) (+ x 1)) b))
(define (sum-squares1 a b)
        (sum square a (lambda (x) (+ x 1)) b))
(define (pi-sum1 a b)
      (sum (lambda (x) (/ 1 (square x))) a
           (lambda (x) (+ x 2)) b))
```

6.001 SICP 15/15

6.001 Notes: Section 6.3

Slide 6.3.1
So we have now seen how procedures can be passed in as arguments to other procedures. But we can also have procedures return procedures as values. Where might that be useful?
Well here is a simple mathematical example. What is a derivative? First, we know that given some function f, such as the two shown here, there are standard rules for finding the derivative of f, which we denote Df, as shown.
We can easily write down a procedure to compute f, but what is D? In particular, can we write a single procedure for D that would take the derivative of any function?



Computing derivatives

$f : x \to x^2$ $\qquad$ $f : x \to x^3$

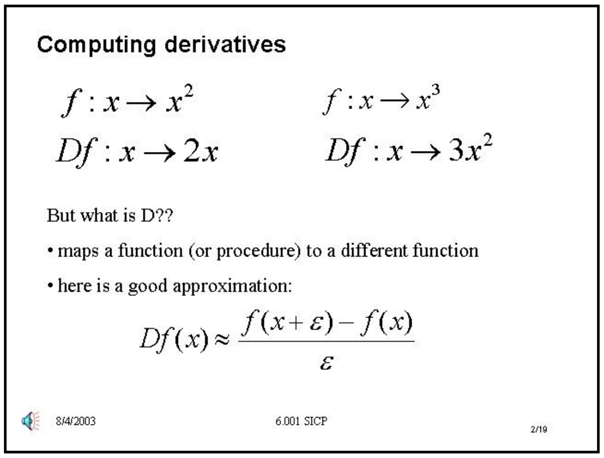
$Df : x \to 2x$ $\qquad$ $Df : x \to 3x^2$

We can easily write f in either case:

```
(define f (lambda (x) (* x x)))
```

But what is D??

8/4/2003 6.001 SICP 1/19



Computing derivatives

$f : x \to x^2$ $\qquad$ $f : x \to x^3$

$Df : x \to 2x$ $\qquad$ $Df : x \to 3x^2$

But what is D??

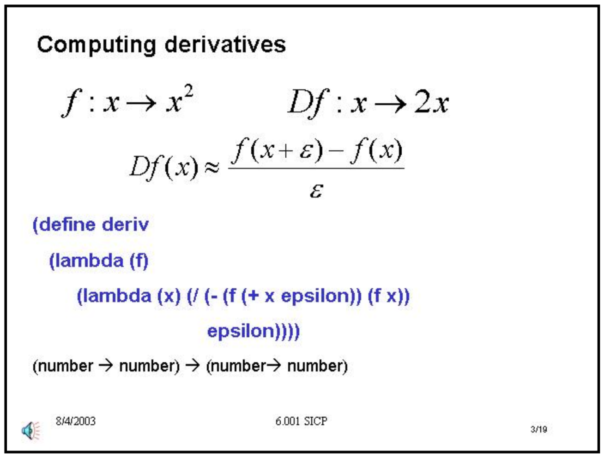
- maps a function (or procedure) to a different function
- here is a good approximation:

$$Df(x) \approx \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$$

8/4/2003 6.001 SICP 2/19

Slide 6.3.2
So what do we know about this beast? Well, it needs to map a function to another function, or said slightly differently, it needs to take as input any procedure that represents a numerical function, and it needs to **return a procedure** with the property that applying that returned procedure to any value will give us a good approximation to the derivative of the original function at that value.
Numerically, the equation shown will do a decent job for us.

Slide 6.3.3
So how do we then capture the idea of D? Well, here it is.
Look carefully at the form. Deriv takes one argument, which we see must be a procedure of one numerical argument. Inside the body of the lambda associated with deriv is a second lambda. This is also a procedure of one numerical argument, that returns a numerical value.
So the type of this procedure is as shown.



Computing derivatives

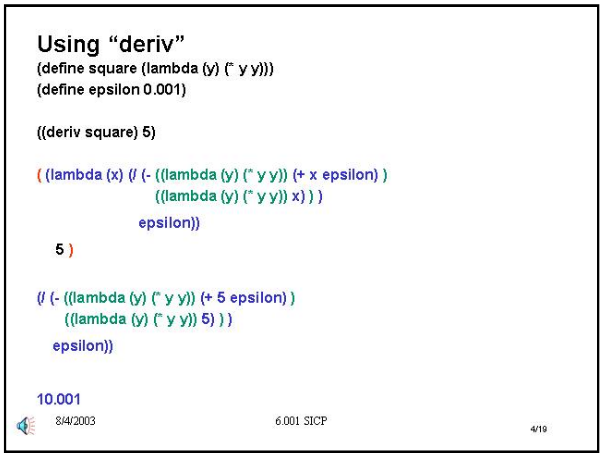
$f : x \to x^2$ $\qquad$ $Df : x \to 2x$

$$Df(x) \approx \frac{f(x+\varepsilon)-f(x)}{\varepsilon}$$

```
(define deriv
  (lambda (f)
     (lambda (x) (/ (- (f (+ x epsilon)) (f x))
                    epsilon))))
```

(number → number) → (number→ number)

8/4/2003 6.001 SICP 3/19



Using “deriv”

```
(define square (lambda (y) (* y y)))
(define epsilon 0.001)

((deriv square) 5)

( (lambda (x) (/ (- ((lambda (y) (* y y)) (+ x epsilon) )
                    ((lambda (y) (* y y)) x) ) )
                epsilon))
   5 )

(/ (- ((lambda (y) (* y y)) (+ 5 epsilon) )
     ((lambda (y) (* y y)) 5) ) )
   epsilon))

10.001
```

8/4/2003 6.001 SICP 4/19

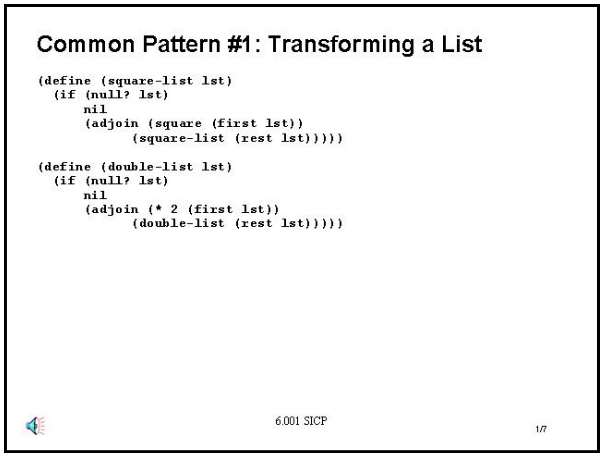
Slide 6.3.4
Now does this make sense? Sure.
Suppose we define square as shown, and we take the derivative. Notice the nested parentheses: the first compound subexpression should return a procedure as a value, for this to make sense. And to see that it does, use the substitution model. We substitute the value of square everywhere in the body of deriv that we see the formal parameter f. The mess that we get does exactly that.
Now, if we complete the process, we substitute 5 for x, which reduces to an expression similar to those we have seen before.
So we see that this kind of higher order procedure also has a

valuable role.

6.001 Notes: Section 6.4

Slide 6.4.1

Now, let's see how we can use "higher order procedures" on data structures. Remember our lists from before. We saw that there were common patterns for using lists, especially creating new lists out of old ones. In particular, we had the idea of generating a list, consing it up as we move down another list. We can generalize that idea, so that instead of just making a copy of a list, I do something to each element of the list as I make the copy.
For example, each of these procedures takes a list as input and generates a new list, transforming each element in a fixed way. The first example squares each element of the input list, the second one doubles each element of the list.
Notice the form is very similar to our **append** example. We walk down the input list one element at a time, however here we actually apply some procedure to the element before adjoining it to the output. Note how the procedures use the selectors and constructors of the data abstraction to extract elements and create elements, and how the recursive structure of the procedure reflects the recursive structure of the data abstraction.



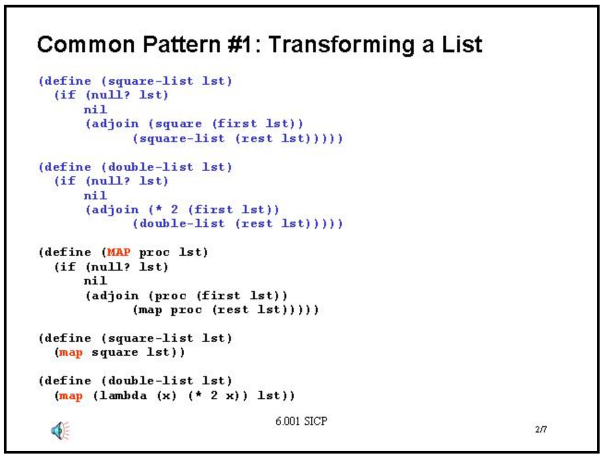
Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))
```

