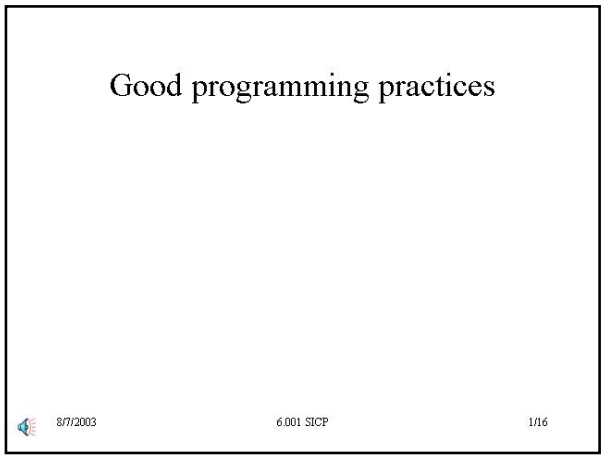
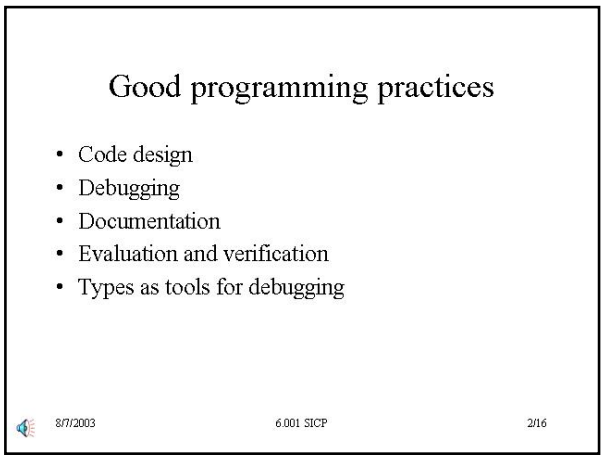
6.001 Notes: Section 7.1

Slide 7.1.1



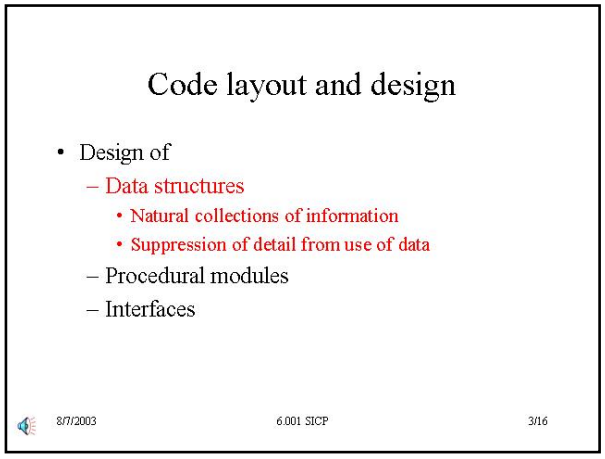
In the past few lectures, we have seen a series of tools for helping us create procedures to compute a variety of computational processes. Before we move on to more complex issues in computation, it is useful to step back and look at more general issues in the process of creating procedures.
In particular, we want to spend a little bit of time talking about **good programming practices**. This sounds a little bit like lecturing about "motherhood and apple pie", that is, a bit like talking about things that seem obvious, apparent, and boring in that everyone understands and accepts them. However, it is surprising how many "experienced" programmers don't execute good programming practices, and we want to get you started on the right track.



Slide 7.1.2

Thus, in this lecture we are going to look briefly at several methodological aspects of creating procedures: designing the components of our code, debugging our code when it doesn't run correctly, writing documentation for our code, and testing our code. We will highlight some standard practices for each stage, and indicate why these practices lead to efficient and effective generation of code.

Slide 7.1.3

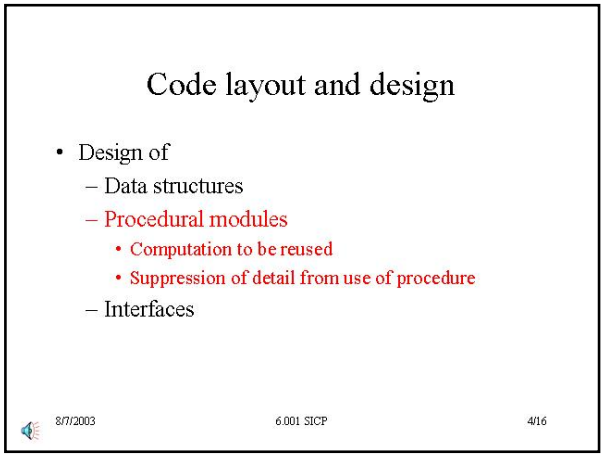


Let's start with the issue of how to design code, given a problem statement. There are many ways to do this, but most of them involve some combination of the following steps:

- Design of data structures
- Design of computational modules
- Design of interfaces between modules

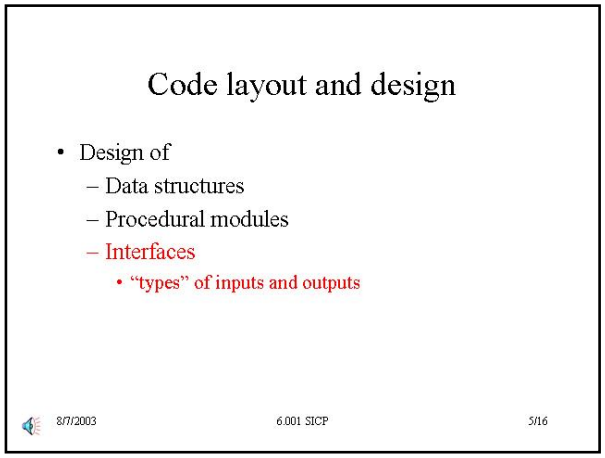
Once we have laid out the general design of these stages, we follow by creating specific instantiations of the actual components. We have not yet talked about data structures in Scheme, and will return to this issue in a few lectures. For our purposes here, the key thing to note is that when designing a computational system, it is extremely valuable to decide what kinds of information naturally should be grouped together, and to then create structures that perform that grouping, while maintaining interfaces to the structures that hide the details. For example, one thinks naturally of a vector as a pairing of an *x* and *y* coordinate. One wants to

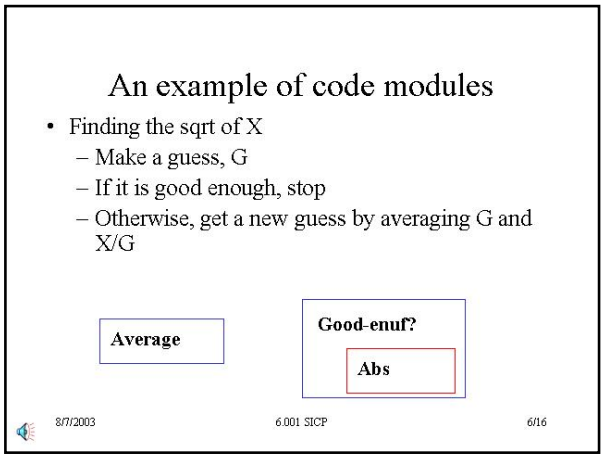
be able to get out the coordinates when needed, but in many cases, one thinks naturally of manipulating a vector as a unit. Similarly, one can imagine aggregating together a set of vectors, to form a polygon, and again one can think of manipulating the polygon as a unit. Thus, a key stage in designing a computational system is determining the natural data structures of the system.



Slide 7.1.4
A second stage in designing a computational system is deciding how best to break the computation into modules or pieces. This is often as much art as science, but there are some general guidelines that help us separate out modules in our design. For example, is there part of the problem that defines a computation that is likely to be used many times? Are there parts of the problem that can be conceptualized in terms of their behavior, e.g. how they convert certain inputs into certain types of outputs, without worrying about the details of how that is done. Does this help us focus on other parts of the computation? Or said a bit differently, can one identify parts of the computation in terms of their role, and think about that role in the overall computation, without having to know details of the computation?
If one can, these parts of the computation are good candidates for separate modules, since we can focus on their use while ignoring the details of how they achieve that computation.

Slide 7.1.5
Finally, given that one can identify data structures, whose information is to be manipulated; and stages of computation, in which that information is transformed; one wants to decide the overall flow of information between the modules. What types of inputs does each module need? What types of data does each module return? How does one ensure that the correct types are provided, in the correct order?
These kinds of questions need to be addressed in designing the overall flow between the computational modules.





