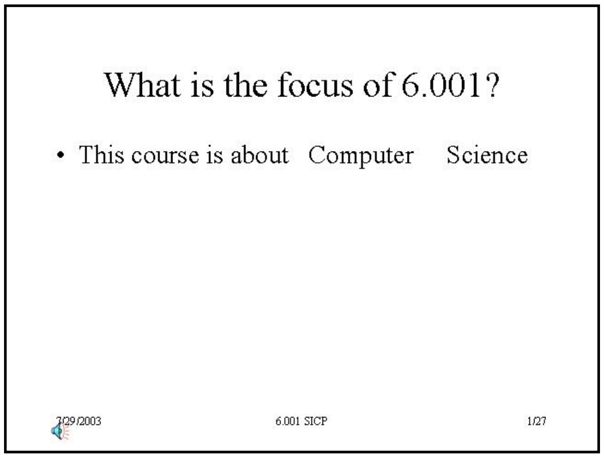
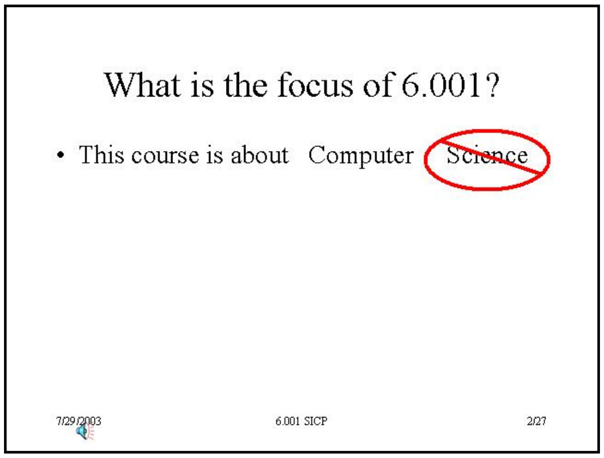
6.001 Notes: Section 1.1

Slide 1.1.1
This first thing we need to do is discuss the focus of 6.001. What is this course all about? This seems quite obvious -- this is a course about computer science. But we are going to claim in a rather strange way that this is not really true.


What is the focus of 6.001?

- This course is about Computer Science

7/29/2003 6.001 SICP 1/27

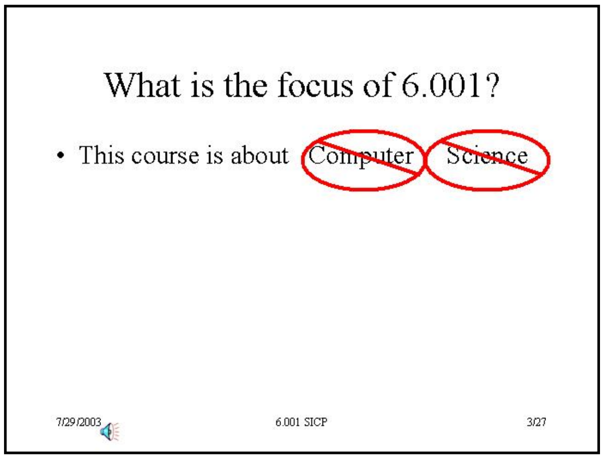

What is the focus of 6.001?

- This course is about Computer ~~Science~~

7/29/2003 6.001 SICP 2/27

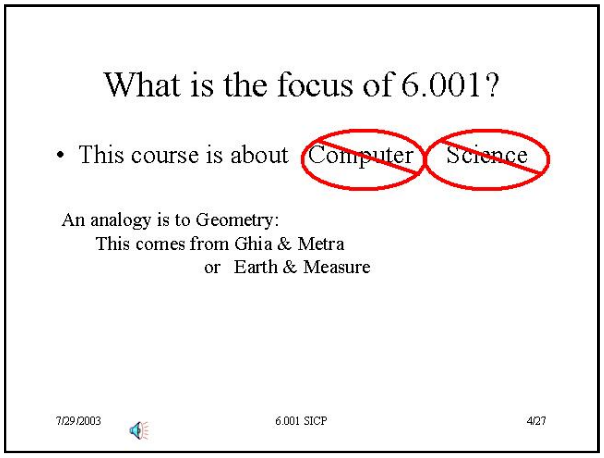
Slide 1.1.2
First of all, it is not really about science. It is really much more about engineering or art than it is about science.

Slide 1.1.3
...and it is not really about computers. Now that definitely sounds strange! But let me tell you why I claim it is not really about computers. I claim it is not really about computers in the same way that physics is not really just about particle accelerators, or biology is not really just about microscopes, or geometry is not really about surveying instruments.


What is the focus of 6.001?

- This course is about ~~Computer~~ ~~Science~~

7/29/2003 6.001 SICP 3/27



What is the focus of 6.001?

- This course is about ~~Computer~~ ~~Science~~

An analogy is to Geometry:
This comes from Ghia & Metra
or Earth & Measure

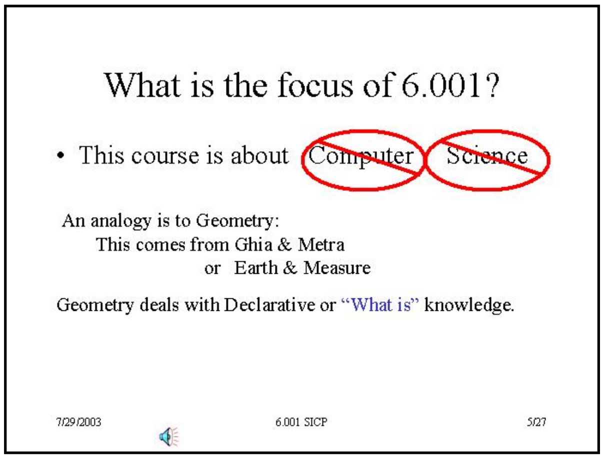
7/29/2003 6.001 SICP 4/27

Slide 1.1.4

In fact, geometry is a good analogy to use here. It has also a terrible name, which comes from two words: **GHIA** or earth, and **METRA** or measurement. And to the ancient Egyptians, that is exactly what geometry was -- instruments for measuring the earth, or surveying. Thousands of years ago, the Nile annually flooded, and eventually retreated, wiping out most of the identifying landmarks. It also deposited rich soil in its wake, making the land that it flooded very valuable, but also very hard to keep track of. As a consequence, the Egyptian priesthood had to arbitrate the restoration of land boundaries after the annual flooding. Since there were no landmarks, they needed a better way of determining boundaries, and they invented geometry, or earth measuring. Hence, to the Egyptians, geometry was surveying -- and about surveying instruments. This is a common effect. When a field is just getting started, it's easy to confuse the essence of the field with its tools, because we usually understand the tools much less well in the infancy of an area. In hindsight, we realize that the important essence of what the Egyptians did was to formalize the notions of space and time which later led to axiomatic methods for dealing with declarative, or **What Is** kinds of knowledge. --- So geometry not really about measuring devices, but rather about declarative knowledge.

Slide 1.1.5

So geometry is not really about surveying, it is actually fundamentally about axioms for dealing with a particular kind of knowledge, known as Declarative, or "what is true" knowledge.



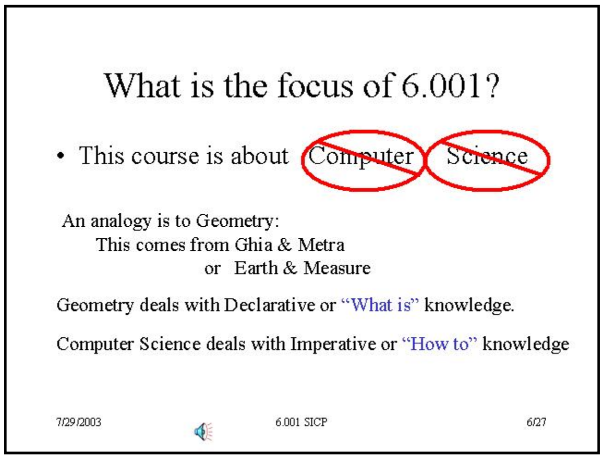
What is the focus of 6.001?

- This course is about ~~Computer~~ ~~Science~~

An analogy is to Geometry:
This comes from Ghia & Metra
or Earth & Measure

Geometry deals with Declarative or "What is" knowledge.

7/29/2003 6.001 SICP 5/27



What is the focus of 6.001?

- This course is about ~~Computer~~ ~~Science~~

An analogy is to Geometry:
This comes from Ghia & Metra
or Earth & Measure

Geometry deals with Declarative or "What is" knowledge.

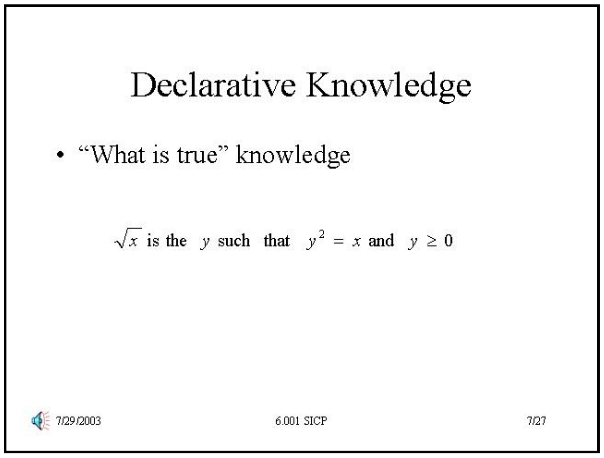
Computer Science deals with Imperative or "How to" knowledge

7/29/2003 6.001 SICP 6/27

Slide 1.1.6

By analogy to geometry, Computer Science is not really about computers -- it is not about the tool. It is actually about the kind of knowledge that computer science makes available to us. What we are going to see in this course is that computer science is dealing with a different kind of knowledge -- **Imperative** or "how to" knowledge. It is trying to capture the notion of a process that causes information to evolve from one state to another, and we want to see how we can uses methods to capture that knowledge.

Slide 1.1.7
First, what is declarative knowledge? It is knowledge that talks about what is true. It makes statements of fact that one can use to try to reason about things. For example, here is a statement of truth about square roots. It provides a definition of a square root. As a consequence if someone were to hand you a possible value for the square root of some x, you could check it by using this definition. But it doesn't tell you anything about how to find the value of square root of x.

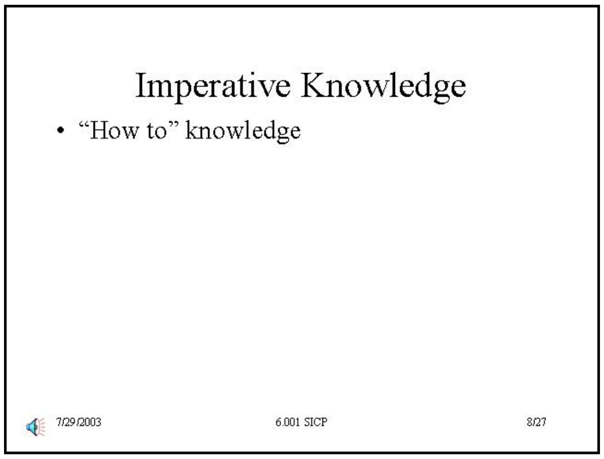


Declarative Knowledge

- "What is true" knowledge

$\sqrt{x}$ is the y such that $y^2 = x$ and $y \geq 0$

7/29/2003 6.001 SICP 7/27



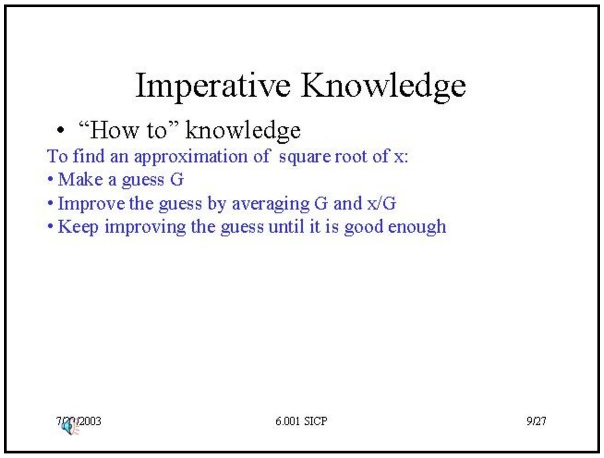
Imperative Knowledge

- "How to" knowledge

7/29/2003 6.001 SICP 8/27

Slide 1.1.8
On the other hand, imperative knowledge talks about "how to" knowledge. It tends to describe a specific sequence of steps that characterize the evolution of a process by which one can deduce information, transforming one set of facts into a new set.

Slide 1.1.9
So here for example is a very old algorithm, that is, a specific piece of imperative knowledge, for find an approximation to the square root of some number, x.

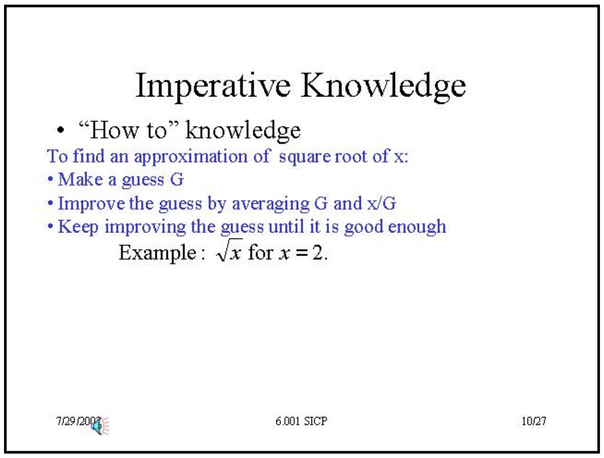


Imperative Knowledge

- "How to" knowledge

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

7/29/2003 6.001 SICP 9/27



Imperative Knowledge

- "How to" knowledge

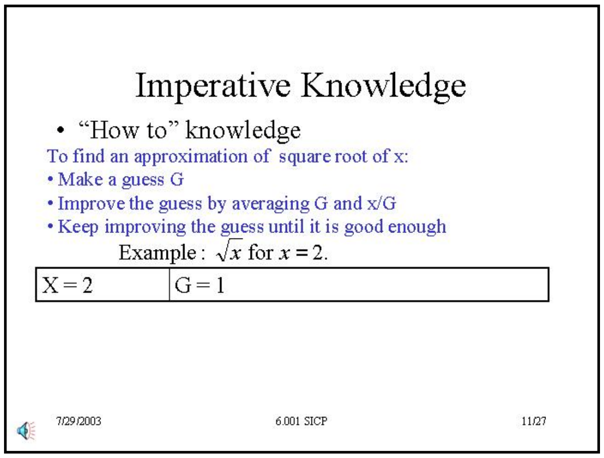
To find an approximation of square root of x:
- Make a guess G
- Improve the guess by averaging G and x/G
- Keep improving the guess until it is good enough

Example: $\sqrt{x}$ for $x = 2$.