6.001 SICP 1/7

Slide 6.4.2

Clearly there is a pattern here: we take a list as input, walk down it an element at a time, do **something** to each element, and construct a new list of the results. Because there is a common pattern, we can capture it as a procedural abstraction. **Map** is that general pattern. It is a higher order procedure that takes as input both a list and a procedure, and simply applies the procedure to each element of the old list to create the corresponding element of the new list. As a consequence, we can easily capture the two examples as instances of **map**, and **map** forms a very common interface for manipulating lists.

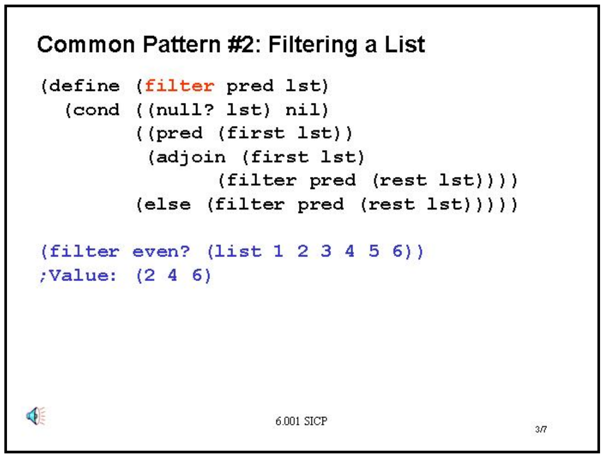


Common Pattern #1: Transforming a List

```
(define (square-list lst)
  (if (null? lst)
      nil
      (adjoin (square (first lst))
              (square-list (rest lst)))))

(define (double-list lst)
  (if (null? lst)
      nil
      (adjoin (* 2 (first lst))
              (double-list (rest lst)))))

(define (MAP proc lst)
  (if (null? lst)
      nil
      (adjoin (proc (first lst))
              (map proc (rest lst)))))

(define (square-list lst)
  (map square lst))

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

6.001 SICP 2/7

Slide 6.4.3



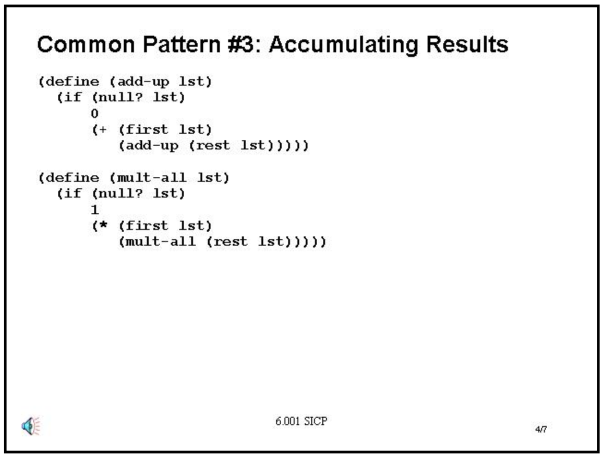
Common Pattern #2: Filtering a List

```
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (first lst))
         (adjoin (first lst)
                 (filter pred (rest lst))))
        (else (filter pred (rest lst)))))

(filter even? (list 1 2 3 4 5 6))
;Value: (2 4 6)
```

6.001 SICP 3/7

Note that in all the previous examples, and indeed for any example using **map**, the length of the output list is the same as the length of the input list. For each element of the input list, I do some process and create a **corresponding** element of the output list. But what if I want a list as output that is different in length, in particular, a list that has corresponding elements for only **some** of the elements of the input list?
For that, we have another conventional interface, called a **filter**. Much like a coffee filter, a **filter** will only let some elements of the input list through. As before, we recursively walk along the list. If the list is empty, we just return an empty list, to guarantee that the result is a list. Otherwise, we apply a predicate to the first element of the list. Note that **pred** must be a procedure that returns a boolean as output, and hence `filter` must be a higher order procedure. If the result of applying the predicate to the element is true, we adjoin a copy of the element onto the output list, otherwise we move on to the next element.

Slide 6.4.4



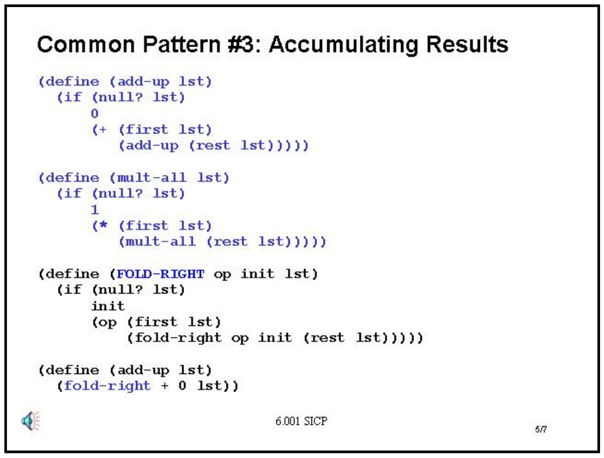
Common Pattern #3: Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
         (add-up (rest lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
         (mult-all (rest lst)))))
```

6.001 SICP 4/7

Finally, suppose we want to gather the elements of the list together into a new value, rather than creating a list as output. For example, we might want to add up all the elements of a list, or multiply them together. Here, we could just recursively walk down the list, adding the first element of the list to whatever we get by recursively doing the same thing to the rest of the list (there is that nice recursion property again!). Of course, we have to be careful about what we return when we reach the empty list. For addition, we add in a 0, for multiplication we return a 1.
Here are two common patterns for converting lists to numbers, and as before we should be able to capture this common pattern.

Slide 6.4.5

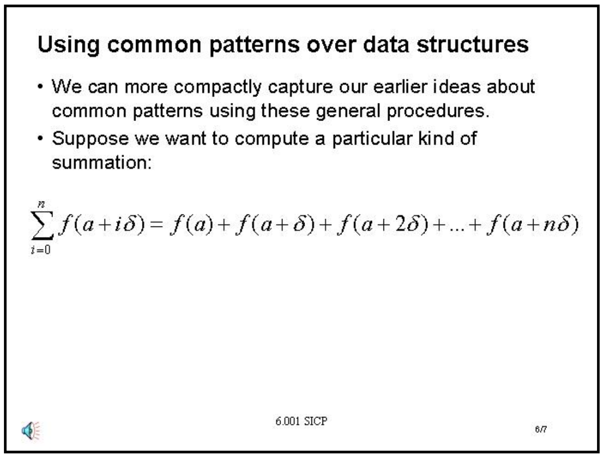


Common Pattern #3: Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (first lst)
         (add-up (rest lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (first lst)
         (mult-all (rest lst)))))

(define (FOLD-RIGHT op init lst)
  (if (null? lst)
      init
      (op (first lst)
          (fold-right op init (rest lst)))))