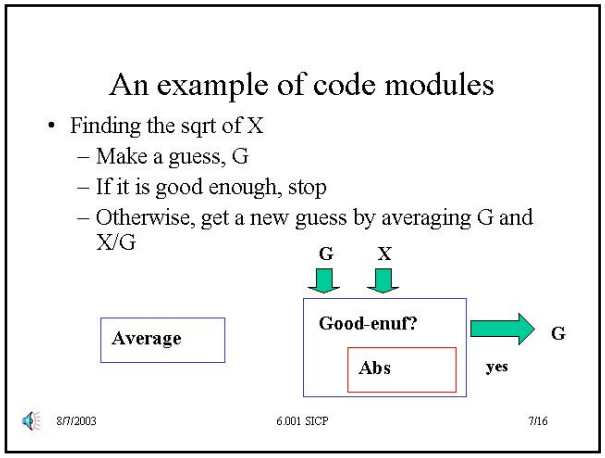
Slide 7.1.6
This is perhaps more easily seen by thinking about an example – and in fact you have already seen one such example, our implementation of `sqrt`. When we implemented our method for square roots, we actually engaged in many of these stages. We didn't worry about data structures, since we were simply interested in numbers. We did, however, spend some effort in separating out modules. Remember our basic computation: we start with a guess; if it is good enough, we stop; otherwise we make a new guess by averaging the current guess, and the ratio of the target number and the guess, and continue.
To design this system, we separated out several modules: the notion of averaging, the notion of measuring "good enough". We saw that some of these modules might themselves rely on other procedural abstractions; for example,

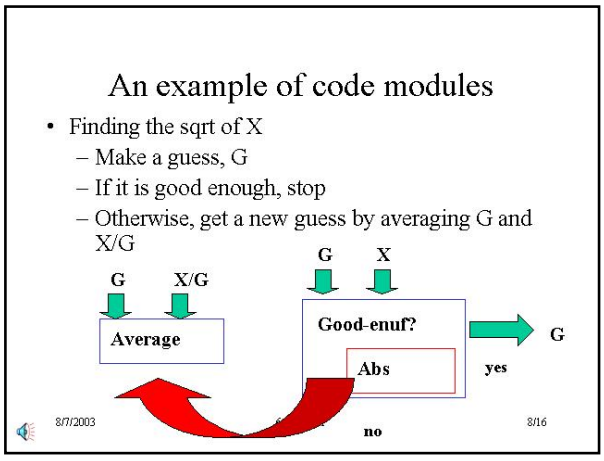
our particular version of “good enough” needed to use the absolute value procedure, though other versions might not.

Slide 7.1.7

Once we had separated out these notions of different computations: *average* and *good-enough*, we considered the overall flow of information through the modules. Note by the way that we can consider each of theses processes as a black box abstraction, meaning that we can focus on using these procedures without having to have already designed the specific implementation of each.

Now what about the flow between these modules? In our case, we began with a guess, and tested to see if it was good enough. If it was, we could then stop, and just return the value of the guess.





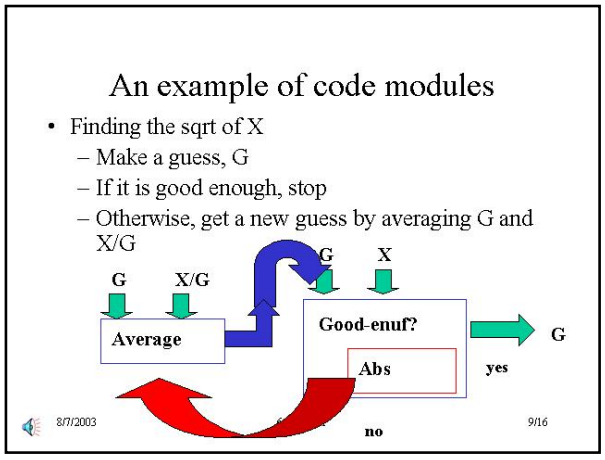
Slide 7.1.8

If it was not, then we needed to average the current guess and the ratio of our target number to the guess.

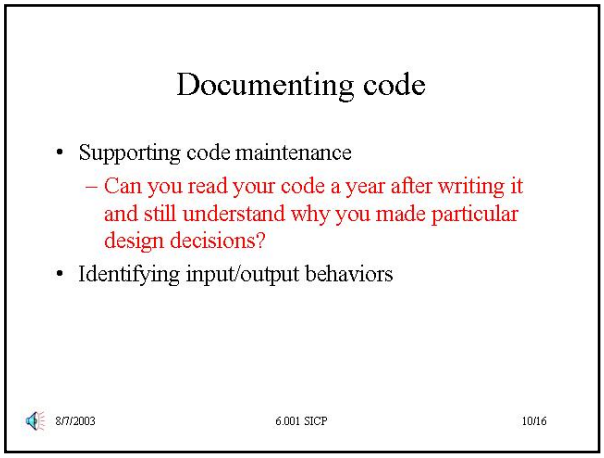
Slide 7.1.9

And then we need to repeat the entire process, with this new value as our new guess.

The point of laying out these modules, or black boxes, is that we can use them to decide how to divide up the code, and how to isolate details of a procedure from its use. As we saw when we implemented our `sqrt` procedure, we can change details of a procedure, such as `average`, without having to change any of the procedures that use that particular component. As well, the flow of information between the modules helps guide us in the creation of the overall set of procedures.



Thus, when faced with any new computational problem, we want to try to engage in the same exercise: block out chunks of the computation that can be easily isolated; identify the inputs and outputs from each chunk; and lay out the overall flow of information through the system. Then we can turn to implementing each of the units separately, and testing the entire system while isolating the effects of each unit.



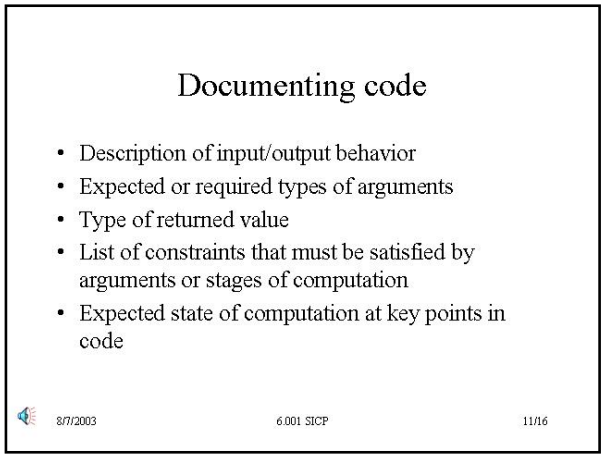
Slide 7.1.10

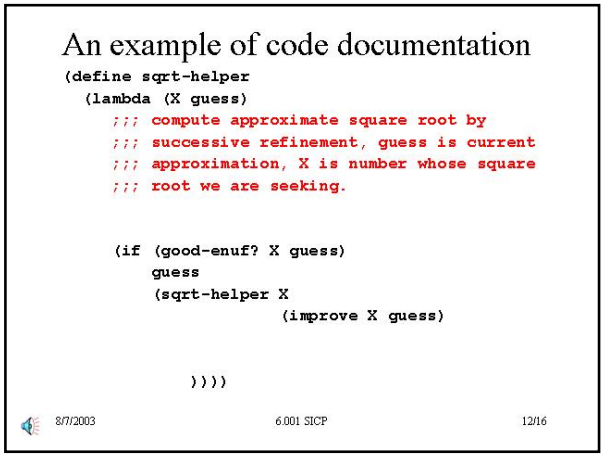
A second key element to good programming practice is code documentation. Unfortunately, this is one of the least well-practiced elements – far too often programmers are in such a hurry to get things written that they skip by the documentation stage. While this may seem reasonable at the time of code creation, when the design choices are fresh in the program creator's mind, six months later when one is trying to read the code (even one's own), it may be very difficult to reconstruct why certain choices were made. Indeed, in many commercial programming settings, more time is spent on code maintenance and modification than on code generation, yet without good documentation it can be very difficult or inefficient to understand existing code and change it.

As well, good documentation can serve as a valuable source of information about the behavior of each module, enabling a programmer to maintain the isolation of the details of the procedural abstraction from the use of that abstraction. This information can be of help when debugging procedures.