7/29/2003 6.001 SICP 10/27

Slide 1.1.10
Okay -- let's test it out. Suppose we want to find the square root of 2. We will see how this sequence of steps describes a process for finding the square root of 2.

Slide 1.1.11
So here we go. We'll create a little chart to keep track of the algorithm. We want to find the square root of 2, so x = 2. And we start with some guess, say G = 1. Our algorithm tells us how to improve this guess, by averaging g and x divided by g.


Imperative Knowledge

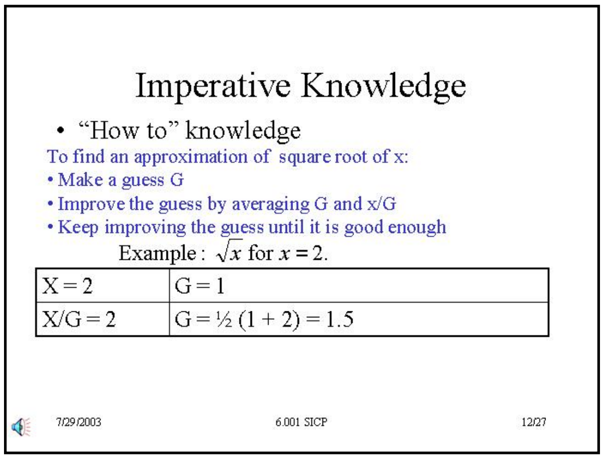
- "How to" knowledge

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

Example : $\sqrt{x}$ for $x = 2$.

X = 2	G = 1

7/29/2003 6.001 SICP 11/27


Imperative Knowledge

- "How to" knowledge

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

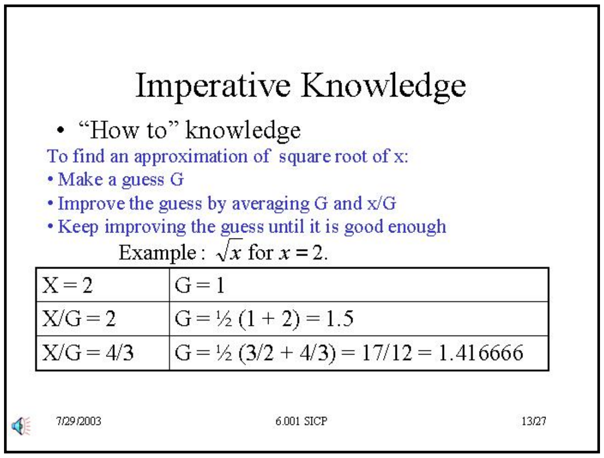
Example : $\sqrt{x}$ for $x = 2$.

X = 2	G = 1
X/G = 2	G = ½ (1 + 2) = 1.5

7/29/2003 6.001 SICP 12/27

Slide 1.1.12
So we compute x/G. And then the algorithm says to get a new guess, G, by averaging the old guess and this ratio -- giving us a better guess for the square root of 2.

Slide 1.1.13
If we decide we are not close enough (i.e. square our current guess is too far away from 2) we can continue. We take our current value for the guess, and compute x divided by that value. Then we get a new guess, by averaging our current guess and this new ratio. And we continue.


Imperative Knowledge

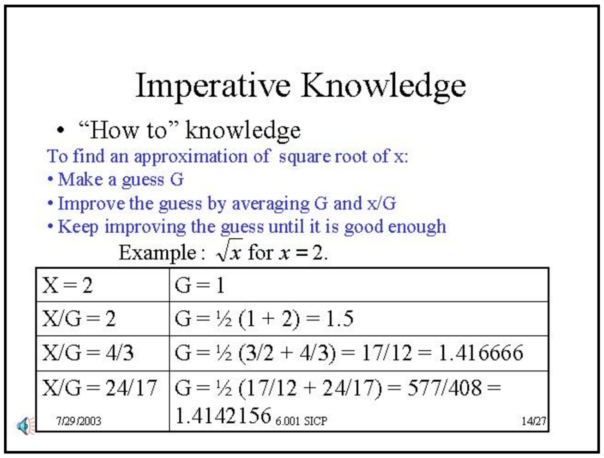
- "How to" knowledge

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

Example : $\sqrt{x}$ for $x = 2$.

X = 2	G = 1
X/G = 2	G = ½ (1 + 2) = 1.5
X/G = 4/3	G = ½ (3/2 + 4/3) = 17/12 = 1.416666

7/29/2003 6.001 SICP 13/27


Imperative Knowledge

- "How to" knowledge

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

Example : $\sqrt{x}$ for $x = 2$.

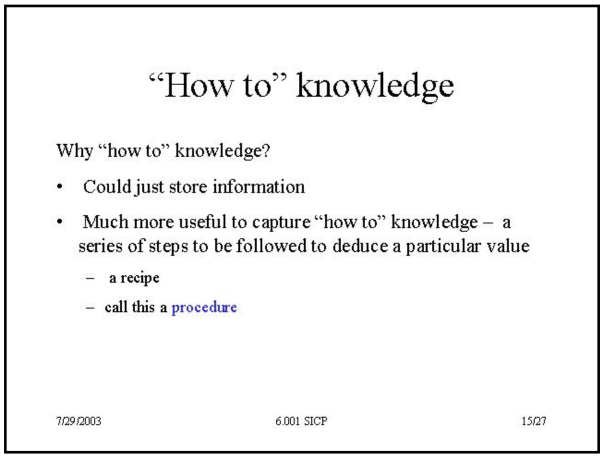
X = 2	G = 1
X/G = 2	G = ½ (1 + 2) = 1.5
X/G = 4/3	G = ½ (3/2 + 4/3) = 17/12 = 1.416666
X/G = 24/17	G = ½ (17/12 + 24/17) = 577/408 = 1.4142156

7/29/2003 6.001 SICP 14/27

Slide 1.1.14
Eventually we get a value for the guess that is close enough, and we stop. Notice how this "algorithm" describes a sequence of steps to follow to deduce some new information from a set of facts. It tells us "how to" do something.
Compare this with the case of the declarative or "what is" version. It simply told us how to recognize a square root if we saw one, but nothing about how to find one.

Slide 1.1.15

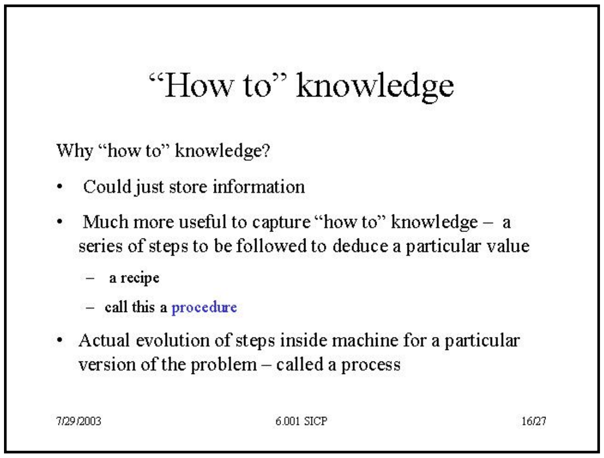
So what we have seen is that imperative and declarative knowledge are very different. One captures statements of fact; the other captures methods for deducing information. It is easy to see why the latter is more interesting. For example, one could in principle imagine trying to collect a giant listing of all possible square roots, and then simply looking up a square root when you need it. Much more convenient is to capture the process of deducing a specific square root as needed. Thus, we are primarily interested in "how to" knowledge -- we want to be able to give the computer instructions to compute a value, and this means we need a way of capturing the "how to" knowledge. In particular, we want to describe a series of specific, mechanical steps to be followed in order to deduce a particular value associated with some problem, using a predefined set of operations. This "recipe" for describing "how to" knowledge we call a **procedure**.



"How to" knowledge

Why "how to" knowledge?
- Could just store information
- Much more useful to capture "how to" knowledge – a series of steps to be followed to deduce a particular value
  - a recipe
  - call this a procedure

7/29/2003 6.001 SICP 15/27



"How to" knowledge

Why "how to" knowledge?
- Could just store information
- Much more useful to capture "how to" knowledge – a series of steps to be followed to deduce a particular value
  - a recipe
  - call this a procedure
- Actual evolution of steps inside machine for a particular version of the problem – called a process

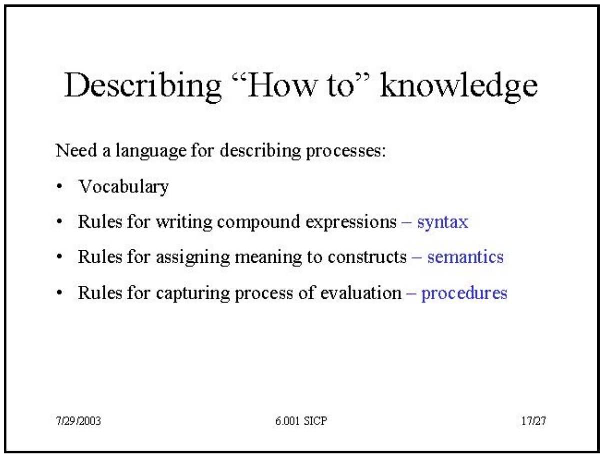
7/29/2003 6.001 SICP 16/27

Slide 1.1.16

When we want to get the computer to actually compute a value, that is, use the "how to" knowledge to find the value associated with a particular instantiation of the problem, we will **evaluate** an expression that applies that procedure to some values. The actual sequence of steps within the computer that cause the "how to" knowledge to evolve is called a **process**. Much of our focus during the term will be in understanding how to control different kinds of processes by describing them with procedures.

Slide 1.1.17

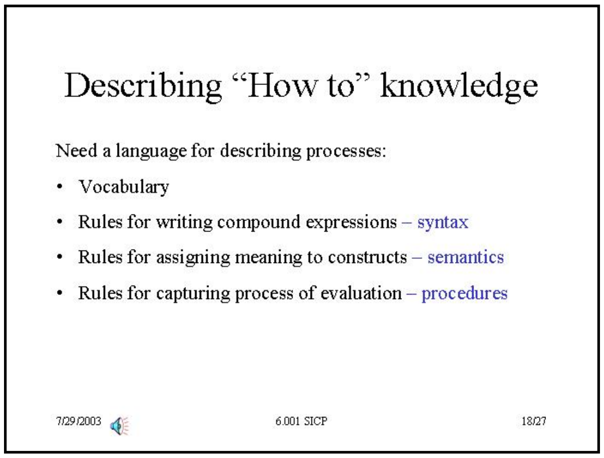
Now we want to create tools that make it easy for us to capture procedures and describe processes, and for that we will need a language. Whatever language we choose to use to describe computational processes, it must have several components.
First, it will have a vocabulary -- a set of words on which we build our description. These will be the basic elements of computation, the fundamental representations of information and the fundamental procedures that we use to describe all other procedures.
Second, it will have a set of rules for legally connecting elements together -- that is, how to build more complex parts of a procedure out of more basic ones. This will be very similar to the **syntax** of a natural language.
Third, it will have a set of rules for deducing the meaning associated with elements of the description. This will be very similar to the **semantics** of a natural language.
And finally, we will need standard ways of combining expressions in our language together into a sequence of steps that actually describe the process of computation.
We will see is that our language for describing procedures will have many of the same features as natural languages, and that we will build methods for constructing more complex procedures out simpler pieces in natural ways.



Describing "How to" knowledge

Need a language for describing processes:
- Vocabulary
- Rules for writing compound expressions – syntax
- Rules for assigning meaning to constructs – semantics
- Rules for capturing process of evaluation – procedures

7/29/2003 6.001 SICP 17/27



Describing "How to" knowledge

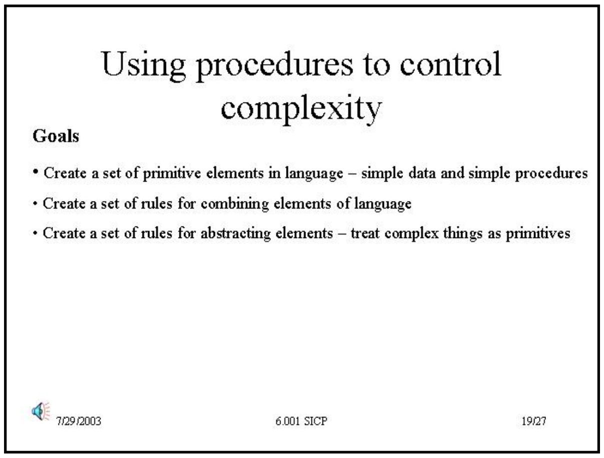
Need a language for describing processes:

- Vocabulary
- Rules for writing compound expressions – syntax
- Rules for assigning meaning to constructs – semantics
- Rules for capturing process of evaluation – procedures

7/29/2003 6.001 SICP 18/27

Slide 1.1.18
One of the things we will see is that it does not take long to describe the rules for connecting elements together, nor to describe the rules for determining meanings associated with expressions in our language of procedures.
Our real goal is to use this language of procedures and processes to help us control complexity in large systems -- that is, to use the language and its elements to design particular procedures aimed at solving a specific problem. We will spend much of the term doing this, both from scratch, and by looking at examples from existing procedures.

