Lecture #24: 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

URL: [https://xkcd.com/297/](https://xkcd.com/297/)
Author: Randall Munrow, licensed under a Creative Commons Attribution-NonCommercial 2.5 License.
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).
Our Subset

• In part, we’ll use Scheme to illustrate the *applicative programming* paradigm—computing with no side-effects (save output of results), no assignments, and only non-destructive operations.

• Therefore, we’ll leave out Scheme features such as assignment, as well as mutable data structures.

• What’s so great about applicative programming?
  - Reasoning about programs can be easier without side-effects.
  - Side-effects and mutations make correct parallel programming more difficult.
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).

• Pairs are like two-element Python 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 
  $(V_1 \cdot V_2)$

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

- In Scheme, the main one is the (linked) list, defined recursively:
  - 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>$(V \cdot ())$</td>
</tr>
<tr>
<td>$(V_1 \ V_2 \ \cdots \ V_n)$</td>
<td>$(V_1 \cdot (V_2 \cdot (\cdots (V_n \cdot ()))))$</td>
</tr>
<tr>
<td>$(V_1 \ V_2 \ \cdots \ V_{n-1} \ . \ V_n)$</td>
<td>$(V_1 \cdot (V_2 \cdot (\cdots (V_{n-1} \cdot V_n))))$</td>
</tr>
</tbody>
</table>

- For our purposes this semester, we'll use the abbreviation exclusively, and won't use structures that require dots.
Examples of Lists

\[(x = 3)\]
\[(+ (* 3 7) (- x))\]

\[(\text{(a+ 28)} \text{(a 26)} \text{(a- 25)})\]
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 in a Scheme program: we say they are self-evaluating.

- Most lists (aka forms) stand for function calls in a Scheme program:
  \[(OP \ E_1 \ \cdots \ \ E_n)\]
  
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 \(E_i\)." (Sound familiar? It's the same as in Python.)

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

Last modified: Tue Mar 30 12:25:07 2021
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, and use quotation marks to distinguish mentions (“Copper” is a six-letter word, not a metal.) from uses (Copper is a metal, not a word.)

• In Scheme, we use 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:

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

• How about

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

• \( (\text{quote } E) \) is a \textit{special form}: an exception to the general rule for evaluating functional forms.

• A few other special forms—lists identified by their \textit{OP}—also have meanings that generally do not involve simply evaluating their operands:

\[
\begin{align*}
(\text{if} \; (> \; x \; y) \; x \; y) & \quad ; \text{Like Python } \ldots \; \text{if } \ldots \; \text{else } \ldots \\
(\text{and} \; (\text{integer?} \; x) \; (> \; x \; y) \; (< \; x \; z)) & \quad ; \text{Like Python } \text{'}\text{and}' \\
(\text{or} \; (\text{not} \; (\text{integer?} \; x)) \; (< \; x \; L) \; (> \; x \; U)) & \quad ; \text{Like Python } \text{'}\text{or}' \\
(\text{lambda} \; (x \; y) \; (/ \; (* \; x \; x) \; y)) & \quad ; \text{Like Python lambda} \\
& \quad ; \text{yields function} \\
(\text{define} \; \text{pi} \; 3.14159265359) & \quad ; \text{Definition, like Python first assignment} \\
(\text{define} \; (f \; x) \; (* \; x \; x)) & \quad ; \text{Function Definition, like Python def}
\end{align*}
\]
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
```

which is the Lisp version of Python’s

"small" if x < 1 else "medium" if x < 3 else "large" if x < 5 else "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:

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

• This defines the symbols in the current environment. The last expression in each definition is evaluated first and then bound to the symbol.
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 \texttt{lambda} special form:

  \[\texttt{scm> \((\lambda(x\ y)(+(*\ x\ x)(*\ y\ y)))\ 3\ 4)\} \]
  \[25\]
  \[\texttt{scm> (define fib}\]
  \[\quad(\lambda(n)(\textbf{if} (<n2)n(+\textbf{(fib\ (-n2))\ (fib\ (-n1))})))\]
  \[\texttt{scm> (fib 5)}\]
  \[5\]

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

  \[\texttt{scm> (define (fib n)}\]
  \[\quad(\textbf{if} (<n2)n(+\textbf{(fib\ (-n2))\ (fib\ (-n1))})\)

Last modified: Tue Mar 30 12:25:07 2021
Numbers

- All the usual numeric operations and comparisons:

