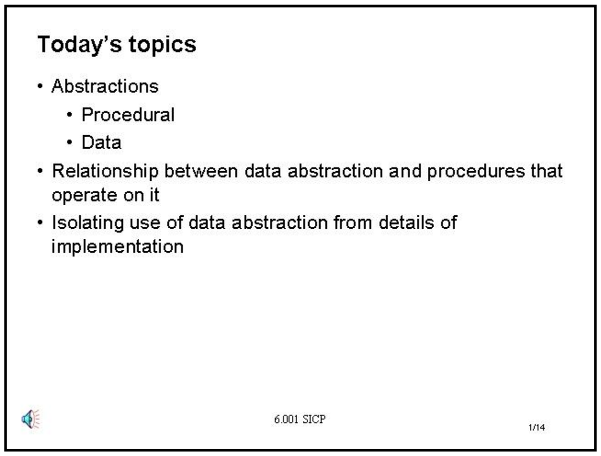
6.001 Notes: Section 5.1

Slide 5.1.1
In this lecture we are going to continue with the theme of building abstractions. Thus far, we have focused entirely on procedural abstractions: the idea of capturing a common pattern of computation within a procedure, and isolating the details of that computation from the use of the concept within some other computation. Today we are going to turn to a complementary issue, namely how to group together pieces of information, or data, into abstract structures. We will see that the same general theme holds: we can isolate the details of how the data are glued together from the use of the aggregate data structure as a primitive element in some computation. We will also see that the procedures we use to manipulate the elements of a data structure often have an inherent structure that mimics the data structure, and we will use this idea to help us design our data abstractions and their associated procedures.

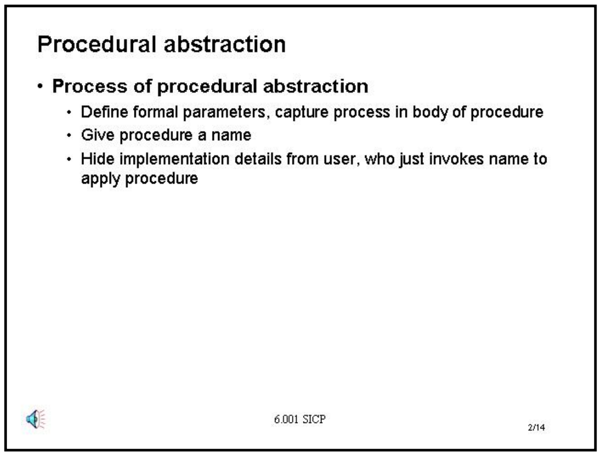


Today's topics

- Abstractions
  - Procedural
  - Data
- Relationship between data abstraction and procedures that operate on it
- Isolating use of data abstraction from details of implementation

6.001 SICP 1/14

Slide 5.1.2
Let's review what we have been looking at so far in the course, in particular, the idea of procedural abstraction to capture computations. Our idea is to take a common pattern of computation, then capture that pattern by formalizing it with a set of parameters that specify the parts of the pattern that change, while preserving the pattern inside the body of a procedure. This encapsulates the computation associated with the pattern inside a lambda object. Once we have abstracted that computation inside the lambda, we can then give it a name, using our `define` expression, then treat the whole thing as a primitive by just referring to the name, and use it without worrying about the details within the lambda.

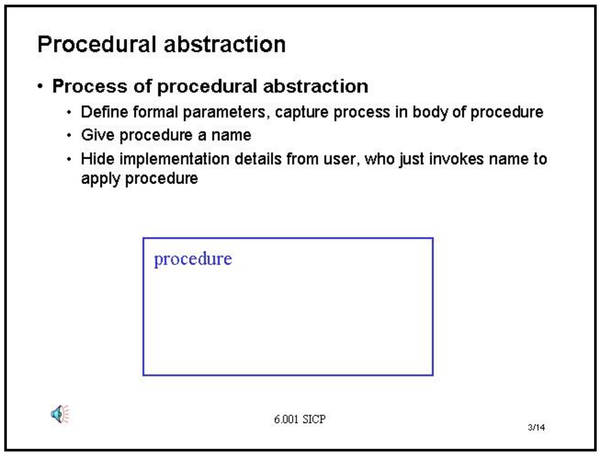


Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

6.001 SICP 2/14

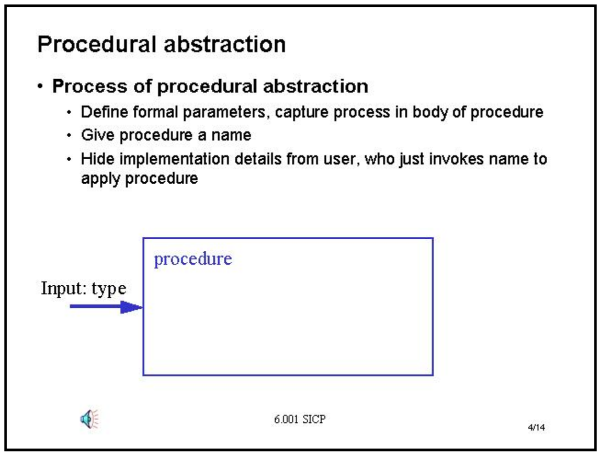
Slide 5.1.3
This means we can treat the procedure as it if is a kind of **black box**.



Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

6.001 SICP 3/14



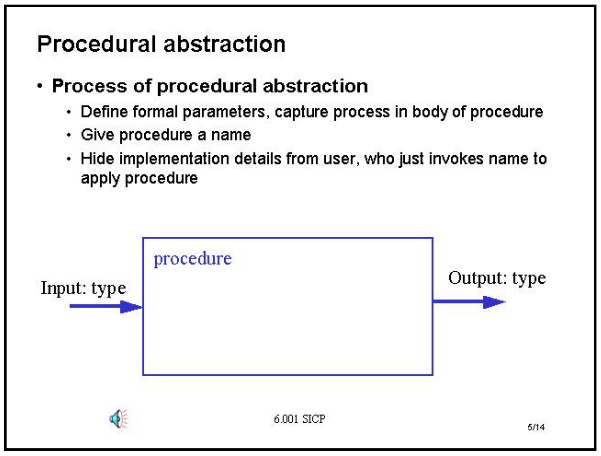
Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

procedure
Input: type

Slide 5.1.4
We need to provide it with inputs of a specified type.

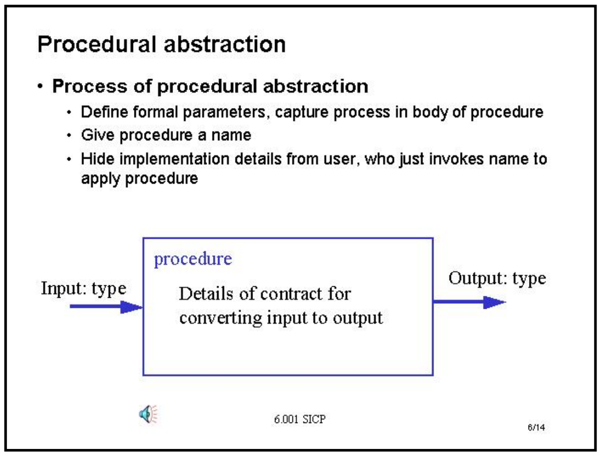
Slide 5.1.5
We know by the contract associated with the procedure, that if we provide inputs of the appropriate type, the procedure will produce an output of a specified type...



Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

procedure
Input: type
Output: type



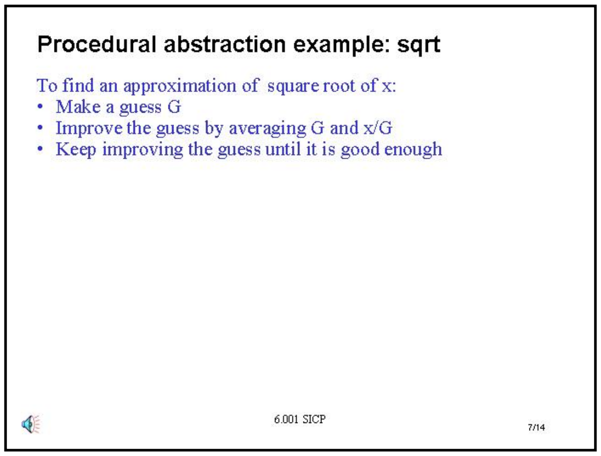
Procedural abstraction

- **Process of procedural abstraction**
  - Define formal parameters, capture process in body of procedure
  - Give procedure a name
  - Hide implementation details from user, who just invokes name to apply procedure

procedure
Details of contract for converting input to output
Input: type
Output: type

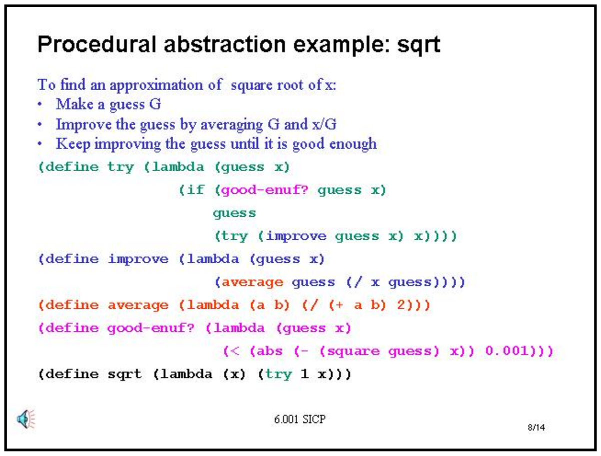
Slide 5.1.6
... and by giving the whole procedure a name, we create this black box abstraction, in which we can use the procedure without knowing details. This means that the procedure will obey the contract that specifies the mapping from inputs to outputs, but the user is not aware of the details by which that contract is enforced.

Slide 5.1.7
So let's use this idea to look at a more interesting algorithm than the earlier ones we've examined. Here, again, is Heron of Alexandria's algorithm for computing good approximations to the square root of a positive number. Read the steps carefully, as we are about to implement them.



Procedural abstraction example: sqrt

To find an approximation of square root of x:
- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough



Procedural abstraction example: sqrt

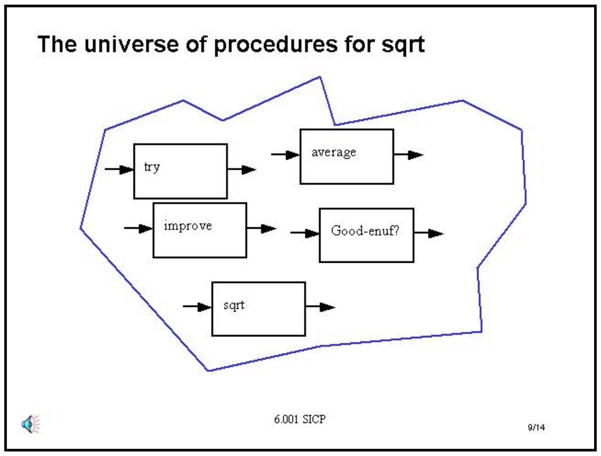
To find an approximation of square root of x:
- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough

```
(define try (lambda (guess x)
                  (if (good-enuf? guess x)
                      guess
                      (try (improve guess x) x))))
(define improve (lambda (guess x)
                      (average guess (/ x guess))))
(define average (lambda (a b) (/ (+ a b) 2)))
(define good-enuf? (lambda (guess x)
                       (< (abs (- (square guess) x)) 0.001)))
(define sqrt (lambda (x) (try 1 x)))
```

6.001 SICP 8/14

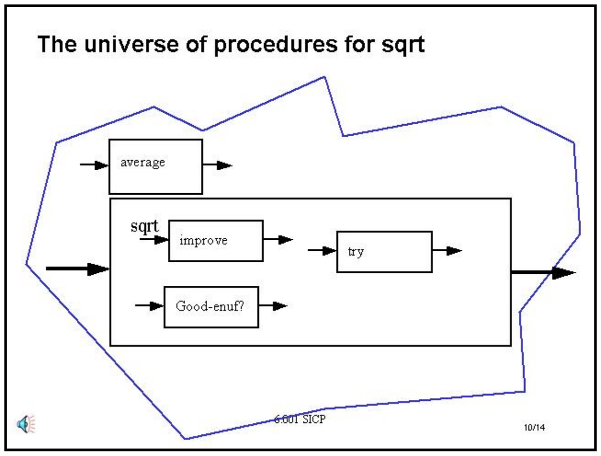
Slide 5.1.8
Now, let's use the tools we seen so far to implement this method. Notice how the first procedure uses the ideas of wishful thinking, and recursive procedures to capture the basic idea of Heron's method. `Try` is a procedure that takes a current guess and the `x`, and captures the top-level idea of the method. It checks to see if the guess is sufficient. If it is, it simply returns the value of that guess. If it is not, then it tries again, with a new guess.
Note how we are using wishful thinking to reduce the problem to another version of the same problem, and to abstract out the idea of both getting a new guess and checking for how good the guess is. These are procedures we can subsequently write, for example, as shown. Finally, notice how the recursive call to `try` will use a different argument for guess, since we will evaluate the expression before substituting into the body.
Also notice the recursive structure of `try` and the use of the special form `if` to control the evolution of this procedure.
The method for `improve` simply incorporates the ideas from the algorithm, again with a procedure abstraction to separate out idea of averaging from the procedure for improving the guess.
Finally, notice how we can build a square root procedure on top of the procedure for `try`.

Slide 5.1.9
If we think of each of these procedures as its own black box abstraction, then we can visualize the universe containing these procedures as shown. Each procedure exists with its own contract, but each is accessible to the user, simply by referring to it by name.
While this sounds fine in principle, there is a problem with this viewpoint. Some of these procedures are general methods, such as `average` and `sqrt`, and should be accessible to the user, who might utilize them elsewhere. Some of them, however, such as `try` or `good-enuf?`, are really specific to the computation for square roots. Ideally we would like to capture those procedures in a way that they can only be used by `sqrt` but not by other methods.



