SI 413: A program is never less than 90% complete, and never more than 95% complete.

Professor Keith Sullivan

Language Paradigms

- Declarative (what the computer is to do)

Functional	Lisp, Scheme, ML, Haskell
Dataflow	Id, Val
Logic	Prolog, SQL, spreadsheets

- Imperative (how the computer should do it)

von Neuman	C, Ada, Fortran, Cobol
Object Oriented	C++, Java, Smalltalk
Scripting	Python, Perl, PHP, Javascript

Imperative Programming Languages

Consider the following C++ fragment:

```
int x = 1;
x = 3;
x = x + 1;
```

- Each statement is a command
- Specifies actions and a specific ordering
- Expressions produce values, but **side effects** are often more important!

Functional Programming Languages

The output of a functional program is a mathematical function of the input. There is no internal state and no side effects.

Key features:

- Referential Transparency
  Value of a function does not depend on its context
  $(3x - x\sin(x)) - (3x - x\sin(x))$
  (2 - i++) - (2 - i++)
- Functions (and types) are first class
  Functions can be passed as arguments, created on-the-fly, and returned from other functions. Functions are data!
  (define (f x y) (x (x y)))
  (f sqrt 16)
- List types

Functional Programming Languages

Other common properties include

- Garbage collection
- Built-in types and operators
- Interpreters rather than compilers
- Extensive polymorphism
- Heavy use of subroutines

The Scheme Language

History of Scheme

- 1958: Lisp language invented by John McCarthy
  (based on Church's lambda calculus, alternative to Turing machines)
- 1958: Steve Russell writes `eval` in machine code,
  creates first Lisp interpreter
- 1962: First Lisp *compiler*, written in Lisp
- 1970s, 80s, 90s: Lisp is the dominant language for AI research
- 1975: Scheme created by Steele & Sussman:
  minimal Lisp dialect focused on functional programming
- 1985: *Structure and Interpretation of Computer Programs*:
  teaching Scheme in first-year at MIT
- 1991: *How to Design Programs*:
  teaching Scheme to beginners based on *design recipes*

Programming Language Vocabulary

Excerpt from the R6RS standard

Invented by Guy Lewis Steele Jr. and Gerald Jay Sussman, Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including functional, imperative, and message passing styles, find convenient expression in Scheme.

Reading this should give you a good overview of what Scheme is about. But first we have to learn what the terms mean!

Scheme

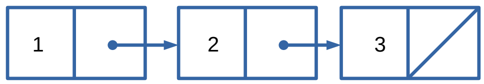
- Everything, and I mean everything, in Scheme looks like: `(function-name arg1 arg2 ...)`
- Lists are the fundamental data structure in Scheme
- No types
- Call by value
- Infix notation:
  `(operation a b)` or `(+ 3 4)`
  Be careful: what does this do: `((+ 3 4))`?
- No loops! Recursion is your frenemy

Scheme building blocks

- Syntax: (*procedure arg1 arg2* ...)
- Arithmetic: +, *, `remainder`, etc.
- Logic: `and`, `or`, `not`, <, etc.
- `define`: Create constants and functions
- `if` and `cond`
- `cons`, `car`, `cdr`

Lists

- Recall singly linked lists
- Scheme lists recursively defined: either null, or a pair whose second value is a list
- Use '() for the empty list
- How to write this list:



Lists

- (list a b c ...)
- For item a and list L, (cons a L) produces a new list starting with a followed by the elements of L
- (car L) first non-empty element of L
- (cdr L) a list with the first item of L removed
- (cXXXr L) shortcut for (cdr (car (car (cdr L)))) → (cdaadr L)
- (append L1 L2) returns list with elements of L1 followed by elements of L2

Dr. Racket

- In your VM, go to the DOWNLOAD section of https://racket-lang.org/ and download the latest installer.
- Run the installer:

```
chmod +x racket-XXXXX
sudo ./racket-XXXXX
```

- Set Dr. Racket preferences. From your home directory:

```
curl faculty.cs.usna.edu/~ksulliv/racket.sh | bash
```

Exercises

- Using only cons, car, cdr, and '(), write an expression to produce the nested list '(3 (4 5) 6)
- Using only cons, car, cdr, and '(), write a function (get2nd L) which takes a list L and returns the 2nd item in the list.
- What's wrong with this list?

```
(cons (cons 8 (cons 6 (cons 7 '())))
      (cons 5 (cons 3 (cons 0 9)))
)
```

  How does the interpreter print it and why?

Recursion on Lists

```
(define (my-func L)
        (if (null? L)
                ; Base case for empty list
                0
                ; Recursive case
                (+ 1 (my-func (cdr L)))))
```

Quoting

- Single quote is shorthand for quote function (quote something) == '(something)
- Quoting in Scheme means "don't evaluate this"
- What does (quote (1 2 3)) do?

