6.001: Structure and Interpretation of Computer Programs

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

Main sources of information on logistics:
  – General information handout
  – Course web page
    - http://sicp.csail.mit.edu/

Course structure

• Lectures
  – Delivered live here, twice a week (Tuesday and Thursday)
  – Versions of lectures also available on the web site, as audio annotated
    PowerPoint. Treat this like a live textbook. Versions are not identical
    to live lecture, but cover roughly same material.
  – Because lecture material is evolving, we strongly suggest that you attend
    live lectures, and use the online lectures as reinforcement.

• Recitations
  – Twice a week (Wednesday and Friday)
  – For Wednesday, 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, some on Tuesday)
  – You should really be there – we provide a “carrot” to encourage you
    – Ask questions, participate in active learning setting

Grades

• 2 mid-term quizzes – 25%
• Final exam – 25%
• 1 introductory project and 5 extended programming
  projects – 40%
• weekly problem sets – 10 %
  BUT YOU MUST ATTEMPT ALL OR COULD
  RESULT IN FAILING GRADE!!
• Participation in tutorials and recitations – up to 5%
  bonus points!!

Contact information

• Web site: http://sicp.csail.mit.edu/
• Course secretary
  – Donna Kaufman, dkauf@mit.edu, 38-409a, 3-4624
• Instructor in charge, lecturer
  – Eric Grimson, welg@csail.mit.edu
• Co-lecturer
  – Rob Miller, rcm@csail.mit.edu

Section Instructors

Prof. Michael Collins

Gerald Dalley

Prof. Berthold Horn

Dr. Kimberle Koile

Prof. Peter Szolovits
Other logistics

- **Problem sets**
  - Are released through the online tutor (see website for link – there is a separate link to register for the tutor)
  - Are due electronically on the date posted
  - Includes lecture problems
    - You should really try to do these problems before the associated recitation
    - If section instructors find that too many students are coming unprepared, we will change these problems to be due on day of associated recitation!
  - First one was posted today!

- **Projects**
  - First one will be 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 – departmental tutoring services, institute tutoring services (ask for help if you think you need it)
- **6.001 Lab** – 34-501
- **Combination**
  - Inner door: 04862*
  - Outer door: 94210 (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
Getting assigned to a recitation

- We are **NOT** going to use the registrar’s recitation assignments
- Please take a few minutes to fill out the sign up sheet
  - Turn in at the end of lecture
- We will post assignments for tomorrow’s section later this afternoon on the course web site

---

6.001

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

---

What is the main focus of 6.001?

- This course is about **Computer Science**
- Geometry was once equally misunderstood.
- Term comes from *ghia* & *metra* or earth & measure – suggests geometry is about surveying
- But in fact it’s about…
- By analogy, computer science deals with *computation*; knowledge about *how to compute* things
- 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: \( \sqrt{2} \) for \( x = 2 \).

\[
\begin{array}{c|c|c|c|c|c}
X & G & \text{X/G} & G = 1/2 (1 + 2) & 1.5 \\
\hline
2 & 1 & 1.5 & & \\
\end{array}
\]

---

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 \( x = 2 \).

\[
\begin{array}{c|c|c|c|c|c}
X & G & \text{X/G} & G = 1/2 (1 + 2) & 1.5 \\
\hline
2 & 1 & 1.5 & & \\
\end{array}
\]
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: \( \sqrt{2} \) for \( x = 2 \).

| \( x = 2 \) | \( G = 1 \) |
| \( x/G = 2 \) | \( G = \frac{1}{2} (1 + 2) = 1.5 \) |
| \( x/G = 4/3 \) | \( G = \frac{1}{2} (3/2 + 4/3) = 17/12 = 1.416666 \) |

"How to" knowledge

Why "how to" knowledge?
- Could just store tons of "what is" information

Describing "How to" knowledge

If we want to describe processes, we will need a language:
- 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: Given a specific problem domain, we need to
• 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 abstraction? → Can create complex procedures while suppressing details

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

Key Ideas of 6.001

• Linguistic perspective on engineering design
  – Primitives
  – Means of combination
  – Means of abstraction
  – Means for capturing common patterns
• Controlling complexity
  – Procedural and data abstractions
  – Recursive programming, higher order procedures
• Functional programming versus object oriented programming
• Metalinguistic abstraction
  – Creating new languages
  – Creating evaluators

But no HASS credit!

Look at generic elements, but also at how to design for specific problem domain

6.001

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

Computation as a metaphor

• Capture descriptions of computational processes
• Use abstractly to design solutions to complex problems
• Use a language to describe processes

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
• What are basic units on which to describe procedures?
  – Need to represent information somehow

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, other encodings
• Words
  – Collections of characters, separated by spaces, other delimiters
Binary numbers and operations

• Unsigned integers

<table>
<thead>
<tr>
<th>Bit place i</th>
<th>(2^{i-1})</th>
<th>(2^{i-2})</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight (b_i)</td>
<td>(2^i)</td>
<td>(2^{i-1})</td>
<td>(2^{i-2})</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\[1 + 2 + 8 = 11\]

Binary numbers and operations

• Addition

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td>+0</td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

\[10101\]

\[111\]

\[11100\]

Binary numbers and operations

• Can extend to signed integers (reserve one bit to denote positive versus negative)

• Can extend to character encodings (use some high order bits to mark characters versus numbers, plus encoding)

Where Are The 0’s and 1’s?

Where Are The 0’s and 1’s?
… we don’t care at some level!

• Dealing with procedures at level of bits is way too low-level!
• From perspective of language designer, simply need to know the interface between
  – Internal machine representation of bits of information, and
  – Abstractions for representing higher-order pieces of information, plus
  – Primitive, or built-in, procedures for crossing this boundary
  • you give the procedure a higher-order element, it converts to internal representation, runs some machinery, and returns a higher-order element

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, together with a “contract” that assures a particular kind of output, given legal input
• Can then focus on using these basic elements to construct more complex processes

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, hence every (almost) expression 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
George Boole

A Founder

An Investigation of the Laws of Thought, 1854
-- “a calculus of symbolic reasoning”

Language elements – primitives

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

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

Language elements – combinations

- How do we create expressions using these procedures?

  (+ (* 2 3) 4) → 10

  (* (+ 3 4)
   (- 8 2)) → 42

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
  \[ \text{score} \rightarrow 23 \]
  - Note that we already did this for +, *, ...
  \[(\text{define total} (+ 12 13)) \rightarrow 92\]
- 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 Loop

A new idea: two worlds

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

\[(\text{define pi} 3.14) \rightarrow \text{"pi --> 3.14"}\]
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

Summary

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