The universe of procedures for sqrt

6.001 SICP 9/14

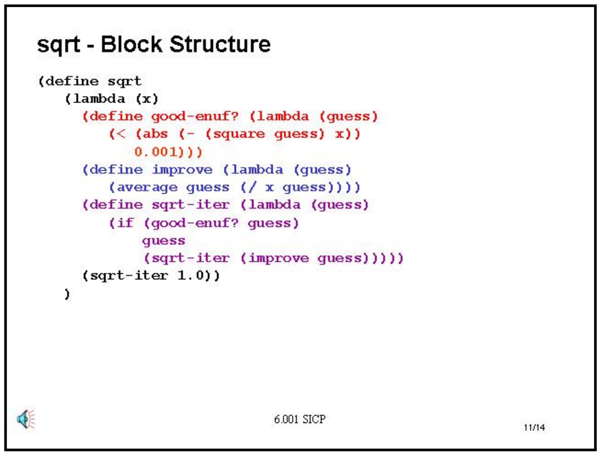


The universe of procedures for sqrt

6.001 SICP 10/14

Slide 5.1.10
Abstractly, this is what we would like to do. We would like to move the abstractions for the special purpose procedures inside of the abstraction for `sqrt` so that only it can use them, while leaving more generally useful procedures available to the user. In this way, these internal procedures should become part of the implementation details for `sqrt` but be invisible to outside users.

Slide 5.1.11



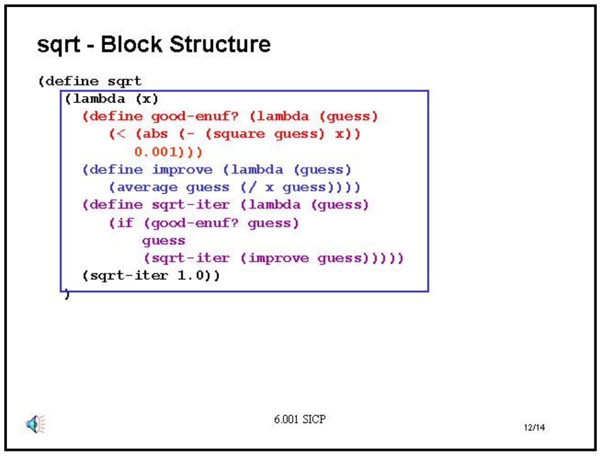
sqrt - Block Structure

```
(define sqrt
   (lambda (x)
     (define good-enuf? (lambda (guess)
        (< (abs (- (square guess) x))
           0.001)))
     (define improve (lambda (guess)
        (average guess (/ x guess))))
     (define sqrt-iter (lambda (guess)
        (if (good-enuf? guess)
            guess
            (sqrt-iter (improve guess)))))
     (sqrt-iter 1.0))
   )
```

6.001 SICP 11/14

And here is how to do this.
Note that the definition of `sqrt` bind this name to a **lambda**.
Within the bounds of that **lambda** we have moved the definitions for `improve`, `good-enuf?`, and `sqrt-iter` (which is what we have renamed `try`). By moving these procedures inside the body of the lambda, they become internal procedures, accessible only to other expressions within the body of that lambda. That is, if we try to refer to one of these names when interacting with the evaluator, we will get an **unbound variable** error. But these names can be referenced by expressions that exist within the scope of this **lambda**.
The rules of evaluation say that when we apply `sqrt` to some argument, the body of this **lambda** will be evaluated. At that point, the internal definitions are evaluated.
The final expression of the **lambda** is the expression `(sqrt-iter 1.0)` which means when `sqrt` is applied to some argument, by the substitution model it will reduce to evaluating this expression, meaning it will begin the recursive evaluation of guesses for the square root.

Slide 5.1.12



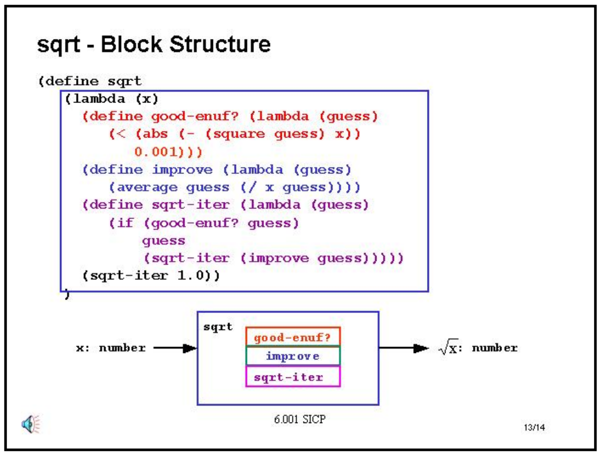
sqrt - Block Structure

```
(define sqrt
   (lambda (x)
     (define good-enuf? (lambda (guess)
        (< (abs (- (square guess) x))
           0.001)))
     (define improve (lambda (guess)
        (average guess (/ x guess))))
     (define sqrt-iter (lambda (guess)
        (if (good-enuf? guess)
            guess
            (sqrt-iter (improve guess)))))
     (sqrt-iter 1.0))
   )
```

6.001 SICP 12/14

In fact we can stress this by drawing a box around the boundary of the outermost lambda. Clearly that boundary exactly scopes the black box abstraction that I wanted.
This is called **block structure**, which you can find, discussed in more detail in the textbook.

Slide 5.1.13

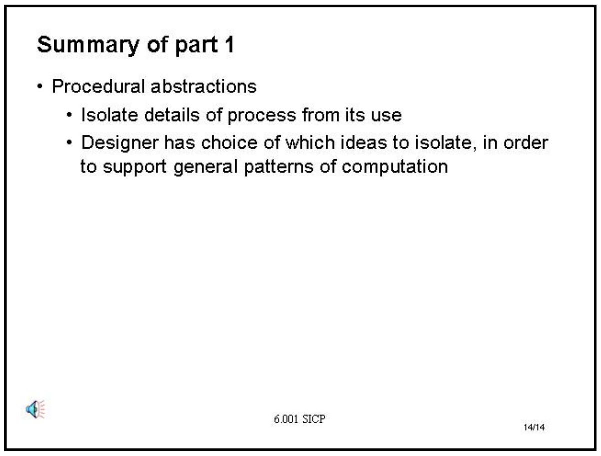


sqrt - Block Structure

```
(define sqrt
   (lambda (x)
     (define good-enuf? (lambda (guess)
        (< (abs (- (square guess) x))
           0.001)))
     (define improve (lambda (guess)
        (average guess (/ x guess))))
     (define sqrt-iter (lambda (guess)
        (if (good-enuf? guess)
            guess
            (sqrt-iter (improve guess)))))
     (sqrt-iter 1.0))
   )
```

6.001 SICP 13/14

Schematically, this means that `sqrt` contains within it only those internal procedures that belong to it, and behaves according to the contract expected by the user, without the user knowing how those procedures accomplish this contract.
This provides another method for abstracting ideas and isolating them from other abstractions.



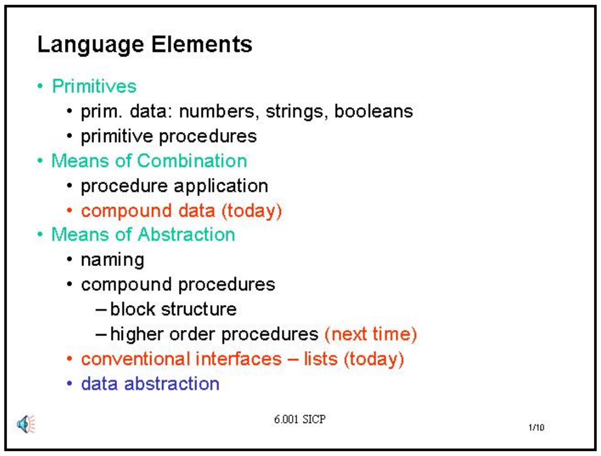
Summary of part 1

- Procedural abstractions
  - Isolate details of process from its use
  - Designer has choice of which ideas to isolate, in order to support general patterns of computation

6.001 SICP 14/14

Slide 5.1.14
So here is the summary of what we have seen in this section.

6.001 Notes: Section 5.2



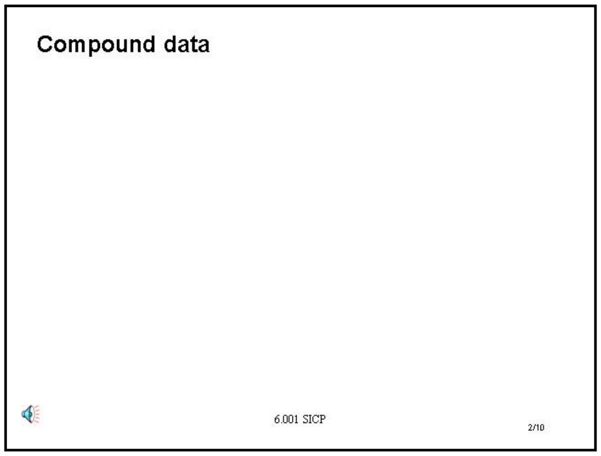
Language Elements

- Primitives
  - prim. data: numbers, strings, booleans
  - primitive procedures
- Means of Combination
  - procedure application
  - compound data (today)
- Means of Abstraction
  - naming
  - compound procedures
    - block structure
    - higher order procedures (next time)
  - conventional interfaces – lists (today)
  - data abstraction

6.001 SICP 1/10

Slide 5.2.1
So let's take that idea of abstraction and build on it. To set the stage for what we are about to do, it is useful to think about how the language elements can be group together into a hierarchy. At the atomic level, we have a set of **primitives**. In Scheme, these include primitive data objects: numbers, strings and Booleans. And these include built-in, or primitive, procedures: for numbers, things like *, +, =, >; for strings, things like string=?, substring; for Booleans, things like and, or, not. To put these primitive elements together into more interesting expressions, we have a **means of combination,** that is, a way of combining simpler pieces into expressions that can themselves be treated as elements of other expressions. The most common one, and the one we have seen in the previous lectures, is procedure application. This is the idea of creating a combination of subexpressions, nested within a pair of parentheses: the value of the first subexpression is a procedure, and the expression captures the idea of applying that procedure to the values of the other expressions, which means as we have seen that we substitute the values of the arguments for the corresponding parameters in the body of the procedure, and proceed with the evaluation. We know that these combinations can themselves be included within other combinations, and the same rules of evaluation will recursively govern the computation. Finally, our language has a **means of abstraction**: a way of capturing computational elements and treating them as if they were primitives; or said another way, a method of isolating the details of a computation from the use of a computation. Our first means of abstraction was **define**, the ability to give a name to an element, so that we could just use the name, thereby suppressing the details from the use of the object. This ability to give a name to something is most valuable when used with our second means of abstraction, capturing a computation within a procedure. This means of abstraction dealt with the idea that a common pattern of computation can be generalized into a single procedure, which covered every possible application of that idea to an appropriate value. When coupled with the ability to give a name to that procedure, we engendered the ability to create an important cycle in our language: we can now create procedures, name them, and thus treat them as if they were themselves primitive elements of the language. The whole goal of a high-level language is to allow us to suppress unnecessary detail in this manner, while focusing on the use of a procedural abstraction to support some more complex computational design.
Today, we are going to generalize the idea of abstractions to include those that focus on data, rather than procedures. So we are going talk about how to create compound data objects, and we are going to examine standard

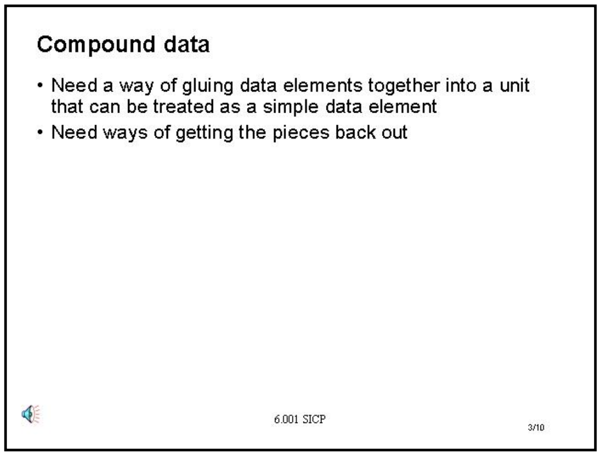
procedures associated with the manipulation of those data structures. We will see that data abstractions mirror many of the properties of procedural abstractions, and we will thus generalize the ideas of compound data into data abstractions, to complement our procedural abstractions.


Compound data

6.001 SICP 2/10

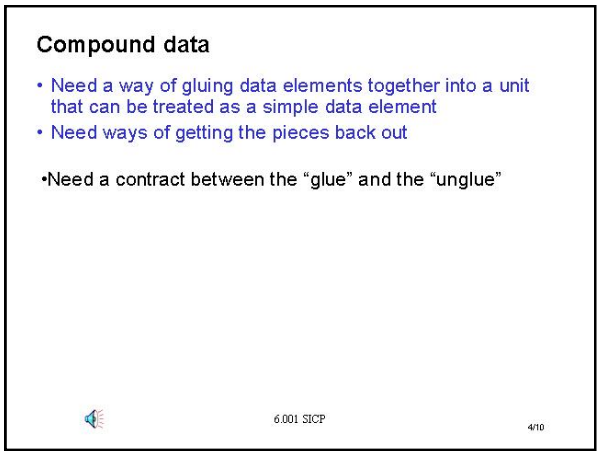
Slide 5.2.2
So far almost everything we've seen in Scheme has revolved around numbers and computations associated with numbers. This has been partly deliberate on our part, because we wanted to focus on the ideas of procedural abstraction, without getting bogged down in other details. There are, however, clearly problems in which it is easier to think in terms of other elements than just numbers, and in which those elements have pieces that need to be glued together and pulled apart, while preserving the concept of the larger unit.

Slide 5.2.3
So our goal is to create a method for taking primitive data elements, gluing them together, and then treating the result as if it were itself a primitive element. Of course, we will need a way of "de-gluing" the units, to get back the constituent parts. What do we mean when we say we want to treat the result of gluing elements together as a primitive data element? Basically we want the same properties we had with numbers: we can apply procedures to them, we can use procedures to generate new versions of them, and we can create expressions that include them as simpler elements.