Slide 1.1.19
In order to capture imperative knowledge, we are going to create languages that describe such processes. This means we will need to specify a set of primitive elements -- simple data and simple procedures, out of which we will capture complex procedures. We will also need a set of rules for combining primitive things into more complex structures. And once we have those complex structures, we will want to be able to abstract them -- give them name so that we can treat them as primitives.

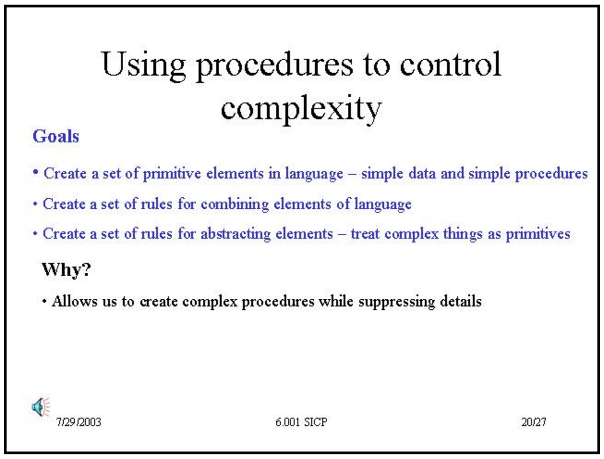


Using procedures to control complexity

Goals

- Create a set of primitive elements in language – simple data and simple procedures
- Create a set of rules for combining elements of language
- Create a set of rules for abstracting elements – treat complex things as primitives

7/29/2003 6.001 SICP 19/27



Using procedures to control complexity

Goals

- Create a set of primitive elements in language – simple data and simple procedures
- Create a set of rules for combining elements of language
- Create a set of rules for abstracting elements – treat complex things as primitives

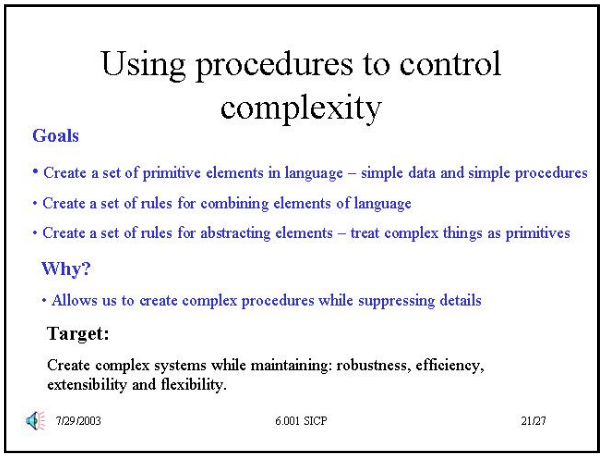
Why?

- Allows us to create complex procedures while suppressing details

7/29/2003 6.001 SICP 20/27

Slide 1.1.20
We will see, as we go through the term, that this cycle of creating complex processes, then suppressing the details by abstracting them into black box units, is a powerful tool for designing, maintaining and extending computational systems.

Slide 1.1.21
Indeed, that is precisely the goal of understanding computation. How can we create methodologies that make it easy to describe complex processes without getting lost in the details? Clearly a well-designed methodology for thinking about computation should enable us to build systems that robustly and efficiently compute results without error, but also should enable us to easily add new capabilities to the system. Thus, our goal is to gain experience in thinking about computation, independent of language details and specifics, in order to control complexity in large, intricate systems.



Using procedures to control complexity

Goals

- Create a set of primitive elements in language – simple data and simple procedures
- Create a set of rules for combining elements of language
- Create a set of rules for abstracting elements – treat complex things as primitives

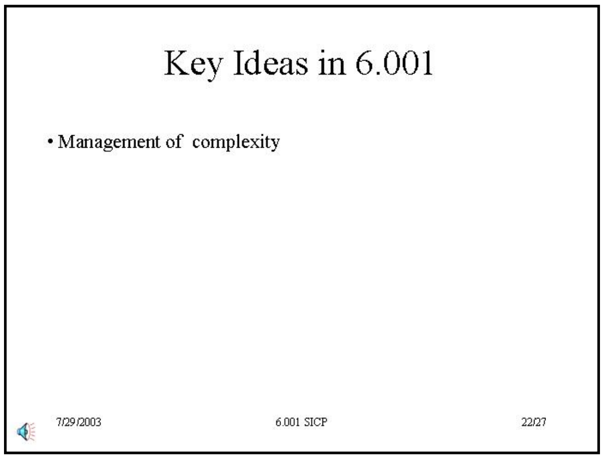
Why?

- Allows us to create complex procedures while suppressing details

Target:

Create complex systems while maintaining: robustness, efficiency, extensibility and flexibility.

7/29/2003 6.001 SICP 21/27



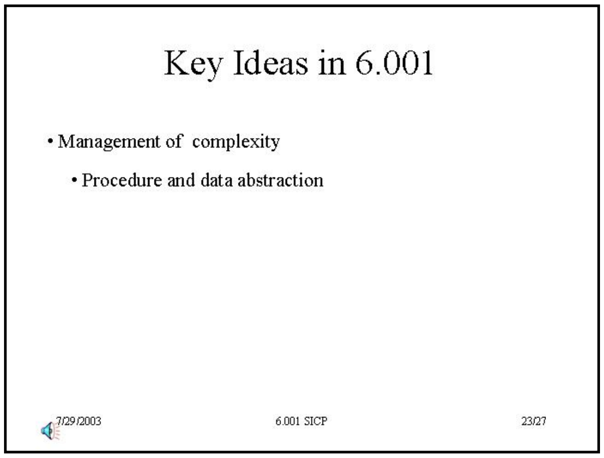
Key Ideas in 6.001

- Management of complexity

7/29/2003 6.001 SICP 22/27

Slide 1.1.22
Thus our goal in 6.001 is to use the ideas of "how to" knowledge, the ideas of describing processes through procedures, to control complexity of large systems. We don't just want to write small programs, we want to understand how the ideas of procedures and their pieces can be used to construct large systems in well-engineered ways.
This means we need tools for handling complex systems, and we are going to see a range of such tools, built on the language of procedures.

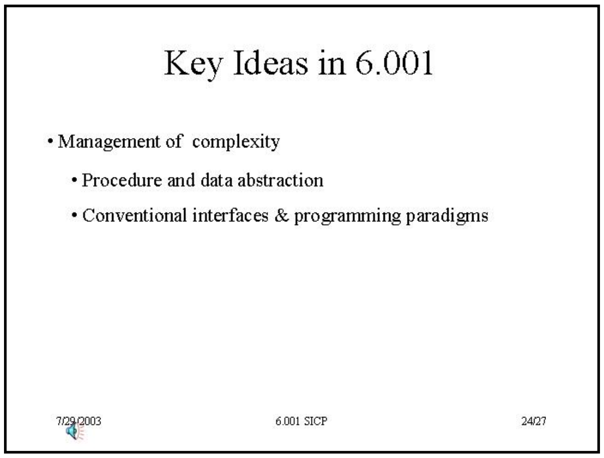
Slide 1.1.23
The first tool we will use for controlling complexity is the idea of an **abstraction**, a black box, if you like. Take the method we just described for computing square roots. While it is useful to know how to do this, one can easily imaging problems in which one simply wants the square root, and one doesn't care how it is derived. Imagine creating a black box that captures the idea of square root -- one simply puts values in the correct slot, and out come appropriate square roots. This idea of isolating the use of a procedure from its actual implementation or mechanism is a central idea that we will use frequently in controlling complexity of large systems.



Key Ideas in 6.001

- Management of complexity
  - Procedure and data abstraction

7/29/2003 6.001 SICP 23/27

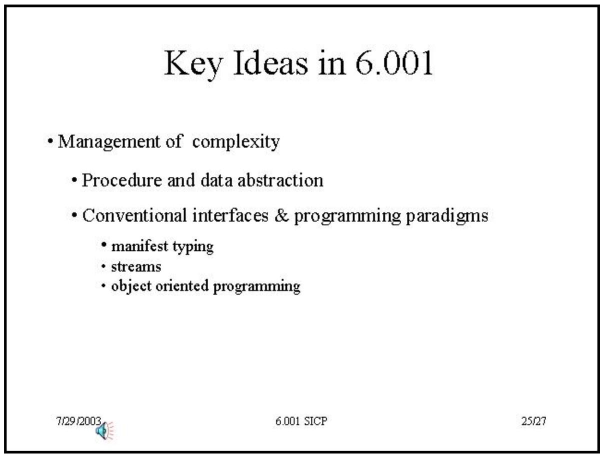

Key Ideas in 6.001

- Management of complexity
  - Procedure and data abstraction
  - Conventional interfaces & programming paradigms

7/29/2003 6.001 SICP 24/27

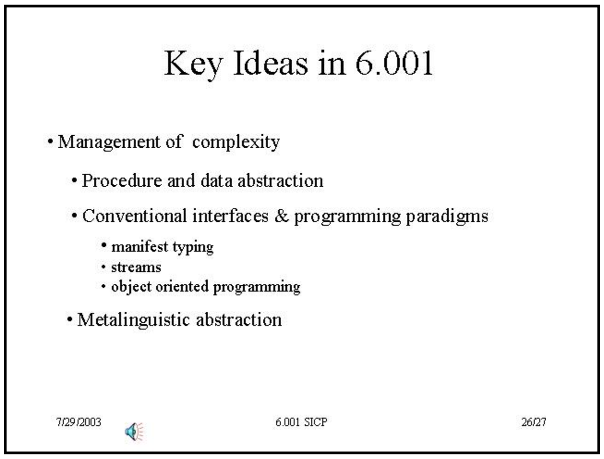
Slide 1.1.24
Not only are black boxes a useful tool for isolating components of a system, they also provide the basis for connecting things together. A key issue is providing methods for connecting together basic units into components that themselves can be treated as basic units. Thus, we will spend a lot of time talking about conventional interfaces -- standard ways of interconnecting simpler pieces.
This is much like hooking up parts of a stereo system. One has standard interfaces by which components can be intertwined, and this can be done without worrying about the internal aspects of the components. Similarly in programming, we will describe conventions for interfacing simpler components to create new elements that can further be connected together.

Slide 1.1.25
In fact, here are three particular kinds of conventional interfaces that we will explore in some detail during the term.


Key Ideas in 6.001

- Management of complexity
  - Procedure and data abstraction
  - Conventional interfaces & programming paradigms
    - manifest typing
    - streams
    - object oriented programming

7/29/2003 6.001 SICP 25/27

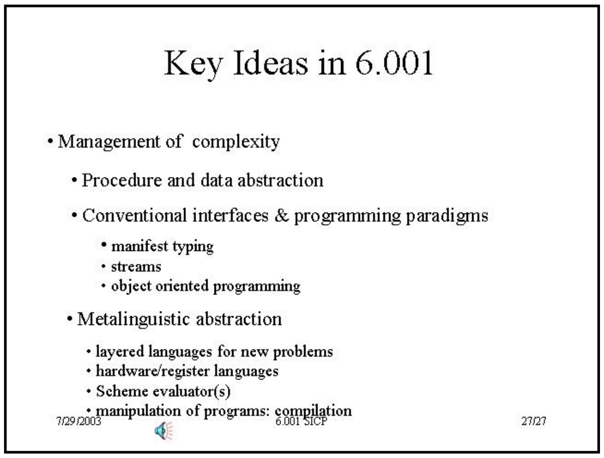

Key Ideas in 6.001

- Management of complexity
  - Procedure and data abstraction
  - Conventional interfaces & programming paradigms
    - manifest typing
    - streams
    - object oriented programming
  - Metalinguistic abstraction

7/29/2003 6.001 SICP 26/27

Slide 1.1.26
We will see as we go through the term that ideas of capturing procedures in black box abstractions, then gluing them together through conventional interfaces will give us considerable power in creating computational machinery. At some point, however, even these tools will not be sufficient for some problems. At this stage, we will generalize these ideas, to create our own languages specifically oriented at some problem domain. This idea of **meta-linguistic abstraction** will provide us with a powerful tool for designing procedures to capture processes, especially as we focus on the idea of what it means to evaluate an expression in a specifically designed language.

Slide 1.1.27
And as a consequence we will see several different mechanisms for creating new languages -- ones designed for particular problems, one designed to interface between higher level languages and the hardware that actually does the work, and ones that get to the heart of any computer language, by focusing on the essence of computation -- evaluation of expressions.



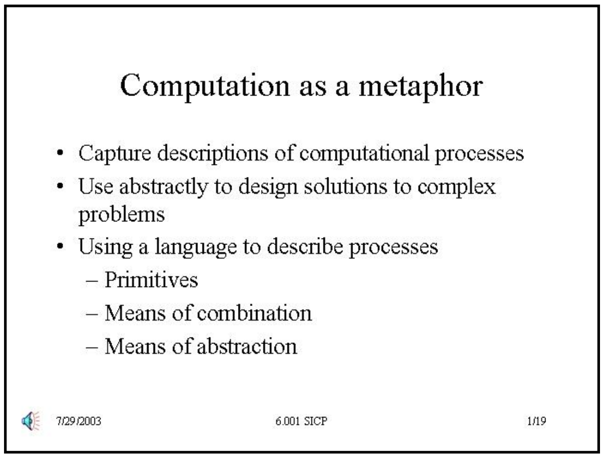
Key Ideas in 6.001

- Management of complexity
  - Procedure and data abstraction
  - Conventional interfaces & programming paradigms
    - manifest typing
    - streams
    - object oriented programming
  - Metalinguistic abstraction
    - layered languages for new problems
    - hardware/register languages
    - Scheme evaluator(s)
    - manipulation of programs: compilation

7/29/2003 6.001 SICP 27/27

6.001 Notes: Section 1.2

Slide 1.2.1
Our goal in this course is to explore computation, especially how thinking about computation can serve as a tool for understanding and controlling complexity in large systems. Of particular interest are systems in which information is inferred from some set of initial data, whether that is finding other information on the web, or computing an answer to a scientific problem, or deciding what control signals to use to guide a mechanical system.

