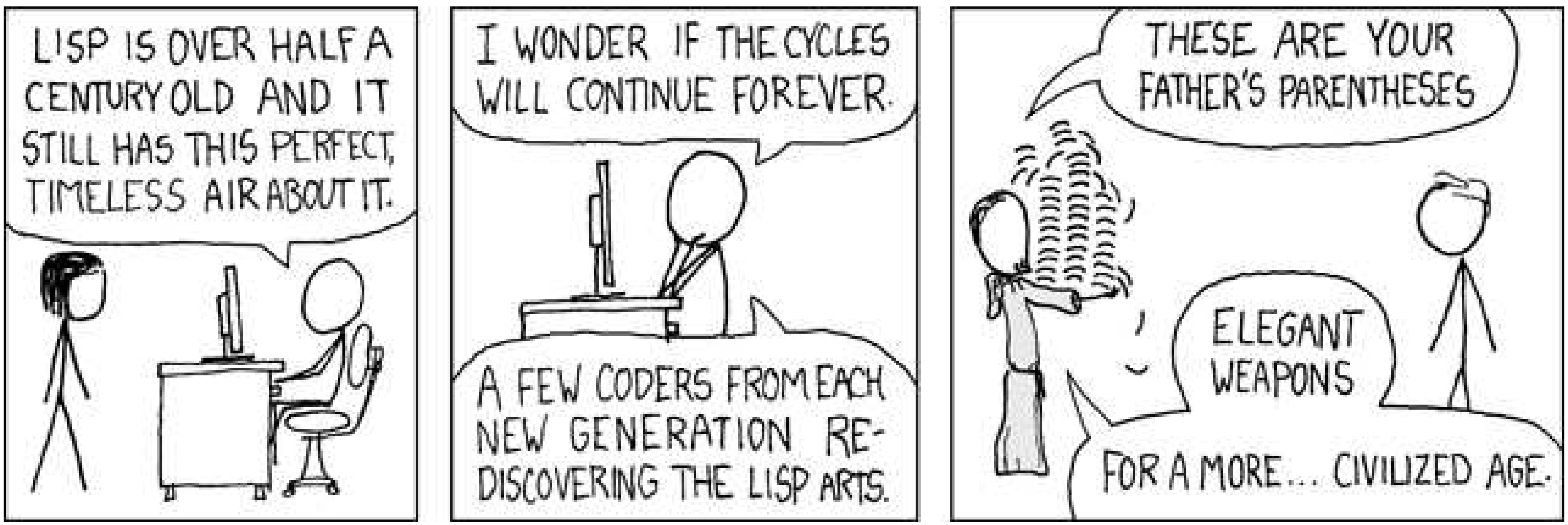
Lecture #26: The Scheme Language

Scheme is a dialect of Lisp:

- "The only programming language that is beautiful."
  —Neal Stephenson
- "The greatest single programming language ever designed"
  —Alan Kay



Scheme Background

- Invented in the 1970s by Guy Steele ("The Great Quux"), whose has also participated in the development of Emacs, Java, and Common Lisp.
- Designed to simplify and clean up certain irregularities in Lisp dialects at the time.
- Used in a fast Lisp compiler (Rabbit).
- Still maintained by a standards committee (although both Brian Harvey and I agree that recent versions have accumulated an unfortunate layer of cruft).

Data Types

- We divide Scheme data into *atoms* and *pairs*.
- The classical atoms:
  - Numbers: integer, floating-point, complex, rational.
  - Symbols.
  - Booleans: #t, #f.
  - The empty list: ().
  - Procedures (functions).
- Some newer-fangled, mutable atoms:
  - Vectors: Python lists.
  - Strings.
  - Characters: Like Python 1-element strings.
- Pairs are two-element tuples, where the elements are (recursively) Scheme values.

Symbols

- Lisp was originally designed to manipulate *symbolic data:* e.g., formulae as opposed merely to numbers.
- Such data is typically recursively defined (e.g., "an expression consists of an operator and subexpressions").
- The "base cases" had to include numbers, but also variables or words.
- For this purpose, Lisp introduced the notion of a *symbol:*
  - Essentially a constant string.
  - Two symbols with the same "spelling" (string) are always the same object.
  - Confusingly, the reader (the program that reads in Scheme programs and data) converts symbols it reads into lower-case first.
- The main operation on symbols, therefore, is *equality.*

Pairs and Lists

- As we've seen, one can build practically any data structure out of pairs.
- The Scheme notation for the pair of values V_1 and V_2 is

  $$(V_1 \;\; . \;\; V_2)$$

- In Scheme, the main one is the *list*, defined recursively like an rlist:
  - The empty list, written "()", is a list.
  - The pair consisting of a value V and a list L is a list that starts with V, and whose tail is L.
- Lists are so prevalent that there is a standard abbreviation: You can write (V . ()) as (V), and (V_1 . (V_2 . (V_3 ...))) as (V_1 V_2 $V_3 \ldots$).