Compound data

- Need a way of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of getting the pieces back out

6.001 SICP 3/10


Compound data

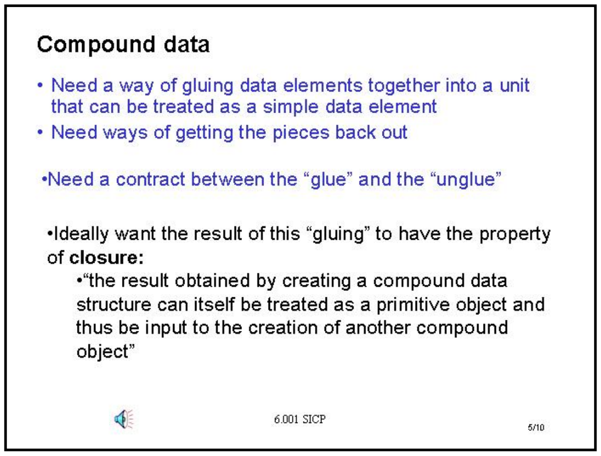
- Need a way of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of getting the pieces back out

•Need a contract between the “glue” and the “unglue”

6.001 SICP 4/10

Slide 5.2.4
The most important point when we "glue" things together is to have a contract associated with that process. This means that we don't really care that much about the details of how we glue things together, so long as we have a means of getting back out the pieces when needed. This means that the "glue" and the "unglue" work hand in hand, guaranteeing that however, the compound unit is created, we can always get back the parts we started with.

Slide 5.2.5
And ideally we would like the process of gluing things together to have the property of **closure**, that is, that whatever we get by gluing things into a compound structure can be treated as a primitive so that it can be the input to another gluing operation. Not all ways of creating compound data have this property, but the best of them do, and we say they are **closed** under the operation of creating a compound object if the result can itself be a primitive for the same compound data construction process.



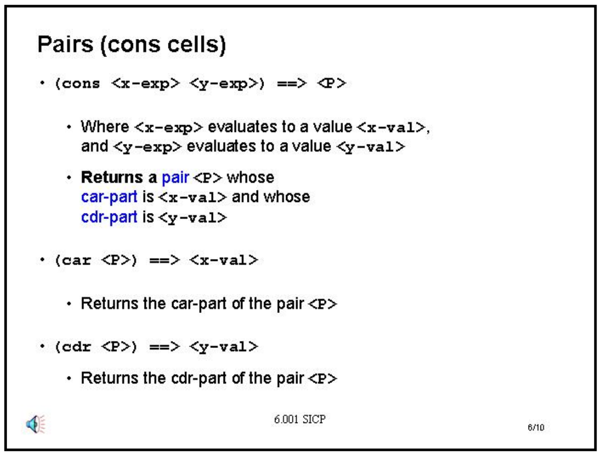
Compound data

- Need a way of gluing data elements together into a unit that can be treated as a simple data element
- Need ways of getting the pieces back out

•Need a contract between the "glue" and the "unglue"

•Ideally want the result of this "gluing" to have the property of **closure:**
  •"the result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object"

6.001 SICP 5/10



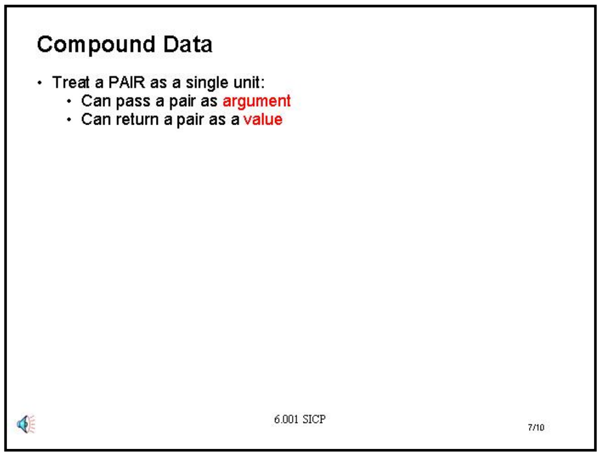
Pairs (cons cells)

- `(cons <x-exp> <y-exp>) ==> <P>`
  - Where `<x-exp>` evaluates to a value `<x-val>`, and `<y-exp>` evaluates to a value `<y-val>`
  - **Returns** a pair `<P>` whose car-part is `<x-val>` and whose cdr-part is `<y-val>`
- `(car <P>) ==> <x-val>`
  - Returns the car-part of the pair `<P>`
- `(cdr <P>) ==> <y-val>`
  - Returns the cdr-part of the pair `<P>`

6.001 SICP 6/10

Slide 5.2.6
Scheme's basic means for gluing things together is called `cons`, short for **constructor**, and virtually all other methods for creating compound data objects are based on `cons`. `Cons` is a procedure that takes two expressions as input. It evaluates each in turn, and then glues these values together into something called a **pair**. Note that the actual pair object is the value returned by evaluating the `cons`. The two parts of a cons pair are called the **car** and the **cdr**, and if we apply the procedures of those names to a pair, we get back the value of the argument that was evaluated when the pair was created.
Note that there is a contract here between **cons**, **car** and **cdr**, in which **cons** glues things together in some arbitrary manner, and all that matters is that when **car**, for example, is applied to that object, it gets back out what we started with.

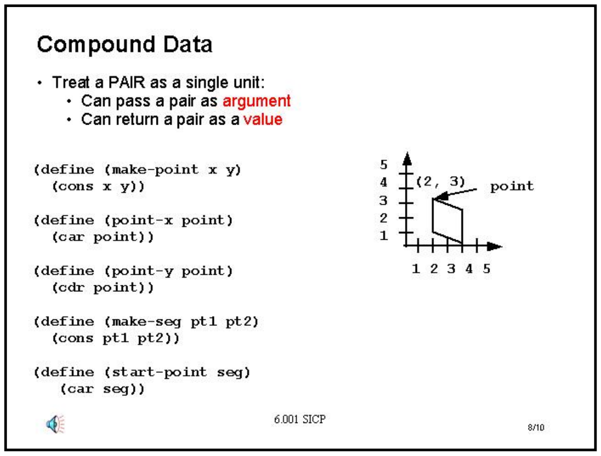
Slide 5.2.7
Note that we can treat a pair as a unit, that is, having built a pair, we can treat it as a primitive and use it anywhere we might use any other primitive. So we can pass a pair in as input to some other data abstraction, such as another pair.



Compound Data

- Treat a PAIR as a single unit:
  - Can pass a pair as argument
  - Can return a pair as a value

6.001 SICP 7/10



Compound Data

- Treat a PAIR as a single unit:
  - Can pass a pair as argument
  - Can return a pair as a value

```
(define (make-point x y)
  (cons x y))

(define (point-x point)
  (car point))

(define (point-y point)
  (cdr point))

(define (make-seg pt1 pt2)
  (cons pt1 pt2))

(define (start-point seg)
  (car seg))
```

(2, 3)
point
1 2 3 4 5
5 4 3 2 1

6.001 SICP 8/10

Slide 5.2.8

In this way, we can create elements that are naturally thought of as simple units that happen to themselves be created out of elements that are also thought of as simple units. This allows us to build up levels of hierarchy of data abstractions.

For example, suppose we want to build a system to reason about figures drawn in the plane. Those figures might be built out of line segments that have start and end points, and those points are built out of **x** and **y** coordinates.

Notice how there is a contract between **make-point** and **point-x** or **point-y**, in which the selectors get out the pieces that are glued together by the constructor. Because they are built on top of **cons, car** and **cdr**, they inherit the same contract that holds there. And in the case of segments, these pieces are glued together as if they are primitives, so that we have **cons** pairs whose elements are also **cons** pairs.

Thus we see how **cons** pairs have the property of closure, in which the result of **consing** can be treated as primitive input to another level of abstraction.

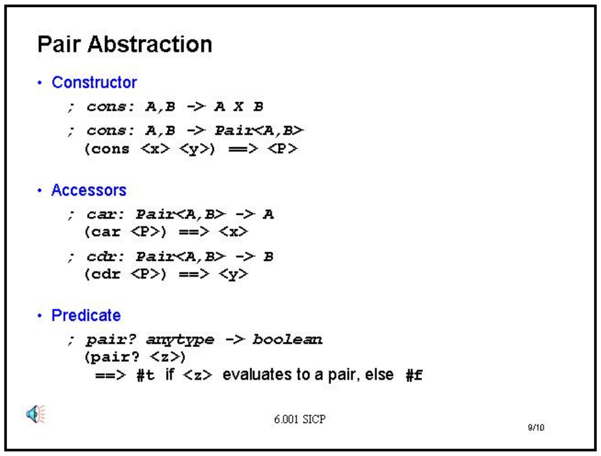
Slide 5.2.9

We can formalize what we have just seen, in terms of the abstraction of a **pair**. This abstraction has several standard parts. First, it has a **constructor**, for making instances of this abstraction. The constructor has a kind of contract, in which objects A and B are glued together to construct a new object, called a Pair, with two pieces inside.

Second, it has some **selectors** or **accessors** to get the pieces back out. Notice how the contract specifies the interaction between the constructor and the selectors, whatever is put together can be pulled back apart using the appropriate selector.

Typically, a data abstraction will also have a **predicate**, here called `pair?`. Its role is to take in any object, and return `true` if the object is of type **pair**. This allows us to test objects for their type, so that we know whether to apply particular selectors to that object.

The key issue here is the contract between the constructor and the selectors. The details of how a constructor puts things together are not at issue, so long as however the pieces are glued together, they can be separated back out into the original parts by the selectors.



Pair Abstraction

- Constructor

```
  ; cons: A,B -> A X B
  ; cons: A,B -> Pair<A,B>
  (cons <x> <y>) ==> <P>
```

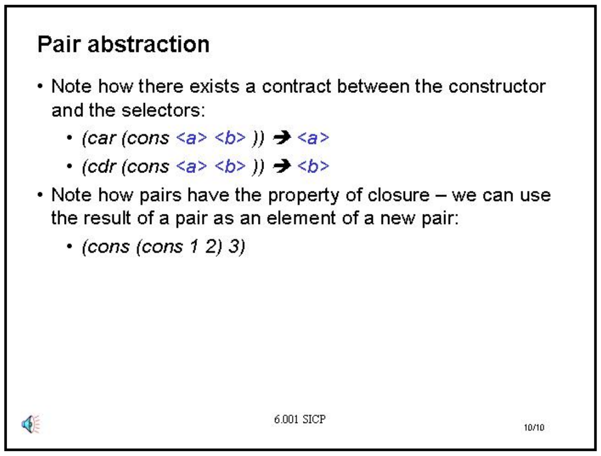
- Accessors

```
  ; car: Pair<A,B> -> A
  (car <P>) ==> <x>
  ; cdr: Pair<A,B> -> B
  (cdr <P>) ==> <y>
```

- Predicate

```
  ; pair? anytype -> boolean
  (pair? <z>)
   ==> #t if <z> evaluates to a pair, else #f
```

6.001 SICP 9/10



Pair abstraction

- Note how there exists a contract between the constructor and the selectors:
  - *(car (cons <a> <b>)) ➔ <a>*
  - *(cdr (cons <a> <b>)) ➔ <b>*
- Note how pairs have the property of closure – we can use the result of a pair as an element of a new pair:
  - *(cons (cons 1 2) 3)*

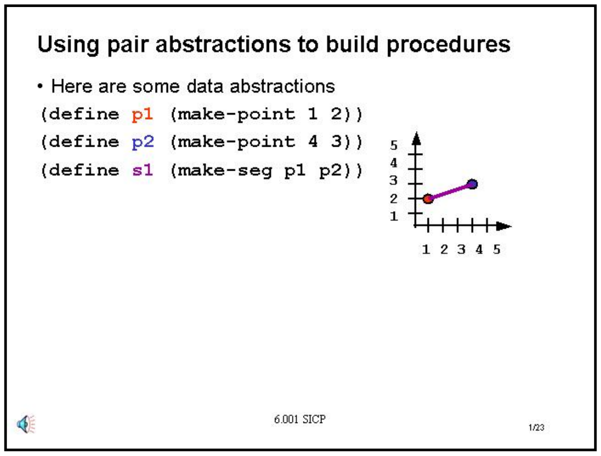
6.001 SICP 10/10

Slide 5.2.10

So here we just restate that idea, one more time, stressing the idea of the contract that defines the interaction between constructor and selectors. And, we stress one more time the idea that pairs are closed, that is, they can be input to the operation of making other pairs.

6.001 Notes: Section 5.3

Slide 5.3.1
So how do we use the idea of pairs to help us in creating computational entities? To illustrate this, let's stick with our example of points and segments. Suppose we construct a couple of points, using the appropriate constructor, and we then glue these points together into a segment.

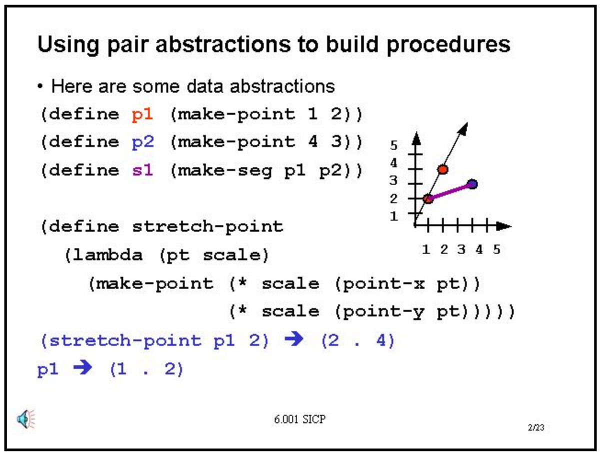


Using pair abstractions to build procedures

- Here are some data abstractions

```
(define p1 (make-point 1 2))
(define p2 (make-point 4 3))
(define s1 (make-seg p1 p2))
```

6.001 SICP 1/23



Using pair abstractions to build procedures

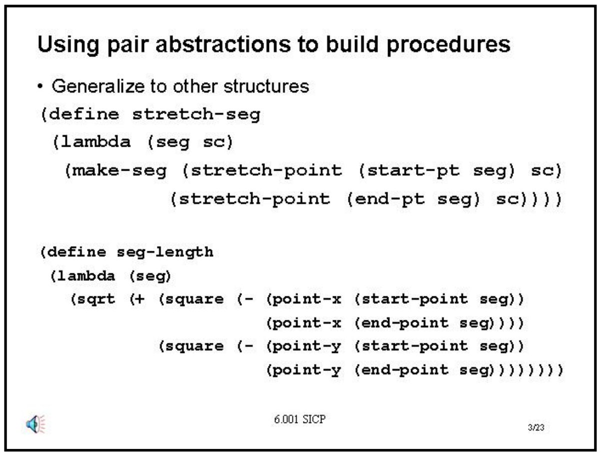
- Here are some data abstractions

```
(define p1 (make-point 1 2))
(define p2 (make-point 4 3))
(define s1 (make-seg p1 p2))

(define stretch-point
  (lambda (pt scale)
    (make-point (* scale (point-x pt))
                (* scale (point-y pt)))))
(stretch-point p1 2) ➔ (2 . 4)
p1 ➔ (1 . 2)
```

6.001 SICP 2/23

Slide 5.3.2
Now suppose we want to think about the operation of stretching a point, that is, pulling (or pushing) it along a line from the origin through the point. Ideally, we would just think about this in terms of operations on elements of a point, without worrying about how the point is actually implemented. We do this with the code shown.
Note how this code creates a **new** data object. If we stretch point P1, we get a new point. Also note, as an aside, how a cons pair prints out, with open and close parentheses, and with the values of the two parts within those parentheses, separated by a dot. Thus, the point created by applying our stretch procedure has a different value for the x and y parts than the original point,
which is still hanging around. Thus, as we might expect from the actual code, we get out the values of the parts of P1, but then make a new data object with scaled versions of those values as the parts.