(define (add-up lst)
  (fold-right + 0 lst))
```

6.001 SICP 5/7

... and here is the generalization, a third standard operation on lists called **fold-right**.
Notice its form: it has three arguments, a procedure **op**, an initial value **init** and a list **lst**. It's structure is to return the initial value, if the list is empty, otherwise it returns the value of applying the operator procedure to the first element of the list, and whatever we get by reducing the remainder of the list.
Notice the nice use of closure and the recursive structure of the lists and procedure to unwrap this procedure into a simpler version of the same problem.
These common interfaces, by the way, are very powerful ways of dealing with sequences of data. They can be applied not only to sequences of numbers, but to sequences of other structures, and we will see that many operations of such structures reduce to combinations of **map**, **filter** and **fold-right**.


Using common patterns over data structures

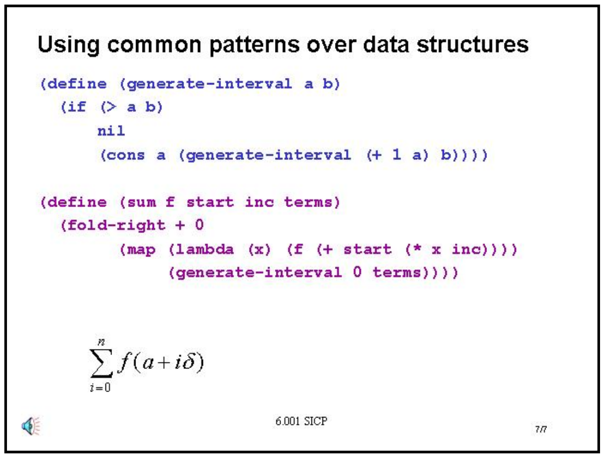
- We can more compactly capture our earlier ideas about common patterns using these general procedures.
- Suppose we want to compute a particular kind of summation:

$$\sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta)$$

6.001 SICP 6/7

Slide 6.4.6
To see how we can use these ideas, let’s go back to the idea of summation, but now for sums of a particular form. Suppose we want to add up the values of a function at a regularly spaced set of intervals. The mathematical expression captures this idea, for n+1 regularly spaced samples, each delta apart, starting at some point a.

Slide 6.4.7
Well, we can easily capture this idea using our notions of operations over lists. To generate a list of integers, we can create a procedure, generate-interval, that cons’es up a list. Given the set of integers between 0 and n, which we can create using this procedure, we can “map” a procedure down that list, computing the function f applied to sample points: note how we multiply each term in the integer list by a constant (inc), then add an offset (start) to reflect the general formula. This creates a new list, of samples of f applied to a set of points. To add them up, we “fold” them together.
So we see that we can combine our idea of higher order procedures on numbers with higher order procedures that operate over data structures.


Using common patterns over data structures

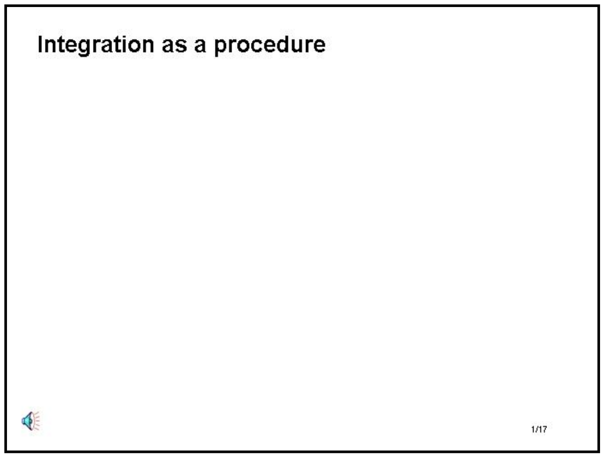
```
(define (generate-interval a b)
  (if (> a b)
      nil
      (cons a (generate-interval (+ 1 a) b))))

(define (sum f start inc terms)
  (fold-right + 0
        (map (lambda (x) (f (+ start (* x inc))))
             (generate-interval 0 terms))))
```

$$\sum_{i=0}^{n} f(a+i\delta)$$

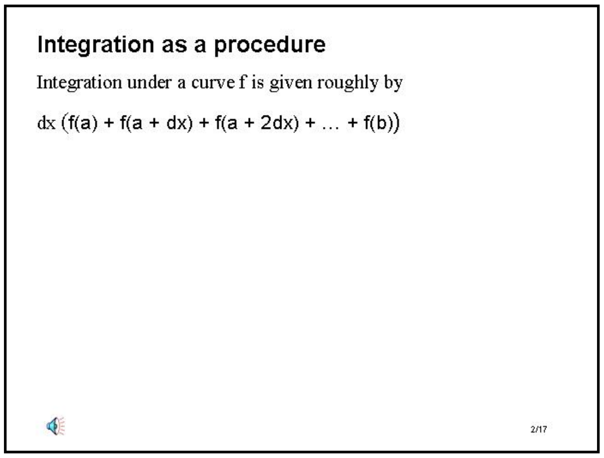
6.001 SICP 7/7

6.001 Notes: Section 6.5

Slide 6.5.1
Let's stop to stress an important point. Why are higher order procedures useful? It is not just to make writing easier. Instead, higher order procedures tremendously increase the expressive power in our language. They form building blocks that suppress unnecessary detail inside of units that we can treat as primitives, thus allowing us to build more complex structures out of units that themselves may be complex structures.
To stress this idea, let's take the concept of **sum** which we just captured as a higher order procedure, and let's see what we can do if we treat sum as a primitive procedure, a primitive building block for other procedures.


Integration as a procedure

1/17



Integration as a procedure

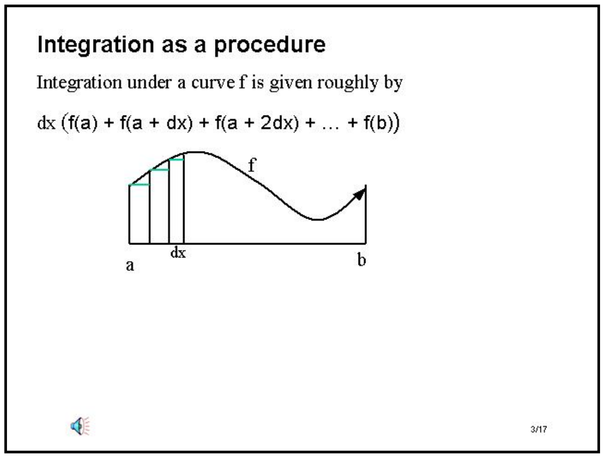
Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))

2/17

Slide 6.5.2
Remember from calculus that integration is just the idea of summation. That is, to integrate a function **f**, we take the area of a set of rectangles that approximate the area under **f**, and this is just a sum of a set of values, all scaled by the width of the rectangles.

Slide 6.5.3
That is just shown here.

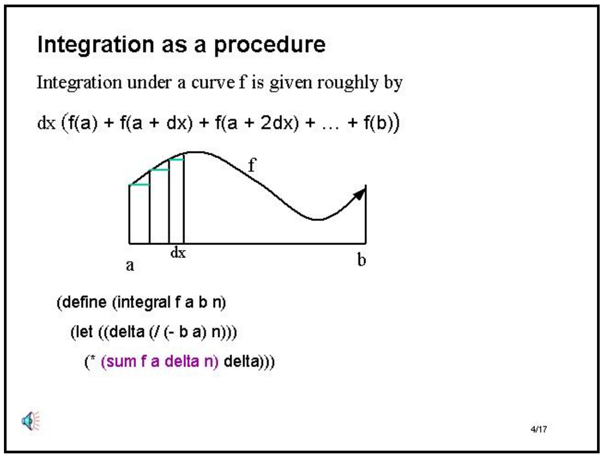


Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))

3/17



Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))

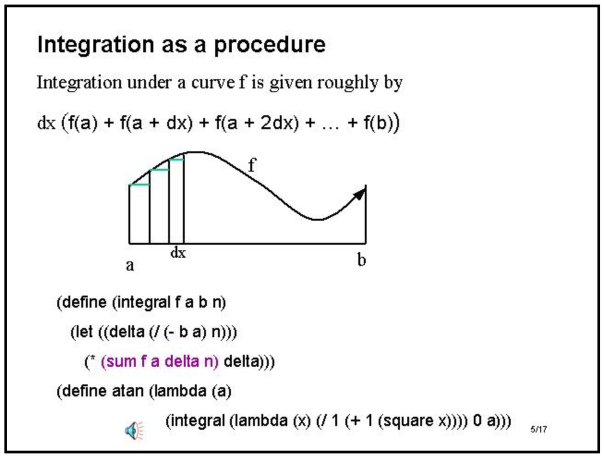
```
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* (sum f a delta n) delta)))
```

4/17

Slide 6.5.4
To capture the idea of integration, we just build on the idea of summation. In particular, we can reduce integration to a simpler problem, that of summation. Thus we simply sum **f** from **a** to **b**. Note the use of **f** as a parameter for the term to be added. We use the number of terms in the summation (or the number of sample points) to determine the increment delta to add at each stage
Note that the inputs to **sum** are all of the expected type. And **sum** we know returns a number, so we can complete the process by multiplying this number by the number **delta.**
Clearly as we increase **n**, the number of terms in the summation, we get a better approximation to the integral

Slide 6.5.5

And once I have the idea of integration, I again can treat this as an abstraction. For example, I can use the fact that inverse trigonometric functions are described as integrals, and thus by building on the idea of integration, I can easily compute the inverse tangent function.



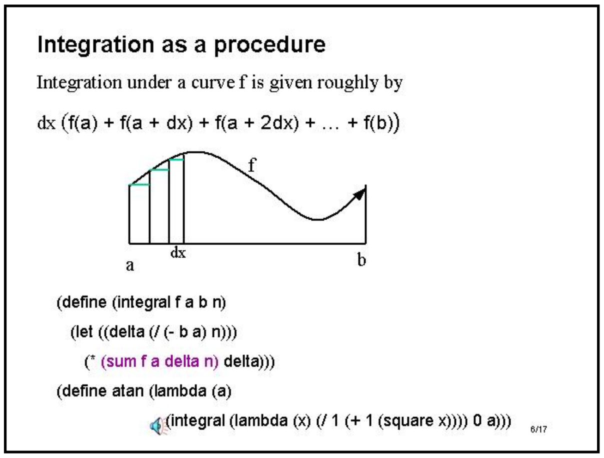
Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))