Programs

- Scheme expressions programs are instances of Lisp data structures ("Scheme is written in Scheme").
- At the bottom, numerals, booleans, characters, and strings are expressions that stand for themselves.
- Most lists stand for function calls:

  $(OP\ E_1\ \cdots\ E_n)$

  as a Scheme expression means "evaluate *OP* and the E_1 (recursively), and then apply the value of *OP*, which must be a function, to the values of the arguments E_i."
- A few lists, identified by their *OP*, are *special forms*, which each have different meanings.

Quotation

- Since programs are data, we have a problem: suppose you want your program to create a piece of data that happens to look like a program?
- How do we say, for example, "Set the variable x to the three-element list (+ 1 2)" without it meaning "Set the variable x to the value 3?"
- The "quote" special form does this: evaluating (quote E) yields E itself as the value, without treating it like a Scheme expression to be evaluated.

```
>>> (+ 1 2)
3
>>> (quote (+ 1 2))
(+ 1 2)
>>> '(+ 1 2)      ; Shorthand. Converted to (quote (+ 1 2))
(+ 1 2)
```

- How about

```
>>> (quote (1 2 '(3 4)))      ;?
```

Symbols

- When evaluated as a program, a symbol acts like a variable name.
- Variables are bound in environments, just as in Python, although the syntax differs.
- To define a new symbol, either use it as a parameter name (later), or use the "define" special form:

```
(define pi 3.1415926)
(define pi**2 (* pi pi))
```

- This (re)defines the symbols in the current environment. The second expression is evaluated first.
- To assign a new value to an existing binding, use the `set!` special form:

```
(set! pi 3)
```

- Here, pi must be defined, and it is that definition that is changed (not like Python).

Function Evaluation

- Function evaluation is just like Python: same environment frames, same rules for what it means to call a user-defined function.
- To create a new function, we use the `lambda` special form:

```
>>> ( (lambda (x y) (+ (* x x) (* y y))) 3 4)
25
>>> (define fib
          (lambda (n) (if (< n 2) n (+ (fib (- n 2) (- n 1))))))
>>> (fib 5)
5
```

- The last is so common, there's an abbreviation:

```
>>> (define (fib n)
          (if (< n 2) n (+ (fib (- n 2) (- n 1)))))
```

Numbers

- All the usual numeric operations and comparisons:

```
>>> (- (quotient (* (+ 3 7 10) (- 1000 8)) 992) 17)
3
>>> (> 7 2)
#t
>>> (< 2 4 8)
#t
>>> (= 3 (+ 1 2) (- 4 1))
#t
>>> (integer? 5)
#t
>>> (integer? 'a)
#f
```

Lists and Pairs

- Pairs (and therefore lists) have a basic constructor and accessors:

```
>>> (cons 1 2)
(1 . 2)
>>> (cons 'a (cons 'b '()))
(1 2)
>>> (define L (a b c))
>>> (car L)
a
>>> (cdr L)
(b c)
>>> (cadr L)    ; (car (cdr L))
b
>>> (cdddr L)   ; (cdr (cdr (cdr L)))
()
```

- And one that is especially for lists:

```
>>> (list (+ 1 2) 'a 4)
(3 a 4)
>>> ; Why not just write ((+ 1 2) a 4)?
```

Conditionals

- The basic control structures are the conditional, which are special forms:

```
>>> (define x 14)
>>> (define n 2)
>>> (if (not (zero? n))      ; Condition
...     (quotient x n)       ; If condition is not #f
...     x)                   ; If condition is #f
7
>>> (and (< 2 3) (> 3 4))
#f
>>> (and (< 2 3) '())
()
>>> (or (< 2 3) (> 3 4))
#t
>>> (or (< 3 2) '())
()
```

Traditional Conditionals

Traditional Lisp had a more elaborate special form:

```
>>> (define x 5)
>>> (cond ((< x 1) 'small)
...       ((< x 3) 'medium)
...       ((< x 5) 'large)
...       (else    'big))
big
```

Binding Constructs: Let

- Sometimes, you'd like to introduce local variables or named constants.
- The `let` special form does this:

```
>>> (define x 17)
>>> (let ((x 5)
...       (y (+ x 2)))
...      (+ x y))
24
```

- This is a *derived form,* equivalent to:

```
>>> ((lambda (x y) (+ x y)) x (+ x 2)))
```

Tail recursion

- With just the functions and special forms so far, can write anything.
- But there is one problem: how to get an arbitrary iteration that doesn't overflow the execution stack because recursion gets too deep?
- Scheme mandates that *tail-recursive functions must work like iterations.*
- This means that in this program:

```
(define (fib n)
     (define (fib1 n1 n2 n)
             (if (< n 2)
                 n2
                 (fib1 n2 (+ n1 n2) (- n 1))))
     (if (= n 0) 0
         (fib1 0 1 n)))
```

- Instead of calling `fib1` recursively, we *replace* the call on `fib1` with the recursive call.