Slide 5.3.3
And we can generalize this idea to handle operations on segments, as well as points. Note how each of these procedures builds on constructors and selectors for the appropriate data structure, so that in examining the code, we have no sense of the underlying implementation. These structures happen to be built out of cons pairs, but from the perspective of the code designer, we rely only on the contract for constructors and selectors for points and segments.

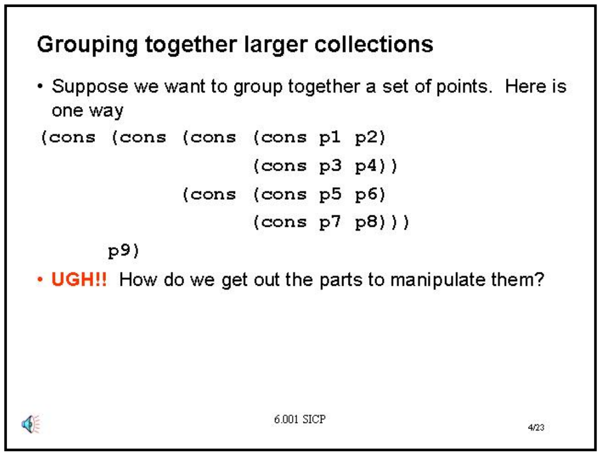


Using pair abstractions to build procedures

- Generalize to other structures

```
(define stretch-seg
 (lambda (seg sc)
  (make-seg (stretch-point (start-pt seg) sc)
            (stretch-point (end-pt seg) sc))))

(define seg-length
 (lambda (seg)
   (sqrt (+ (square (- (point-x (start-point seg))
                       (point-x (end-point seg))))
            (square (- (point-y (start-point seg))
                       (point-y (end-point seg)))))))
```

6.001 SICP 3/23



Grouping together larger collections

- Suppose we want to group together a set of points. Here is one way

```
(cons (cons (cons (cons p1 p2)
                  (cons p3 p4))
            (cons (cons p5 p6)
                  (cons p7 p8)))
      p9)
```

- **UGH!!** How do we get out the parts to manipulate them?

6.001 SICP 4/23

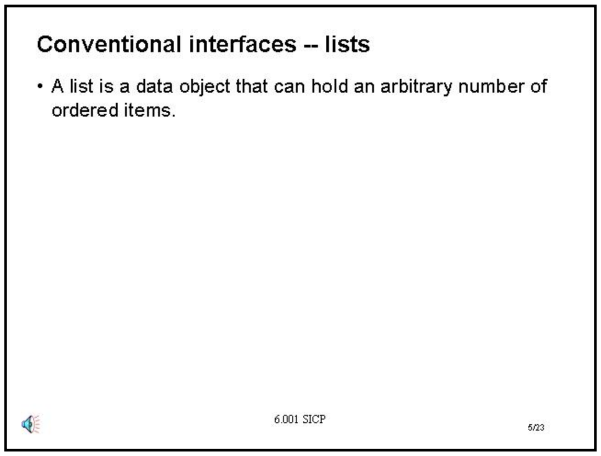
Slide 5.3.4

Now, suppose we decide that we want to take a group of points (which might have segments defined between adjacent points) and manipulate these groups of points. For example, a figure might be defined as a group of ordered points, with segments between each consecutive pair of points. And we might want to stretch that whole group, or rotate it, or do something else to it. How do we group these things together?

Well, one possibility is just to use a bunch of cons pairs, such as shown here. But while this is a perfectly reasonable way to glue things together, it is going to be a bear to manipulate. Suppose we want to stretch all these points? We would have to write code that would put together the right collections of car's and cdr's to get out the pieces, perform a computation on them, and then glue them back together again. This will be a royal pain! It would be better if we had a more convenient and conventional way of gluing together groups of things, and fortunately we do.

Slide 5.3.5

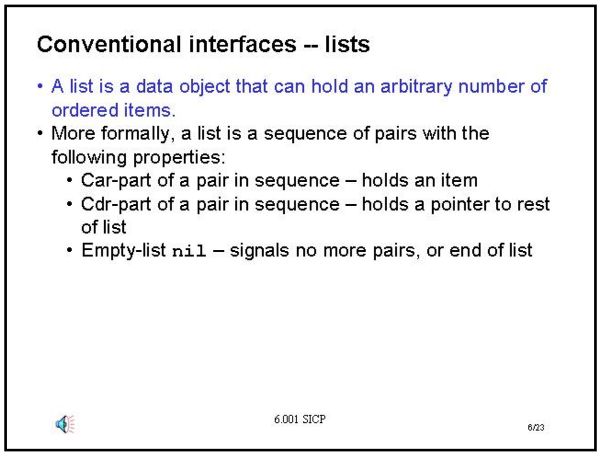
Pairs are a nice way of gluing two things together. However, sometimes I may want the ability to glue together arbitrary numbers of things, and here pairs are less helpful. Fortunately, Scheme also has a primitive way of gluing together arbitrary sets of objects, called a **list**, which is a data object with an arbitrary number of ordered elements within it.



Conventional interfaces -- lists

- A list is a data object that can hold an arbitrary number of ordered items.

6.001 SICP 5/23



Conventional interfaces -- lists

- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
  - Car-part of a pair in sequence – holds an item
  - Cdr-part of a pair in sequence – holds a pointer to rest of list
  - Empty-list `nil` – signals no more pairs, or end of list

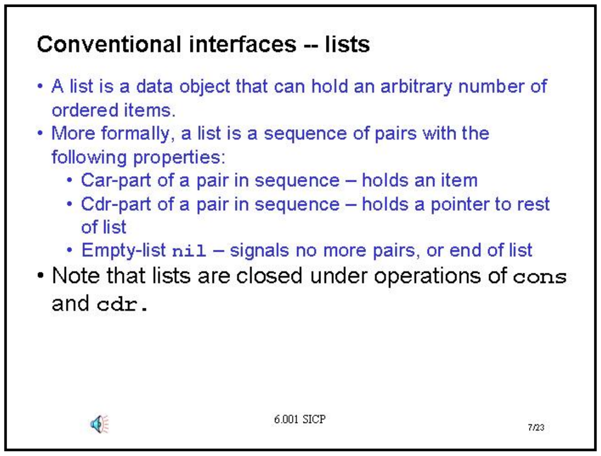
6.001 SICP 6/23

Slide 5.3.6

Of course, we could make a list by just **cons**ing together a set of things, using however many pairs we need. But it is much more convenient to think of a list as a basic structure, and here is more formally how we define such a structure. A list is a sequence of pairs, with the following properties. The car part of a pair in the list holds the next element of the list. The cdr part of a pair in the list holds a pointer to the rest of the list. We will also need to tell when we are at the end of the list, and we have a special symbol, **nil**, that signals the fact that there are no more pairs in the list.

Slide 5.3.7

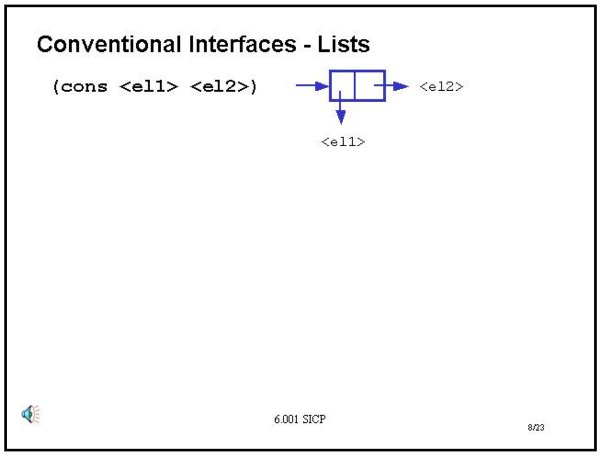
Another way of saying this is that lists are sequences of pairs ending in the empty list. Under that view, we see that lists are closed under the operations of **cons** and **cdr**. To see this, note that if we are given a sequence of pairs ending in the empty list, and we cons anything onto that sequence, we get another sequence of pairs ending in the empty list, hence a list. Similarly, if we take the cdr of a sequence of pairs ending in the empty list, we get a smaller sequence of pairs ending in the empty list, hence a list. This property of closure says we can use lists as primitives within other lists.
The only trick is what happens when I try to take the cdr of an empty list. The result depends on the Scheme implementation, as in some cases it is an error, while for other Schemes it is the empty list. The latter view is nice when considering the closure property as it preserves the notion that the cdr of a list is a list.



Conventional interfaces -- lists

- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
  - Car-part of a pair in sequence – holds an item
  - Cdr-part of a pair in sequence – holds a pointer to rest of list
  - Empty-list `nil` – signals no more pairs, or end of list
- Note that lists are closed under operations of `cons` and `cdr`.

6.001 SICP 7/23



Conventional Interfaces - Lists

```
(cons <el1> <el2>)
```

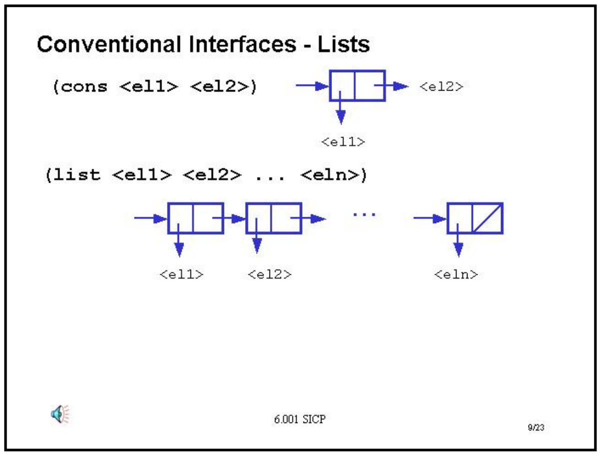
6.001 SICP 8/23

Slide 5.3.8

To visualize this new conventional way of collecting elements, called a list, we use **box-and-pointer** notation. First, a cons pair is represented by a pair of boxes. The first box contains a pointer to the value of the first argument to **cons** and the second box contains a pointer to the value of the second argument to **cons**. The pair also has a pointer into it, and that pointer is the value returned by evaluating the **cons** expression, and represents the actual pair.

Slide 5.3.9

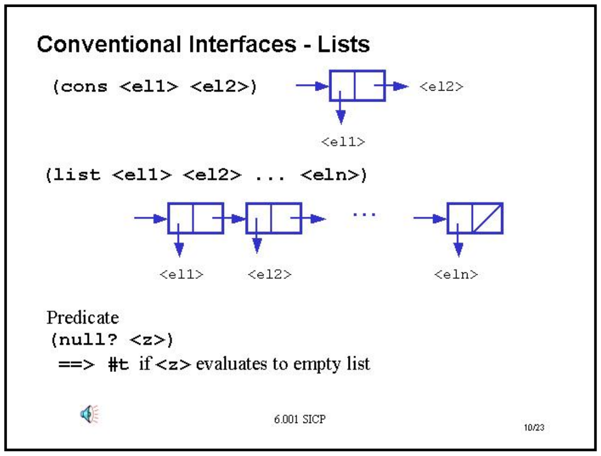
A list then simply consists of a sequence of pairs, or boxes, which we conventionally draw in a horizontal line. The **car** element of each box points to an element of the sequence, and the **cdr** element of each box points to the rest of the list. The empty list is indicated by a diagonal line in the last **cdr** box. One can see that the list is much like a skeleton. The cdrs define the spine of the skeleton, and hanging off the cars are the ribs, which contain the elements. Also notice how this visualization clearly defines the closure property of lists, since taking the cdr of a list gives us a new sequence of boxes ending the empty list.



Conventional Interfaces - Lists

```
(cons <el1> <el2>)
(list <el1> <el2> ... <eln>)
```

6.001 SICP 9/23



Conventional Interfaces - Lists

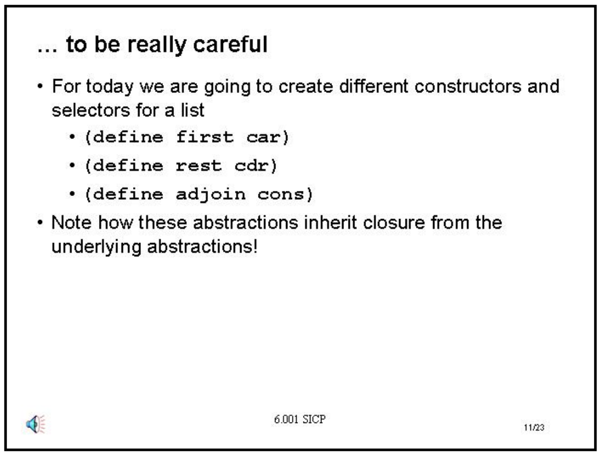
```
(cons <el1> <el2>)
```

```
(list <el1> <el2> ... <eln>)
```

Predicate
`(null? <z>)`
`==> #t` if `<z>` evaluates to empty list

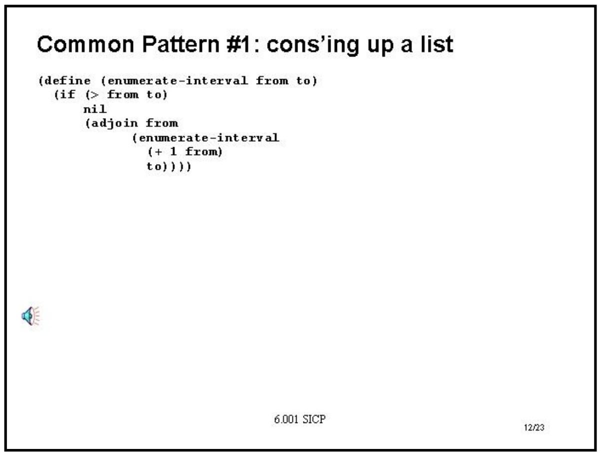
Slide 5.3.10
To check if something is a list, we need two things. First, we have a predicate, `null?` that checks to see if an object is the empty list. Then, to check if a structure is a list, we can just use `pair?` to see if the structure is a sequence of pairs. Actually, we really should check to see that the sequence ends in the empty list, but conventionally we just check to see that the first element is a pair, that is, something made by **cons**.

