6.001: Structure and Interpretation of Computer Programs

• Today
  – The structure of 6.001
  – The content of 6.001
  – Beginning to Scheme

Course structure

• Lectures
  – Delivered live here, twice a week (Tuesday and Thursday)
  – Versions of lectures also available on the web site, as audio annotated Power Point. Treat this like a live textbook. Versions are not identical to live lecture, but cover same material.
• Recitations
  – Twice a week (Wednesday and Friday)
  – Don’t go to recitation assigned by registrar: check the web site for your assigned section. If you have conflict, contact course secretary by EMAIL only.
  – You are expected to have attended the lecture (or listened to the online version) before recitation
  – Opportunity to reinforce ideas, learn details, clarify uncertainties, apply techniques to problems
• Tutorials
  – Once a week (typically Monday)
  – Mandatory
  – Ask questions, participate in active learning setting

Contact information

• Web site: http://sicp.csail.mit.edu/
• Course secretary
  – Donna Kaufman, dkauf@mit.edu, 38-409a, 3-4624
• Instructor in charge
  – Eric Grimson, welg@csail.mit.edu
• Lecturer
  – Trevor Darrell, trevor@csail.mit.edu

Section Instructors

Regina Barzilay
Mike Collins
Piotr Indyk
David Karger
Kimberle Koile
Jacob Strauss
Other logistics

- Problem sets
  - Are released through the online tutor (see website for link – will create login for you when first used)
  - Are due electronically on the date posted
  - Includes lecture problems which are due on day of associated recitation
  - First one was posted today!
- Projects
  - First one was released today
  - Check website for updates

Other Issues

- Collaboration – Read description on web site
- Use of bibles – see description on web site
- Time spent on course
  - Survey shows 15-18 hours/week
  - Seeking help
    - Lab assistants
    - Other sources
- 6.001 Lab – 34-501
- Combination
  - Inner door: ????
  - Outer door: ???? (evenings, weekends)
Other Issues

- Slides: You have most of them.
- Because sometimes…
  - there are answers to problems
  - there are jokes
  - it’s good to pay attention

6.001

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

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 computation; knowledge about how to compute something
- Imperative knowledge

Declarative Knowledge

- “What is true” knowledge
  \[ \sqrt{x} \text{ is the } y \text{ such that } y^2 = x \text{ and } y \geq 0 \]

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 \( x = 2 \).

<table>
<thead>
<tr>
<th>X</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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 \( x = 2 \).

<table>
<thead>
<tr>
<th>X</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>X/G</td>
<td>G = \frac{1}{2} (1 + 2) = 1.5</td>
</tr>
</tbody>
</table>

\[ x = 2 \quad G = 1 \]
\[ \frac{x}{G} = 2 \quad G = \frac{1}{2} (1 + 2) = 1.5 \]
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 $x = 2$.

<table>
<thead>
<tr>
<th>$x$</th>
<th>$G_0$</th>
<th>$G_1$</th>
<th>$G_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1.5</td>
<td>1.414</td>
</tr>
<tr>
<td>$x/G$=2</td>
<td>$G = \frac{1}{2} (1 + 2) = 1.5$</td>
<td>$G = \frac{1}{2} (3/2 + 4/3) = 17/12 = 1.416666$</td>
<td>1.4142156</td>
</tr>
<tr>
<td>$x/G$=4/3</td>
<td>$G = \frac{1}{2} (1 + 2) = 1.5$</td>
<td>$G = \frac{1}{2} (3/2 + 4/3) = 17/12 = 1.416666$</td>
<td>1.4142156</td>
</tr>
<tr>
<td>$x/G$=24/17</td>
<td>$G = \frac{1}{2} (17/12 + 24/17) = 1.4142156$</td>
<td>$G = \frac{1}{2} (3/2 + 4/3) = 17/12 = 1.416666$</td>
<td>1.4142156</td>
</tr>
</tbody>
</table>

“How to” knowledge

Why “how to” knowledge?

• Could just store tons of “what is” information

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

15 minutes
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.

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

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

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

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
Binary numbers and operations

- **Unsigned integers**
  
<table>
<thead>
<tr>
<th>N-1</th>
<th>N-2</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{n-1}$</td>
<td>$2^{n-2}$</td>
<td>$2^1$</td>
<td>$2^0$</td>
<td>$b_n$</td>
<td>$b_{n-1}$</td>
</tr>
</tbody>
</table>

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

- **Addition**

  \[
  \begin{array}{c|c|c|c|c|c}
  & 0 & 0 & 1 & 1 \\
  + 0 & +1 & +0 & +1 \\
  \hline
  0 & 1 & 1 & 10 \\
  \end{array}
  \]

  \[
  \begin{array}{c|c}
  10101 & 111 \\
  11100 & 11100 \\
  \end{array}
  \]

- 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!!

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

Representing basic information, Again

- Numbers
  - Primitive element – single binary variable
    - Takes on one of two values (0 or 1)
    - Represents one bit (binary digit) of information

Where *Are* The 0’s and 1’s?
Where *Are* The 0’s and 1’s?

Where *Are* The 0’s and 1’s?

Our language for 6.001

• Scheme
  – Invented in 1975
• Dialect of Lisp
  – Invented in 1959

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.

Kinds of Language Constructs

• Primitives
• Means of combination
• Means of abstraction

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
Language elements – primitives

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

Language elements – combinations

- How do we create expressions using these procedures?

Language elements - combinations

- Can use nested combinations – just apply rules recursively

Language elements -- abstractions

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

(define score 23)
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

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))
  
  - This creates a loop in our system, can create a complex thing, name it, treat it as primitive

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

Read-Eval-Print

A new idea: two worlds
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

(define pi 3.14)

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

Primitive procedures are just values

A primitive proc that multiplies its arguments

Summary

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