```
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* (sum f a delta n) delta)))
(define atan (lambda (a)
          (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```

5/17



Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))

```
(define (integral f a b n)
  (let ((delta (/ (- b a) n)))
    (* (sum f a delta n) delta)))
(define atan (lambda (a)
          (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```

6/17

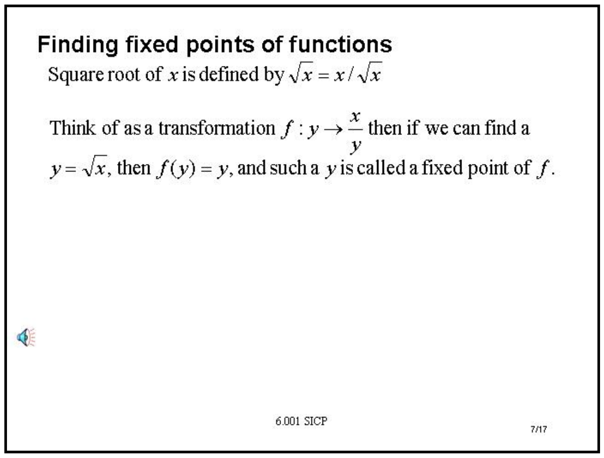
Slide 6.5.6

The point is that by capturing common patterns of computation, whose parameters may themselves be patterns of computation, we can quickly abstract out complex operations without getting lost in the details.

Slide 6.5.7

Notice how we have generalized things. We have moved functions that mapped numbers to numbers, to more general methods that work independent of the function. For example, **integral** is a procedure that takes any function as input, instead of just a number, and produces a number as output.

Let's turn to other ways of capturing common patterns in higher order procedures. Let's go back to square root, but think about it a different way. One way to think about the process of square root is to note that the square root of **x** is defined as the value that is equal to **x** divided by the same value.



Finding fixed points of functions

Square root of x is defined by $\sqrt{x} = x/\sqrt{x}$

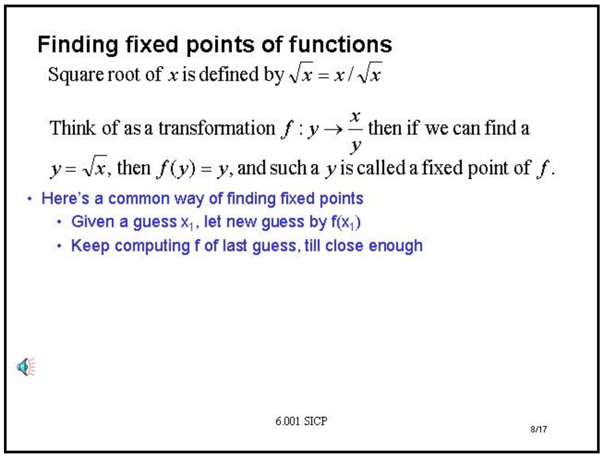
Think of as a transformation $f : y \rightarrow \frac{x}{y}$ then if we can find a $y = \sqrt{x}$, then $f(y) = y$, and such a y is called a fixed point of f.

6.001 SICP

7/17

If we think of the transformation that maps values of **y** into the ratio of **x** over **y**, then we can see that if we can somehow find a value for **y** that is the actual square root of **x**, then the value of **f** applied to that **y** will be equal to its input. Such a point is called a **fixed point**, a point at which the transformation **f** has the same input and output value.

This suggests another way of finding square roots, namely to see if we can find a guess for the input to **f** that happens to be a fixed point of **f**, for this particular **f**, or in other words, see if we can find fixed points of a function.



Finding fixed points of functions

Square root of x is defined by $\sqrt{x} = x/\sqrt{x}$

Think of as a transformation $f : y \to \frac{x}{y}$ then if we can find a $y = \sqrt{x}$, then $f(y) = y$, and such a y is called a fixed point of f.

- Here's a common way of finding fixed points
  - Given a guess x_1, let new guess by $f(x_1)$
  - Keep computing f of last guess, till close enough

6.001 SICP 8/17

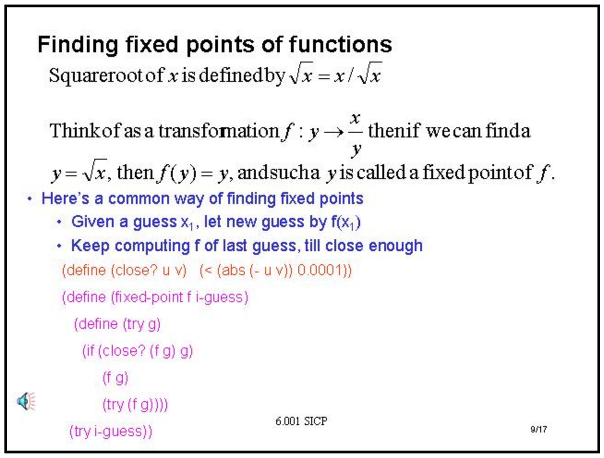
Slide 6.5.8

Here is a good way to find fixed points. Given a guess, evaluate **f** of that guess, then **f** of the result, then **f** of that result, until we get a guess that is good enough.

Slide 6.5.9

That has a familiar ring to it, as it sounds somewhat like what we did in the earlier lecture when we talked about square roots. The procedure `fixed-point` has two parameters, a procedure `f` and a guess. In the body of `fixed-point`, we use the auxiliary procedure `try`. This procedure takes a guess as input, and checks to see if the value of the input procedure applied to that guess is close to guess, i.e. if it is a fixed point. If it is not, then we "try" again, using the value of f(guess) as our next guess.

Notice the nice recursive structure to `try`, which is itself a higher order procedure, since it finds fixed points for any procedure that maps numbers to numbers.



Finding fixed points of functions

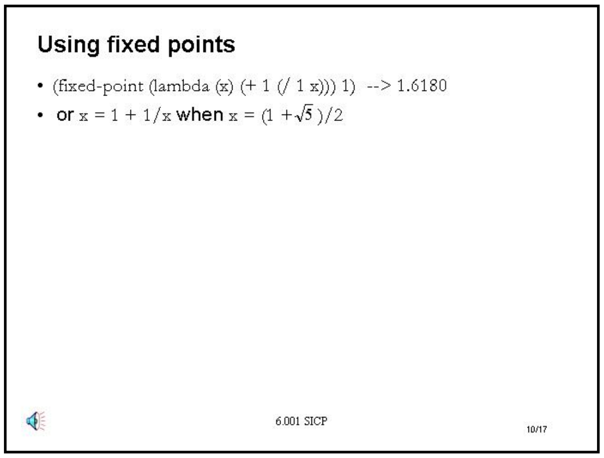
Square root of x is defined by $\sqrt{x} = x/\sqrt{x}$

Think of as a transformation $f : y \to \frac{x}{y}$ then if we can find a $y = \sqrt{x}$, then $f(y) = y$, and such a y is called a fixed point of f.

- Here's a common way of finding fixed points
  - Given a guess x_1, let new guess by $f(x_1)$
  - Keep computing f of last guess, till close enough

