### Lecture #22: The Scheme Language

**Scheme Background**

- The programming language Lisp is the second-oldest language still in use (introduced in 1958).
- Scheme is a Lisp dialect invented in the 1970s by Guy Steele, who has also participated in the development of Java, and Common Lisp.
- Designed to simplify and clean up certain irregularities in Lisp at the time.
- Used in a fast Lisp compiler (Rabbit).
- Still maintained by a standards committee (although both Brey 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 like two-element Python lists, where the elements are recursively Scheme values.

---

**Symbols**

- Lisp was originally designed to manipulate *symbolic data*: expressions 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 other symbols.
- 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 numbers * + / wide-ranging ?@*!!
  - (a + 289) (a . 269) (a - 255)

---

**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 from pairs.
- In Scheme, the main one is the (linked) *list*, defined recursively on list:
  - The empty list, written "()", is a list.
  - The pair consisting of a value $V$ and a list $L$ is a list that has $V$ as its head 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>$(V . O)$</td>
</tr>
<tr>
<td>$(V_1 V_2 \ldots V_n)$</td>
<td>$(V_1 . (V_2 . \ldots (V_n . O)))$</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**

- $(3 . 2)$
- $(x = 3)$
- $(+ (\ast 3 \ 7) (- x))$
- $((a+ . 289) (a . 269) (a - . 255))$
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: \((OP E_1 \ldots E_n)\). As a Scheme expression means "evaluate \(OP\) and the \(E_i\) (recursively), and then apply the value of \(OP\), which must be a function, values of the arguments \(E_i\)."

Examples:

- \((> 3 2)\); \(3 > 2\) \(\Rightarrow\) #t
- \((- (/ (* (+ 3 7 10) (- 1000 8)) 992))\)
- \(17\)
- \((pair? (list 1 2))\); \(\Rightarrow\) #t

Quotation

Since programs are data, we have a problem: How do we say, "Set the variable \(x\) to the three-element list \((+ 1 2)\)" without 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 evaluate its operands.

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

- \((\text{quote } (+ 2 3))\)
- \(\Rightarrow\) 5

How about:\n
\((\text{quote } (quote (1 2 (3 4))))\) ;?
### Loops and Tail Recursion (II)

- This means that in this program:

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Python</th>
</tr>
</thead>
</table>
| (define (fib n)) | def fib(n):
| (define (fib1 n1 n2 k)) | def fib1(n1, n2, k):
| (if (= k n) n2) | return \
| (fib1 n2) | n2 if k == n \
| (+ n1 n2) | else fib1(n2, n1+\n| (+ k 1)))) | return 0 if n == 0 \
| (if (= n 0) 0 (fib1 0 1 1)) | else fib1(0, 1) |

Rather than having one call of `fib1` recursively call itself, we forget the outer call on `fib1((fib1 0 1 1))` with the recursive call `((fib1 1 2 3))`, and then replace that with `(fib1 1 2 3)`, then `(fib1 2 3 4)`, etc.

- At each inner tail-recursive call, in other words, we forget the sequence of calls that got us there, so the system need not use memory to go deeper.

### 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:

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

### Binding Constructs: Let

- Sometimes, you’d like to introduce local variables or constants.

- The `let` special form does this:

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

- This is a derived form, equivalent to:

```scheme
(let ((x 5) (y (+ x 2)))
  (+ x y))
```

### Lists and Pairs

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

```scheme
(define (cons a b)
  (list a b))
```

- And one that is especially for lists:

```scheme
(list (+ 1 2) 'a 4)
```

### Numbers

- All the usual numeric operations and comparisons:

```
(- (quotient (* (+ 3 7 10) (- 1000 8)) 992) 17)
(quotient 3 2)
(> 7 2)
(< 2 4 8)
(integer? 5)
(integer? 'a)
```

### Loops and Tail Recursion

- With just the functions and special forms so far, can write a loop.

- But there is one problem: how to get an arbitrary iteration doesn’t overflow the execution stack because recursion goes deep?

- In Scheme, tail-recursive functions must work like iterations.

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

  (if n 0
      (fib1 0 1 1)))
```

- Rather than having one call of `fib1` recursively call itself, we forget the outer call on `fib1((fib1 0 1 1))` with the recursive call `((fib1 1 2 3))`, and then replace that with `(fib1 1 2 3)`, then `(fib1 2 3 4)`, etc.

- At each inner tail-recursive call, in other words, we forget the sequence of calls that got us there, so the system need not use memory to go deeper.

### 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:

```
(sum 0 '(1 2 3))
```

### Binding Constructs: Let

- Sometimes, you’d like to introduce local variables or constants.

- The `let` special form does this:

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

- This is a derived form, equivalent to:

```scheme
(let ((x 5) (y (+ x 2)))
  (+ x y))
```

### Lists and Pairs

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

```scheme
(define (cons a b)
  (list a b))
```

- And one that is especially for lists:

```scheme
(list (+ 1 2) 'a 4)
```

### Numbers

- All the usual numeric operations and comparisons:

```
(- (quotient (* (+ 3 7 10) (- 1000 8)) 992) 17)
(quotient 3 2)
(> 7 2)
(< 2 4 8)
(integer? 5)
(integer? 'a)
```

### Loops and Tail Recursion

- With just the functions and special forms so far, can write a loop.

- But there is one problem: how to get an arbitrary iteration doesn’t overflow the execution stack because recursion goes deep?

- In Scheme, tail-recursive functions must work like iterations.

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

  (if n 0
      (fib1 0 1 1)))
```

- Rather than having one call of `fib1` recursively call itself, we forget the outer call on `fib1((fib1 0 1 1))` with the recursive call `((fib1 1 2 3))`, and then replace that with `(fib1 1 2 3)`, then `(fib1 2 3 4)`, etc.

- At each inner tail-recursive call, in other words, we forget the sequence of calls that got us there, so the system need not use memory to go deeper.