Lecture 1: Scheme and black-box abstractions

Links

- Slides: https://docs.google.com/presentation/d/1sFfJ2bANtAXR7cJN79Un-7gWCHkiH8rSkEtrqDubr3A/edit?usp=sharing
- DrRacket: https://racket-lang.org/
- **Code: https://pastebin.com/Jd6TbCbW**
- Syllabus: https://docs.google.com/document/d/1KxD2l3xKJTLi2VTOAw5bibweRg1IVgCmeTe_QAmCU2k/edit?usp=sharing
- Pre-class survey: https://forms.gle/FxTyVoQYoVUHQAoYA
- **Lecture 1 feedback: https://forms.gle/JjsXSdZdEMhmdkP67**
- The book: https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-4.html#%_toc_start
    - If you want a fuller understanding of the contents, you can read section 1.1.
    - There are exercises at the end of this file that refer to section 1.1 exercises.

Teacher introduction

- **Pleng (Thanadol Chomphoochan)** is an MIT junior studying Computer Science and Engineering. His interests are broad. He started with algorithms and data structures for competitive programming in high school, and recently spent more time on low-level systems, language design, and formal verification.
- **Adhami (Khaleel Al-Adhami)** is also another MIT junior studying CS. He co-founded MIT OpenCode, a club that works on developing and contributing to free and open source software (FOSS). Has a lot of experience with web development.
- **Give us feedback:** We are both first-time teachers for HSSP (though we do have teaching experience elsewhere). There is a lot to improve. Please feel free to give us feedback!
    - Raise your hand whenever you have questions or want something clarified.
    - If I'm going too fast, please feel free to raise your hand (maybe hesitantly; I'll get the message). In fact, Adhami is here to help control my pace!
    - Meet us after class or email us if you have any questions or issues.

Updated: 2023-02-25 4:44pm

    - Send anonymous feedback via Google form.
- **Logistics**
    - Lectures are 80 minutes long, with a bit of problem solving mixed in (about 3-5 min each). There will be a break in the middle.
    - Syllabus and course notes are available on the course website.
- **Contact information**
    - Email: C15493-teachers@esp.mit.edu

Why this course?

- **This course's name is "Learn Programming Like It's 1986,"** which is meant to catch your attention. A more fitting title might be "A Purist's Introduction to Programming." Either way, this is a programming course. If you're here, we assume you want to learn a thing or two about programming.
- You may ask: What's up with "1986"?
- **6.001:** This course is based on MIT's historical course, 6.001. There are freely available lectures for the course, recorded in 1986, hence the title of this class. The associated textbook is *Structure and Interpretation of Computer Programs* (SICP), which is touted by some as the best programming textbook.
- **Style:** The course is mainly college-style lectures with emphasis on theory. I understand that part of the appeal of programming is being able to build cool things, and that certainly is very important! However, this class is meant to give you a *specific perspective* on programming—a bottom-up way of learning how to program from first principles, as you could tell from the course name. We'll give you opportunities to practice in class so you have a bit of familiarity, so you can learn programming on your own after this. But most importantly, we hope that you are able to gain appreciation for the art of designing programs—the theory of it, just for the sake of it.
- **Agenda for today:**
    - Talk a bit about what programming is, at a high level.
    - We will teach the syntax of Scheme and write a few basic arithmetic programs. In the process, we introduce concepts like definitions, functions, conditionals, and recursion.
    - Hopefully we'll be able to get to recursion by the end of today and see how to write the example from the beginning of the lecture in Scheme. Don't worry if it feels a bit too fast the first time: This is mostly a preview. Next week we'll review this a bit more.