Slide 7.1.11

As with designing procedural modules, the creation of good documentation is as much art as science. Nonetheless, here are some standard elements of well-documented code. We are going to illustrate each of these with an example.



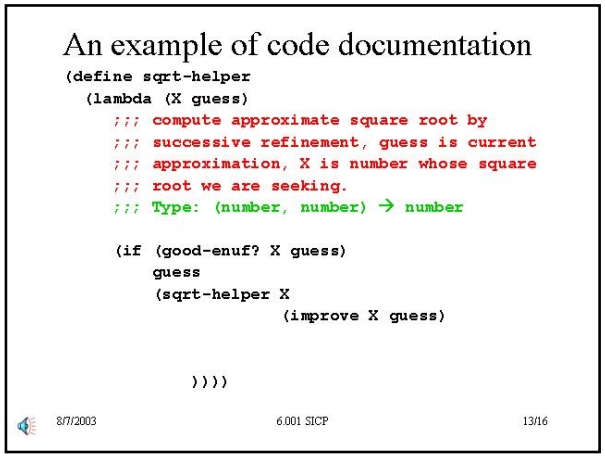


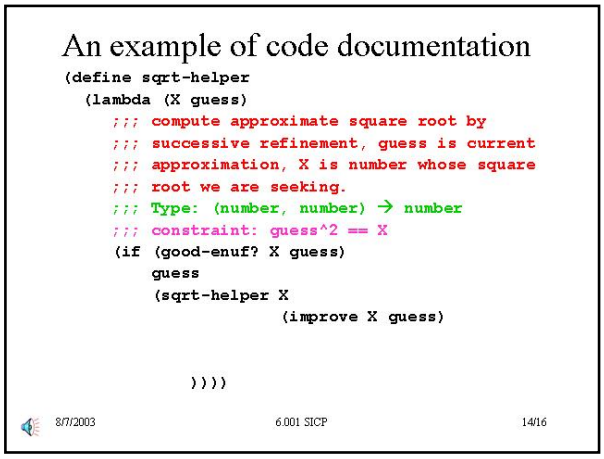
Slide 7.1.12

First, describe the goal of the procedure. Is it intended to part of some other computation (as this helper function is)? If so, what is the rough description of the process? Note that here we have been a bit cryptic (in order to fit things on the slide) and we might well want to say more about "successive refinement" (though we could defer that to the documentation under the `improve` procedure). We also identify the role of each argument to the procedure.

Slide 7.1.13

Second, describe the types of values used in the computation. In this case, the inputs or parameters are both numbers, and the returned value is also a number. Actually, if we were more careful here, we would require that X be a positive number, and we would place a check somewhere to ensure that this is true.



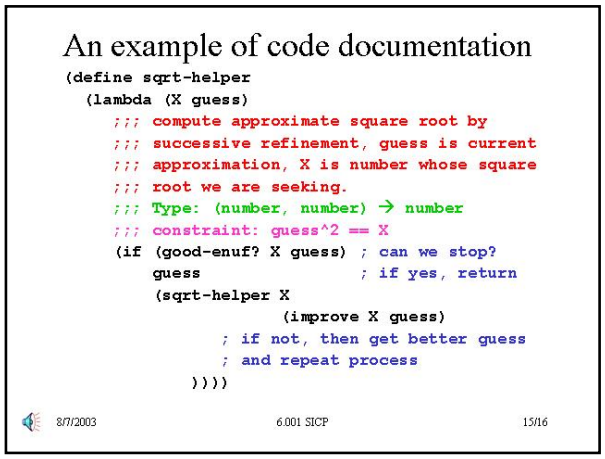


Slide 7.1.14

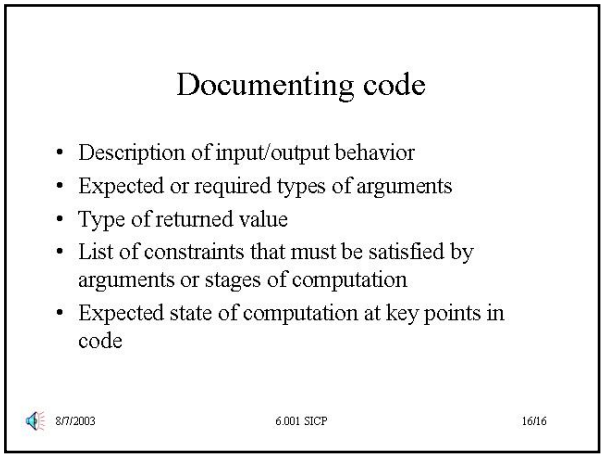
Third, describe constraints, either desired or required, on the computation. Here, we know that squaring the guess should get us something close to the target value, although we really don't guarantee this until we reach the termination stage.

Slide 7.1.15

And fourth, describe the expected state of the computation and the goal at each stage in the process. For example, here we indicate what `good-enuf?` should do, namely test if our approximation is sufficiently accurate. Then we indicate that if this is the case, we can stop and what value to return to satisfy the contract of the entire procedure. And we indicate how to continue the process, though we could probably say a bit more about what `improve` should do.



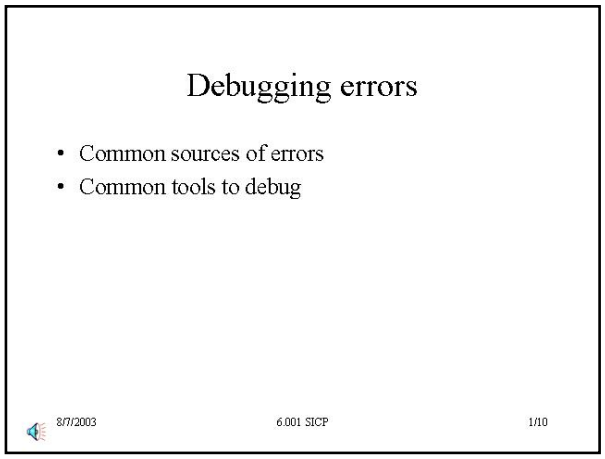
Notice how we can use the documentation to check some aspects of our procedure's "contract". Here, we have indicated that the procedure should return a number. By examining the `if` expression, we can see that in the consequent clause, if the input parameter `guess` is a number, then we are guaranteed to return a number. For the alternative clause, we can use induction to reason that given numbers as input, we also return a number, and hence the entire procedure returns a value of the correct type.

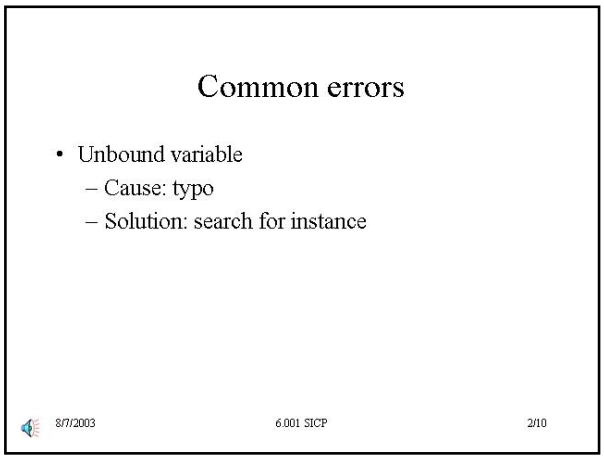


Slide 7.1.16
In general, taking care to meet each of the stages when you create code will often ensure an easier time when you have to refine or replace code. Getting into the habit of doing this every time you write something, even if you are only minutes away from some problem set deadline, will greatly improve your productivity!

6.001 Notes: Section 7.2

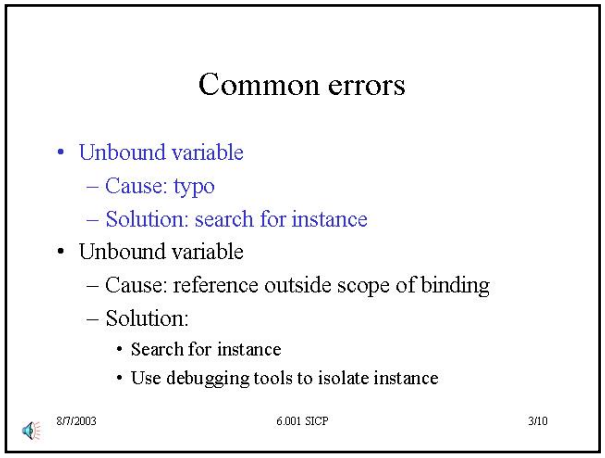
Slide 7.2.1
While we would like to believe that the code we write will always run correctly, the first time we try it, experience shows that this is a fortunate happenstance. Typically, especially with complex code, things will not work right, and we need to debug our code. Debugging is in part an acquired skill – with lots of practice you will develop your own preferred approach. Here, we are going to describe some of the common sources of errors in code, and standard tools for finding the causes of the errors and fixing them.



