6.001: Structure and Interpretation of Computer Programs

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

6.001 – contact information
- Course secretary
  - Donna Kaufman, dkauf@mit.edu, 38-409a, 3-4624
- Course head
  - Prof. Eric Grimson, welg@csail.mit.edu
- Lecturers
  - Prof. Trevor Darrell, trevor@csail.mit.edu
  - Prof. Peter Szolovits, welg@csail.mit.edu
- Web site: 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 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. We are going to do section assignments based on the scheduling form that you fill out today. Please 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

6.001: Structure and Interpretation of Computer Programs
- Grades
  - 2 mid-term quizzes – 25%
  - Final exam – 25%
  - 1 introductory project and 4 extended programming projects – 30%
  - weekly problem sets – 10 % BUT MUST ATTEMPT ALL OR COULD RESULT IN FAILING GRADE!!
  - Participation in tutorials and recitations – 10%

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
6.001: Structure and Interpretation of Computer Programs

- Time expectations
  - Survey shows 15-18 hours/week
  - Seeking help
    - Lab assistants
    - Other sources
  - 6.001 Lab – 34 01
  - Combination – xxxxx
  - Collaboration and use of bibles…

Collaboration and Bible Policy

- See web for full discussion
- Problem sets – individual work
- Projects – working with one or two others encouraged:
  - each person must be involved in all aspects of project
  - must list all collaborators
  - must write up work separately

A database of 6.001 problem sets and solutions is available on the 6.001 Web page and in the 6.001 locker on Athena.
- Can use for independent study, or for extra problems.
- **Do not use bibles as a source of code or solutions to any of this year’s assignments.**
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

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 \( x = 2 \).

| \( x/G = \frac{24}{17} \) | \( G = \frac{4}{3} \) | \( X/G = \frac{17}{12} \) | \( G = \frac{17}{12} + \frac{24}{17} = \frac{577}{408} \) = 1.4142156 |
| \( X/G = \frac{3}{2} \) | \( G = \frac{3}{2} + \frac{4}{3} = \frac{17}{12} \) = 1.416666 |
| \( X/G = 2 \) | \( G = \frac{2}{1} \) | \( X/G = 1 \) | \( G = \frac{1}{1} \) |

“How to” knowledge

Why “how to” knowledge?

- Could just store tons of “what is” information (e.g. tables of logs from the 1970’s)
- Much more useful to capture “how to” knowledge – a series of steps to be followed to deduce a particular value (e.g. the mechanism underlying the log function on a calculator)
  - 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) and process (actual computation)

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
### Using procedures to control complexity

- 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?** — Develop complex procedures while suppressing details

`Build complex systems while maintaining robustness, efficiency, 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

### 6.001: Structure and Interpretation of Computer Programs

- Today
  - The structure of 6.001
  - The content of 6.001
  - Scheme Basics

### Computation and complex problem solving

- Capture descriptions of computational processes
- Use abstraction 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

### Primitives for representing basic information – what is the right level?

- 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
Binary numbers and operations

• Unsigned integers

\[
\begin{array}{c|cccc}
\text{Bit place} & 0 & 1 & 2 & 3 \\
\hline
\text{Weight} & 2^0 & 2^1 & 2^2 & 2^3 \\
\text{Value} & 1 & 0 & 1 & 1 \\
\end{array}
\]

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

\[
2^3 + 2^1 + 2^0 = 8 + 2 + 1 = 11
\]

• Addition

\[
\begin{array}{c|c|c|c|c}
\text{0} & \text{0} & +1 & \text{1} & \text{1} \\
\hline
\text{0} & 1 & 1 & \text{10} \\
\text{1} & \text{1} & \text{11} \\
\text{1} & \text{0} & \text{1100} & \text{28} \\
\end{array}
\]

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!!

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

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, - 3 5 , 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: length, string=?
  – Booleans: boolean/and, boolean/or, not

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

Language elements – combinations

• How do we create expressions using these procedures?
  \( (+ 2 3) \)
  \( \text{Open paren} \)
  \( \text{Expression whose value is a procedure} \)
  \( \text{Close paren} \)
  \( \text{Other expressions} \)

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

Language elements – combinations

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

  `score \rightarrow 23`
  - Note that we already did this for `+`, `*`, ...

**(define total (+ 12 13))**

  `(* 100 (/ score total)) \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

Mathematical operators are just
names

**(+ 3 5)** \(\rightarrow 8\)

**(define fred +)** \(\rightarrow\) **undef**

**(fred 4 6)** \(\rightarrow 10\)

- 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