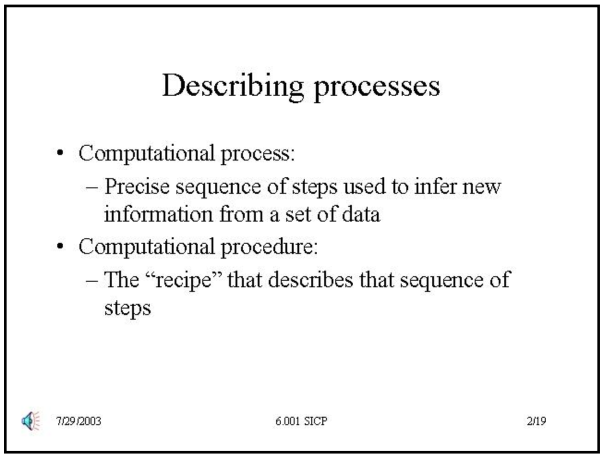


Computation as a metaphor

- Capture descriptions of computational processes
- Use abstractly to design solutions to complex problems
- Using a language to describe processes
  - Primitives
  - Means of combination
  - Means of abstraction

7/29/2003 6.001 SICP 1/19

For such systems to work, they need some process by which such inference takes place, and our goal is to be able to reason about that process. In using computation as a metaphor to understand complex problem solving, we really want to do two things. We want to capture descriptions of **computational processes.** And we want to use the notion of a computational process as an abstraction on which we can build solutions to other complex problems.
We will see that to describe processes, we need a language appropriate for capturing the essential elements of processes. This means we will need fundamental primitives (or atomic elements of the language), means of combination (ways of constructing more complex things from simpler ones) and means of abstraction (ways of treating complex things as primitives so that we can continue this process).

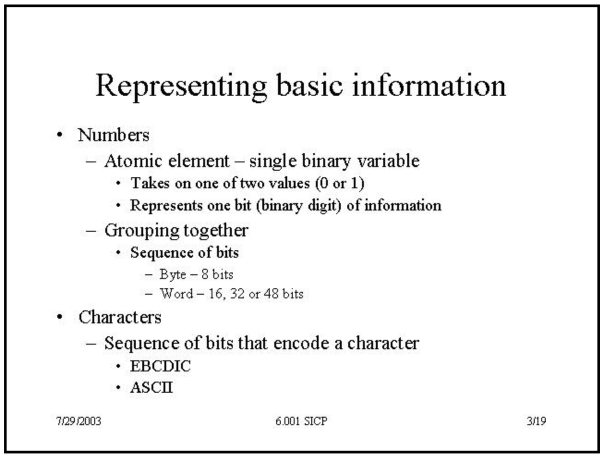


Describing processes

- Computational process:
  - Precise sequence of steps used to infer new information from a set of data
- Computational procedure:
  - The "recipe" that describes that sequence of steps

7/29/2003 6.001 SICP 2/19

Slide 1.2.2
So let's begin our discussion of a language for describing computational processes. To do this, we really need to provide a definition of, or at least some intuition behind, the idea of a computational process. In simple terms, a computational process is a precise sequence of steps by which information (in the form of numbers, symbols, or other simple data elements) is used to infer new information. This could be a numerical computation (e.g. a set of steps to compute a square root, as we saw earlier), or it could be a symbolic computation (finding a piece of information on the Web), or some other inference based on information.
While the computational process refers to the actual evolution of information in the computation, we also want to be able to capture a description of the actual steps in the computation, and we refer to this recipe as a **computational procedure**: a description of the steps in the computation. Thus, our language will allow us to describe "recipes" and to use these descriptions on particular instances of problems, that is, will "bake solutions" using the recipes.

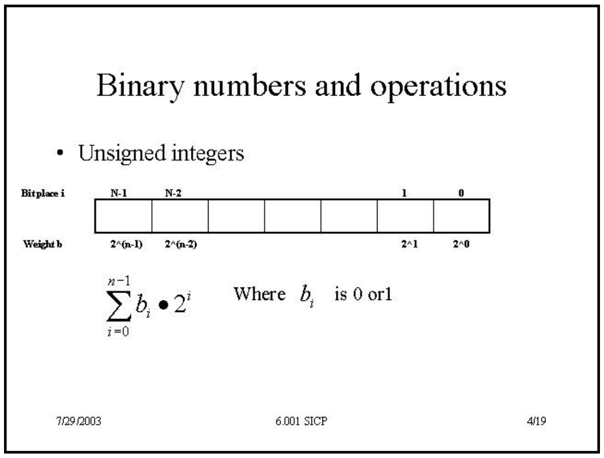


Representing basic information

- Numbers
  - Atomic element – single binary variable
    - Takes on one of two values (0 or 1)
    - Represents one bit (binary digit) of information
  - Grouping together
    - Sequence of bits
      - Byte – 8 bits
      - Word – 16, 32 or 48 bits
- Characters
  - Sequence of bits that encode a character
    - EBCDIC
    - ASCII

7/29/2003 6.001 SICP 3/19

Slide 1.2.3
First, we need to understand how we are going to representation information, which we will use as the basis for our computational processes. To do this, we need to decide on representations for numeric values and for symbolic ones. Let's start with numbers.
To represent a number, we start with the most atomic element. Because ultimately we will represent information internally inside the computer, we will use electronic signals to do so. These are most conveniently represented by using a high voltage or current or a low voltage or current to represent fundamental values. Thus, the most primitive element in representing a value is a binary variable, which takes on one of two values: a zero or a one. This variable represents one **bit**, or *binary digit,* or information.
Of course, we need to group these bits together to represent other numbers, which we do typically in groupings of 8 bits (a *byte*) or in groupings of 16, 32, or 48 bits (a *word).*
Once we have sequences of bits, we can use them not only to represent other numbers, but we can envision encodings, in which numbers or bit sequences are used to represent characters. And characters can further be group together to represent symbolic words. Though we won't worry about it much in this course, there are standard encoding schemes for using bit sequences to represent characters as well as numbers. Typically the first few bits in a sequence are used as a tag to distinguish a number from an encoding for a character.



Binary numbers and operations

- Unsigned integers

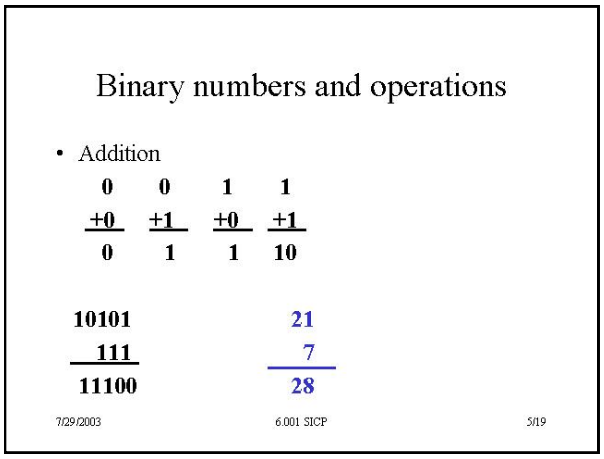
Bitplace i
N-1
N-2
1
0
Weight b
2^(n-1)
2^(n-2)
2^1
2^0

$$\sum_{i=0}^{n-1} b_i \bullet 2^i \quad \text{Where } b_i \text{ is 0 or1}$$

7/29/2003 6.001 SICP 4/19

Slide 1.2.4
Now let's spend a minute thinking about binary representations for numbers and operations to manipulate them. We said we could use a sequence of bits to represent a number, for example, a positive integer. We can identify each place in the sequence (by convention the lowest order bit is at the right). And we can weight each bit by a different power of 2, so that we are able to represent all possible integers (up to some limit based on the number of bits we are using). Mathematically, we can capture this in the equation shown on the slide, and this gives us a way of representing unsigned integers.

Slide 1.2.5
Now what about simple arithmetic operations on binary integer representations? Well, the rules for addition are just what you would expect. And we can do standard addition by simply carrying bits, just as you would in decimal arithmetic. You can see this by checking the binary addition at the bottom left, and confirming that conversion of this result to digital form gives you what is shown on the right.
There are similar rules for binary multiplication.

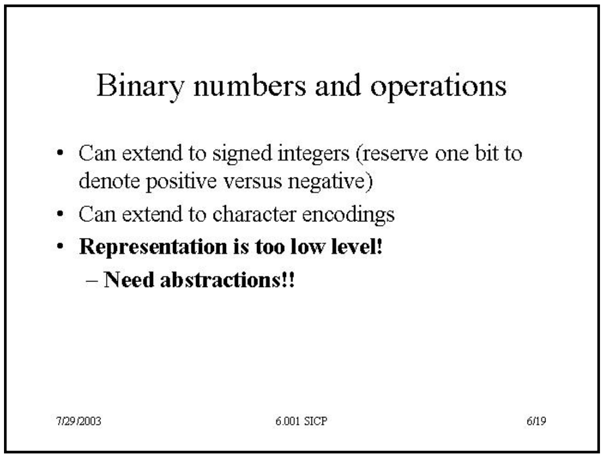


Binary numbers and operations

- Addition

```
  0     0     1     1
 +0    +1    +0    +1
  0     1     1    10

10101          21
  111           7
11100          28
```

7/29/2003 6.001 SICP 5/19



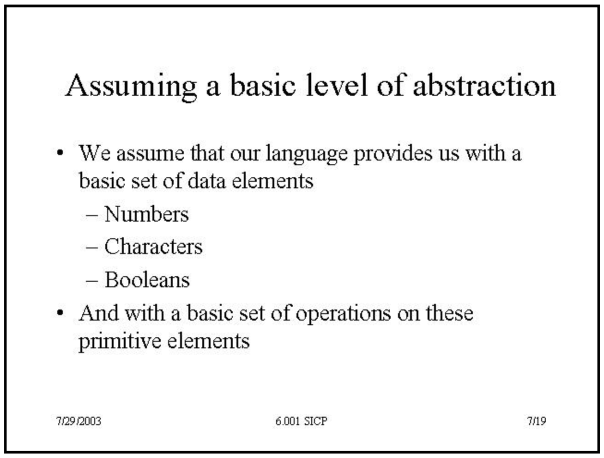
Binary numbers and operations

- Can extend to signed integers (reserve one bit to denote positive versus negative)
- Can extend to character encodings
- **Representation is too low level!**
  – **Need abstractions!!**

7/29/2003 6.001 SICP 6/19

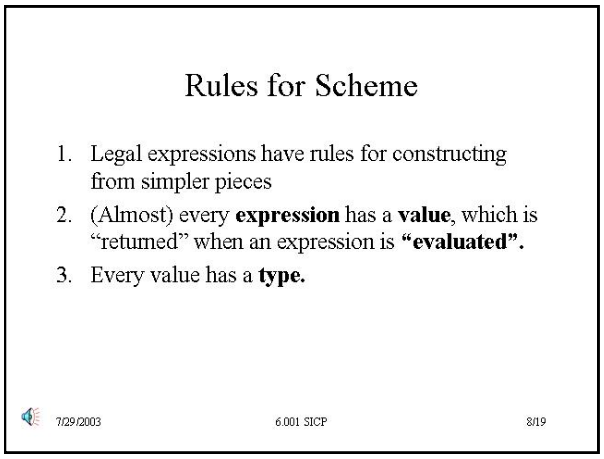
Slide 1.2.6
One can build on this to create signed integers (using one bit, typically the highest order bit) to represent the sign (positive or negative). And can extend this to represent real (or scientific) numbers, and to represent encodings for characters (using some high order bits to denote a character, and then using some standard encoding to relate bit sequences to characters).
The problem is that this is clearly too **low level!** Imagine trying to write a procedure to compute square roots, when all you can think about are operations on individual bits. This quickly gets bogged down in details, and is generally mind-numbingly boring. So we need to incorporate a level of abstraction. We are going to assume that we are given a set of primitive objects,
and a set of basic operations, and we are going to build on top of that level of abstraction to deal with higher-level languages for computation.

Slide 1.2.7
So we will assume that our language provides a built-in set of data structures: numbers, characters, and Boolean values (true or false). And we will assume that our language provides a built-in set of primitive operations for manipulating numbers, characters and Booleans. Our goal is to build on top of this level of abstraction to capture the essence of a computational process.