Slide 5.3.11
Since we have built lists out of cons pairs, we can use `car` and `cdr` to get out the pieces. But just for today, we are going to separate the operations on lists from operations on pairs, by defining special selectors and constructor for lists, as shown. Thus we have a way of getting the first and rest of the elements of a list, and for putting a new element onto the front of a list. Notice how these operations inherit the necessary closure properties from their implementation in terms of cons, car and cdr.



... to be really careful

- For today we are going to create different constructors and selectors for a list
  - `(define first car)`
  - `(define rest cdr)`
  - `(define adjoin cons)`
- Note how these abstractions inherit closure from the underlying abstractions!

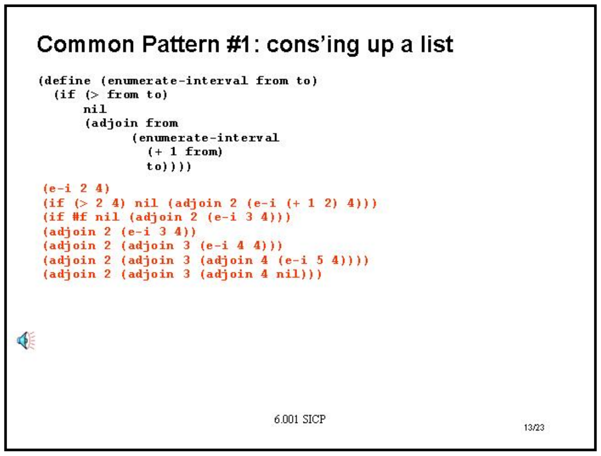


Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
             (enumerate-interval
               (+ 1 from)
               to))))
```

Slide 5.3.12
A key point behind defining a new data abstraction is that it should make certain kinds of operations easy to perform. We expect to see that with lists, and in what follows we are going to explore that idea, looking for standard kinds of operations associated with lists.
One common operation is the creation of lists. Here is a simple example of this, which generates a sequence (or list) of numbers between two points. Notice the nice recursive call within this procedure. It says, to generate such a list, cons or glue the value of **from** onto whatever I get by creating the interval from **from + 1** to **to**. This is reducing things to a simpler version of the same problem, terminating when I just have an empty list.
Notice how the constructor relies on the data abstraction contract. If this procedure works correctly on smaller problems, then the **adjoin** operation is guaranteed to return a new list, since it is gluing an element onto a list, and by closure this is also a list. This kind of inductive reasoning allows us to deduce that the procedure correctly creates the right kind of data structure.

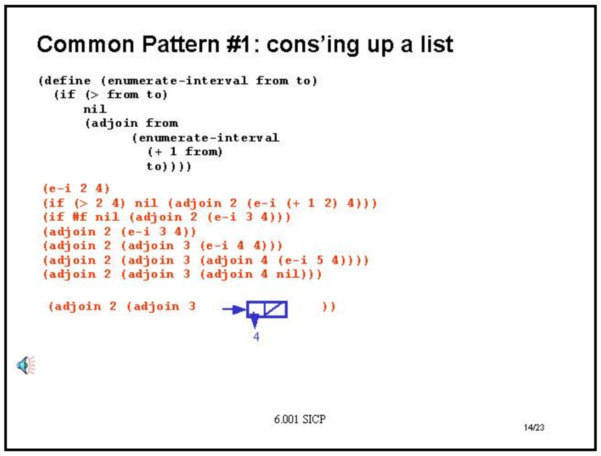
Slide 5.3.13
Here is a trace of the substitution model running on an example of this. Notice how the **if** clause unwinds this into an adjoining of an element onto a recursive call to the same procedure with different arguments. Since we must get the values of the subexpressions before we can apply the adjoin operation, we expand the recursive call out again, creating another deferred adjoin operation. And we keep doing this until we get down to the base case, returning the value of the empty list. Notice that this now leaves us with an expression that will create a list, a sequence terminating in the special symbol for an empty list.



Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
              (enumerate-interval
                (+ 1 from)
                to))))

(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))
```

6.001 SICP 13/23



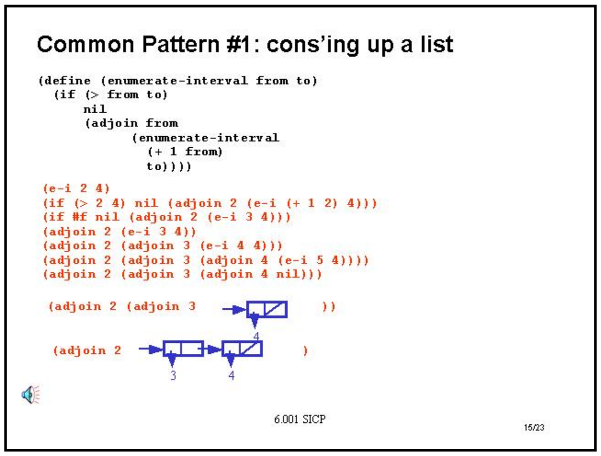
Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
              (enumerate-interval
                (+ 1 from)
                to))))

(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))

 (adjoin 2 (adjoin 3            ))
```

6.001 SICP 14/23

Slide 5.3.14
Now we are ready to evaluate the innermost expression, which actually creates a cons pair. To demonstrate this, we have drawn in the pair to show that this is the value returned by **adjoin**. Notice how it has the correct form for a list.

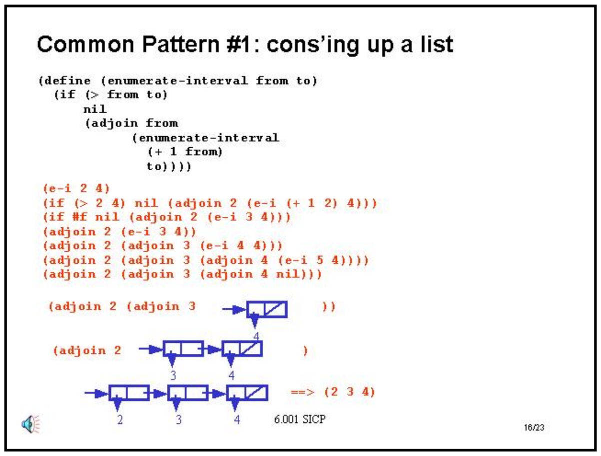
Slide 5.3.15
The next evaluation adjoins another element onto this list, with the cdr pointer of the newly created cons pair pointing to the value of second argument, namely the previously created list.



Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
              (enumerate-interval
                (+ 1 from)
                to))))

(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))

 (adjoin 2 (adjoin 3            ))

  (adjoin 2                  )
```

6.001 SICP 15/23



Common Pattern #1: cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      nil
      (adjoin from
              (enumerate-interval
                (+ 1 from)
                to))))

(e-i 2 4)
(if (> 2 4) nil (adjoin 2 (e-i (+ 1 2) 4)))
(if #f nil (adjoin 2 (e-i 3 4)))
(adjoin 2 (e-i 3 4))
(adjoin 2 (adjoin 3 (e-i 4 4)))
(adjoin 2 (adjoin 3 (adjoin 4 (e-i 5 4))))
(adjoin 2 (adjoin 3 (adjoin 4 nil)))

 (adjoin 2 (adjoin 3            ))

  (adjoin 2                  )

                         ==> (2 3 4)
```

6.001 SICP 16/23

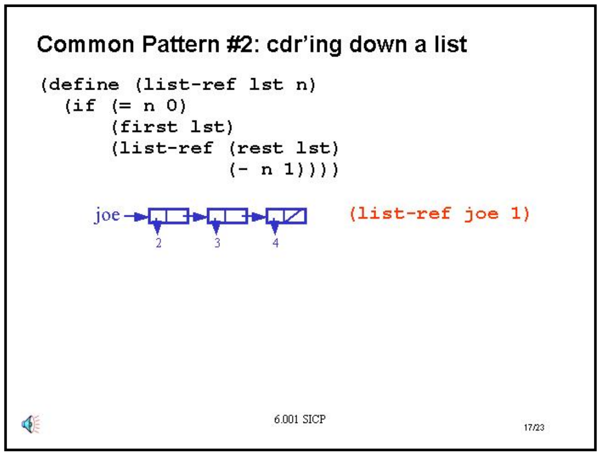
Slide 5.3.16
And this of course then leads to this structure. This prints out as shown, which is the printed form for a list.
Notice the order in which the pairs are created, and notice the set of deferred operations associated with the creation of this list.

Slide 5.3.17

In addition to procedures that can "cons up" a list, we have procedures that can walk down a list, also known as "cdring down" a list.

Here is a simple example, which finds the nth element of a list, where by convention a list starts at 0. Notice how we use the recursive property of a list to do this. If our index is 0, then we want the first element. Otherwise, we know that the nth element of a list will be the n-1st element of the rest of the list, by the closure property of lists. So we can recursively reduce this to a simpler version of the same problem.

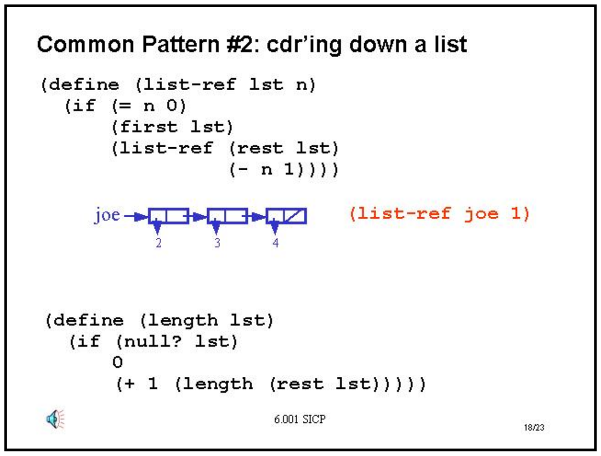
For the example shown, we will first check to see if n=0. Since it does not, we take the list pointed to by `joe`, extract the rest of that list by literally taking the pointer out of the cdr part of the first box, and call list-ref on that structure, with a decrement of `n`. This recursive call will now have n=0, so we return the first element of the new list, namely the element 3.



Common Pattern #2: cdr'ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (first lst)
      (list-ref (rest lst)
                (- n 1))))
```

joe
2
3
4
(list-ref joe 1)
6.001 SICP
17/23



Common Pattern #2: cdr'ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (first lst)
      (list-ref (rest lst)
                (- n 1))))
```

joe
2
3
4
(list-ref joe 1)

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (rest lst)))))
```

6.001 SICP
18/23

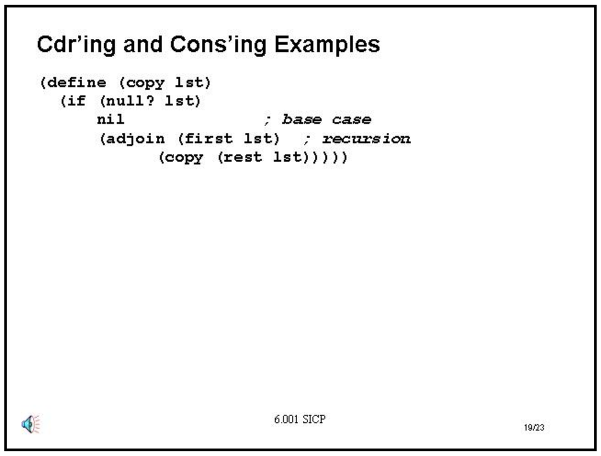
Slide 5.3.18

A related procedure is to count up the number of elements in a list, using the same sort of recursive reasoning. `Length` of a list is defined as 1 more than the length of the rest of a list, which by the closure property is also a list. The base case is the empty list, which has zero length.

Slide 5.3.19

Now we can put the two ideas together, cdring down one list while consing up a new one as the return value. Here is an example, which creates a copy of a list.

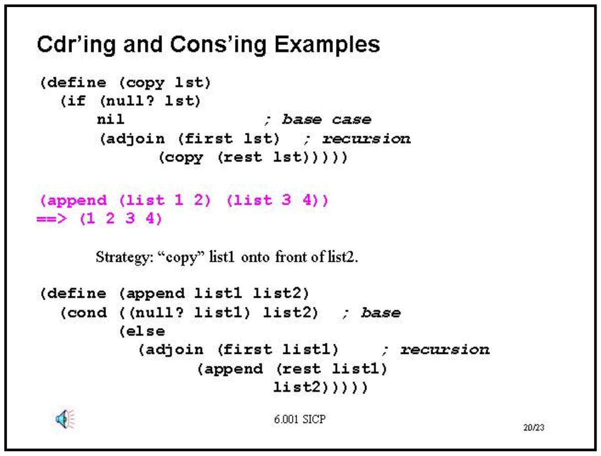
Notice the form. If we are given an empty list, we just return an empty list and are done. If not, then we use the recursive property of lists. We adjoin the first element of the input list onto whatever we get by copying the rest of the list. But by closure, the rest of the input list is a list, so copy is guaranteed by induction to give us back a list. And by what we just saw, we see that copy will create a copy of the list in exactly the same order.