Quoting

- Quote is why '() means an empty list
- Also works for non-empty lists: '(a b c)
- Its recursive! '(1 '(2 3) 4) == (list 1 (list 2 3) 4)
- What does this code do?

```
(define x 3)
'(1 2 x)
(list 1 2 x)
```

Symbols

- A new data type: symbols
- Comparable to an identifier in other languages
- Immutable
- symbol? is useful
- Use eqv? for comparisons

To make a symbol use a single quote: 'these 'are 'all 'symbols
Typical uses:

- Names from a short list
- Tag data (cons 10.5 'feet)

Let

A way to get local variables

```
(let ((a 3)
      (b 7))
   (* a b))
```

```
(let ((x (* 3 4)))
   (+ x 8))
```

C

```
int a = 4;
int b = 12
a = b - a;
return a*2;
```

Scheme

```
(let ((a 4))
  (let ((b 12))
    (let ((a (- b a)))
       (* a 2))))
```

Let

```scheme
(define (lmax L)
  (cond [(null? (cdr L))
         ; if list has one element, return it
          (car L)]
          ; if first element is largest, return it
         [(>= (car L) (lmax (cdr L)))
          (car L)]
          ; else return recursive call
         [else (lmax (cdr L))]))
```

What's the run time in the worst case?

Let

```scheme
(define (lmax L)
  (if (null? (cdr L))
      (car L)
      (let ((rest-max (lmax (cdr L))))
        (if (>= (car L) rest-max)
            (car L)
            rest-max))))
```

Ah, `let` gets this back to $O(n)$ as we would like.

Syntactic Building Blocks

- Atoms: code fragments that cannot be split
  Examples: characters, integers
- Values: code fragments that cannot be evaluated any further
  Examples: atoms, lists, arrays
- Expressions: code fragments that can be evaluated to produce a value
  Examples: arithmetic, function calls
- Statements: a complete standalone instruction or command
  - In Scheme, every expression is also a statement.
  - In C++, most statements end in a semicolon.

A Scheme program is just a series of definitions and expressions (statements)

Scheme Grammar

```
expressionsequence : expression |
         expressionsequence expression
expression : atom | ( expressionsequence )
atom : identifier | number | boolean
```

Read-Eval-Print-Loop

1. Read an expression from the user or a program
2. Evaluate the expression
3. Print the resulting value
4. Go to step 1

There is a Scheme "print" function. But I'm not going to tell you about it yet.

Scheme is lists!

Everything in Scheme that looks like a list... *is a list*!
Scheme evaluates a list by using a general rule:

- First, turn a list of expressions (e1 e2 e3 ...) into a list of values (v1 v2 v3 ...) by recursively evaluating each e1, e2, etc.
- Then, apply the procedure v1 to the arguments v2, v3, ...

Can you think of any exceptions to this rule?
What if v1 is not a procedure?

Special Forms

The only exceptions to the evaluation rule are the special forms.

Special forms we have seen: `define`, `if`, `cond`, `and`, `or`.

What makes these "special" is that they
do not (always) evaluate (all) their arguments.

Example: evaluating (5) gives an error, but

```
(if #f
    (5)
    6)
```

just returns 6 — it never evaluates the "(5)" part.

Scheme evaluation and unevaluation

We can use the built-in function `eval` to evaluate a Scheme expression within Scheme!

- Try (`eval` (`list` + 1 2))
- Even crazier: (`eval` (`list` '`define` 'y 100))

What is the opposite (more properly, the *inverse*) of `eval`?

This makes Scheme *homoiconic* and *self-extensible*

Exercises

- Write your own version of the built-in `append` function, at least for the case when there are exactly two lists as arguments.
- Write a function (has—digit? n d) which takes a number and a digit and returns true or false depending on whether n has the digit d in its base-10 representation. You can assume that d is between 1 and 9.
- Write your own version of the built-in list? function, which takes *anything* as its argument and returns true or false depending on whether that thing is actually a list.

Exercises

- Write a function (another x) that takes *anything* as input and returns something else of that same type (an example). The only rule is that it can't return the same thing as the input. At the very least, your function should work for symbols and numbers.
- Write a function (negate expr) that takes a single argument expr and does one of two things. If expr is a list that looks like (`not` X), then it returns X (where X can be absolutely anything). Otherwise, the function returns a list (`not` expr).

More exercises

1. Write a Scheme function (f x y) that computes the formula $5x^2y + \sin(x^2y + 1) + \cos(x^2y + 2)$ at any given point.
2. Simulate the following Java code as a series of nested lets:

```
int x = 1;
x += 3;
x *= 12;
return x;
```

Exercises

- Write a function (my—or a b) that works similar to the built-in `or` boolean function, but returns a symbol 'true or 'false as appropriate.
- Write a function that takes a list of numbers and adds them up using the + function. (Hint: first build this expression using `cons`, then evaluate it using `eval`.)
- Repeat #2 using the built-in `apply` function.