Assuming a basic level of abstraction

- We assume that our language provides us with a basic set of data elements
  - Numbers
  - Characters
  - Booleans
- And with a basic set of operations on these primitive elements

7/29/2003 6.001 SICP 7/19

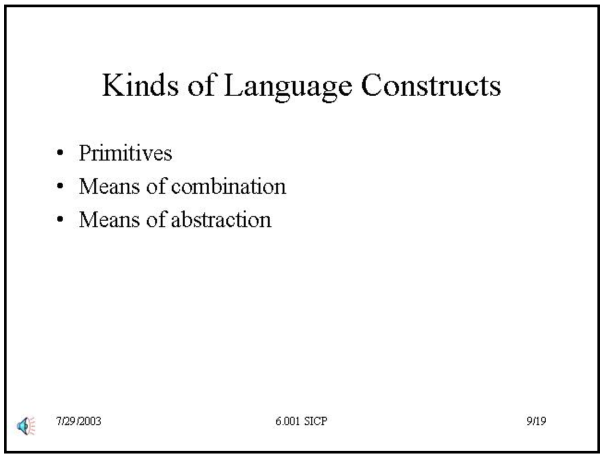

Rules for Scheme

1. Legal expressions have rules for constructing from simpler pieces
2. (Almost) every **expression** has a **value**, which is "returned" when an expression is **"evaluated".**
3. Every value has a **type.**

7/29/2003 6.001 SICP 8/19

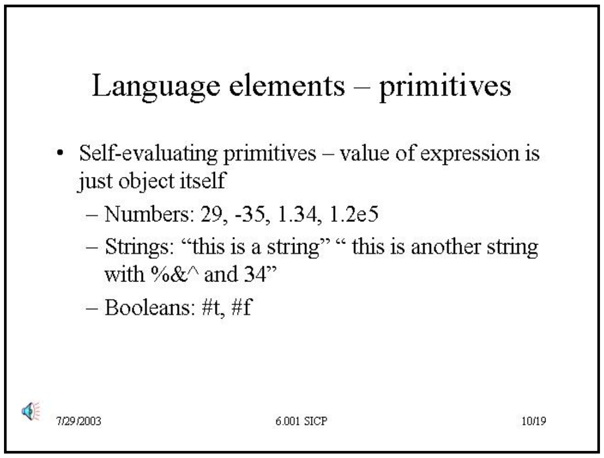
Slide 1.2.8
Thus, we are going to first describe our language for capturing computational processes, and then look at using it to solve complex problems. In this course the language we are going to use is called Scheme - a variant of a language called LISP, both of which were invented here at MIT some time ago.
Everything we write in Scheme will be composed of a set of expressions, and there is a simple set of rules that tell us how to create legal expressions in this language. These rules are similar to the **syntax** of a natural language, like English. They tell us the simplest legal expressions, and give us rules for constructing more complex legal expressions from simpler pieces.
Similar, almost every expression in Scheme has a meaning or value associated with it. The **semantics** of the language will tell us how to deduce the meaning associated with each expression - or, if you like, how to determine the value associated with a particular computation. In Scheme, with only a few exceptions, we will see that evaluating every expression results in a value being returned as the associated meaning.
As we build up our vocabulary in Scheme, we will find rules of syntax and semantics associated with each new type of expression.
Finally, we will also see that every value in Scheme has a **type** associated with it. Some of these types are simple; others are more complex. Types basically define a taxonomy of expressions, and relate legal ways in which those expressions can be combined and manipulated. We will see that reasoning about the types of expressions will be very valuable when we reason about capturing patterns of computation in procedures.

Slide 1.2.9
As we build up our language, looking at the syntax and semantics of expressions in that language, we will also see that these expressions very nicely break up into three different components. We have **primitives** - our most basic atomic units, on top of which everything else is constructed. We have ways of gluing things together - our **means of combination** - how to combine smaller pieces to get bigger constructs. And finally we have a **means of abstraction** - our way of taking bigger pieces, then treating them as primitives so that they can be combined into bigger constructs, while burying or suppressing the internal details of the pieces.


Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

7/29/2003 6.001 SICP 9/19



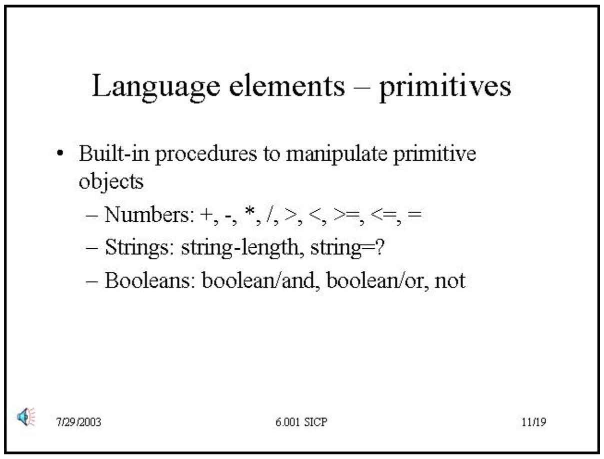
Language elements – primitives

- Self-evaluating primitives – value of expression is just object itself
  - Numbers: 29, -35, 1.34, 1.2e5
  - Strings: “this is a string” “ this is another string with %&^ and 34”
  - Booleans: #t, #f

7/29/2003 6.001 SICP 10/19

Slide 1.2.10
Let’s start with the primitives -- the basic elements. These are the simplest elements on top of which we will build all our other computational constructs.
The simplest of these are the **self-evaluating** expressions, that is, things whose value is the object or expression itself. These include **numbers, strings** and **Booleans**.
Numbers are obvious, but include integers, real numbers, and scientific notation numbers.
Strings are sequences of characters, including numbers and special characters, all delimited by double quotes. These represent symbolic, as opposed to numeric, data.
Booleans represent the logical values of true and false. These represent logical, as opposed to symbolic or numeric, data.

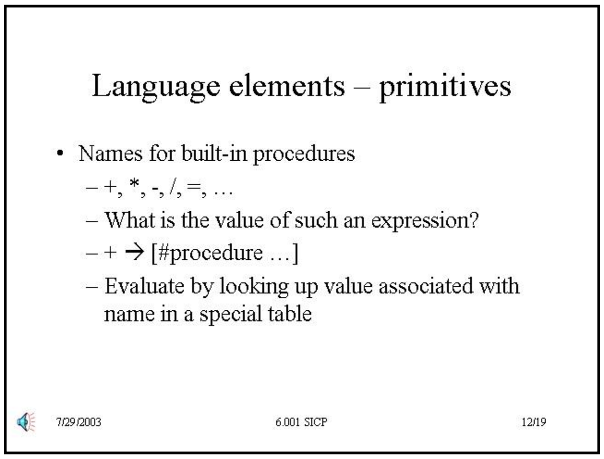
Slide 1.2.11
Of course we want more than just primitive objects, we need ways of manipulating those objects. For example, for numbers we have a set of built-in, or predefined, procedures. Thus, the symbol + is a name for the primitive procedure or operation of addition, and similarly for other arithmetic operations, including comparison operations.
Strings have an associated set of operations, for comparing strings or extracting parts of strings.
And Booleans have an associated set of logical operations.
Think of these as abstractions: they are machinery that performs the operations described by a set of known rules.



Language elements – primitives

- Built-in procedures to manipulate primitive objects
  - Numbers: +, -, *, /, >, <, >=, <=, =
  - Strings: string-length, string=?
  - Booleans: boolean/and, boolean/or, not

7/29/2003 6.001 SICP 11/19



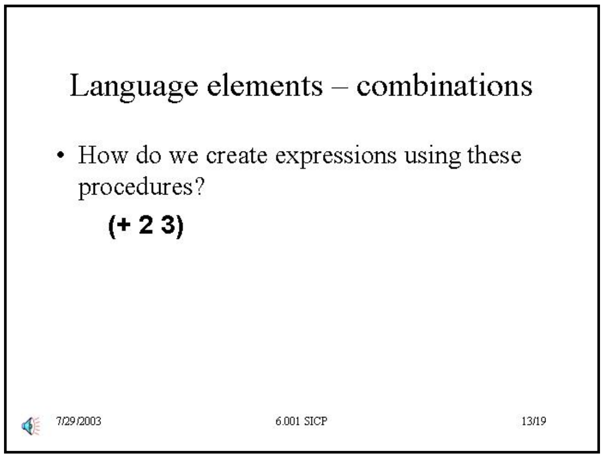
Language elements – primitives

- Names for built-in procedures
  - +, *, -, /, =, …
  - What is the value of such an expression?
  - + $\rightarrow$ [#procedure …]
  - Evaluate by looking up value associated with name in a special table

7/29/2003 6.001 SICP 12/19

Slide 1.2.12
Before we actually show the use of these primitive, or built-in, procedures, we pause to stress that these names are themselves expressions. By our earlier discussion, this suggests that this expression, +, should have a value. This sounds like a strange thing for many computer languages, but in Scheme we can ask for the value associated with a symbol or name. In this case, the value or meaning associated with this built-in symbol is the actual procedure, the internal mechanism if you like, for performing addition. In fact our rule for evaluating the expression, +, is to treat it as a symbol, and look up the value associated with it in a big table somewhere. Shortly we will see how that table is created.

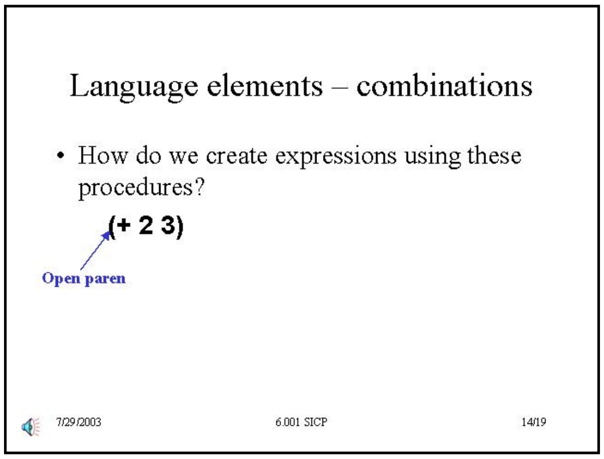
Slide 1.2.13
Given numbers and procedures, we want to use them together. Ideally, we should be able to apply operations like * or + to numbers to get new values. This leads to means of combination - our way of constructing larger expressions out of simpler ones. In Scheme our standard means of combination consists of an expression that will apply a procedure to a set of arguments in order to create a new value, and it has a very particular form ....


Language elements – combinations

- How do we create expressions using these procedures?

(+ 2 3)

7/29/2003 6.001 SICP 13/19


Language elements – combinations

- How do we create expressions using these procedures?

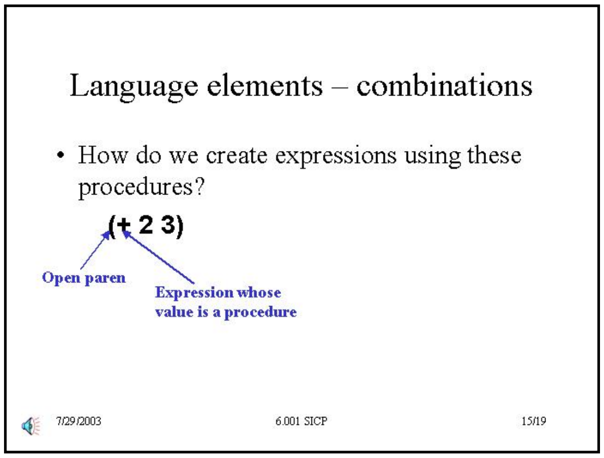
(+ 2 3)

Open paren

7/29/2003 6.001 SICP 14/19

Slide 1.2.14
... consisting of an open parenthesis ...

Slide 1.2.15
... followed by an expression whose value, using the rules we are describing, turns out to be a procedure ...


Language elements – combinations

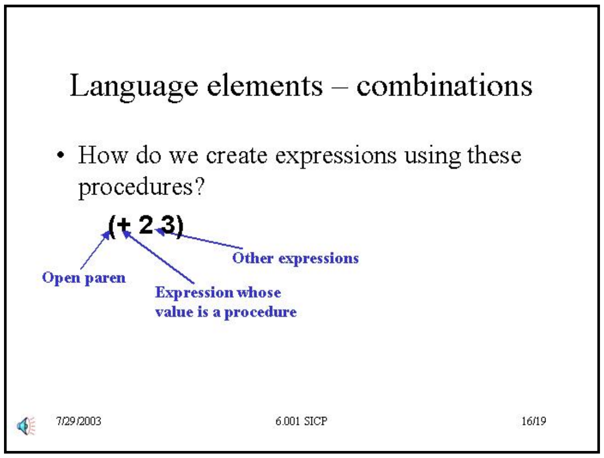
- How do we create expressions using these procedures?

(+ 2 3)

Open paren

Expression whose value is a procedure

7/29/2003 6.001 SICP 15/19


Language elements – combinations

- How do we create expressions using these procedures?

(+ 2 3)

Other expressions

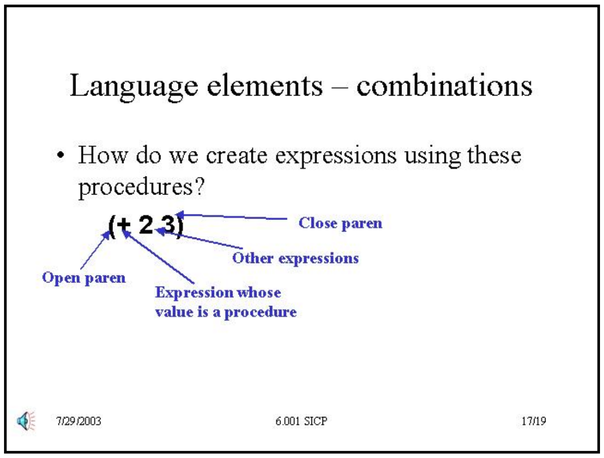
Open paren

Expression whose value is a procedure

7/29/2003 6.001 SICP 16/19

Slide 1.2.16
... followed by some number of other expressions, whose values are obtained using these same rules ...

Slide 1.2.17
... followed by a matching close parenthesis. This form **always** holds for a combination.

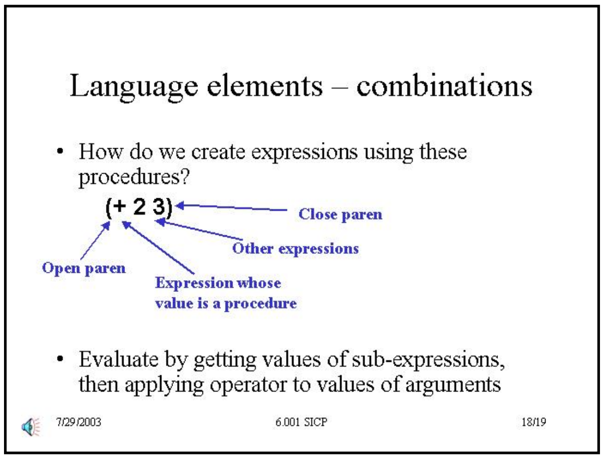


Language elements – combinations

- How do we create expressions using these procedures?