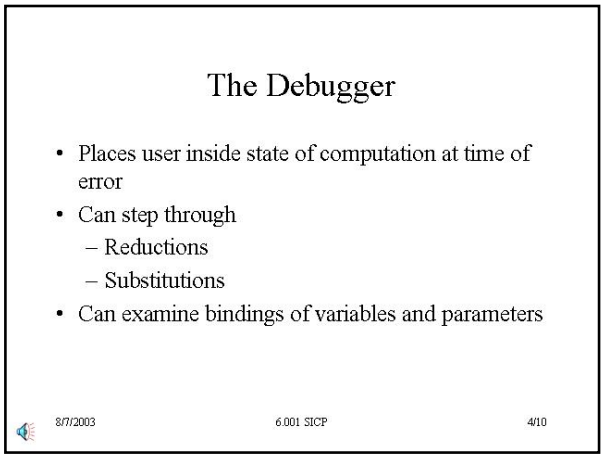


Slide 7.2.2
A common and simple bug in code arises when we use an **unbound variable.** From the perspective of Scheme, this means that somewhere in our code we try to reference (or look up the value of) a variable that does not have one. This can occur for several reasons. The simplest is that we mistyped – a spelling error. The solution in this case is pretty straightforward – simply search through the code file using editor tools to find the offending instance and correct it.

Slide 7.2.3



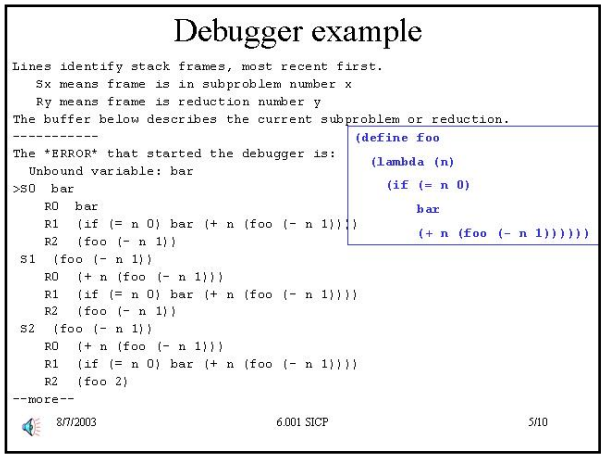
Sometimes, however, we are using a legal variable (that is, one that we intended to hold some value) but the evaluator still complains that this variable is unbound. How can that be? Remember that in Scheme a variable gets bound to a value in one of several ways. We may define it at "top level", that is, we may directly tell the interpreter to give a variable some value. We may define it internally within some procedure. Or, we may use it as a formal parameter of a procedure, in which case it gets locally bound to a value when the procedure is applied. In the last two cases, if we attempt to reference the variable outside the scope of the binding, that is, somewhere outside the bounds of the lambda expression in which the variable is being used, we will get an unbound variable error. This means that we have tried to use a variable outside its legal domain, and we need to correct this. This probably means we have a coding error, but we can isolate the problem either by searching for instances of the variable in the code file, or by using the debugger.



Slide 7.2.4

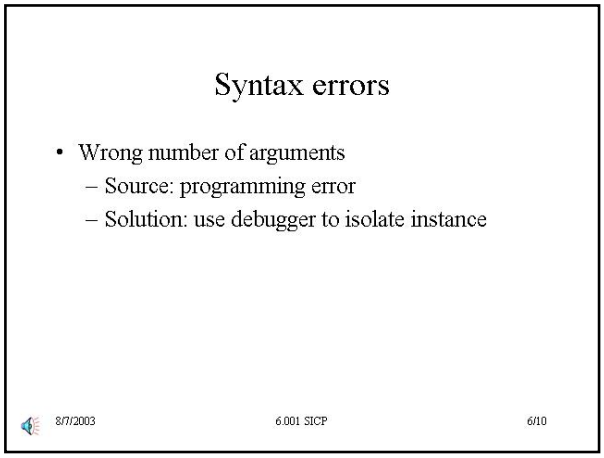
So what does a debugger do to help us find errors? Each programming language will have its own flavor of debugger; for an interpreted language like Scheme, the debugger actually places us inside the state of the computation. That is, when an error occurs, the debugger provides us access to the state of the computation at the time of the error, including access to the values of the variables within the computation. Moreover, we can step around inside the environment of the computation: we can work back up the chain of computational steps, examining what values were produced during **reductions** (where computation is reduced to a simpler expression), and examining what values were produced during **substitutions** (where the computation was converted to a simpler version of itself).

Slide 7.2.5



For example, here is a simple procedure, which we have called with argument 2. Notice what happens when we hit the unbound variable error and enter the debugger. We are placed at the spot in the computation at which the error occurred. If we choose to step back through the chain of evaluations, we can see what expressions were reduced to get to this point, and what recursive versions of the same problem were invoked in reaching this stage.

In this case, we note that `foo` was initially called with argument 2, and after a reduction through an `if` expression, we arrived at an expression that contained within it a simpler version of the same problem. This reduction stage repeated again, until we apparently reached the base case of the `if` expression, where we hit the unbound variable. We can see in this simple case that our unbound error is coming from within the body of `foo` and is in the base case of the decision process.

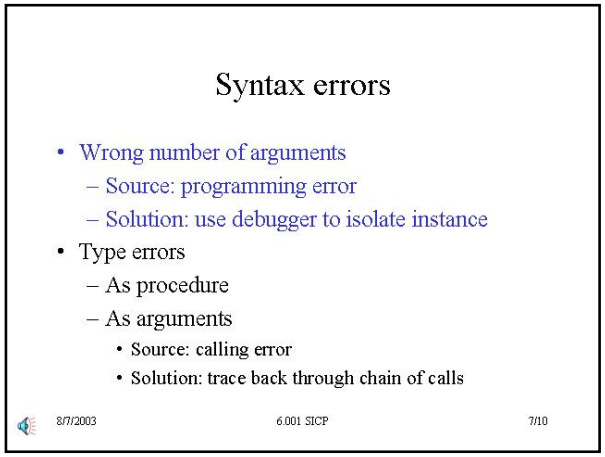


Slide 7.2.6

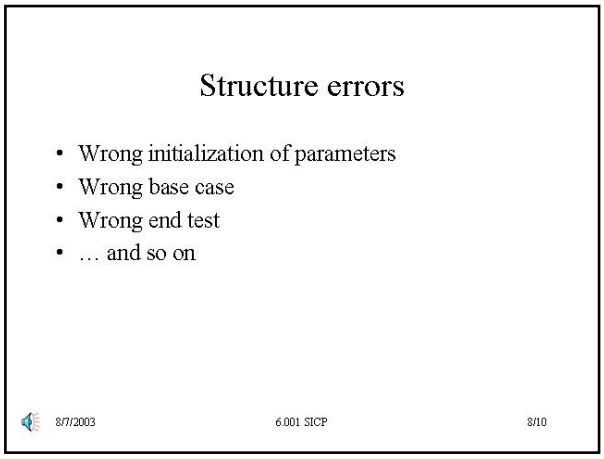
A second class of errors deals with mistakes in syntax – creating expressions that do not satisfy the programming language's rules for creating legal expressions. A simple one of these is an expression in which the wrong number of arguments is provided to the procedure. If this occurs while attempting to evaluate the offending expression, we will usually be thrown into the debugger – a system intended to help us determine the source of the error. In Scheme, the debugger provides us with information about the environment in which the offending expression occurred. It supplies tools for examining the values associated with variable names, and for examining the sequence of expressions that have been evaluated leading up to this error. By stepping through the frames of the debugger, we can often isolate where in our code the incorrect expression resides.

Slide 7.2.7

A more insidious syntax error occurs when we use an expression of the wrong type somewhere in our code. If we use an expression whose value is not a procedure as the first subexpression of a combination, we will get an error that indicates we have tried to apply a non-procedure object. As before, the debugger can often help us isolate the location of this error, though it may not provide much insight into why an incorrect object was used as a procedure. For that, we may have to trace back through our code, to determine how this value was supplied to the offending expression.