Cdr'ing and Cons'ing Examples

```
(define (copy lst)
  (if (null? lst)
      nil               ; base case
      (adjoin (first lst)  ; recursion
              (copy (rest lst)))))
```

6.001 SICP
19/23



Cdr'ing and Cons'ing Examples

```
(define (copy lst)
  (if (null? lst)
      nil                ; base case
      (adjoin (first lst)  ; recursion
             (copy (rest lst)))))

(append (list 1 2) (list 3 4))
==> (1 2 3 4)
```

Strategy: "copy" list1 onto front of list2.

```
(define (append list1 list2)
  (cond ((null? list1) list2)   ; base
        (else
          (adjoin (first list1)    ; recursion
                (append (rest list1)
                        list2)))))
```

6.001 SICP 20/23

Slide 5.3.20

Well copying is boring, but here is how we can use this idea. Suppose we want to glue two lists together into one long list, as shown in the example.

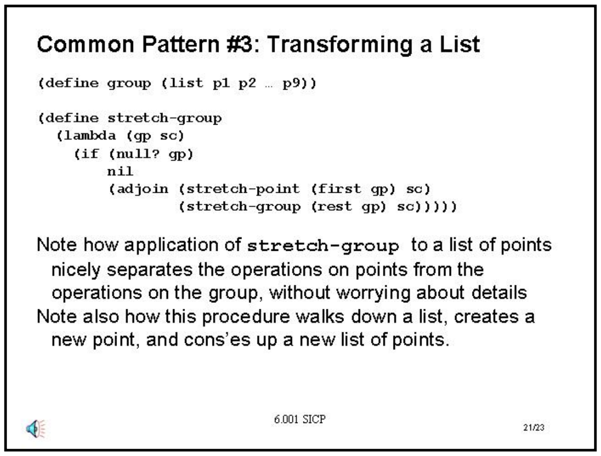
Let's just use the same strategy, implemented in a handy procedure called `append`. For the base case, notice that if we do not have anything in the first list, we just want the second list. Otherwise, we use the same idea as before. We adjoin the first element of the first list onto whatever we get by appending the rest of the first list onto the second list. Using the same closure properties, we see that this will create a list, and adjoin will then put the first element on the front of this new list.

Notice how we are using the recursive properties of lists to build this procedure, and in particular notice how the recursive structure of the procedure nicely mimics the recursive structure of the data object.

Slide 5.3.21

So now we can put these ideas to work in handling more complex structures. Let's group together a set of points using a list. Then we can easily write a procedure to stretch the entire group, by building on the wonderfully recursive nature of the list. Here it is.

Note how we subdivide the problem of stretching a group into the operation of stretching a point (using the procedure appropriate for points), and then adding that to the result we would get by stretching the rest of the group. Because of the recursive nature of a list, we know that the rest of a list is a list, so we can use induction to conclude that stretch-group applied to the smaller collection will return a new group, and thus adjoining the new element to the front of this will clearly give us back a group.



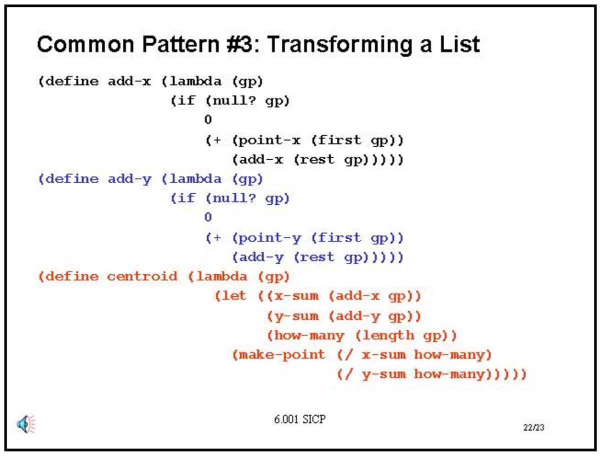
Common Pattern #3: Transforming a List

```
(define group (list p1 p2 … p9))

(define stretch-group
  (lambda (gp sc)
    (if (null? gp)
        nil
        (adjoin (stretch-point (first gp) sc)
                (stretch-group (rest gp) sc)))))
```

Note how application of `stretch-group` to a list of points nicely separates the operations on points from the operations on the group, without worrying about details

Note also how this procedure walks down a list, creates a new point, and cons'es up a new list of points.

6.001 SICP 21/23



Common Pattern #3: Transforming a List

```
(define add-x (lambda (gp)
               (if (null? gp)
                   0
                   (+ (point-x (first gp))
                      (add-x (rest gp))))))
(define add-y (lambda (gp)
               (if (null? gp)
                   0
                   (+ (point-y (first gp))
                      (add-y (rest gp))))))
(define centroid (lambda (gp)
                  (let ((x-sum (add-x gp))
                        (y-sum (add-y gp))
                        (how-many (length gp))
                    (make-point (/ x-sum how-many)
                                (/ y-sum how-many)))))
```

6.001 SICP 22/23

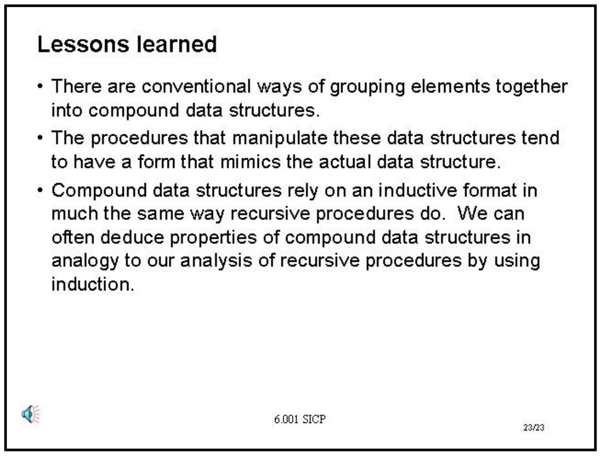
Slide 5.3.22

And if we want to find the midpoint (or centroid) of the group, we can put together the pieces we built earlier, as shown here. Add-x and add-y have a structure very similar to our earlier examples: they simply cdr down the list, gathering up information as they go in a set of deferred additions. Each of these will get the sum of the x and y values of all the points in a group. To find the midpoint we need to get the average x and y value, so we need to know how many elements are in the group, which we get using length. We can combine this information to create a new point at the middle of the group.

Note the new form **let.** You can find the details in the textbook, but it suffices to think of this as an expression in which each of the names in the first set of expressions (x-sum, y-sum and how-many) are bound to the values of the expressions following those names. Then within the confines of the let expression, those names are simply local names for those values, and are substituted for just as we would in the standard substitution model.

Slide 5.3.23

So to summarize: we have seen that languages often provide conventional ways of grouping data elements into structures, here either in pairs or as arbitrarily long collections. Associated with these conventional structures are methods for operating on them, and these procedures often have a form that mimics the structure. For example, in procedures that convert lists into lists, we see that the recursive step usually involves using the selectors to get out the parts of the list, operating on each, and then using the constructor to reassemble the parts into a list. This form means that the same inductive proofs we used to reason about our recursive procedures will also apply here. We can often deduce properties of our procedures and their associated data structures by relying on the fact that inductively the procedure operates correctly on smaller sized data structures.



Lessons learned

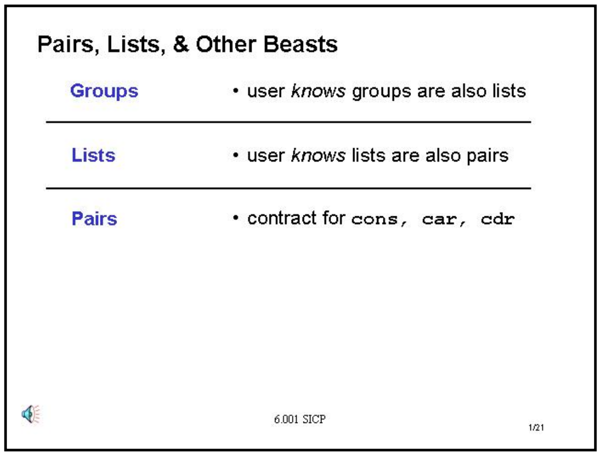
- There are conventional ways of grouping elements together into compound data structures.
- The procedures that manipulate these data structures tend to have a form that mimics the actual data structure.
- Compound data structures rely on an inductive format in much the same way recursive procedures do. We can often deduce properties of compound data structures in analogy to our analysis of recursive procedures by using induction.

6.001 SICP 23/23

6.001 Notes: Section 5.4

Slide 5.4.1

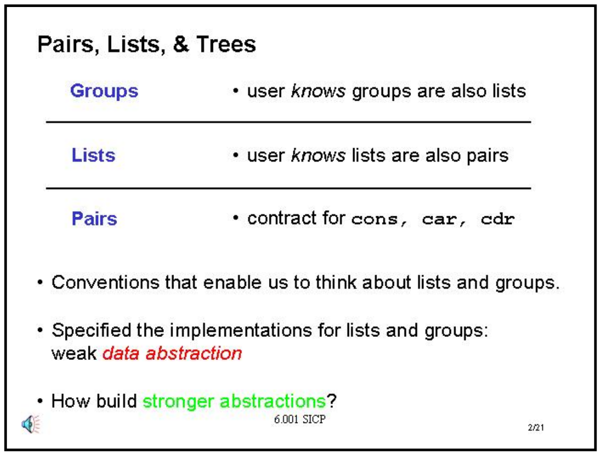
So let's step back and examine what we have built so far. We've basically built a hierarchy of data abstractions, each of which is constructed out of simpler ones. At the bottom are **pairs** where we have a basic contract from Scheme, about how `cons`, `car` and `cdr` interact as a data abstraction. On top of that, we have built lists, but where the user can take advantage of the fact that lists are also pairs. In fact, we do this directly by using `car` and `cdr` as our selectors for lists. On top of that, we have just built groups, but again the user can take advantage of the knowledge that groups are constructed as lists of lists, but where she is shielded from the fact that lists are implemented as pairs.



Pairs, Lists, & Other Beasts

Groups	• user *knows* groups are also lists
Lists	• user *knows* lists are also pairs
Pairs	• contract for `cons, car, cdr`

6.001 SICP 1/21



Pairs, Lists, & Trees

Groups	• user *knows* groups are also lists
Lists	• user *knows* lists are also pairs
Pairs	• contract for `cons, car, cdr`

- Conventions that enable us to think about lists and groups.
- Specified the implementations for lists and groups: weak *data abstraction*
- How build stronger abstractions?

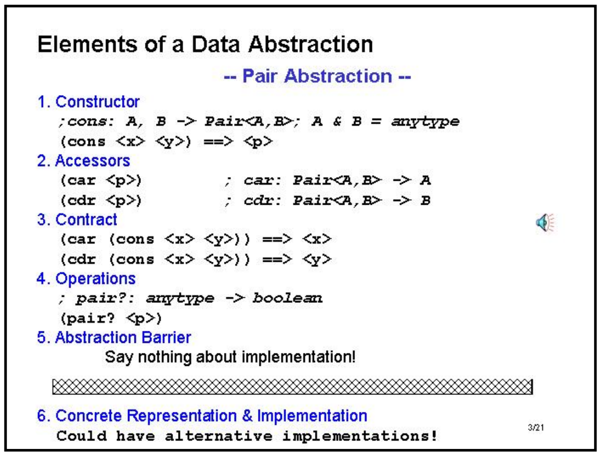
6.001 SICP 2/21

Slide 5.4.2

In essence, we have built a set of abstraction barriers in which the implementation details of a data structure are only weakly separated from the use of that structure. This means that we rely on the user showing discipline when applying procedures to data structures, expecting them not to use procedures that directly take advantage of the implementation of the structure.

Sometimes we are much better off imposing **strong** abstraction barriers between structures, thereby not allowing a user to take advantage of the underlying representation.

Said another way, we have tried to separate out data structures that have different conventions for usage, enabling us to think about the structures as ways of organizing data, but we've allowed the user to cross over the barrier between those structures to get at the underlying implementation. What happens if we decide to make those barriers much stronger, thereby shielding the people who use the abstractions from the underlying representation?



Elements of a Data Abstraction

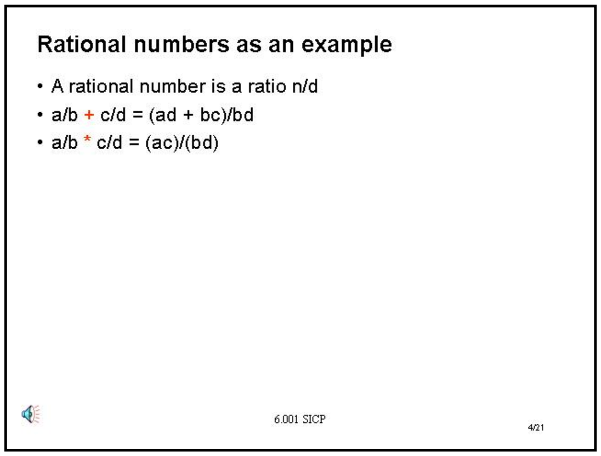
-- Pair Abstraction --

1. Constructor
```
;cons: A, B -> Pair<A,B>; A & B = anytype
(cons <x> <y>) ==> <p>
```
2. Accessors
```
(car <p>)          ; car: Pair<A,B> -> A
(cdr <p>)          ; cdr: Pair<A,B> -> B
```
3. Contract
```
(car (cons <x> <y>)) ==> <x>
(cdr (cons <x> <y>)) ==> <y>
```
4. Operations
```
; pair?: anytype -> boolean
(pair? <p>)
```
5. Abstraction Barrier

Say nothing about implementation!

6. Concrete Representation & Implementation
```
Could have alternative implementations!
```
3/21

Slide 5.4.3
This then leads us to the notion of a rigorous **data abstraction**. For example, for **pairs** here is what we expect in such an abstraction. We'll have a constructor for gluing pieces together, and it will have a contract between the types of the inputs and the type of the data structure constructed.
We'll have selectors or accessors for getting back out the pieces that are glued together.
And most importantly, we will have a contract between the constructor and the selectors, so that whatever we glue together with the constructor we can get back out using the appropriate selector.
Notice that this contract says **nothing about how** the abstraction is implemented, only that its behavior is as documented.
We will have some standard operations on the abstraction, typically predicates for identifying an instance of the data abstraction.
All of this is separated from the actual implementation by what we call an **abstraction barrier**. Think of this as a wall that separates the use of an abstraction from the implementation of an abstraction. This means that a user of the abstraction can freely write procedures to manipulate the structures, just relying on the constructor and selectors, without any knowledge of how the structure is actually made.