(+ 2 3)
Close paren
Other expressions
Open paren
Expression whose value is a procedure

7/29/2003 6.001 SICP 17/19



Language elements – combinations

- How do we create expressions using these procedures?

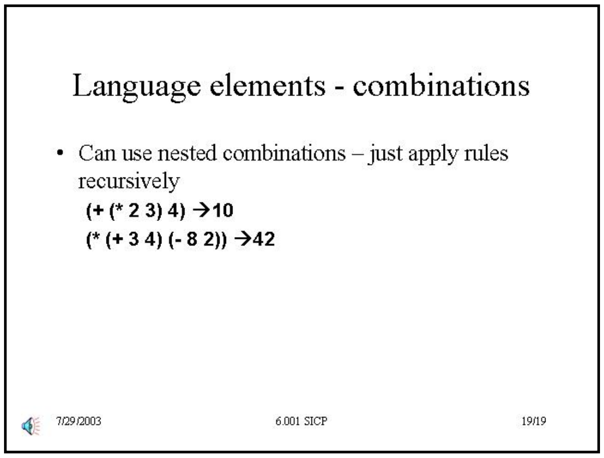
(+ 2 3)
Close paren
Other expressions
Open paren
Expression whose value is a procedure

- Evaluate by getting values of sub-expressions, then applying operator to values of arguments

7/29/2003 6.001 SICP 18/19

Slide 1.2.18
So there is the syntax for a combination - an open parenthesis, an expression whose value is a procedure, some other set of values, and a close parenthesis.
What about the semantics of a combination? To evaluate a combination, we evaluate all the sub-expressions, in any order, using the rules we are developing. We then apply the value of the first expression to the values of the other expressions.
What does apply mean? For simple built-in procedures, it just means take the underlying hardware implementation and do the appropriate thing to the values, e.g. add them, multiply them, etc.

Slide 1.2.19
This idea of combinations can be nested arbitrarily. We can use a combination whose parts are themselves combinations, so long as the resulting value can legally be used in that spot. To evaluate combinations of arbitrary depth, we just recursively apply these rules, first getting the values of the sub-expressions, then applying the procedure to the arguments, and further reducing the expression.
So, for example, to get the value of the first expression, we get the values of + (by lookup), and 4 because it is self-evaluating. Because the middle subexpression is itself a combination, we apply the same rules to this get the value 6, before completing the computation.



Language elements - combinations

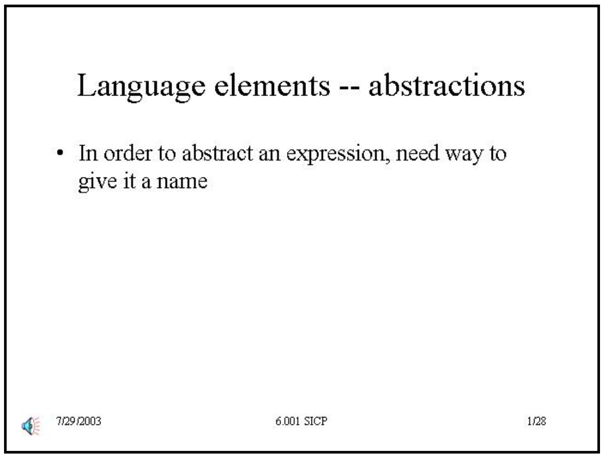
- Can use nested combinations – just apply rules recursively

```
(+ (* 2 3) 4) →10
(* (+ 3 4) (- 8 2)) →42
```

7/29/2003 6.001 SICP 19/19

6.001 Notes: Section 1.3

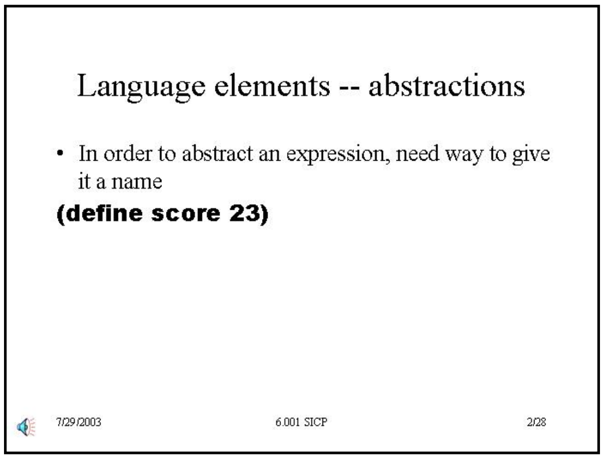
Slide 1.3.1
So far we have basic primitives -- numbers, and simple built-in procedures, and we have means of combination -- ways of combining all those pieces together to get more complicated expressions. But at this stage all we can do is write out long, complicated arithmetic expressions. We have no way of abstracting expressions. We would like to be able to give some expression a name, so that we could just refer to that name (as an abstraction) and not have to write out the complete expression each time we want to use it.



Language elements -- abstractions

- In order to abstract an expression, need way to give it a name

7/29/2003 6.001 SICP 1/28



Language elements -- abstractions

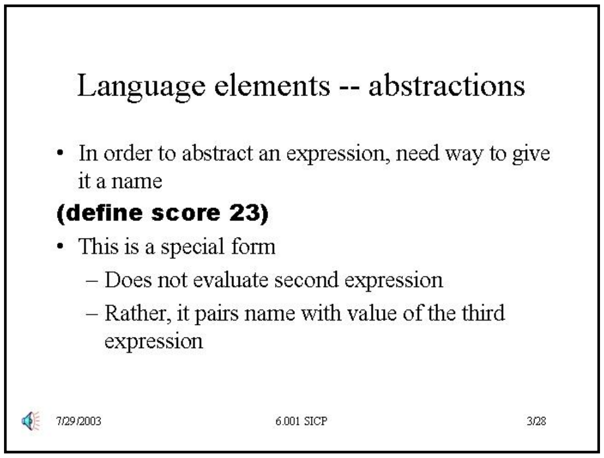
- In order to abstract an expression, need way to give it a name

(define score 23)

7/29/2003 6.001 SICP 2/28

Slide 1.3.2
In Scheme, our standard way for doing that is to use a particular expression, called a **define**. It has a specific form, an open parenthesis (as before), followed by the keyword **define**, followed by an expression that will serve as a name (typically some sequence of letters and other characters), followed by a expression whose value will be associated with that name, followed by a close parenthesis.

Slide 1.3.3
We say that this expression is a **special form**, and this means it does not follow the normal rules of evaluation for a combination. We can see why we want that here. If we applied the normal rules for a combination, we would get the value of the expression **score** and the value of **23**, then apply the **define** procedure. But the whole point of this expression is to associate a value with **score** so we can't possibly use the normal rules to evaluate **score**.
So instead, we will use a different rule to evaluate this special form. In particular, we just evaluate the last sub-expression, then take the name without evaluating it (score in this case) and pair that name together with the deduced value in a special structure we call an **environment**. For now, think of this as a big table into which pairings of names and values can be made, using this special **define** expression.



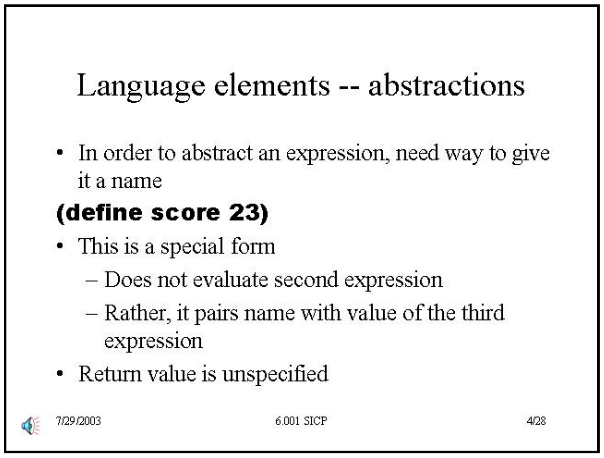
Language elements -- abstractions

- In order to abstract an expression, need way to give it a name

(define score 23)

- This is a special form
  - Does not evaluate second expression
  - Rather, it pairs name with value of the third expression

7/29/2003 6.001 SICP 3/28


Language elements -- abstractions

- In order to abstract an expression, need way to give it a name

(define score 23)

- This is a special form
  - Does not evaluate second expression
  - Rather, it pairs name with value of the third expression
- Return value is unspecified

7/29/2003 6.001 SICP 4/28

Slide 1.3.4

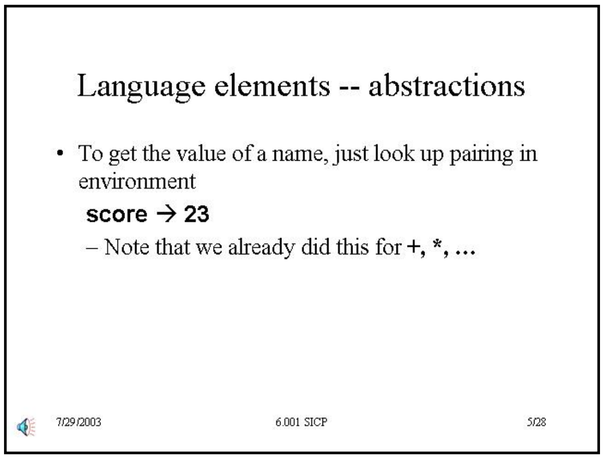
Because our goal is to associate a name with a value, we don't actually care what value is returned by the **define** expression, and in most Scheme implementations we leave that as unspecified.

Thus, our means of abstraction gives us a way of associated a name with an expression, allowing us to use that name in place of the actual expression.

Slide 1.3.5

Once we have the ability to give names to values, we also need the ability to get the value back out. And that's easy. To get the value of a name in Scheme, we simply lookup the pairing of the name in that table we created. Thus, if we evaluate the expression **score** we simply lookup the association we made when we defined **score** in that table, in this case, 23, and return that value.

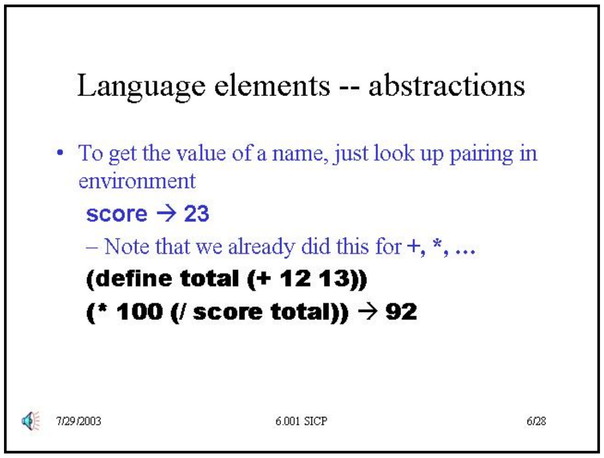
Notice that this is exactly what we did with built-in primitives. If we give the name **+** to Scheme, it looks up the association of that symbol, which in this case is the built in addition procedure, and that procedure is actually returned as the value...


Language elements -- abstractions

- To get the value of a name, just look up pairing in environment

  score → 23
  - Note that we already did this for **+**, *****, …

7/29/2003 6.001 SICP 5/28


Language elements -- abstractions

- To get the value of a name, just look up pairing in environment

  score → 23
  - Note that we already did this for **+**, *****, …

  (define total (+ 12 13))

  (* 100 (/ score total)) → 92

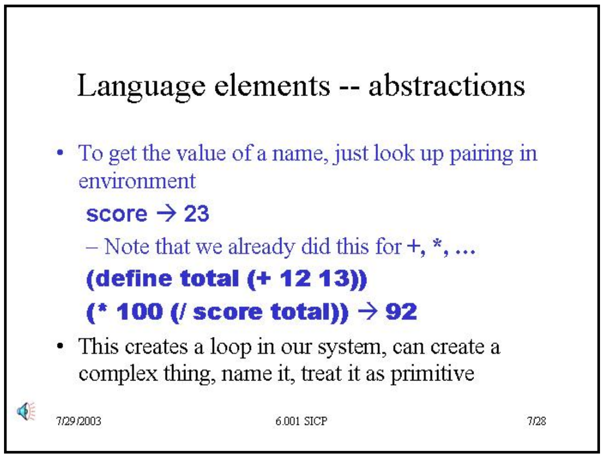
7/29/2003 6.001 SICP 6/28

Slide 1.3.6

... and of course now we can use names in any place we would have used it's associated expression.

In the example shown here, we can define **total** to have the value of the subexpression, and by our previous rules, we know that this reduces to 25. Now if we evaluate the last expression, our rules say to first evaluate the subexpressions. The symbol ***** is easy, as is the number **100**. To get the value of the last sub-expression, we recursively apply our rules, in this case looking up the value of **score**, and the value of **total**, then applying the value of **/** to the result, and finally, applying the multiplication operator to the whole thing.