The harder error to isolate is one in which one of the argument expressions to a combination is of the wrong type. The reason this is harder to track down is that the cause of the creation of an incorrect object type may have occurred far upstream, that is, some other part of our code may have created an incorrect object, which has been passed through several levels of procedure calls before causing an error. Tracking down the original source of this error can be difficult, as we need to chase our way back through the sequence of expression evaluations to find where we accidentally created the wrong type of argument.

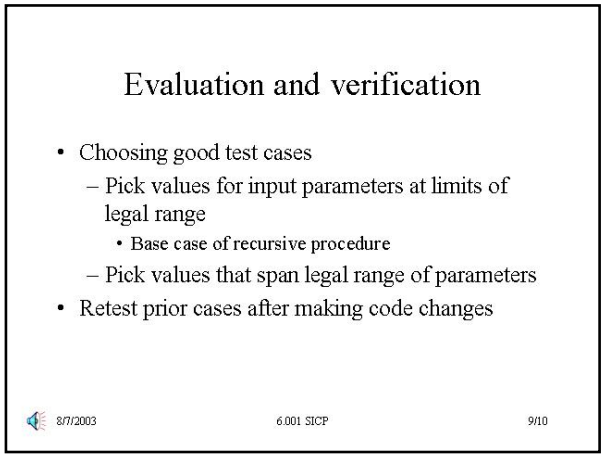


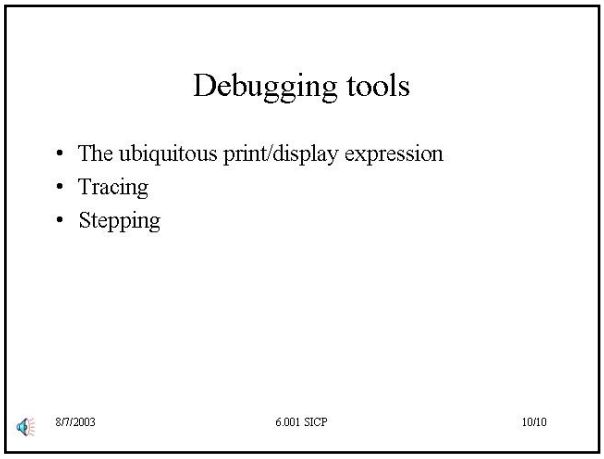
Slide 7.2.8

The most common sorts of errors, though, are structural ones. This means that our code is syntactically valid – composed of correctly phrased expressions, but the code does not compute what we intended, because we have made an error somewhere in the code design. This could be for a variety of reasons: we started a recursive process with the wrong initial values, or we are ending at the wrong place, or we are updating parameters incorrectly, or we are using the wrong procedure somewhere, and so on. Finding these errors is tougher, since the code may run without causing a language error, but the results we get are erroneous.

Slide 7.2.9

This is where having good test cases is important. For example, when testing a recursive procedure, it is valuable to try it using the base case values of the parameters, to ensure that the procedure is terminating at the right place, and returning the right value. It is also valuable to select input parameter values that sample or span the range of legal values – does it work with small values, with large values; does changing the input value by a small increment cause the expected change in output value?





Slide 7.2.10

And what do we do if we find we have one of these structure errors? Well, our goal is to isolate the location of our misconception within the code, and to do this, there are two standard tools.

The most common one is to use a **print** or **display** expression – that is, to insert into our code, expressions that will print out for us useful information at different stages of the computation. For example, we might insert a display expression within the recursive loop of a procedure, which will print out information about the values of parameters. This will allow us to check that parameters are being updated correctly, and that end cases are correctly seeking the right termination point. We might similarly print out the values of intermediate computations within recursive loops, again to ascertain that the computation is operating with the values we expect, and is computing the values we expect.

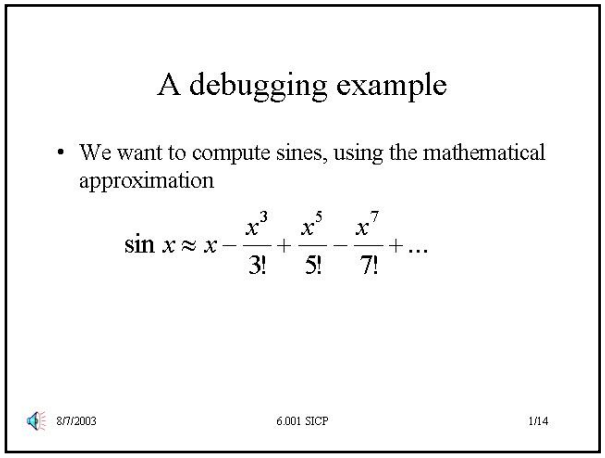
A related tool, supplied for example with Scheme, is a **tracer**. This allows us to ask the evaluator to inform us about the calling conventions of procedures – that is, to print out the values of the parameters supplied before each application of the procedure we designate, and the value returned by each such procedure call. This is similar to our use of display expressions, but is handled automatically for us. It applies only to parameters of procedure calls, however, so that if we want to examine for detailed states of the computation, we need to fall back on the **display** tactic.

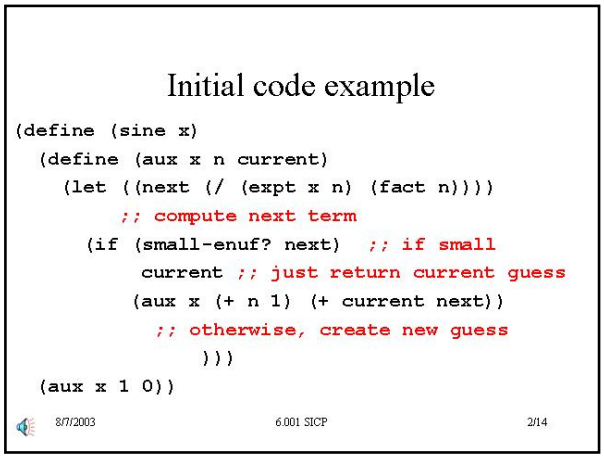
In some cases, it may help to actually walk through the substitution model, that is, to see each step of the evaluation. Many languages, including Scheme, provide a means for doing this – in our case called the **stepper**. This is a mechanism that lets us control each step of the substitution model in the evaluation of the expression. It is obviously tedious, but works best when we need to isolate a very specific spot at which an error is occurring, and we don't want to insert a ton of display expressions.

Perhaps the best way to see the role of these tools is to look at an example, which we do next.

6.001 Notes: Section 7.3

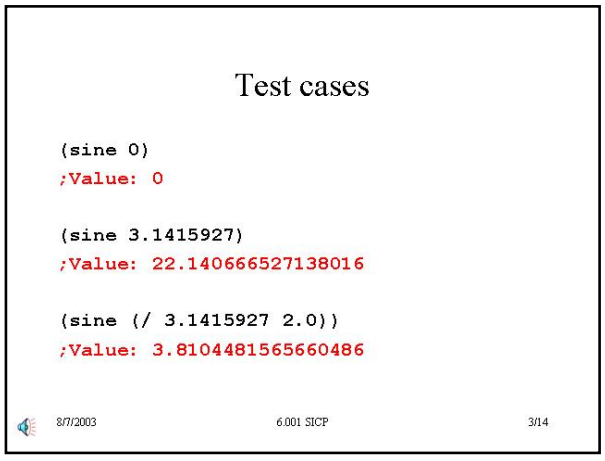
Slide 7.3.1
Let's use an example of a debugging session to highlight these ideas. This will be primarily to fix a structural error, but we will see how the other tools come into play as we do this. Suppose we want to compute an approximation to the *sine* function. Here is a mathematical approximation that will give us a pretty good solution. So let's try coding this up.