Essence of Programming

- In math, we like to study *what is true*. That's declarative knowledge.
    - Let's take for example, what we know about square roots. What is a square root? It's a number such that if you multiply it by itself you get the number you want. For example, the square root of 16 is 4 because 4 times 4 is 16. The square root of 2 is approximately 1.414; it is irrational.
    - We can write this down more formally like this: "*sqrt(x)* is the *y* such that $y*y = x$ and $y \geq 0$."
    - This is **declarative knowledge.** It tells us *what is true about square roots*. It doesn't tell us how to find it.
- Let's contrast this with **imperative knowledge**. How do you find a square root?
    - One way is to guess. Another way is to search a table.
    - Here's a really cool, very systematic way of estimating square roots—a piece of imperative knowledge we'll show here: "To find an approximation to *sqrt(x)*, first make a guess *g*. Improve the guess by averaging *g* and *x/g*. Keep improving until the guess is good enough."
    - Example:
        - To find a square root of *3*, let's guess the square root is *g=1*.
        - *g=1,* so *3/g = 3*. The average is 2. We use this as the next guess.
        - *g=2,* so *3/g = 1.5*. The average is 1.75. We use this as the next guess.
        - *g=1.75* so *3/g = 1.71429*. The average is 1.73214. We use this as the next guess.
        - *g=1.73214* so *3/g = 1.73196*. The average comes out to be about 1.732. Good enough.
    - Out of curiosity—raise your hands if you have seen this algorithm before.
- Programming is really all about expressing these **processes**.
    - Processes are like magical spirits inside your computer.
    - **Procedures** are magical incantations that allow you to manipulate those processes. There are rules and languages.
- **Lisp:** We will use Scheme. Scheme is a very minimal language in the family of Lisp languages. In fact, I claim that in only half an hour, you will have learned essentially all of Scheme. This is like learning the rules of chess, but this is very different from saying you have become a good chess player. Scheme is a great language for illustrating core ideas in computer science, without forcing you to deal with the quirks of languages like C, Java, Python, and others.
- **Like learning math:** Scheme is especially suited for blackboard teaching, so you don't have to rely on having access to computers. In some way it will feel almost like learning

math. (Though we will introduce the DrRacket IDE, integrated development environment, a bit later in the lecture today.)

- Obviously, CS is not about finding square roots. We want to talk about complex systems, like video games or scientific models. Those systems are too large for anyone to hold in their own head. Those systems are possible because there are **techniques for controlling complexity**. This is what CS is about, and what we will learn in the upcoming weeks will help us practice exactly that.

Black-box abstractions

- **Building boxes:** To be able to describe big systems, we need a way to take an idea and build a box about it, so we can use it elsewhere without thinking about the details. For example, we can have a "find a square root" box that takes in a number and outputs another number, e.g. it takes in *x* and outputs *sqrt(x)*.
- **Combining boxes:** To compute *sqrt(x)+sqrt(y)*, we can simply use the square root box twice. Note that the names "x" and "y" don't really mean anything inherently. They're just there to represent the idea of "any inputs" and allow us to distinguish the inputs.
    - Another example: right triangle hypotenuse. *sqrt(x*x + y*y)*.
- **Higher-order boxes:**
    - Consider a different problem: How do you find a value *x* such that *cos(x) = x*? Amazingly, due to some math properties, you can start with any guess and then keep applying the cosine function over and over. You will continually get closer to the answer you want.
    - **Question:** What is similar between the process for finding the square root of *x* and finding the fixed point of *cos*?
    - They're all like this: "Start with any guess *g*. Keep applying function *f* repeatedly on the guess until the guess is good enough."
        - For the square root of *x*, the function is *f(g) = average(g, x/g)*.
        - For the fixed point of cosine, the function is *f(g) = cos(g)*.
        - Note those functions can be thought of as boxes as well!
    - You can imagine we have a box called "do fixed-point-iteration on a function box." This box can take in a function, i.e. a box, and outputs the fixed point! e.g. it can take it the function *f(g) = average(g, x/g)*.
    - Furthermore, we can actually use this to create the square root box so it works for any input *x*.

Updated: 2023-02-25 4:44pm

Other techniques for controlling complexity

- **Conventional interfaces:** We want to enable generic operations. For example, the notion of "addition" doesn't apply to just numbers. They also apply to, e.g. adding vectors. To add two vectors, you add the coordinates. How do we encapsulate this idea so you can just talk about adding two vectors directly without talking about coordinates? How do you talk about adding functions together pointwise? We can do this by establishing conventional interfaces—something that people agree upon.
- **Metalinguistic abstraction:** We can build an entirely new language to help express the kind of ideas we want to be able to talk about.
- We do not have time to go through all of these for this class, but the book does if you are interested.