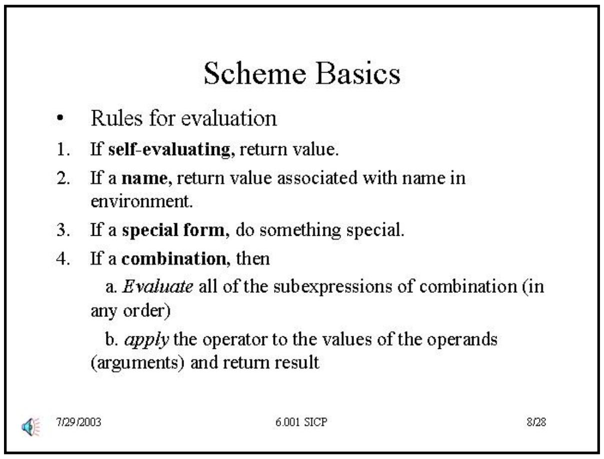
Slide 1.3.7
Notice that this creates a very nice loop in our system. We can now create complex expressions, give them a name, and then by using that name, treat the whole expression as if it were a primitive. We can refer to that expression by name, and thus can write new complex expressions involving those names, give the resulting expression a name, and treat it is a new primitive, and so on. In this way, we can bury complexity behind the names, and create new primitive elements in our language.



Language elements -- abstractions

- To get the value of a name, just look up pairing in environment
  score → 23
  – Note that we already did this for +, *, …
  (define total (+ 12 13))
  (* 100 (/ score total)) → 92
- This creates a loop in our system, can create a complex thing, name it, treat it as primitive

7/29/2003 6.001 SICP 7/28



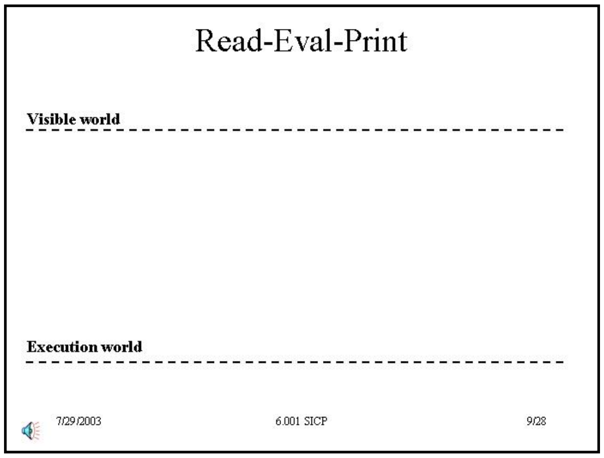
Scheme Basics

- Rules for evaluation
1. If **self-evaluating**, return value.
2. If a **name**, return value associated with name in environment.
3. If a **special form**, do something special.
4. If a **combination**, then
   a. *Evaluate* all of the subexpressions of combination (in any order)
   b. *apply* the operator to the values of the operands (arguments) and return result

7/29/2003 6.001 SICP 8/28

Slide 1.3.8
So here is a summary of the rules of evaluation we have seen so far.

Slide 1.3.9
Because these ideas of evaluation are important, let's take another look at what happens when an expression is evaluated. Remember that our goal is to capture computation in expressions, and use those expressions to compute values. We have been describing both the forms of expressions, and how one deduces values of expressions. When we consider the second stage, we can separate out two different worlds, or two different ways of looking at what happens during evaluation. One world is the **visible** world. This is what we see when we type an expression at the computer and ask it to perform evaluation, leading to some printed result. Below that world is the **execution** world. This is what happens within the computer (we'll see a lot more details about this later in the term), including both how objects are represented and how the actual mechanism of evaluation takes place. We want to see how these two worlds interact to incorporate the rules of semantics for determining values of expressions.

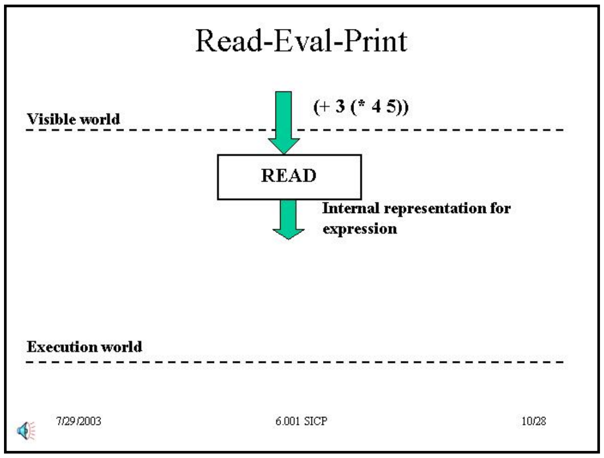


Read-Eval-Print

Visible world

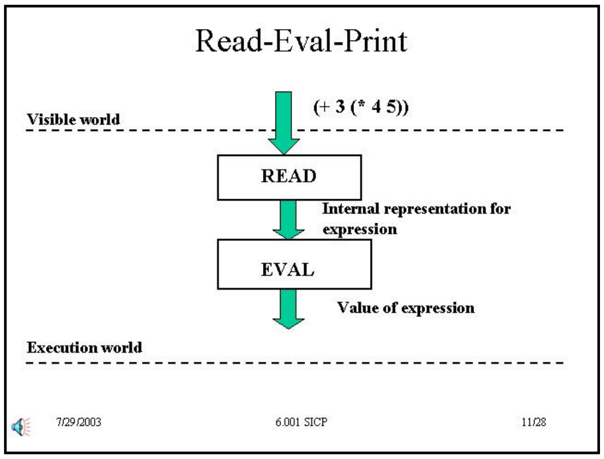
Execution world

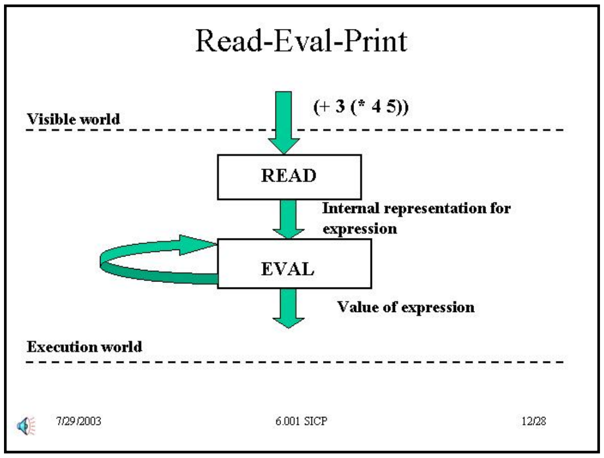
7/29/2003 6.001 SICP 9/28



Slide 1.3.10
When an expression is entered into the computer from our visible world, it is first processed by a mechanism called a **reader**, which converts this expression into an internal form appropriate for the computer.

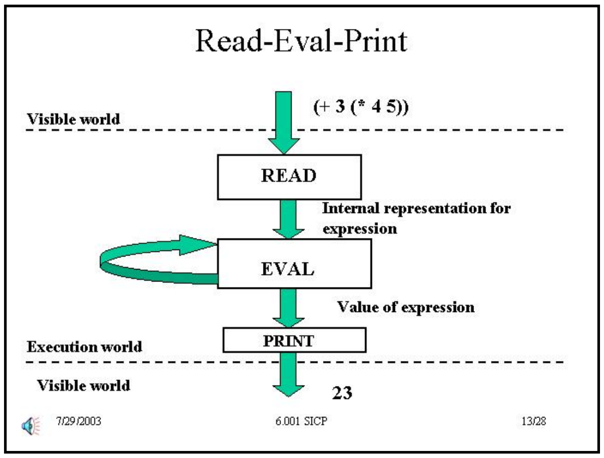
Slide 1.3.11
That form is then passed to a process called an **evaluator**. This encapsulates our rules for evaluation, and reduces the expression to its value.

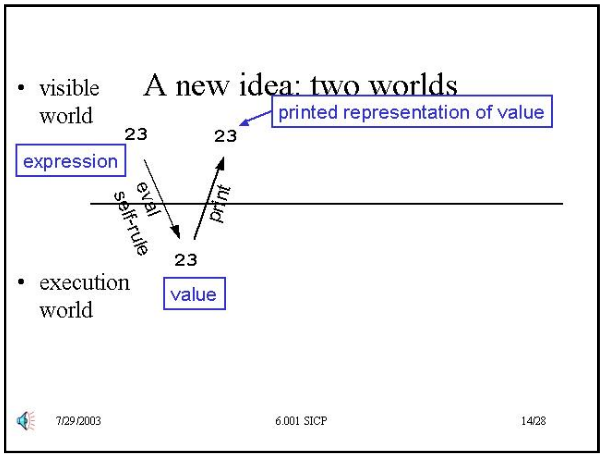




Slide 1.3.12
Note that this may involve a recursive application of the evaluation rules, if the expression is a compound one, as we saw with our nested arithmetic expressions.

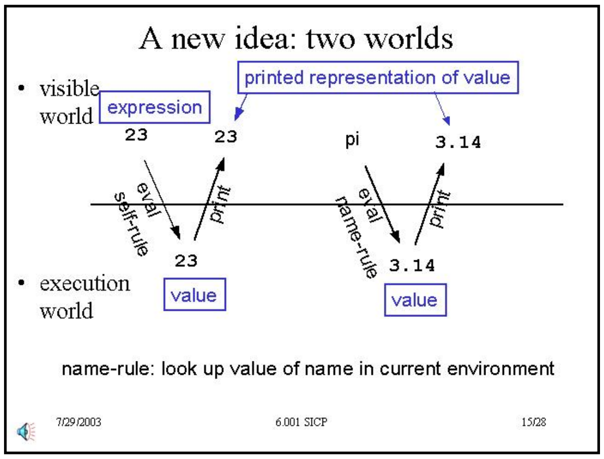
Slide 1.3.13
And the result is then passed to a **print** process, which converts it into a human readable form, and outputs it onto the screen.

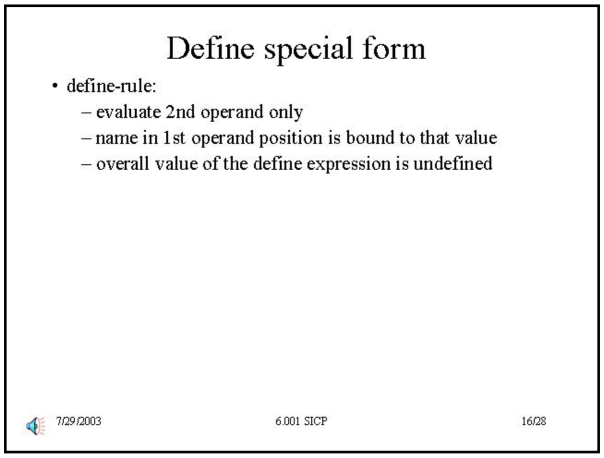




Slide 1.3.14
Suppose, for example, we type the expression **23** into the computer, and ask for its value. The computer basically recognizes what type of expression this is (self-evaluating in this case) and therefore applies the rule for self-evaluating expressions. This causes the expression to be converted into an internal representation of itself, in this case some binary representation of the same number. For self-evaluating expressions, the value is the expression itself, so the computer simply returns that value to the print procedure, which prints the result on the screen for us to see.

Slide 1.3.15
A second kind of primitive object is a name for something, typically created by evaluating a **define** expression. Remember that such a **define** created a pairing of a name and a value in a structure we call an environment. When we ask the computer to evaluate an expression such as **pi**, it recognizes the type of expression (a name), and applies the name rule. This causes the computer internally to find the pairing or association of that name in the environment, and to return that value as the value of the expression. This gets handed to the print procedure, which prints the result on the screen. Note that the internal representation of the value may be different from what is printed out for us to see.





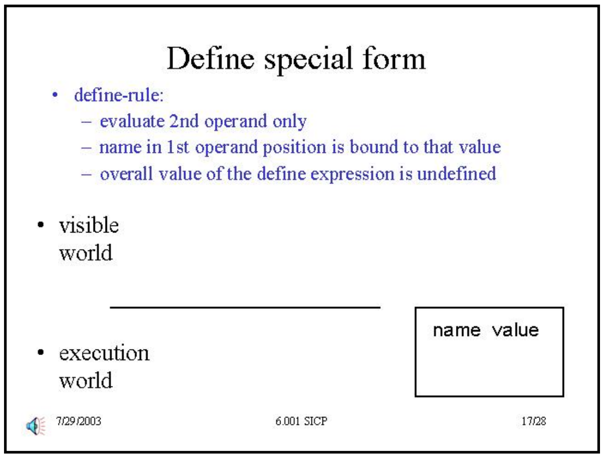
Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined

7/29/2003 6.001 SICP 16/28

Slide 1.3.16
What about special forms? Well, the first one we saw was a **define**. Here the rules are different. We first apply our evaluation rules to the second sub-expression of the **define**. Once we have determined that value, we then take the first subexpression (without evaluation) and create a pairing of that name and the computed value in a structure called an environment.
Since the goal of the define expression is to create this pairing, we don't really care about a value of the **define** expression itself. It is just used for the side effect of creating the pairing. Thus, typically we leave the value returned by a define expression as unspecified.

Slide 1.3.17
So let's see what happens in our two worlds view. Suppose we type in a **define** expression and evaluate it. What happens?

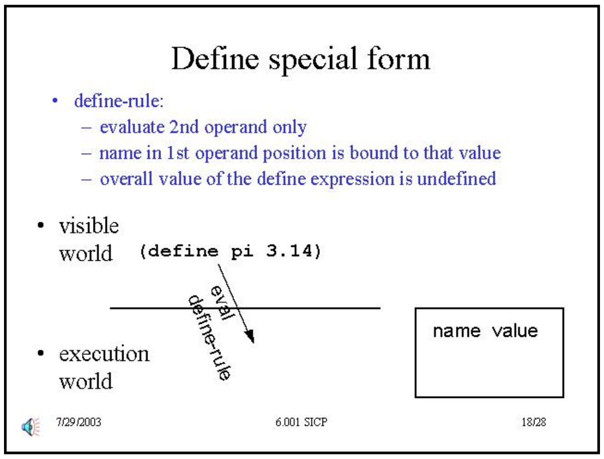


Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined
- visible world
- execution world

name value
7/29/2003
6.001 SICP
17/28

Slide 1.3.18
Here's an example. The evaluator first identifies the type of expression, in this case by recognizing the key word **define** at the beginning of the compound expression. Thus it applies the rule we just described.