  ```sml
  scm> (- (quotient (* (+ 3 7 10) (- 1000 8)) 992) 17)
  3
  scm> (/ 3 2)
  1.5
  scm> (quotient 3 2)
  1
  scm> (quotient -3 2) ; quotient rounds towards 0 (not like Python)
  -1
  scm> (> 7 2)
  #t
  scm> (= 3 (+ 1 2))
  #t
  scm> (integer? 5)
  #t
  scm> (integer? 'a)
  #f
  ```
Lists and Pairs

- Pairs (and therefore lists) have a basic constructor (`cons`) and accessors (`car` and `cdr`):
  
  ```scm
  scm> (cons 1 2)
  (1 . 2)
  scm> (cons 'a (cons 'b '())) ; Like Link("a", Link("b", Link.empty))
  (a b)
  scm> (define L '(a b c))
  scm> (car L) ; Like L.first
  a
  scm> (cdr L) ; Like L.rest (Pamela suggests cdr = see da rest)
  (b c)
  scm> (car (cdr L))
  b
  scm> (cdr (cdr (cdr L)))
  ()
  ```

- And one that is especially for lists:
  
  ```scm
  scm> (list (+ 1 2) 'a 4)
  (3 a 4)
  scm> ; Why not just write ((+ 1 2) a 4)?
  scm> ; Or '((+ 1 2) a 4)?
  ```
Equivalence Operations

```scheme
(scm> (= 1 (- 2 1)) ; Works for numbers only
  #t
(scm> (eqv? 1 2) ; Works for numbers, empty list, booleans, symbols
  #f
(scm> (eqv? 1 (- 2 1))
  #t
(scm> (define L '([1 2 3]))
(scm> (eqv? L L) ; Like Python's "is" elsewhere
  #t
(scm> (eqv? L '(1 2 3))
  #f
(scm> (eq? L '(1 2 3)) ; eq? is Python's "is" (might not work for numbers)
  #f
(scm> ; equal? is like eqv?, but also does deep equality for pairs (and
(scm> ; therefore also for lists.
(scm> (equal? '(((1 2) 3 (4)) (list (list 1 2) 3 (list 4))))
  #t
(scm> (eqv? '(((1 2) 3 (4)) (list (list 1 2) 3 (list 4))))
  #f
```
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 a correct Scheme implementation, *tail-recursive functions work like iterations.*
Loops and Tail Recursion (II)

- So for this program:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (sumsq n)</td>
<td>def sumsq(n):</td>
</tr>
<tr>
<td>(define (sumsq1 s n)</td>
<td>def sumsq1(s, n):</td>
</tr>
<tr>
<td>(if (&lt;= n 0) s</td>
<td>if n &lt;= 0:</td>
</tr>
<tr>
<td>(sumsq1 (+ s (* n n))</td>
<td>return s</td>
</tr>
<tr>
<td>(- n 1)))</td>
<td>return sumsq1(s + n * n,</td>
</tr>
<tr>
<td>(sumsq1 0 n))</td>
<td>n - 1)</td>
</tr>
<tr>
<td>(sumsq 1000)</td>
<td>return sumsq1(0, n)</td>
</tr>
<tr>
<td></td>
<td>sumsq(1000)</td>
</tr>
</tbody>
</table>

  The typical Python implementation of \texttt{sumsq1} will execute \texttt{return s} at the time when there are 1000 other calls on \texttt{sumsq1} that have not yet returned. This often results in an exception.

- But in a correct Scheme implementation, each recursive call of \texttt{sumsq1} replaces the call from which it occurs.

- At each inner tail call, in other words, we forget the sequence of calls that got us there, so the system need not use more memory to go deeper.
Tail Recursion: A Simple Example

- We can think of the execution of \((\text{sumsq1} \ 1000)\) as a sequence of steps in which one call is replaced by another:

\[
\begin{align*}
(\text{sumsq} & \ 1000) \\
\Rightarrow & \ (\text{sumsq1} \ 0 \ 1000) \\
\Rightarrow & \ (\text{if} \ (\leq \ 1000 \ 0) \ 0 \ (\text{sumsq1} \ (+ \ 0 \ (* \ 1000 \ 1000)) \ (- \ 1000 \ 1))) \\
\Rightarrow & \ (\text{sumsq1} \ (+ \ 0 \ (* \ 1000 \ 1000)) \ (- \ 1000 \ 1)) \\
\Rightarrow & \ (\text{sumsq1} \ 1000000 \ 999) \\
\Rightarrow & \ (\text{if} \ (\leq \ 999 \ 0) \ 1000000 \ (\text{sumsq1} \ (+ \ 1000000 \ (* \ 999 \ 999)) \ (- \ 999 \ 1))) \\
\Rightarrow & \ (\text{sumsq1} \ (+ \ 1000000 \ (* \ 999 \ 999)) \ (- \ 999 \ 1)) \\
\Rightarrow & \ (\text{sumsq1} \ 1998001 \ 998) \\
\Rightarrow & \ ... \\
\Rightarrow & \ (\text{sumsq1} \ 333833500 \ 0) \\
\Rightarrow & \ (\text{if} \ (\leq \ 0 \ 0) \ 333833500 \ (\text{sumsq1} \ (+ \ 333833500 \ (* \ 0 \ 0)) \ (- \ 0 \ 1))) \\
\Rightarrow & \ 333833500
\end{align*}
\]