```
(define (close? u v)   (< (abs (- u v)) 0.0001))
(define (fixed-point f i-guess)
  (define (try g)
    (if (close? (f g) g)
        (f g)
        (try (f g))))
  (try i-guess))
```

6.001 SICP 9/17



Using fixed points

- (fixed-point (lambda (x) (+ 1 (/ 1 x))) 1) --> 1.6180
- or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

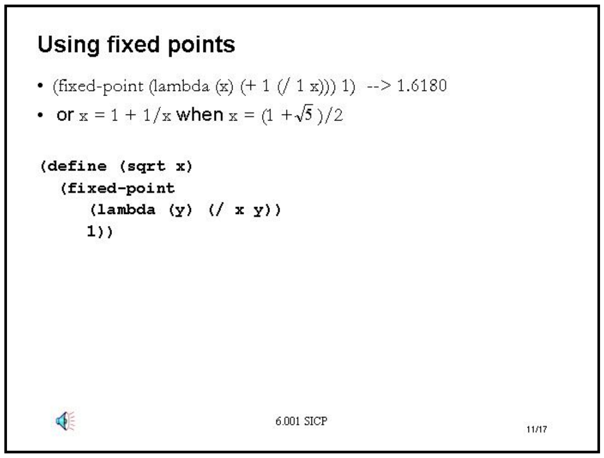
6.001 SICP 10/17

Slide 6.5.10

Given that I have captured the idea of a fixed point in a procedure, I can now use it as an abstraction. For example, the **golden ratio** is defined as that value **x** that is equal to 1 + 1/x, and thus by using **fixed-point** with the appropriate input procedure, I can compute that value. The point is that by capturing the idea of a fixed point, I can abstract away the details of the computation and just use the concept itself as a black box abstraction.

Slide 6.5.11

But to get back to our square root idea, we can now easily compute square root, by using the concept of fixed points. In this case, we just find the fixed point of the procedure that maps values of **y** to **x/y**. Notice how **fixed-point** takes as input a procedure and a number, and returns a number as value, capturing that higher order computational pattern.
Also notice how crisply we have captured the computation of square root. It is simply reduced to the concept of a fixed point of an appropriate function.

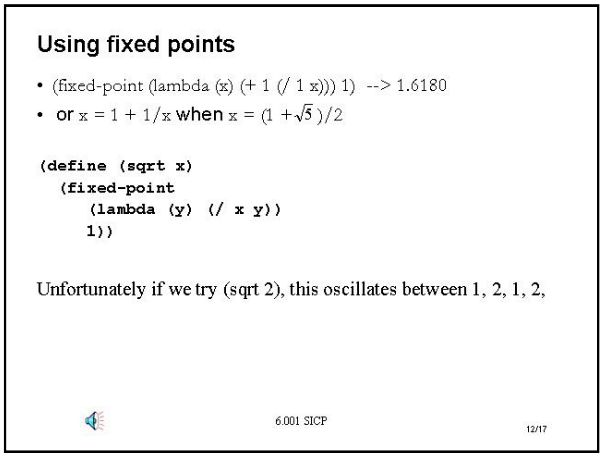


Using fixed points

- (fixed-point (lambda (x) (+ 1 (/ 1 x))) 1) --> 1.6180
- or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

6.001 SICP 11/17



Using fixed points

- (fixed-point (lambda (x) (+ 1 (/ 1 x))) 1) --> 1.6180
- or $x = 1 + 1/x$ when $x = (1 + \sqrt{5})/2$

```
(define (sqrt x)
  (fixed-point
     (lambda (y) (/ x y))
     1))
```

Unfortunately if we try (sqrt 2), this oscillates between 1, 2, 1, 2,

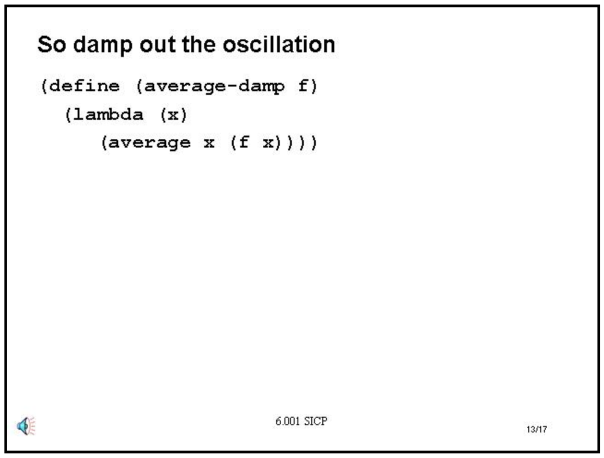
6.001 SICP 12/17

Slide 6.5.12

If we go ahead and use this procedure, we unfortunately discover that while the definition captures the concept of a square root, the actual computation doesn't quite do the right thing. Since we start the fixed point computation off with an initial value of 1, it's next guess will be 2, as you can see by looking back at the code for `try`. And thus it's next guess will be 1, and the process will simply oscillate between these two values.

Slide 6.5.13

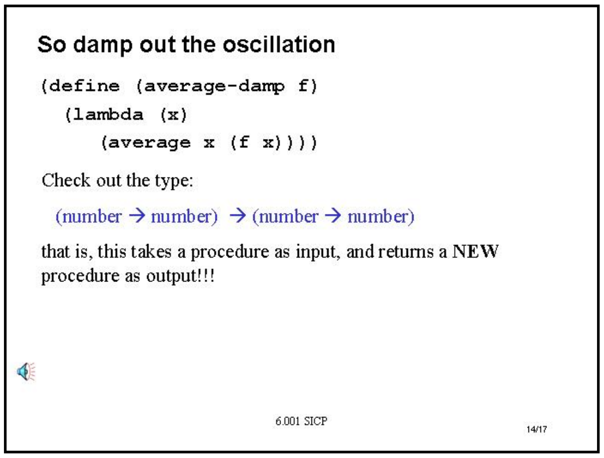
So I have a problem, but it is not a coding problem, rather it is a conceptual problem. Since I have an undamped oscillation here, I need to damp it down, similar to what you've seen in physics courses.
One nice way to damp down an oscillation is to take the function that is oscillating, and average it with its input. In other words, for any function **f**, we can get a new function that is the average of an input value **x** and **f** applied to that input value.
Here is a procedure that captures this idea, but notice its form. It takes as input a procedure that maps numbers to numbers, much like what we have seen before. But it produces as output a **procedure**! Thus, `average-damp` produces a function that can be applied to any input value.



So damp out the oscillation

```
(define (average-damp f)
  (lambda (x)
     (average x (f x))))
```

6.001 SICP 13/17



So damp out the oscillation