Rational numbers as an example

- A rational number is a ratio n/d
- a/b + c/d = (ad + bc)/bd
- a/b * c/d = (ac)/(bd)

6.001 SICP

4/21

Slide 5.4.4
Now, let's drive this point home with a more extended example, and remember that the point we are trying to drive home is the separation of the details of implementation of an abstraction from the use of that abstraction. We are going to place this solid barrier between the implementation and use of an abstraction, and we are going to see why having that barrier makes it much easier for us to create useful systems.
The example we are going to consider is that of **rational numbers** and simple arithmetic operations on rationals. I'll remind you that a rational number is just a ratio of two integers, a **numerator** over a **denominator**. Associated with rationals are certain operations, for example we can add two rationals
together, or multiply two rationals together, using the rules shown in the slide.
Notice that we are really cheating here, and in fact I have put the operation on the rationals in red to distinguish that these are operations on rationals, as opposed to operations on integers. In fact, the rules for operations on rationals really decompose into simpler operations on integers. The operations on the right hand side are just operations on normal integers, followed by the creation of a rational (which is what the division sign denotes here).
Now, let's see how we can use the ideas of data abstractions to build a system for manipulating rational numbers.

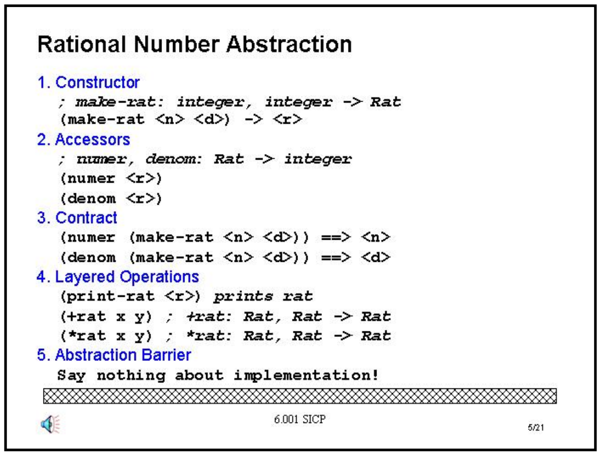
Slide 5.4.5

First, here's our data abstraction for rational numbers, which has the form we expect. It has a constructor, `make-rat` which takes two integers as input, and produces a rational, with some kind of internal representation that we haven't specified. We'll need selectors for getting back out the pieces of a rational. Notice the type associated with `numer` and `denom`.

Most important is the contract between these parts. We can apply either of the selectors to the object created by the constructor, and be guaranteed to get back the appropriate piece, independent of how the object is actually represented.

We will also have some operations that apply to rationals, such as rational addition or rational multiplication, and notice the types of these operations. They take two rationals as input, and produce a rational as output.

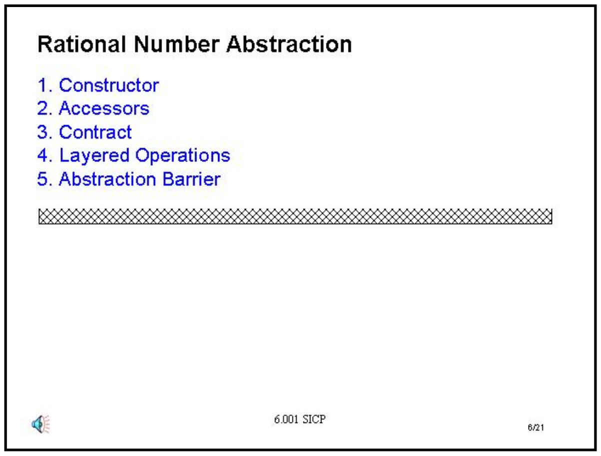
Of course, we are going to separate all of this from the actual implementation of rationals. We should ideally be able to take any implementation of rationals that satisfies these contracts and build our own procedures to use those objects, just by relying on the constructor and selectors, and the contract between them.



Rational Number Abstraction

1. Constructor
```
; make-rat: integer, integer -> Rat
(make-rat <n> <d>) -> <r>
```
2. Accessors
```
; numer, denom: Rat -> integer
(numer <r>)
(denom <r>)
```
3. Contract
```
(numer (make-rat <n> <d>)) ==> <n>
(denom (make-rat <n> <d>)) ==> <d>
```
4. Layered Operations
```
(print-rat <r>) prints rat
(+rat x y) ; +rat: Rat, Rat -> Rat
(*rat x y) ; *rat: Rat, Rat -> Rat
```
5. Abstraction Barrier
```
Say nothing about implementation!
```

6.001 SICP 5/21



Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

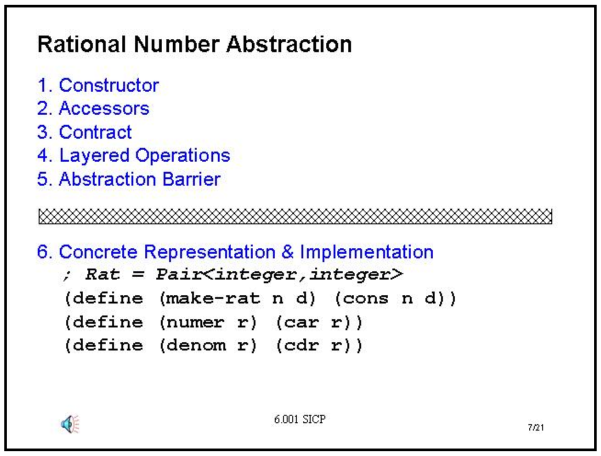
6.001 SICP 6/21

Slide 5.4.6

Now let's think about actually building an implementation for rationals. How do we go about building an implementation that satisfies the contract defined in the previous slide, while providing all the pieces we laid out?

Slide 5.4.7

Well here is a simple implementation for rationals. We could use **pairs** as our underlying representation for rationals. Using this, we can easily construct the constructor and selectors for rationals, using `cons, car` and `cdr`. While this seems like an obvious way to build rationals, there is an important point here. In particular, we can enforce the contract that we require between `make-rat, numer` and `denom` by inheriting the contract between `cons, car` and `cdr`. At the same time, by relying on separate constructors and selectors for rationals, we shield the user from the implementation details, and we will see shortly why that is important.

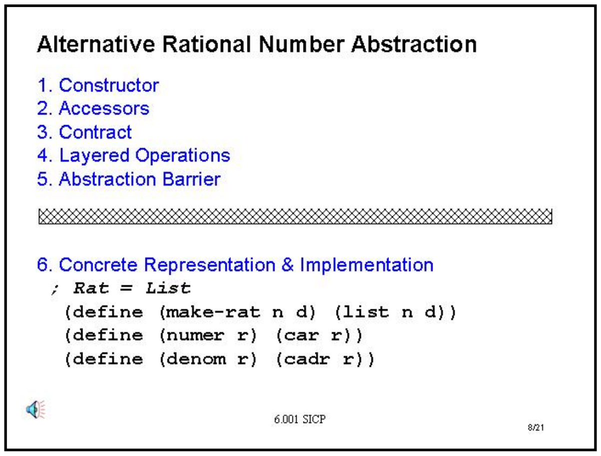


Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation
```
; Rat = Pair<integer,integer>
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

6.001 SICP 7/21



Alternative Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Layered Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
 ; Rat = List
   (define (make-rat n d) (list n d))
   (define (numer r) (car r))
   (define (denom r) (cadr r))
```

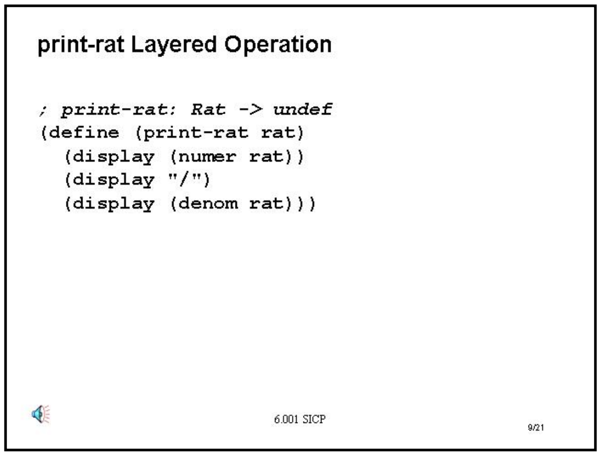
6.001 SICP 8/21

Slide 5.4.8

In fact, here is one reason why this is important. Here is an alternative implementation of rationals, using `list` as the constructor, and `car` and `cadr` (which gets the second element of a list or the `car` or the `cdr` of the list) as the selectors. Does this satisfy our contract? Sure! Can a user tell which implementation we are using, the list or the pair version? No. The whole point of shielding the user from the implementation is to separate out those issues from the use of the abstraction. Why would we want alternative representations? We'll see some advantages shortly, but the key point here is that from the user's perspective each implementation is equally valid, as each satisfies the contract.

Slide 5.4.9

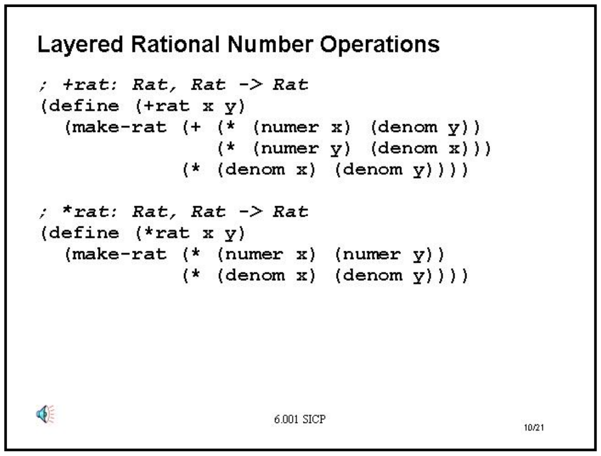
For example, here is a useful procedure for rationals, which will print out the value of a rational by separating the numerator and denominator by a "slash". The key question is can this procedure tell which implementation of rationals we are using? The answer, of course, is **no**. The procedure uses the selectors to get out the pieces, and cannot tell which implementation is being used so long as the right selector for that implementation is employed.



print-rat Layered Operation

```
; print-rat: Rat -> undef
(define (print-rat rat)
  (display (numer rat))
  (display "/")
  (display (denom rat)))
```

6.001 SICP 9/21



Layered Rational Number Operations

```
; +rat: Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

; *rat: Rat, Rat -> Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

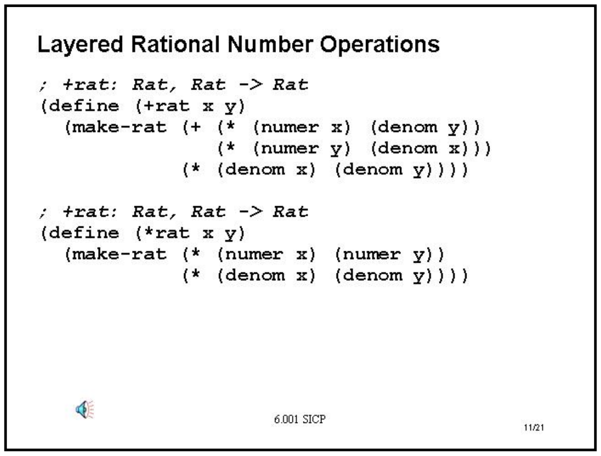
6.001 SICP 10/21

Slide 5.4.10

My operations for arithmetically manipulating rationals have a common form that we are going to see many times. Look at the first one carefully. The type contract says that given two rationals as input, we return a rational as output. The body of the procedure uses the selectors to extract out simpler objects, (in this case integers) then applies more primitive procedures to those objects (for example, multiplying integers and adding integers) then uses the constructor to glue these more primitive elements together into the complex object that needs to be returned.

This is an important and common process. When manipulating data abstractions, we tend to use the selectors to get out simpler pieces, execute more primitive operations on those pieces, then use the constructor to glue the new pieces together into a new complex object.

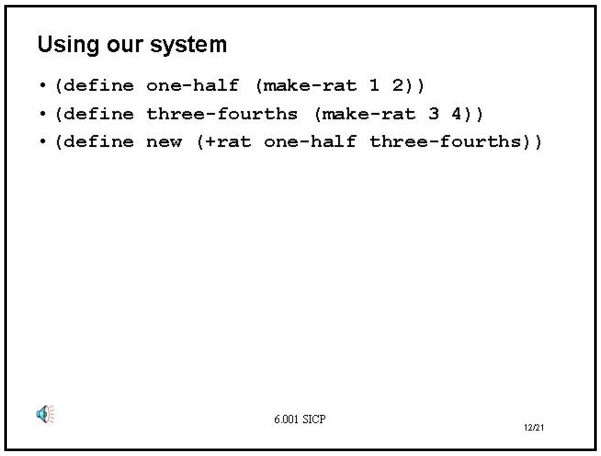
Slide 5.4.11
Also, notice how types can help us reason through these procedures. Remember my contract for the selectors `numer` and `denom`, which said that given rationals as input, they would return integers. Thus in the body of these procedures I can see that I am safe in using integer arithmetic operations on the values returned by these selectors. And the contract for `make-rat` says that given two integers (as specified by the bodies of these procedures), we will return a rational number; hence my contract for these procedures is met, given that the contracts for the constructors and selectors are met.



Layered Rational Number Operations

```
; +rat: Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

; *rat: Rat, Rat -> Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

6.001 SICP 11/21



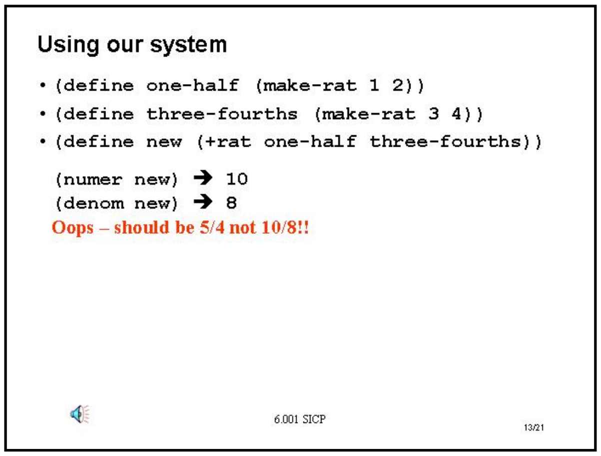
Using our system

- `(define one-half (make-rat 1 2))`
- `(define three-fourths (make-rat 3 4))`
- `(define new (+rat one-half three-fourths))`

6.001 SICP 12/21

Slide 5.4.12
Okay, let's use this little system. We can create a couple of simple rationals as shown, making `one-half` and `three-fourths` to be the expected rationals. Let's define `new` to be the rational obtained by adding these two together, and we know by our previous work, that this should pull apart these two rationals, manipulate the underlying integer pieces, then reassemble a new rational that is the sum of these two.