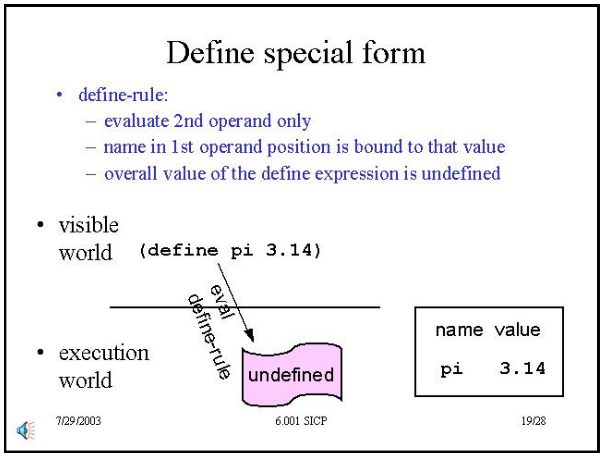


Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined
- visible world
- execution world

(define pi 3.14)
eval
define-rule
name value
7/29/2003
6.001 SICP
18/28

Slide 1.3.19
As we saw, the evaluator now takes the second sub-expression and evaluates it, using the same rules. In this case, we have a number, so the self-evaluation rule is applied. That value is then paired with the name, or first sub-expression, gluing these two things together in a table somewhere (we don't worry about the details of the table or environment for now, we'll discuss that in detail later in the term). As noted, the actual value of the **define** expression is not specified, and in many Scheme implementations we use a particular, "undefined", value.

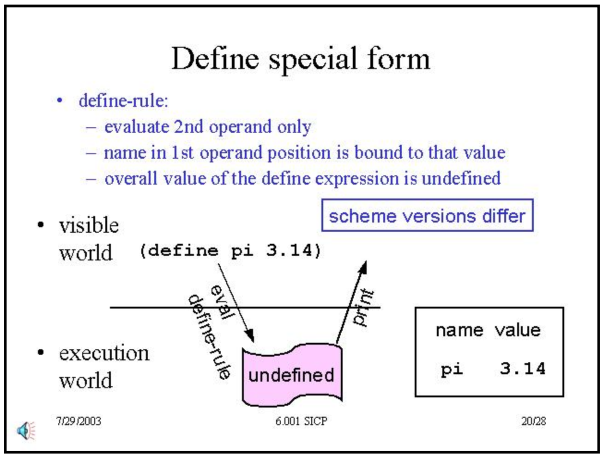


Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined
- visible world
- execution world

(define pi 3.14)
eval
define-rule
undefined
name value
pi 3.14
7/29/2003
6.001 SICP
19/28

Slide 1.3.20
As a consequence, the value that gets returned back up to the visible world may vary in different implementations of Scheme.

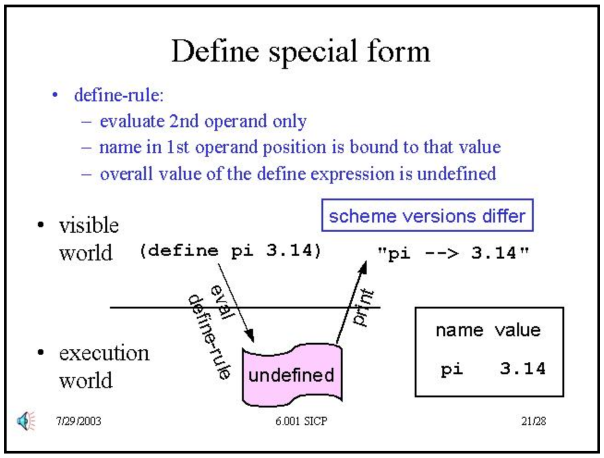


Define special form

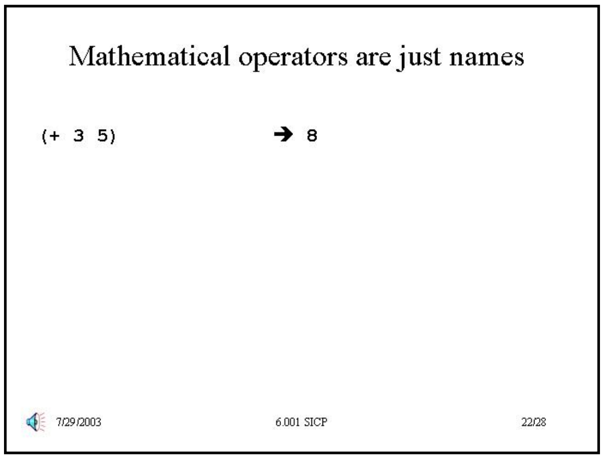
- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined
- visible world
- execution world

scheme versions differ
(define pi 3.14)
eval
define-rule
print
undefined
name value
pi 3.14
7/29/2003
6.001 SICP
20/28

Slide 1.3.21
Most versions of Scheme will show us some information about what binding was just created, but in general, we will not rely on this, since it does vary with implementations of Scheme.


Define special form

- define-rule:
  - evaluate 2nd operand only
  - name in 1st operand position is bound to that value
  - overall value of the define expression is undefined

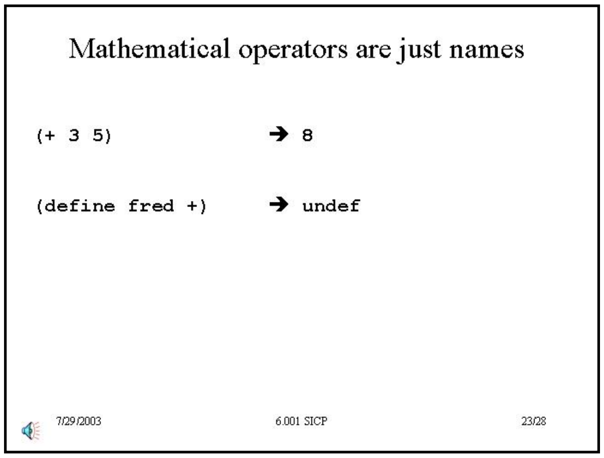

Mathematical operators are just names

```
(+ 3 5)            → 8
```

7/29/2003
6.001 SICP
22/28

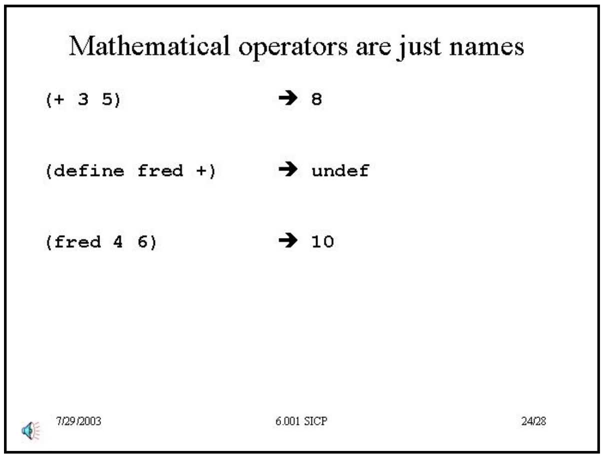
Slide 1.3.22
These rules hold for any expression. If we just have a simple combination involving a built-in arithmetic procedure, we know that we get the values of the other sub-expressions (using the self-evaluation rule), then apply the value associated with the symbol **+** to those values, thus executing an addition operation.

Slide 1.3.23
But suppose we do something strange, like this. Our rule for **defines** says that we get the value associated with **+** and bind it together with the name **fred** in our environment. Remember that **+** is just a name, so we use our name rule to find its value, which is in fact the actual internal procedure for addition.


Mathematical operators are just names

```
(+ 3 5)            → 8

(define fred +)    → undef
```

7/29/2003
6.001 SICP
23/28

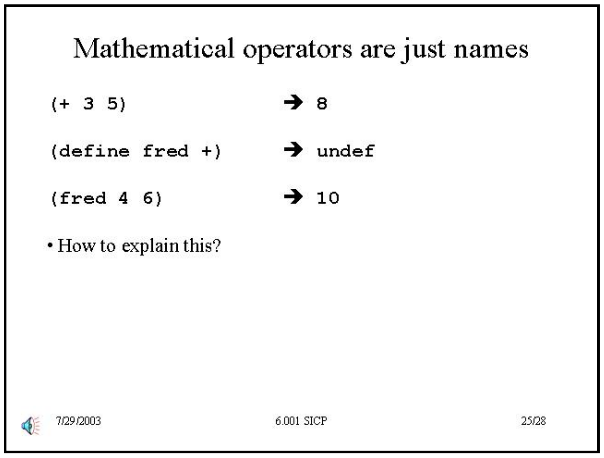

Mathematical operators are just names

```
(+ 3 5)            → 8

(define fred +)    → undef

(fred 4 6)         → 10
```

7/29/2003
6.001 SICP
24/28

Slide 1.3.24
Now, let's apply **fred** to some arguments. Notice that this just appears to do addition.

Slide 1.3.25
Is that really right?
Yes it is, and let's think about why.

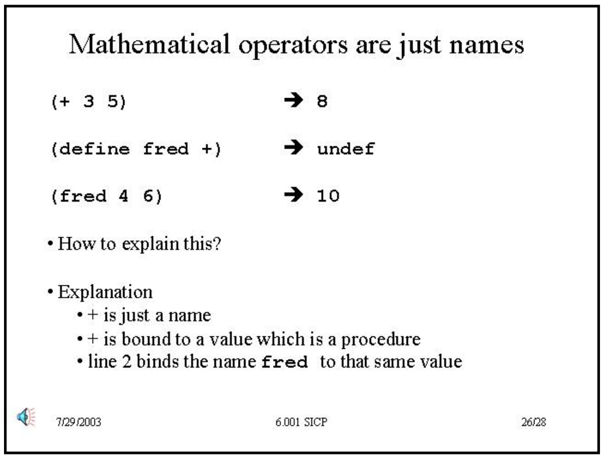


Mathematical operators are just names

```
(+ 3 5)              → 8
(define fred +)      → undef
(fred 4 6)           → 10
```

• How to explain this?

7/29/2003 6.001 SICP 25/28



Mathematical operators are just names

```
(+ 3 5)              → 8
(define fred +)      → undef
(fred 4 6)           → 10
```

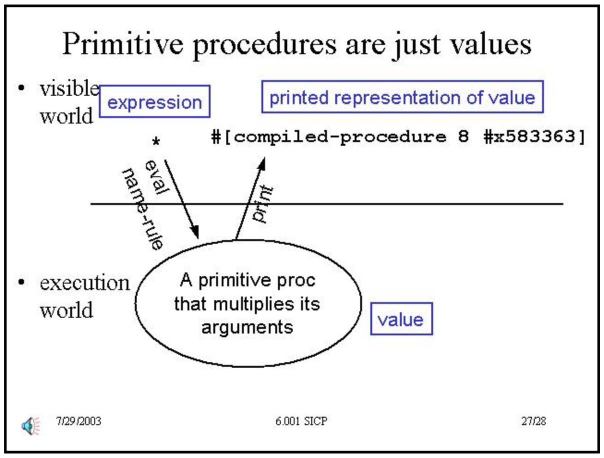
• How to explain this?

• Explanation
  • + is just a name
  • + is bound to a value which is a procedure
  • line 2 binds the name `fred` to that same value

7/29/2003 6.001 SICP 26/28

Slide 1.3.26
Well, our rules really explain this. The **define** expression says to pair the name **fred** with the value of **+**. Note that **+** is just a name, it happens to be one that was created when Scheme was started. Its value, we saw, is a procedure, the internal procedure for addition. Hence, the **define** expression creates a binding for fred to addition.
Thus, when we evaluate the combination, our rule says to get the values of the sub-expressions, and hence the name **fred** is evaluated using the name rule, to get the addition procedure. This is then applied to the values of the numbers to generate the displayed result.

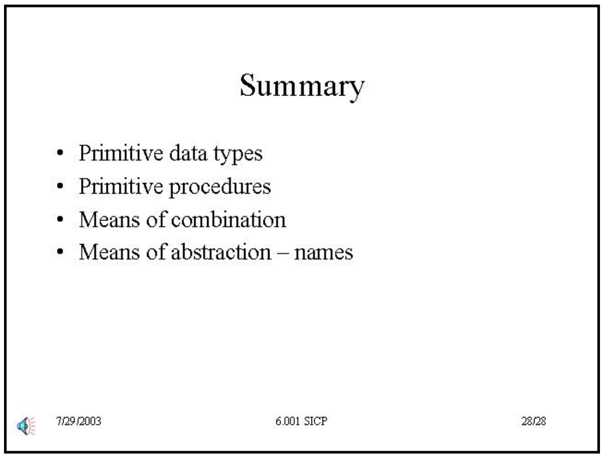
Slide 1.3.27
As an example, if I ask for the value associated with one of these built-in names, my rules explain what happens. Since this is a name, its value is looked up in the environment. In this case, that value is some internal representation of the procedure for multiplication, for example, a pointer to the part of the internal arithmetic unit that does multiplication. That value is returned as the value of this expression, and the print procedure then displays the result, showing some representation of where the procedure lies within the machine. The main issue is to see that this symbol has a value associated with it, in this case a primitive procedure.



Primitive procedures are just values

• visible world
expression
printed representation of value
*
#[compiled-procedure 8 #x583363]
eval
name-rule
print
• execution world
A primitive proc that multiplies its arguments
value

7/29/2003 6.001 SICP 27/28



Summary

- Primitive data types
- Primitive procedures
- Means of combination
- Means of abstraction – names

7/29/2003 6.001 SICP 28/28

Slide 1.3.28

Thus what we have seen so far is ways to utilize primitive data and procedures, ways to create combinations, and ways to give names to things. In the next section, we will turn to the idea of capturing common patterns in procedures.