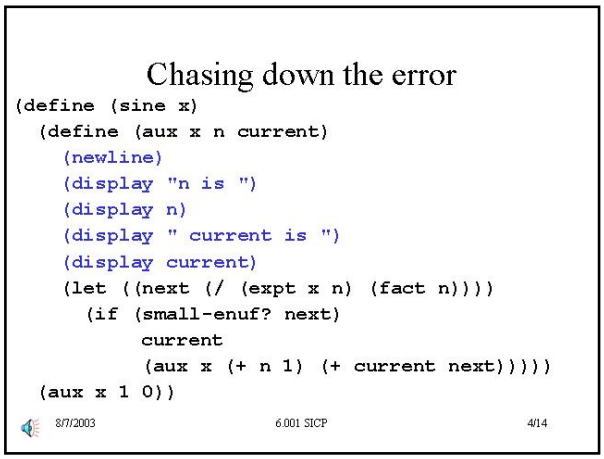




Slide 7.3.2
So here is a first attempt at some code to do this. We will assume that `fact` and `small-enuf?` already exist. The basic idea behind this procedure is quite similar to what we did for square roots. We start with a guess. We then see how to improve the guess, in this case by computing the next term in the approximation, which we would like to add in. If this improvement is small enough, we are done and can return the desired value. If not, we repeat the process with a better guess, by adding in the improvement to the current guess.

Slide 7.3.3
Now, let's try it out on some test cases. One nice test case is the base case, of x equal to 0. That clearly works. Another nice test case is when x is equal to pi, where we know the result should also be close to 0. Oops! That didn't work. Nor does the code work for x equal to pi half. Both of these latter cases give results that are much too large.

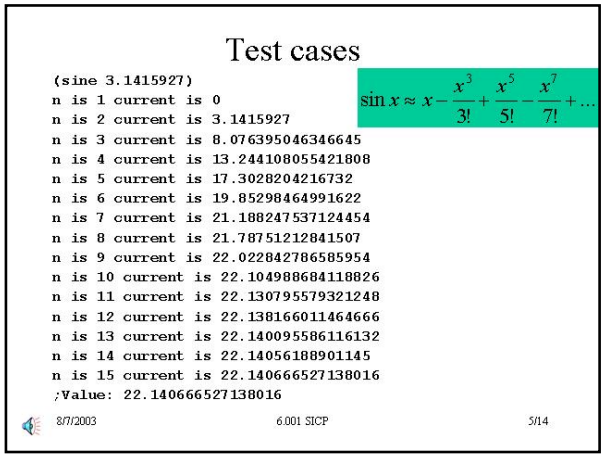


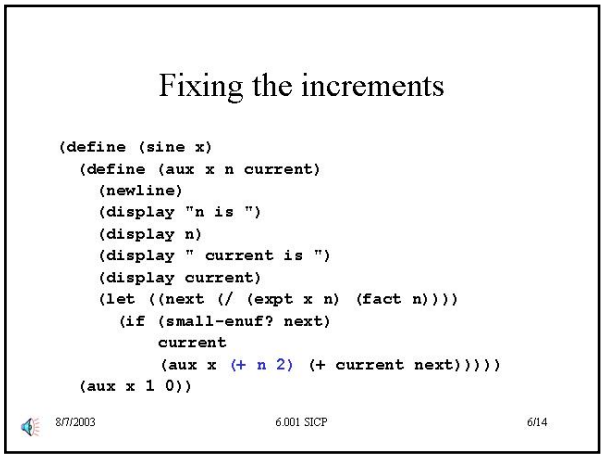


Slide 7.3.4
Okay, we need to figure out where our conceptual error lies. Let's try to isolate this by tracing through the computation. In particular, we will add some **display** expressions that will show us the state of the computation each time through the recursion.

Slide 7.3.5

And let's try this again. Here we have used the test case of x equal to pi. And we can see the trace of the computation. If we compare this to the mathematical equation we can see one problem. We really only want terms where n is odd, but clearly we are getting all terms for n. So we need to fix this. Most likely this is because we are not changing our parameters properly.





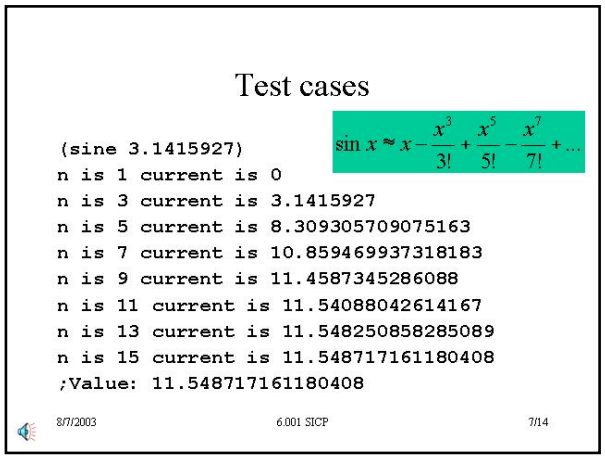
Slide 7.3.6

So here is the correction. We will need to increment our parameter by 2 each time, not by 1 – an easy mistake to make, and to miss!

Slide 7.3.7

So let's try this again. Hmm. We have gotten better as we are only computing the odd terms for n, but we are still not right. If we look again at the mathematical equation, we can see that we should be alternating signs on each term. Or said another way, the successive approximations should go up, then down, then up, then down, and so on. Note that we could have also spotted this if we had chosen to display the value of `next` at each step.

So we need to keep track of some additional information, in this case whether the term should be added or subtracted from the current guess.



```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
               (+ n 2)
               (+ current (* addit next))
               (* addit -1)))))
  (aux x 1 0))
```

8/7/2003 6.001 SICP 8/14

Slide 7.3.8

Well, we can handle that. We add another parameter to our helper procedure, which keeps track of whether to add the term (if the value is 1) or whether to subtract the term (if the value is -1). And of course we will need to change how we update the guess, and how we update the value of this parameter.

Slide 7.3.9

Oops! We blew it somewhere! We could enter the debugger to locate the problem, but we can already guess that since we changed the aux procedure, that must be the cause.

```
(sine 3.1415927)
;The procedure #[compound-procedure 12 aux] has
  been called with 3 arguments; it requires
  exactly 4 arguments.
;Type D to debug error, Q to quit back to REP
  loop: q
```

8/7/2003 6.001 SICP 9/14

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ") (display n)
    (display " current is ") (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x
               (+ n 2)
               (+ current (* addit next))
               (* addit -1)))))
  (aux x 1 0 -1))
```

8/7/2003 6.001 SICP 10/14

Slide 7.3.10

And clearly the solution is to make sure we call this procedure with the right number of arguments. Notice that in this case it is easy to spot this error, but in general, we should get into the habit of checking all calls to a procedure when we alter its set of parameters.

Slide 7.3.11

Now, if we try this on the test case of x equal pi, this works! But if we try it on the test case of pi half, it doesn't! The answer should be close to 1, but we are getting something close to -1. Note that this reinforces why we want to try a range of test cases – if we had stopped with x equal pi, we would not have spotted this problem.

```
(sine 3.1415927)
n is 1 current is 0
n is 3 current is -3.1415927
n is 5 current is 2.026120309075164
n is 7 current is -.5240439191678563
n is 9 current is .07522067212275974
n is 11 current is -6.925225410112354e-3
n is 13 current is 4.452067333052508e-4
;Value: 4.452067333052508e-4

(sine (/ 3.1415927 2.0))
n is 1 current is 0
n is 3 current is -1.57079635
n is 5 current is -.9248322238656045
n is 7 current is -1.004524855998199
n is 9 current is -.999843101378741
;Value: -.999843101378741
```

8/7/2003 6.001 SICP 11/14

```
(define (sine x)
  (define (aux x n current addit)
    (newline)
    (display "n is ")
    (display n)
    (display " current is ")
    (display current)
    (let ((next (/ (expt x n) (fact n))))
      (if (small-enuf? next)
          current
          (aux x (+ n 2)
               (+ current (* addit next)) (* addit -1)))))
  (aux x 1 0 1))
```

8/7/2003 6.001 SICP 12/14

Slide 7.3.12

Here is the bug. We started with the wrong initial value – a common error. By fixing this, we can try again and …

Slide 7.3.13

… finally we get correct performance. Note how we have used **printing** of values to isolate changes, as well as using the **debugger** to find syntax errors.