```
(define (average-damp f)
  (lambda (x)
     (average x (f x))))
```

Check out the type:

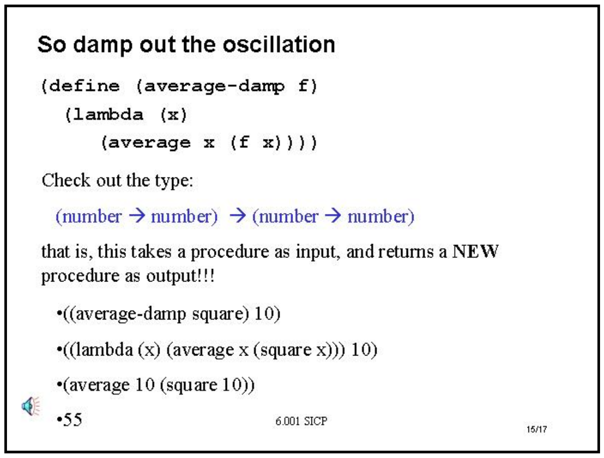
(number → number) → (number → number)

that is, this takes a procedure as input, and returns a **NEW** procedure as output!!!

6.001 SICP 14/17

Slide 6.5.14
So here is the type of this procedure, demonstrating that it is a new kind of higher order procedure.

Slide 6.5.15
Does this really work? Well, let's just try it using our substitution model. I am going to evaluate an average damp of square on the value 5. Notice that the first subexpression of this expression is itself a compound expression, something we haven't seen before. By but our type discussion, this should be right, since we expect that value of the subexpression to be a procedure. Indeed, if I substitute the value of square (which we know is a procedure but for convenience we will just use the primitive name here) into the body of average-damp in place of the formal parameter x, we just get the second expression. Notice how this body is a lambda expression, but this is just what we need, as evaluating that lambda expression will give us a procedure.
In fact, we know by the substitution model that this second expression reduces to the body of the lambda, with the value 5 substituted for the formal parameter, leading to the third expression, and that now is simply a procedure applied to two numbers (since we know that square maps numbers to numbers), giving us what we expect.
Thus, by our substitution model, this works quite nicely, as average damp takes a procedure as input and produces an appropriate procedure as output.



So damp out the oscillation

```
(define (average-damp f)
  (lambda (x)
     (average x (f x))))
```

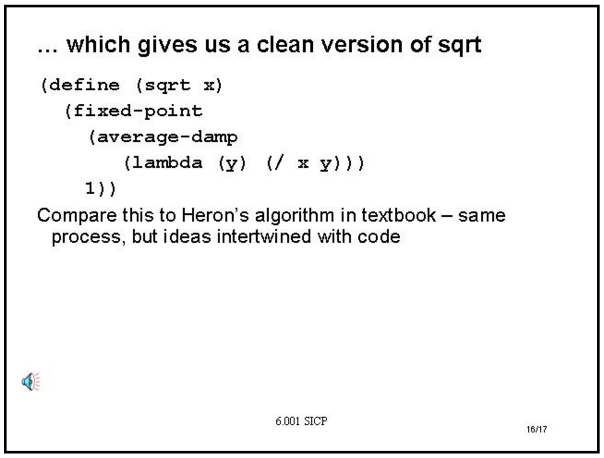
Check out the type:

(number → number) → (number → number)

that is, this takes a procedure as input, and returns a **NEW** procedure as output!!!

- ((average-damp square) 10)
- ((lambda (x) (average x (square x))) 10)
- (average 10 (square 10))
- 55

6.001 SICP 15/17



... which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x y)))
    1))
```

Compare this to Heron's algorithm in textbook – same process, but ideas intertwined with code

6.001 SICP 16/17

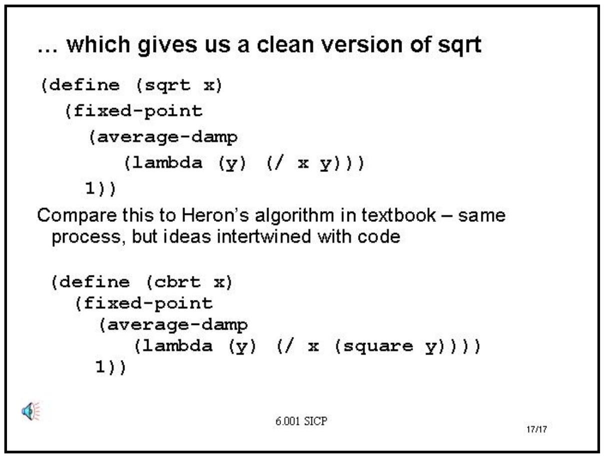
Slide 6.5.16
And this just brings us back to square roots. We can fix our problem by finding the fixed point of the average damp of our original procedure.
The key things to notice are how **average-damp** cleanly isolates the problem of damping from the problem of fixed points, and how **average-damp** nicely satisfies the type contract for fixed point, that is, it takes a procedure as input, and produces a procedure as output, exactly forming the type of input needed by **fixed-point**.
Why is this useful? Well compare this code to the algorithm in the textbook (or the one we talked about earlier). This is exactly the same process, but now the ideas underlying the process have been cleanly separated into appropriate computational pieces, a fixed point process and a damping process.

Slide 6.5.17

... and why is this relevant? Well, suppose I also want to compute cube roots. Given the separation I have made, this is easy. I just compute a fixed point of an average damp of a different function.

On the other hand, if I were to go back to Heron of Alexandria's algorithm for square roots, I would have to change a lot of different things in the code, and this leaves lots of opportunity for coding errors.

Thus, these higher order procedures nicely capture patterns of computation, while suppressing unnecessary detail. This makes it easier for the programmer to focus on the problem of interest, without getting lost in the details of the code itself.



... which gives us a clean version of sqrt

```
(define (sqrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x y)))
    1))
```

Compare this to Heron's algorithm in textbook – same process, but ideas intertwined with code

```
(define (cbrt x)
  (fixed-point
    (average-damp
       (lambda (y) (/ x (square y))))
    1))
```

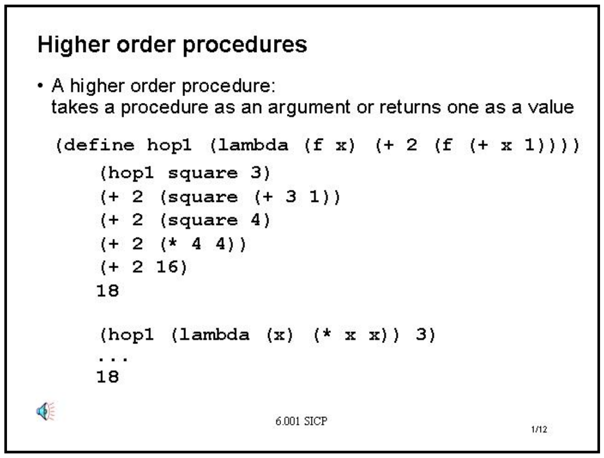
6.001 SICP 17/17

6.001 Notes: Section 6.6

Slide 6.6.1

So we have introduced an interesting new concept in this lecture, the idea of a higher order procedure. Our goal was to show how we could capture common patterns in a procedure, where those patterns might be over numbers, or over procedures or processes themselves. By doing this, we isolate the computational pattern or process from the details of the process itself, thus making it easier to create new computational processes.

Let's revisit this idea of higher order procedures again, to reinforce the main elements. Remember that a higher order procedure is a procedure that either takes procedures as input, or produces a procedure as output.



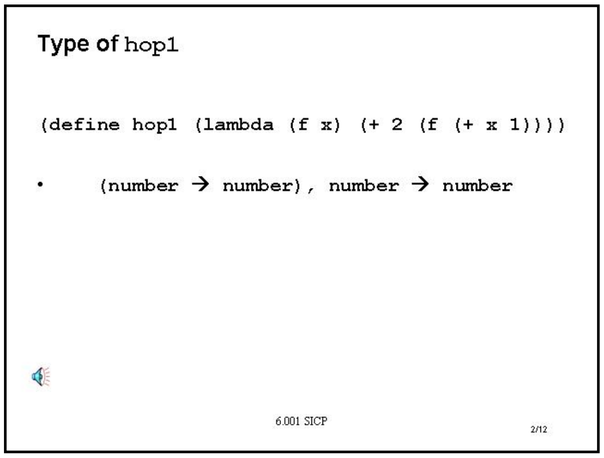
Higher order procedures

- A higher order procedure: takes a procedure as an argument or returns one as a value

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
    (hop1 square 3)
    (+ 2 (square (+ 3 1))
    (+ 2 (square 4)
    (+ 2 (* 4 4))
    (+ 2 16)
    18