Slide 5.4.13
This sounds pretty straightforward. What happens if we look at the pieces of `new`? The numerator is 10, and the denominator is 8, so this rational is 10/8. But wait a minute! Shouldn't this be 5/4?
We would expect to get 5/4 here, but we got an equivalent, but not identical answer. If we look at the details of how we implemented rational addition, we see that in fact this is the answer we should get, since we simply take the cross products of the numerators and denominators and form their ratio. So in this case we get the "right" answer but not the "correct" answer (or the answer we were expecting).



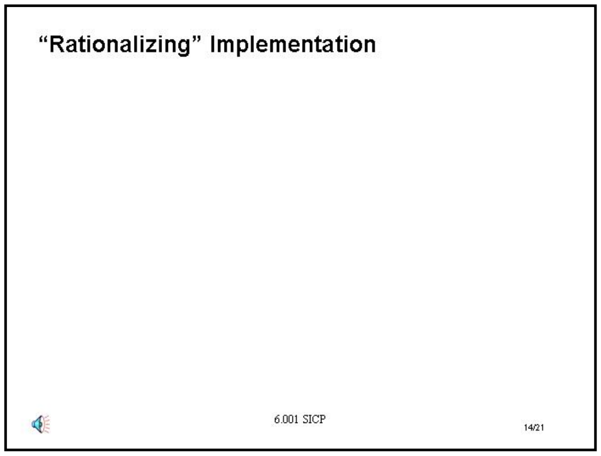
Using our system

- `(define one-half (make-rat 1 2))`
- `(define three-fourths (make-rat 3 4))`
- `(define new (+rat one-half three-fourths))`

```
(numer new) ➔ 10
(denom new) ➔ 8
```

Oops – should be 5/4 not 10/8!!

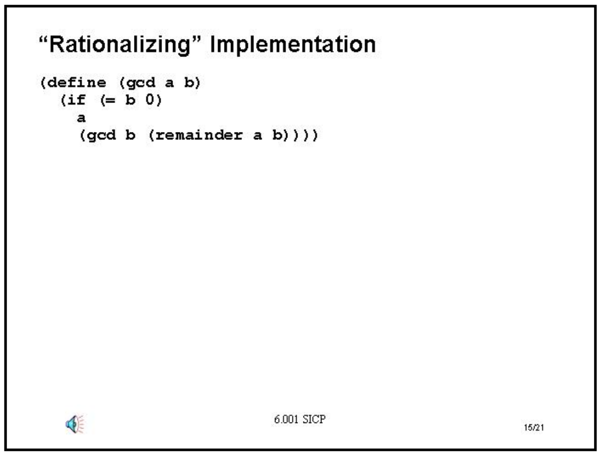
6.001 SICP 13/21


"Rationalizing" Implementation

6.001 SICP 14/21

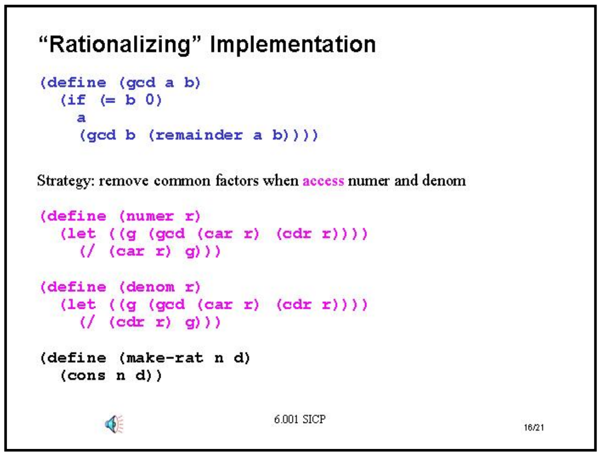
Slide 5.4.14
And in fact that is exactly the point of this whole exercise! Given that we can shield the user from the implementation of an abstraction, we should be able to fix this problem without affecting procedures created by the user.
So what is the "problem" here? We are not "rationalizing" the rationals, that is, we are not reducing our constructed rationals to simplest terms. Suppose we were to compute the greatest common divisor between the numerator and the denominator, and reduce both by this term. This would give us a simpler rational. The point, however, is to do this in a way that is invisible to the user.

Slide 5.4.15
To do that, we will just use an implementation of `gcd`, which computes the greatest common divisor of two integers, as shown here. Let's use this to make "better" rationals.


"Rationalizing" Implementation

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

6.001 SICP 15/21


"Rationalizing" Implementation

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

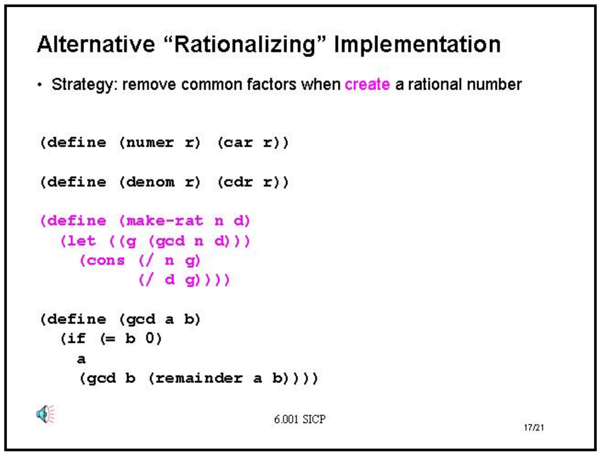
Strategy: remove common factors when access numer and denom

```
(define (numer r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (car r) g)))

(define (denom r)
  (let ((g (gcd (car r) (cdr r))))
    (/ (cdr r) g)))

(define (make-rat n d)
  (cons n d))
```

6.001 SICP 16/21

Slide 5.4.16
... and here we have a choice. The first strategy is to use `gcd` when we access the parts of a rational. Thus, we will construct rationals the normal way, just gluing together the parts. But when we ask for the parts of the rational, we will first compute the gcd of the numerator and denominator, then reduce the selector part by that amount. This will fix the problem we saw in the previous slide, always reducing a rational to its lowest possible form.
But notice that since we built all of our operations just using the constructor and selectors, they will all continue to work, even with this change underneath them. That was exactly the point of separating the implementation of the abstraction from its use. So long as our constructors and selectors satisfy the contract, and so long as we always use them constructor and selectors in creating procedures to manipulate the abstractions (i.e. the only procedures that cross that abstraction barrier), we are free to create new implementations of the abstraction without having to recode any of those procedures.

Slide 5.4.17
And to drive that point home, here is an alternative way to create "rational" rationals. In this case, we will reduce rationals to lowest possible terms when we construct them, that is, when we apply `make-rat`. Here, the selectors can just be the straightforward implementation, since the constructor now takes care of the reduction.
As before, the contract is still satisfied, and thus any procedure that uses these abstraction interfaces will still work without any change.
Why have these alternatives? Well, if we expect to have a system that creates relatively few rationals but accesses them a lot, then for efficiency we should do the reduction at construction time. But if we expect the opposite load, we can use the other implementation.

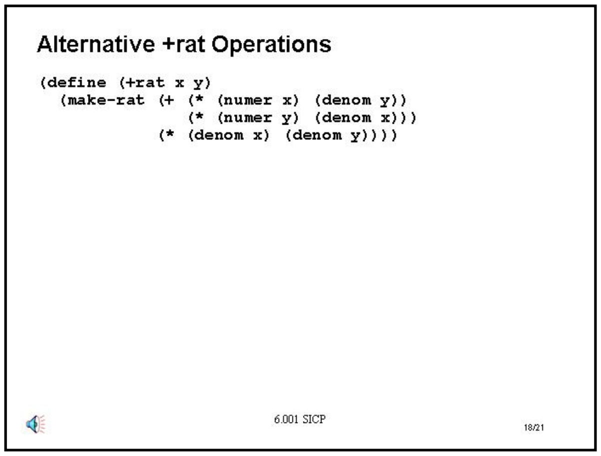


Alternative "Rationalizing" Implementation

- Strategy: remove common factors when create a rational number

```
(define (numer r) (car r))

(define (denom r) (cdr r))

(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g)
          (/ d g))))

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

6.001 SICP 17/21



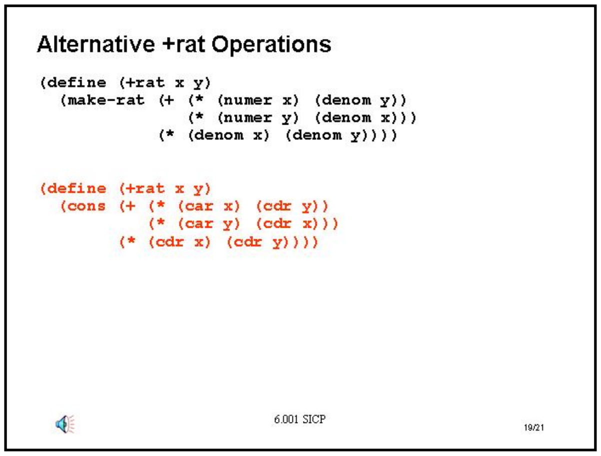
Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

6.001 SICP 18/21

Slide 5.4.18
To finish making this point, think about what would happen if we didn't use the data abstractions. For example, here is our version of `+rat` built on top of the abstraction. Suppose instead we had directly implemented this procedure by relying on knowledge that rationals were constructed as pairs, thus directly using `cons, car` and `cdr`.

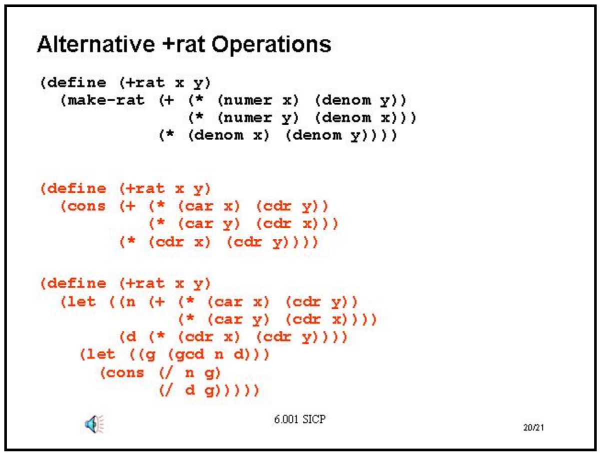
Slide 5.4.19
...then here is that implementation. It looks exactly like the original ; we have simply stripped off the abstraction barrier.



Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))


(define (+rat x y)
  (cons (+ (* (car x) (cdr y))
           (* (car y) (cdr x)))
        (* (cdr x) (cdr y))))
```

6.001 SICP 19/21



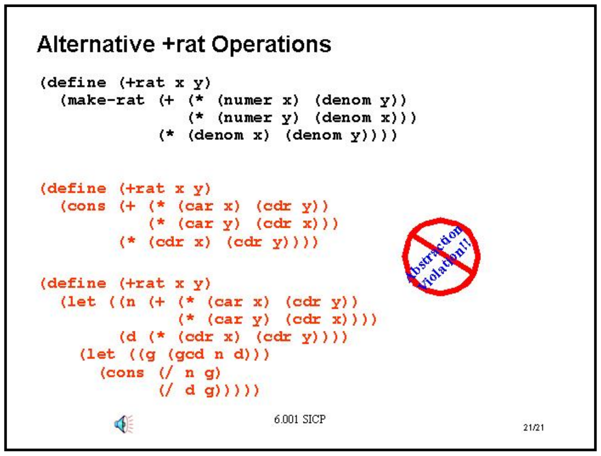
Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (+rat x y)
  (cons (+ (* (car x) (cdr y))
           (* (car y) (cdr x)))
        (* (cdr x) (cdr y))))

(define (+rat x y)
  (let ((n (+ (* (car x) (cdr y))
              (* (car y) (cdr x))))
        (d (* (cdr x) (cdr y))))
    (let ((g (gcd n d)))
      (cons (/ n g)
            (/ d g)))))
```

6.001 SICP 20/21

Slide 5.4.20
But now, suppose we decide to implement the idea of incorporating the reduction by the **gcd** into the creation of rationals. In this case, we have to somehow figure out which parts of this code correspond to the numerator and denominator and replacing them with this grungy looking code. And it is not just the amount of work that is a problem here, it is that we have to do this with every procedure that manipulates rationals, and we have to do it in a way that guarantees we correctly identify the pieces to change. In the previous case, we just had to change the interface to the abstraction, not all the procedures that used them.

Slide 5.4.21
Thus the key point is the following: By isolating the details of an abstraction from the use of that abstraction, and by maintaining a discipline in which we never violate that abstraction barrier, we can cleanly separate changes to the implementation from use of the implementation. Thus, instead of having to change every procedure that uses rationals, we can obtain different systems just by changing the underlying representation and ensuring that the constructor and selectors (the only procedures that cross the abstraction barrier) satisfy the abstraction's contract.



Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

(define (+rat x y)
  (cons (+ (* (car x) (cdr y))
           (* (car y) (cdr x)))
        (* (cdr x) (cdr y))))

(define (+rat x y)
  (let ((n (+ (* (car x) (cdr y))
              (* (car y) (cdr x))))
        (d (* (cdr x) (cdr y))))
    (let ((g (gcd n d)))
      (cons (/ n g)
            (/ d g)))))
```

Abstraction Violation!!

6.001 SICP 21/21