```
(sine (/ 3.1415927 2.0))
n is 1 current is 0
n is 3 current is 1.57079635
n is 5 current is .9248322238656045
n is 7 current is 1.004524855998199
n is 9 current is .999843101378741
;Value: .999843101378741
(sine 3.1415927)
n is 1 current is 0
n is 3 current is 3.1415927
n is 5 current is -2.026120309075164
n is 7 current is .5240439191678563
n is 9 current is -.07522067212275974
n is 11 current is 6.925225410112354e-3
n is 13 current is -4.452067333052508e-4
;Value: -4.452067333052508e-4
(sine 0)
n is 1 current is 0
;Value: 0
```

8/7/2003 6.001 SICP 13/14

Summary

- Display parameters to isolate errors
- Test cases to highlight errors
- Check range of test cases
- **Use these tricks and tools!**

8/7/2003 6.001 SICP 14/14

Slide 7.3.14

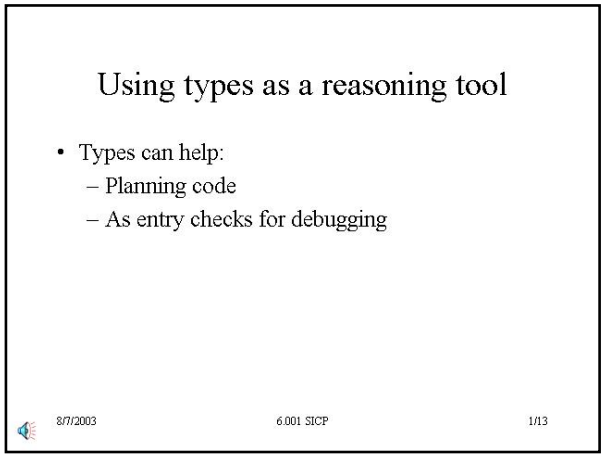
In general, we want you to get into the habit of doing the same things. Developing good programming methodology habits **now** will greatly help you when you have to deal with large, complex, bodies of code. Good programming discipline means being careful and thorough in the creation and refinement of code of all sizes and forms, so start exercising your "programming muscles" now!

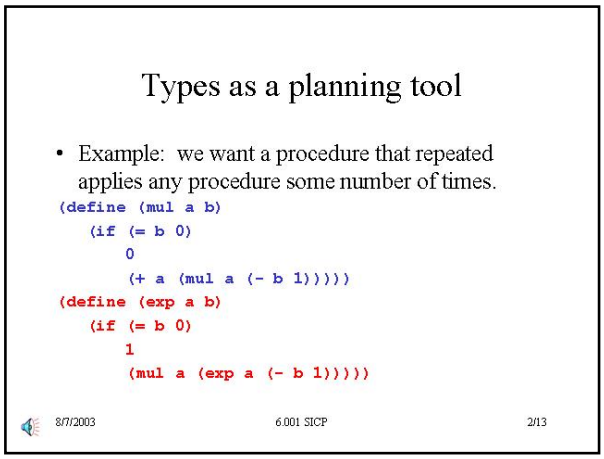
6.001 Notes: Section 7.4

Slide 7.4.1

One other tool that we have in our armamentarium of debugging is the use of **types**. In particular, type specifications, that is, constraints on what types of objects are passed as arguments to procedures and what types of objects are returned as values by procedures, can help us both in planning and designing code, and in debugging existing code.

Here, we are going to briefly explore both of these ideas, both to demonstrate why careful program practice can lead to efficient generation of robust code; and to illustrate why thinking about types of procedures and objects is a valuable practice.





Slide 7.4.2

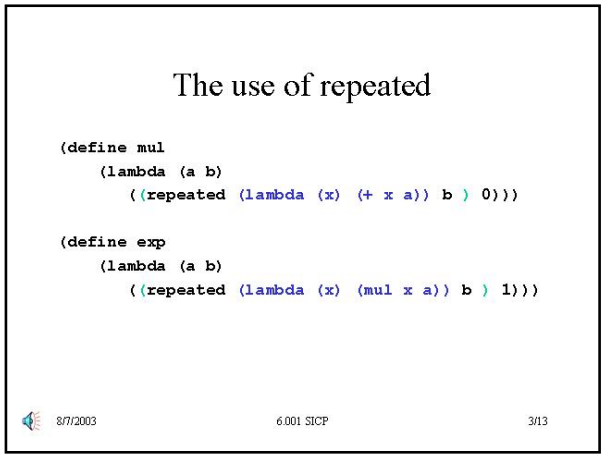
To motivate the idea of types as a tool in designing code, let's consider an example. Suppose we want to create a procedure, let's call it `repeated`, that will apply any other procedure some specified number of times. Since this is a vague description, let's look at a specific motivating example.

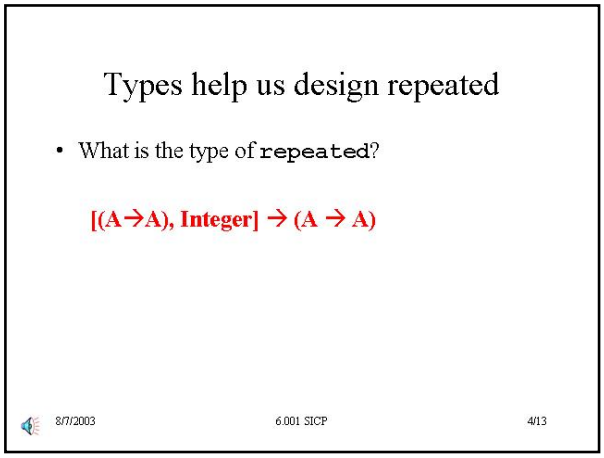
We saw earlier the idea that we could implement multiplication as a successive set of additions, and that we could implement exponentiation as a successive set of multiplications. If we look at these two procedures, we can see that there is a general pattern here. There is a base case value to return (0 in one case, 1 in the other). And there is the idea of applying an operation to an input value and the result of repeating that process one fewer times. `Repeated` is intended to capture that common pattern of operation.

Slide 7.4.3

So here is what we envision: we want our `repeated` procedure to take a procedure to repeat, and the number of times to repeat it. It should return a procedure that will actually do that, when applied to some value. Here we can see that the procedure being applied would change in each case, and the initial value to which to apply it would change, but otherwise the overall operation is the same.

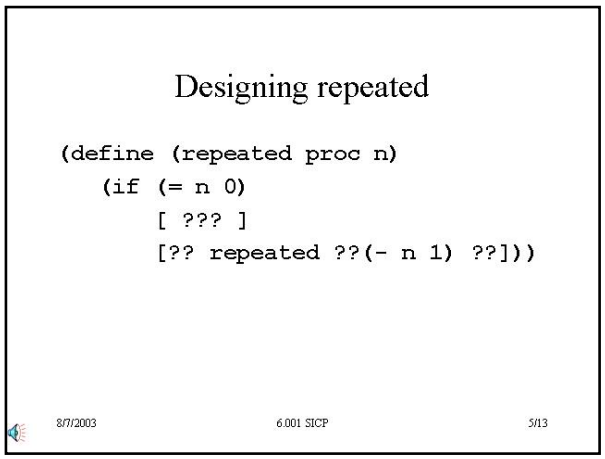
The question is: how to we create `repeated`, and why does the call to `repeated` have that funny structure, with two open parens?

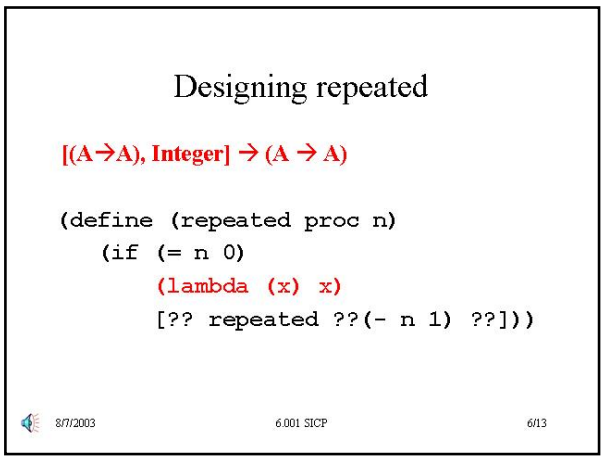




Slide 7.4.4
First, what is the type of `repeated`?
Well, from the previous slide, we know that as given, it should take two arguments. The first should be a procedure of one argument. We don't necessarily know what type of argument this procedure should take (in the two examples shown, the input was a number, but we might want to be more general than this). What we do know, is that whatever type of argument this procedure takes, it needs to return a value of the same type, since it is going to apply that procedure again to that value. Hence the first argument to `repeated` must be a procedure of type `A to A`.