    (hop1 (lambda (x) (* x x)) 3)
    ...
    18
```

6.001 SICP 1/12

Here is a simple example. Note that by the rules of evaluation, we know that **f** must be a procedure that maps numbers to numbers, given how it is used in the body of the **lambda**. And we also know that we can apply **hop1** to names for procedures or to pure **lambda** expressions that return procedures as values. Our substitution model demonstrates that this defines a correct process.



Type of `hop1`

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
```

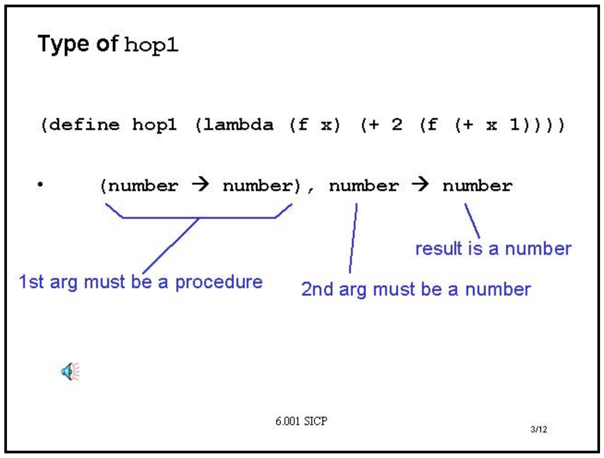
- (number → number), number → number

6.001 SICP 2/12

Slide 6.6.2

And what is the type of **hop1**? By looking at where **x** is used in the body, we know it must be a number, since addition is applied to it. We also know that **f** must be a procedure since it is used as the first subexpression of a combination. Moreover, it must be a procedure of one argument, a number, since addition produces a number, and it must return a number, since its value will be used by addition again. And finally, the whole procedure we know will produce a number as output, since the result of the outermost addition operation is the value of the body.

Slide 6.6.3
So here is the encapsulation of that analysis.



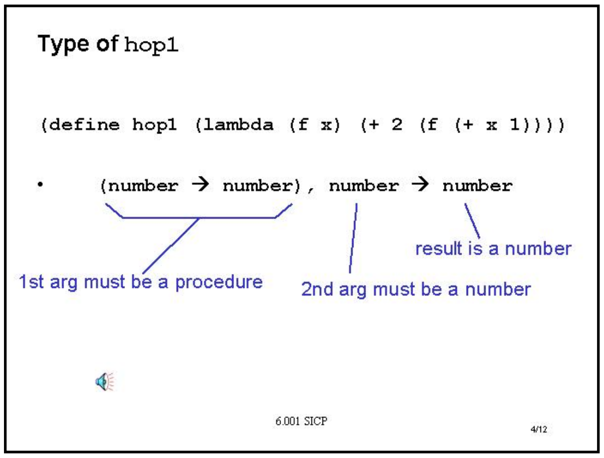
Type of hop1

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
```

• (number → number), number → number

1st arg must be a procedure
2nd arg must be a number
result is a number
6.001 SICP
3/12

Slide 6.6.4
And this nicely demonstrates several key points. First, we see how reasoning about types of objects allows us to deduce the values of expressions, and thus how we can use that analysis to help us design procedures to capture particular patterns. Second, we see how capture common patterns, now across procedures rather than just numbers, allows us to treat complex things as primitives, suppressing detail and focusing on the computation itself.



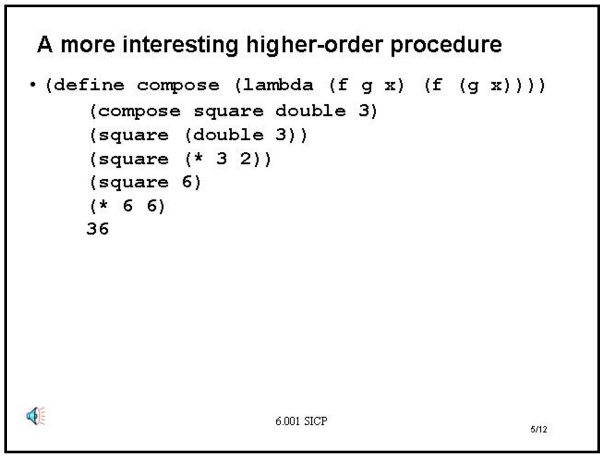
Type of hop1

```
(define hop1 (lambda (f x) (+ 2 (f (+ x 1)))))
```

• (number → number), number → number

1st arg must be a procedure
2nd arg must be a number
result is a number
6.001 SICP
4/12

Slide 6.6.5
Here is a more interesting higher order procedure. It takes as input two procedures, and applies the first procedure to the result of applying the second procedure to some argument. Notice how the trace of the substitution model nicely demonstrates the evolution of the process captured by **compose**.

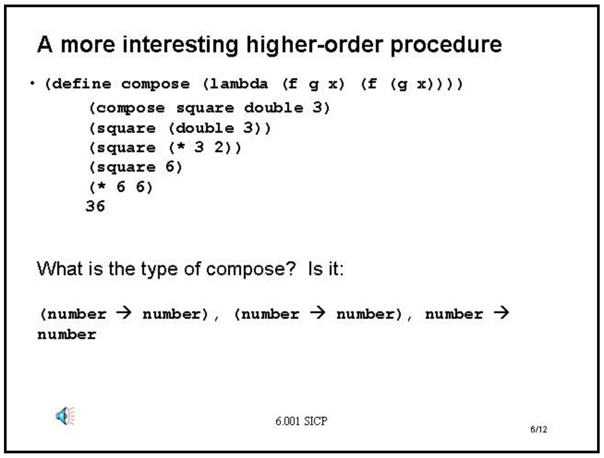


A more interesting higher-order procedure

```
• (define compose (lambda (f g x) (f (g x))))
     (compose square double 3)
     (square (double 3))
     (square (* 3 2))
     (square 6)
     (* 6 6)
     36
```

6.001 SICP
5/12

Slide 6.6.6
So what is the type of **compose**? By our example, it would seem that it should be of the form shown here.



A more interesting higher-order procedure

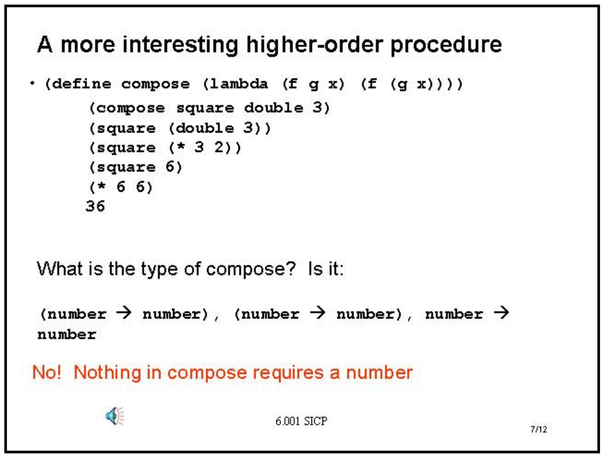
```
• (define compose (lambda (f g x) (f (g x))))
     (compose square double 3)
     (square (double 3))
     (square (* 3 2))
     (square 6)
     (* 6 6)
     36
```

What is the type of compose? Is it:

(number → number), (number → number), number → number

6.001 SICP
6/12

Slide 6.6.7
But wait! Is there anything that says compose requires a number? In fact, if we look at the body of **compose** we see that nowhere is there a procedure application that shows that **x** must be a number, and thus, nothing says that **f** or **g** have to be procedures on numbers either! Thus, **compose** is much more general.



A more interesting higher-order procedure

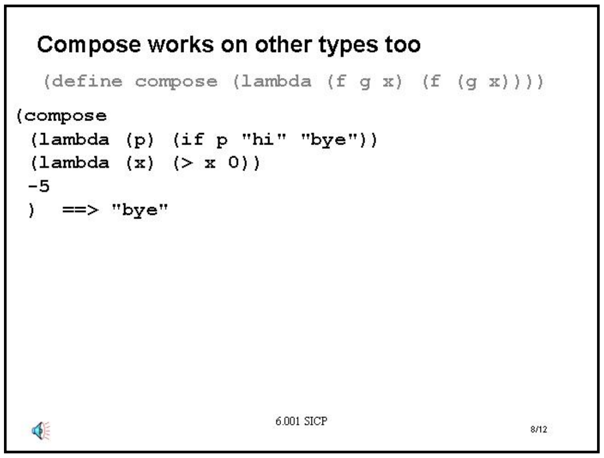
```
• (define compose (lambda (f g x) (f (g x))))
    (compose square double 3)
    (square (double 3))
    (square (* 3 2))
    (square 6)
    (* 6 6)
    36
```

What is the type of compose? Is it:

(number → number), (number → number), number → number

No! Nothing in compose requires a number

6.001 SICP 7/12



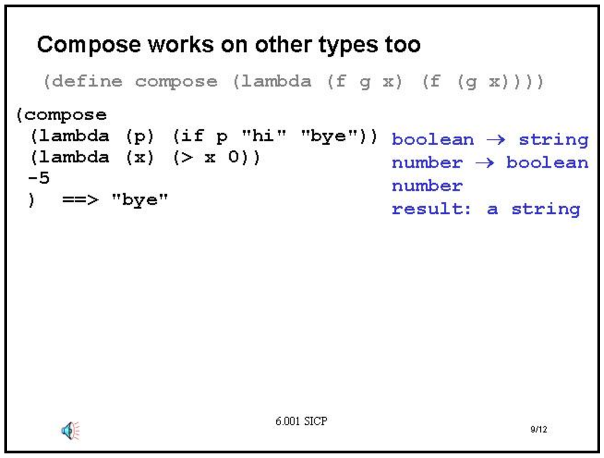
Compose works on other types too