Scheme syntax

- Let's start learning Lisp now!
- When someone wants to show you a language, you don't ask "how many characters there are." You ask about those three fundamental elements of a powerful language:
    - Primitives: What are the "things" we have?
    - Means of combination: How do we combine things?
    - Means of abstraction: Can we combine things and give them a name?
- **Primitives:** First thing first we have numbers. If you put a number into DrRacket, it gives you back that number as a result!
- **Means of combination:** You can perform math operations on them.
    - Let's say: I wonder what 9 plus 10 is. In Lisp, we would write that as *(+ 9 10)*.
    - Lisp uses the **prefix notation** to describe combinations. In the example above, *+* is the **operation** or the **procedure**. The numbers are the **operands** or **arguments**.
    - Note that we can nest the combinations inside each other! For example, to compute the volume of a sphere with radius 20, we can write: `(* (/ 4 3) 3.141592653 20 20 20)`
    - **Parentheses have meanings!** In math, we usually use parentheses to clarify groupings and we can sometimes omit them. In Lisp, parentheses denote precisely the act of applying the operand (the first thing in the list) to the operands (the rest of the list). You can't leave them out. You also can't have extra.
    - **Pretty printing**: Sometimes we write really complicated expressions. The convention is to line up the operands. (See example code.)

Updated: 2023-02-25 4:44pm

    - **Practice**: Write the following math expressions in prefix notation:
        - $$\frac{5 + 4 + (2 - (3 - (6 + \frac{4}{5})))}{3(6 - 2)(2 - 7)}$$
- **Means of abstraction:**
    - In Lisp, you can define constants. This is the simplest kind of abstraction.
        - *(define pi 3.141592653)*
        - *(define radius 20)*
        - Compute *(* (/ 4 3) pi radius radius radius)*.
        - Can also *(define volume (* (/ 4 3) pi radius radius radius))* and compute *volume* as well. Try: compute *(* volume 5).*
    - Another more powerful abstraction is to give names to procedures. For example, I can give a name to the idea of multiplying a number by itself.
        - *(define (sq x) (* x x))*
            - Reads: To square *x*, multiply *x* by *x*.
            - Can then compute, for example: *(sq (+5 7))* and *(sq (sq (sq (sq (sq 2)))))*
        - **In-class exercise—**Fill in the following:
          ```
          (define (sphere-volume radius)
            ...)
          ```
        - It's virtually impossible to distinguish between the built-in operations and user-defined operations. That's the point—it shouldn't matter!
        - Actually, note that the operands themselves are primitive **values** as well! For example, if you write *+* on its own, it stands for the addition operation.
- **Conditionals:** We need a way to represent case analysis in our program to be able to express more complicated ideas like finding the absolute value.
    - In math, absolute value is defined like this:

      $$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$
    - In Lisp, so far, we have primitives (numbers) and applying operations. Now we introduce a **special form** called the "cond" form, which looks like this in general:

        -
          ```
          (cond [condition₁ expression₁]
                [condition₂ expression₂]
                …
                [conditionₙ expressionₙ]
                [else expression_else])  ;else clause is optional
          ```

    - This form works by first checking each condition from top to bottom. If a condition is true, the expression on the right hand side is the result. You can optionally have the "else" clause at the bottom.
    - Important: The order matters! Once a condition is satisfied, all clauses below it are skipped.
    - In Racket, you can use brackets instead of parentheses. We usually use brackets for clauses in conditionals, so things are more readable.
    - For writing conditions, you can use operations like < and = to perform comparisons. Those are called **predicates/comparison operators**.

    -
      ```
      (define (abs x)
        (cond [(> x 0) x]
              [(= x 0) x]
              [else (- x)]))
      ```
    - Note the middle *(= x 0)* case is redundant.
    - Another way to write this is to use the `if` form:
      ```
      (define (abs x)
        (if (> x 0)
             x
             (- x)))
      ```
- Note: Truth values (called **booleans**, denoted #t and #f) are also primitives. We also have **strings** as primitives as well, denoted by writing quotation marks around the text.
- **In-class exercise:** Define *compute-grade*.

