

6.001: Structure and Interpretation of Computer Programs

- Today
 - The structure of 6.001
 - The content of 6.001
 - Beginning to Scheme

9/7/2005

6.001 SICP

1/44

What is the focus of 6.001?

- This course is about **Computer Science**
- Geometry was once equally misunderstood.
 - The term comes from *ghia* & *metra* or earth & measure
 - But in fact it's about...
- Computer Science deals with *imperative* or "how to" knowledge

9/7/2005

6.001 SICP

15/44

Declarative Knowledge

- "What is true" knowledge

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

9/7/2005

6.001 SICP

16/44

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 $x = 2$.

$X = 2$	$G = 1$

9/7/2005

6.001 SICP

17/44

"How to" knowledge

Why "how to" knowledge?

- Could just store tons of "what is" information
- Much more useful to capture "how to" knowledge – a series of steps to be followed to deduce a particular value
 - a recipe
 - called a **procedure**
- Actual evolution of steps inside machine for a particular version of the problem – called a **process**
- Distinguish between procedure (recipe for square root in general) and process (computation of specific result)

9/7/2005

6.001 SICP

23/44

Describing "How to" knowledge

Need a language for describing processes:

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

9/7/2005

6.001 SICP

24/44

Using procedures to control complexity

Goals:

- Create a set of primitive elements– simple data and 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? -- Can create complex procedures while suppressing details

Target:

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

9/7/2005

6.001 SICP

25/44

Key Ideas in 6.001

- Management of complexity:
 - Procedure and data abstraction
 - Conventional interfaces & programming paradigms
 - manifest typing
 - streams
 - object oriented programming
 - Metalinguistic abstraction:
 - creating new languages
 - evaluators

9/7/2005

6.001 SICP

26/44

Computation as a metaphor

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

9/7/2005

6.001 SICP

28/44

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 in general, independent of specific instance

9/7/2005

6.001 SICP

29/44

Representing basic information

- Numbers
 - Primitive 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

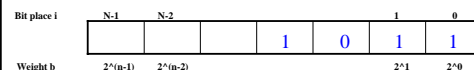
9/7/2005

6.001 SICP

34/44

Binary numbers and operations

- Unsigned integers



$$\sum_{i=0}^{n-1} b_i \cdot 2^i \quad \text{where } b_i \text{ is 0 or 1}$$

9/7/2005

6.001 SICP

35/44

Binary numbers and operations

- Addition

$$\begin{array}{r} 0 \quad 0 \quad 1 \quad 1 \\ +0 \quad +1 \quad +0 \quad +1 \\ \hline 0 \quad 1 \quad 1 \quad 10 \end{array}$$

$$\begin{array}{r} 10101 \\ \quad 111 \\ \hline 11100 \end{array}$$

9/7/2005

6.001 SICP

36/44

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!!**

9/7/2005

6.001 SICP

37/44

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
- Can then focus on using these basic elements to construct more complex processes

9/7/2005

6.001 SICP

38/44

Our language for 6.001

- Scheme
 - Invented in 1975
- Dialect of Lisp
 - Invented in 1959

9/7/2005

6.001 SICP

39/44

Rules for describing processes in Scheme

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

9/7/2005

6.001 SICP

40/44

Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

9/7/2005

6.001 SICP

41/44

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

9/7/2005

6.001 SICP

42/44

Language elements – primitives

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

9/7/2005

6.001 SICP

44/44

Language elements – primitives

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

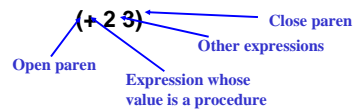
9/7/2005

6.001 SICP

45/44

Language elements – combinations

- How do we create expressions using these procedures?



