Lecture #23: 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

- The programming language Lisp is the second-oldest programming language still in use (introduced in 1958).

- Scheme is a Lisp dialect Invented in the 1970s by Guy Steele ("The Great Quux"), who 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.
- Typically, such data is 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 by default the same object (but usually, case is ignored).
- The main operation on symbols is *equality*.
- Examples:
  
  a bumblebee numb3rs * + / wide-ranging !?@*!!

(As you can see, symbols can include non-alphanumeric characters.)
Pairs and Lists

• The Scheme notation for the pair of values $V_1$ and $V_2$ is
  $$(V_1 . V_2)$$

• As we’ve seen, one can build practically any data structure out of pairs.

• 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:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(V)$</td>
<td>$(V . ())$</td>
</tr>
<tr>
<td>$(V_1 V_2 \ldots V_n)$</td>
<td>$(V_1 . (V_2 . (\ldots (V_n . ()))))$</td>
</tr>
<tr>
<td>$(V_1 V_2 \ldots V_{n-1} . V_n)$</td>
<td>$(V_1 . (V_2 . (\ldots (V_{n-1} . V_n))))$</td>
</tr>
</tbody>
</table>
Examples of Pairs and Lists

(3 . 2)

(x = 3)

(+ (* 3 7) (- x))

((a+ . 289) (a . 269) (a- . 255))
Programs

- Scheme expressions and programs are *instances of Lisp data structures* ("Scheme programs are Scheme data").
- At the bottom, numerals, booleans, characters, and strings are expressions that stand for themselves.
- Most lists (aka *forms*) stand for function calls:
  
  \[(OP \, E_1 \, \cdots \, E_n)\]

  as a Scheme expression means “evaluate \(OP\) and the \(E_i\) (recursively), and then apply the value of \(OP\), which must be a function, to the values of the arguments \(E_i\).”

- Examples:
  
  \[
  (> \, 3 \, 2) \quad ; \quad 3 > 2 \quad \Rightarrow \quad \text{#t}
  
  (- \, (/ \, (* \, (+ \, 3 \, 7 \, 10) \, (- \, 1000 \, 8)\)) \, 992) \, 17) \\
  \quad ; \quad ((3 + 7 + 10) \cdot (1000 - 8))/992 - 17
  
  (pair? \, (list \, 1 \, 2)) \quad ; \quad \Rightarrow \quad \text{#t}
Quotation

• Since programs are data, we have a problem: How do we say, eg., “Set the variable \( x \) to the three-element list \((+ 1 2)\)” without it meaning “Set the variable \( x \) to the value 3?”

• In English, we call this a use vs. mention distinction.

• For this, we need a special form—a construct that does not simply evaluate its operands.

• \( (\text{quote } E) \) yields \( E \) itself as the value, without evaluating it as a Scheme expression:

\[
\begin{align*}
\text{scm}> ( + 1 2 ) \\
3 \\
\text{scm}> (\text{quote } (+ 1 2)) \\
(+ 1 2) \\
\text{scm}> '(+ 1 2) \quad ; \text{Shorthand. Converted to (quote (+ 1 2))} \\
(+ 1 2)
\end{align*}
\]

• How about

\[
\begin{align*}
\text{scm}> (\text{quote } (1 2 ' (3 4))) \quad ; ?
\end{align*}
\]
Special Forms

- (quote E) is a **special form**: an exception to the general rule for evaluating functional forms.

- A few other special forms—lists identified by their OP—also have meanings that generally do not involve simply evaluating their operands:

  (if (> x y) x y) ; Like Python ... if ... else ... 

  (and (integer?) (> x y) (< x z)) ; Like Python 'and'

  (or (not (integer? x)) (< x L) (> x U)) ; Like Python 'or'

  (lambda (x y) (/ (* x x) y)) ; Like Python lambda
  ; yields function

  (define pi 3.14159265359) ; Definition
  (define (f x) (* x x)) ; Function Definition
  (set! x 3) ; Assignment ("set bang")
Traditional Conditionals

Also, the fancy traditional Lisp conditional form:

```
scm> (define x 5)
scm> (cond ((< x 1) 'small)
            ((< x 3) 'medium)
            ((< x 5) 'large)
            (#t 'big))
big
```
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:
  
  ```lisp
  (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:
  
  ```lisp
  (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:

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

• The last is so common, there’s an abbreviation:

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

- All the usual numeric operations and comparisons:

  scm> (- (quotient (* (+ 3 7 10) (- 1000 8)) 992) 17)
      3
  scm> (/ 3 2)
      1.5
  scm> (quotient 3 2)
      1
  scm> (> 7 2)
      #t
  scm> (< 2 4 8)
      #t
  scm> (= 3 (+ 1 2) (- 4 1))
      #t
  scm> (integer? 5)
      #t
  scm> (integer? 'a)
      #f
Lists and Pairs

• Pairs (and therefore lists) have a basic constructor and accessors:
  
  scm> (cons 1 2)  
  (1 . 2)  
  scm> (cons 'a (cons 'b '()))  
  (a b)  
  scm> (define L (a b c))  
  scm> (car L)  
  a  
  scm> (cdr L)  
  (b c)  
  scm> (cadr L) ; (car (cdr L))  
  b  
  scm> (cdddr L) ; (cdr (cdr (cdr L)))  
  ()  

• And one that is especially for lists:
  
  scm> (list (+ 1 2) 'a 4)  
  (3 a 4)  
  scm> ; Why not just write ((+ 1 2) a 4)?
Binding Constructs: Let

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

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

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

```scheme
scm> ((lambda (x y) (+ x y)) 5 (+ x 2))
```
Loops and 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?
• In Scheme, tail-recursive functions must work like iterations.
Loops and Tail Recursion (II)

This means that in this program:

### Scheme

(define (fib n)
    (define (fib1 n1 n2 k)
        (if (= k n) n2
            (fib1 n2 (+ n1 n2) (+ k 1)))
    (if (= n 0) 0 (fib1 0 1 1)))

### Python

```python
def fib(n):
    def fib1(n1, n2, k):
        return n2 if k == n else fib1(n2, n1+n2, k+1)
    return 0 if n == 0 else fib1(0, 1, 1)
```

To call fib1 recursively, we **replace** the call on fib1 with the recursive call.
A Simple Example

- Consider

```scheme
(define (sum init L)
  (if (null? L) init
   (sum (+ init (car L)) (cdr L))))
```

- Here, can evaluate a call by substitution, and then keep replacing subexpressions by their values or by simpler expressions:

```scheme
(sum 0 '(1 2 3))
(if (null? '(1 2 3)) 0 (sum ...))
(if #f 0 (sum (+ 0 (car '(1 2 3))) (cdr '(1 2 3))))
(sum (+ 0 (car '(1 2 3))) (cdr '(1 2 3)))
(sum (+ 0 1) '(2 3))
(sum 1 '(2 3))
(if (null? '(2 3)) 1 (sum ...))
(if #f 1 (sum (+ 1 (car '(2 3))) (cdr '(2 3))))
(sum (+ 1 (car '(2 3))) (cdr '(2 3)))
etc.
```