The second argument to `repeated` must be an integer, since we can only apply an operation an integer number of times. Actually, it should probably be a non-negative integer.
And as we argued, the returned object needs to be a procedure of the same type: `A to A` because the idea is to use `repeated` recursively on itself.

Slide 7.4.5
Okay, now how does this help us in designing the actual procedure?
We know the rough form that `repeated` should take. It should have a test for the base case, which is when there are no more repetitions to make. In the base case, it needs to do something, which we have to figure out. And in the recursive case, we expect to use `repeated` to solve the smaller problem of repetition, plus some additional operations, which we need to figure out.





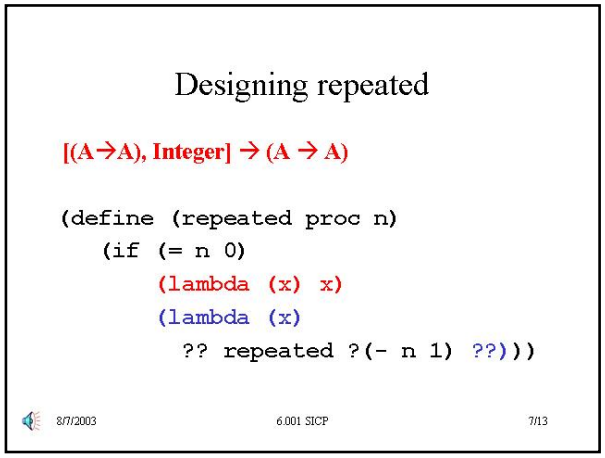
Slide 7.4.6
For the base case, what do we know?
We know that by the type information, this must return a procedure of a single argument that returns a value of the same type.
We also know that if we are in the base case, there is really nothing to do. We don't want to apply our procedure any more times. Hence, we can deduce that we need to return a procedure that serves as the **identity**: it simply returns whatever value was passed in.

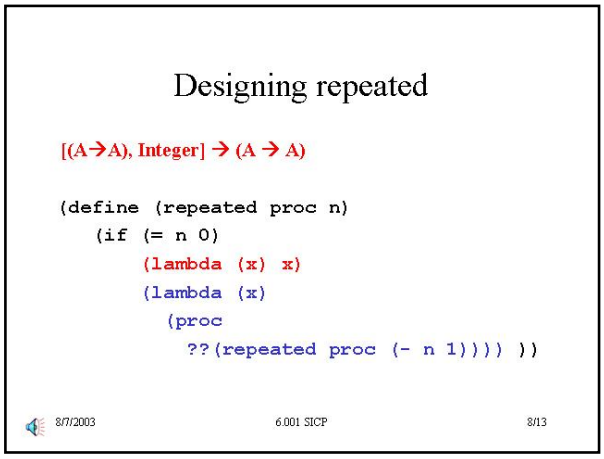
Slide 7.4.7

Now, what about the recursive case?

Well, the idea is to apply the input procedure to the result of repeating the operation `n-1` times. How do we use this idea to figure out the correct code?

First, we know that whatever we write must have type `A to A` by the specification of `repeated`.





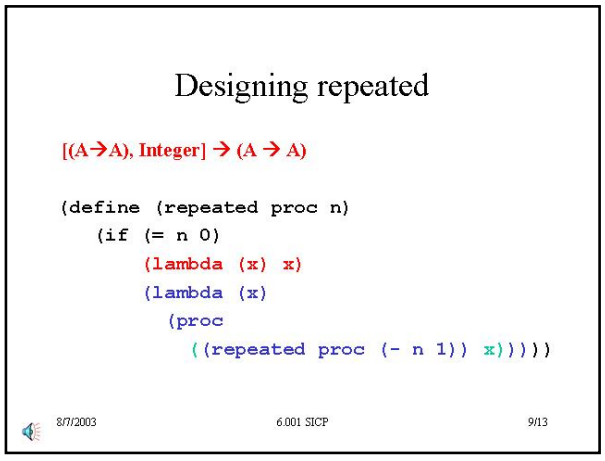
Slide 7.4.8

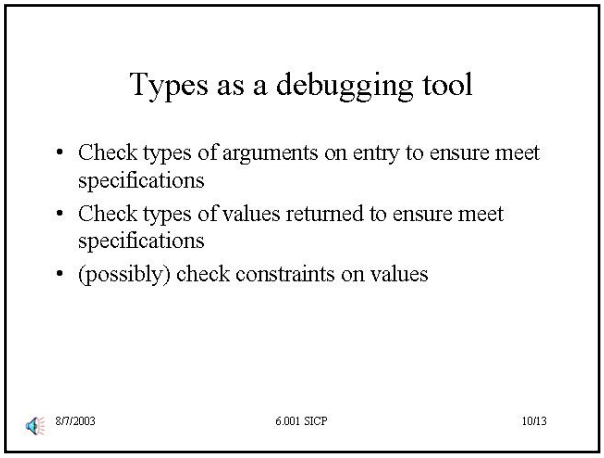
Next we know that we want to apply the input procedure `proc` to the result of solving the same problem, `n-1` times. So we ought to have something that has these pieces in it.

Slide 7.4.9

But let's check the types. We know that `repeated` has type `A to A`, and the `proc` expects only an argument of type `A`. So clearly we need to apply `repeated` to an argument before passing the result on to `proc`. Hence we have the form shown.

Note how this fairly complex piece of code can be easily deduced by using types of procedures to determine interactions. Of course, to be sure we did it right, we should now test this on some test cases, for example, by running `mul` or `exp` on known cases.



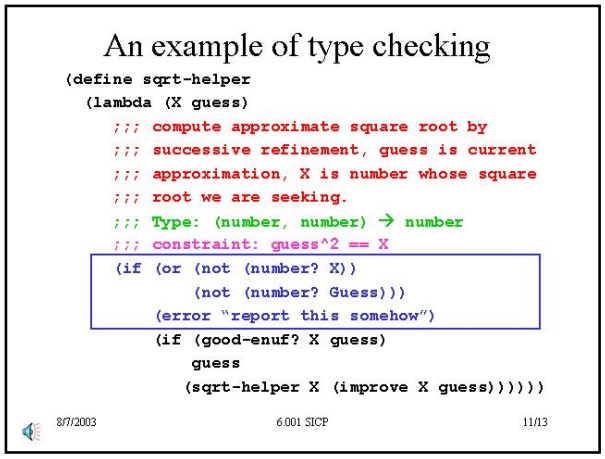


Slide 7.4.10

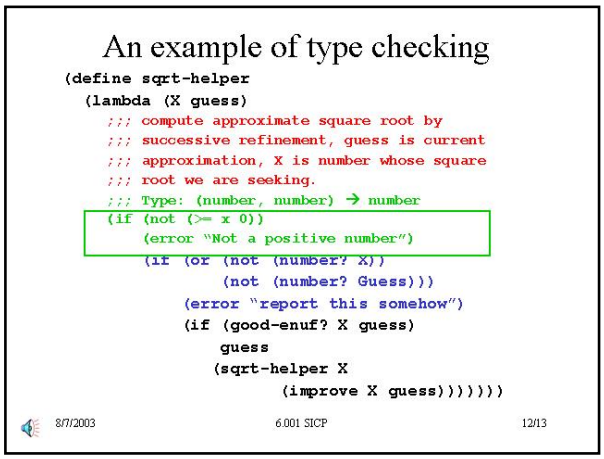
A second way that types can help us is in debugging code. In particular, we can use the information about types of arguments and types of return values explicitly to check that procedures are interacting correctly. And in some cases, where there are constraints on the actual values being returned, we can also enforce a check.

Slide 7.4.11

As an example, here is our `sqrt` code from before. One of the conditions we have is that the input arguments need to be numbers. And we could check that numbers are being correctly passed in by inserting an explicit check. In this case, it is probably redundant since the only code that calls `sqrt-helper` is itself, but in general, when multiple procedures might be involved, you can see how this check is valuable. Note that one can insert this check only when debugging, as a tool for deducing what procedure is incorrectly supplying arguments. But one can also use it regularly, if you want to ensure robust operation of the code.



Clearly one could add a check on the return value in a similar fashion.



Slide 7.4.12

But there are other things one could use to ensure correct operation. For example, the number whose square root we are seeking should be a positive number, and we could check that as shown.

Thus we see that types also serve as a useful tool on good programming methodologies.

Slide 7.4.13

To summarize, we have seen a set of tools for good programming practices: ways of designing code, debugging code, evaluating code, and using knowledge of code structure to guide the design.