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

9/7/2005

6.001 SICP

46/44

Language elements - combinations

- Can use nested combinations – just apply rules recursively
 - `(+ (* 2 3) 4) → 10`
 - `(* (+ 3 4) (- 8 2)) → 42`

9/7/2005

6.001 SICP

47/44

Language elements -- abstractions

- In order to abstract an expression, need way to give it a name
(define score 23)

9/7/2005

6.001 SICP

48/44

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

9/7/2005

6.001 SICP

50/44

Scheme Basics

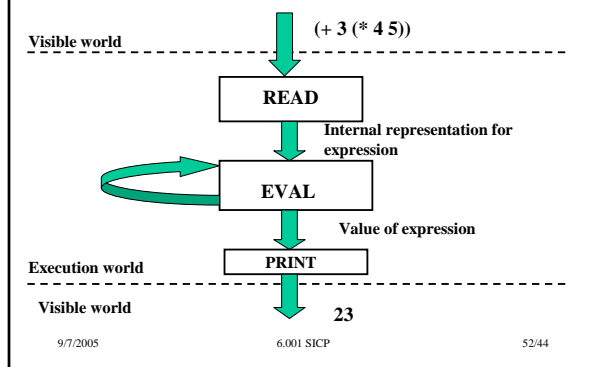
- Rules for evaluation
 - If **self-evaluating**, return value.
 - If a **name**, return value associated with name in environment.
 - If a **special form**, do something special.
 - If a **combination**, then
 - Evaluate all of the subexpressions of combination (in any order)
 - apply the operator to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

9/7/2005

6.001 SICP

51/44

Read-Eval-Print

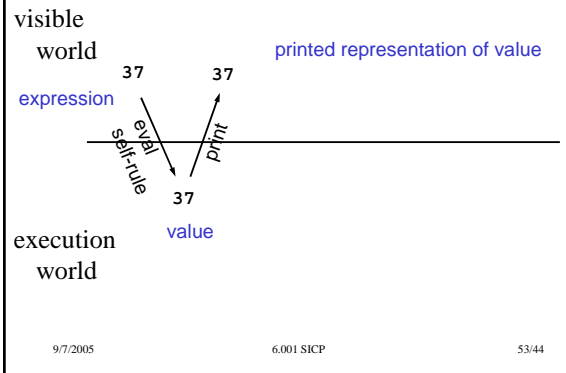


9/7/2005

6.001 SICP

52/44

A new idea: two worlds



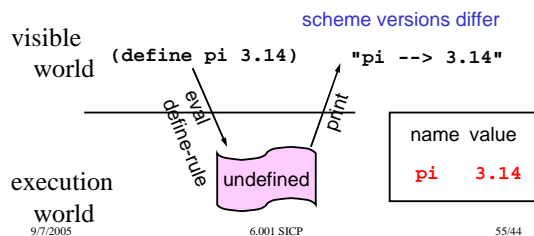
9/7/2005

6.001 SICP

53/44

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



9/7/2005

6.001 SICP

55/44

Mathematical operators are just names

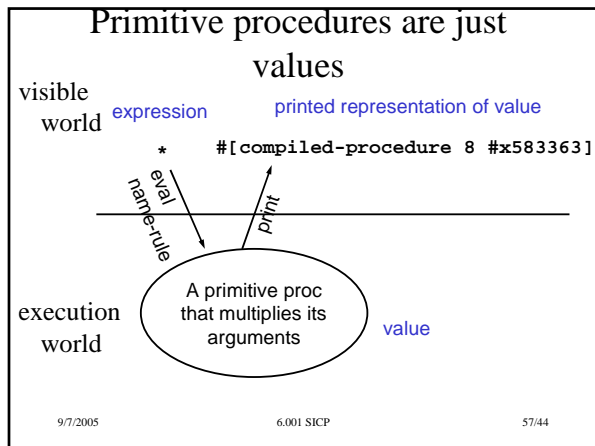
```
(+ 3 5)           → 8
(define fred +)   → undef
(fred 4 6)        → 10
```

- 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

9/7/2005

6.001 SICP

56/44



- ### Summary
- Primitive data types
 - Primitive procedures
 - Means of combination
 - Means of abstraction – names
- 9/7/2005 6.001 SICP 58/44