```
 (define compose (lambda (f g x) (f (g x))))
(compose
 (lambda (p) (if p "hi" "bye"))
 (lambda (x) (> x 0))
 -5
 ) ==> "bye"
```

6.001 SICP 8/12

Slide 6.6.8
In fact, **compose** works perfectly well on other kinds of procedures as shown here.

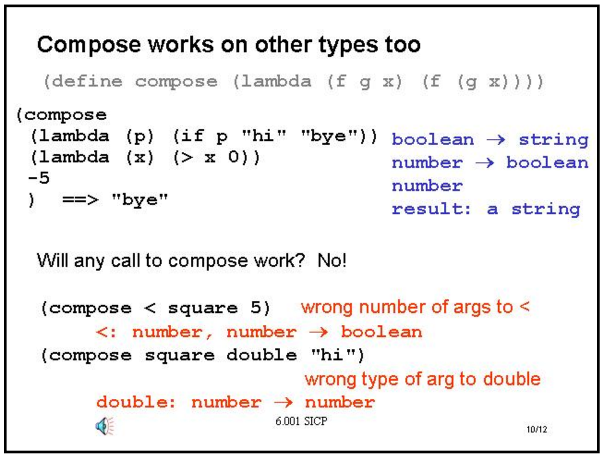
Slide 6.6.9
I can use types to help me reason through why this works. I am going to apply the second procedure to the third argument as the innermost expression in the body of compose. Since the third argument is a number, I need the second argument to be a procedure that applies to numbers, as this one does. Now, this particular procedure produces a boolean as output, so I need the first argument to be a procedure that takes booleans as input. This one does, and returns a string as output, thus I can see that the output of the whole expression will be a string as well.



Compose works on other types too

```
 (define compose (lambda (f g x) (f (g x))))
(compose
 (lambda (p) (if p "hi" "bye"))   boolean → string
 (lambda (x) (> x 0))             number → boolean
 -5                               number
 ) ==> "bye"                      result: a string
```

6.001 SICP 9/12



Compose works on other types too

```
 (define compose (lambda (f g x) (f (g x))))
(compose
 (lambda (p) (if p "hi" "bye"))   boolean → string
 (lambda (x) (> x 0))             number → boolean
 -5                               number
 ) ==> "bye"                      result: a string
```

Will any call to compose work? No!

```
(compose < square 5)   wrong number of args to <
     <: number, number → boolean
(compose square double "hi")
                       wrong type of arg to double
     double: number → number
```

6.001 SICP 10/12

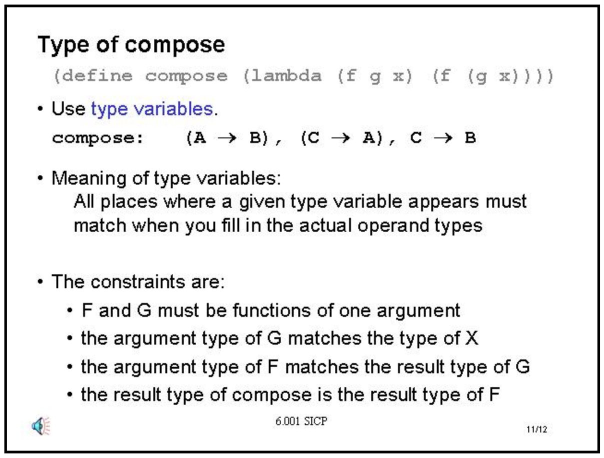
Slide 6.6.10
Now, while compose is more general than just procedures on numbers, it doesn't work on everything. Here are two examples that fail. In the first case, the parameter **f** will be applied to the wrong number of arguments, and in the second case, we try to apply the parameter **g** to the wrong kind of argument.
So how do we use types to figure out the right way to use compose?

Slide 6.6.11

Well, let's go back to our earlier analysis, but be a bit more general. We know that the parameter **x** can be of any type, so rather than declaring a specific type, we'll just use a variable to represent that type, call it C.

Then we can see that the parameter **g** must be a procedure of one argument, whose type must match that of its input, hence must be of type C again. This procedure can return any type as its output, so let's call that type A, thus the second parameter of **compose** must be a mapping of type C to type A.

By the same reasoning, we get the type of the first parameter of **compose**. And since the result of compose is the result of that parameter, the output type of **compose** must match the output type of **f**.



Type of compose

```
(define compose (lambda (f g x) (f (g x))))
```

- Use type variables.

```
compose:   (A → B), (C → A), C → B
```

- Meaning of type variables:
  All places where a given type variable appears must match when you fill in the actual operand types

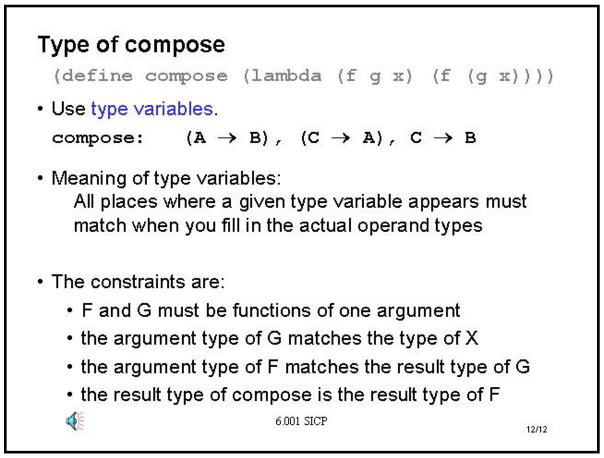
- The constraints are:
  - F and G must be functions of one argument
  - the argument type of G matches the type of X
  - the argument type of F matches the result type of G
  - the result type of compose is the result type of F

6.001 SICP 11/12

Slide 6.6.12

To summarize, we have looked at the idea of capturing common patterns, where those patterns may over operations on numbers, or more generally over operations on operations themselves. This led to the idea of higher order procedures as a powerful tool for capturing complex processes and treating them as primitive building blocks.

We also saw that by using types to reason about objects and about procedures applied to objects, we can analyze the evolution of a process, deciding what kinds of inputs are required for each stage of the computation.



Type of compose

```
(define compose (lambda (f g x) (f (g x))))
```

- Use type variables.

```
compose:   (A → B), (C → A), C → B
```

- Meaning of type variables:
  All places where a given type variable appears must match when you fill in the actual operand types

- The constraints are:
  - F and G must be functions of one argument
  - the argument type of G matches the type of X
  - the argument type of F matches the result type of G
  - the result type of compose is the result type of F

6.001 SICP 12/12