    -
      ```
      (define (compute-grade score)
        (cond [(and (>= score 90) (<= score 100)) "A"]
              …))
      ```

Installing DrRacket

- Actually, we will be using a language called "Racket," which for our intents and purposes, is basically an improved Scheme.
- DrRacket is an integrated development environment (IDE) we use for this class.
    - You can download and install DrRacket from the internet pretty easily.
    - You can put the code in the interaction pane at the bottom (or at the right, depending on where you put it) and get the results back immediately.
    - You can also put the code in the definition pane at the top, then click Run.
    - Do coding at the top. Use the bottom pane for experimentation.
- In class, we typed in all the previous examples we've written on the board so far to demonstrate DrRacket's working.

Updated: 2023-02-25 4:44pm

Square root demonstration

- Now, we'll demonstrate how to write a complicated program using what we know so far. This is also a look into what my thought process is usually like when writing programs.
- Recall: In order to find the square root of x, we have to first choose a guess. Given a guess, if the guess is good enough, we stop, otherwise we improve the guess and try again.
- We could write this idea as the "try" procedure

    -
      ```
      (define (try guess x)
        (if (good-enough? guess x)
            guess
            (try (improve guess x) x)))
      ```
- Then square root can be defined as follows: `(define (sqrt x) (try 1 x))`. Doesn't matter what the starting guess is.
- At this point, we don't know how "good-enough?" and "improve" work yet. That's fine - we are engaging in "wishful thinking." We assume that they work, then we fill in the details for them later.
- One thing about wishful thinking, however, is that we have to make sure we give *good-enough* and *improve* enough data to work with. For example, if *good-enough?* takes only one argument (*guess*), there is no way it could possibly do its job.
- See the attached code file for everything we implemented.
- It is also possible to hide all those helper functions within the definition of square root. This kind of block structure allows us to hide unnecessary details.

~~Number guessing game~~

- ~~The number guessing game! I am thinking of a number between 1 and 100. You need to guess what it is. I will tell you if it is too low, too high, or correct. How do you minimize the number of times you have to ask me in the worst case scenario?~~
- ~~First idea, very naive solution: Just try 1, 2, 3, …~~
- ~~Better idea: Try midpoint.~~
- ~~We are writing this strategy in Scheme. We assume the system has given us:~~
    - ~~A procedure (guess x), which, when called, will give us the result of 'h, 'l or 'c for too high, too low, or correct.~~
    - ~~You can compare this result using the eq? operator.~~

Updated: 2023-02-25 4:44pm

Extra Practice

The book is here:
https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-4.html#%_toc_start

- Exercise 1.1
- Exercise 1.3
- Exercise 1.4
- Implement this function: *(define (count-real-roots a b c) …)*!
  - For example, the polynomial x^2-1 (a=1, b=0, c=-1) has two roots (i.e. two values of x such that the polynomial becomes zero). So, calling (count-real-roots 1 0 -1) should output 2.
- Exercise 1.7 (challenging)

Applied Projects

- Just in case you want to spend **A LOT** of time getting practical experience.
- These are the topics. Choose one:
  - Calculating progressive tax rate
    - Input: Income, Output: Total tax you have to pay
  - Doomsday algorithm (calculate day of the week given any date)
    - Input: Month Day Year (as numbers), Output: Numbers representing Sunday through Saturday
  - Barcode checksum
    - Input: A number of a string that looks like a number, Output: whether it's correct or not.
  - Writing roman numerals
    - Input: A number. Output: A string.
  - "Number to words" generator
    - (number->words 3245) should output "three thousands, two hundreds and forty five"
    - Work incrementally! Try smaller procedures that accept only one digit numbers first, then two, …
    - You can use string-append to concatenate strings together. For example, (string-append "hi" "ho") outputs "hiho".
- Study the topic, and see if there's a way to automate it in Racket!
- **If you do choose to take a stab at a project, please let me know!**

Updated: 2023-02-25 4:44pm