Scheme
Announcements
Scheme is a Dialect of Lisp

What is Lisp?

• LISt Processor language – all code consists of either primitives or lists (combinations)
• One of the oldest high-level programming languages
• Dependent on Linked List-like structures
• Functional, not object-oriented programming

Why are we learning Scheme?

• To learn an extremely elegant language
• To see how to learn new programming languages now that you’ve already learned Python
• To lay the foundation for learning about how to write interpreters
What’s important in a programming language?

Computation

Variables

Variable assignments

Creating functions

Calling functions

Conditional flow
### Scheme Expressions

Scheme programs consist of expressions, which can be:

- **Primitive expressions:** 2 3.3 true + quotient
- **Combinations:** (quotient 10 2) (not true)
  - Combinations look like lists! Lists are created with parentheses and delimited with spaces.

Numbers are self-evaluating; symbols are bound to values.

**Call expressions** include an operator and 0 or more operands in parentheses.

```scheme
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
   (+ (* 2 4)
     (+ 3 5)))
   (+ (- 10 7)
     6))
```

"quotient" names Scheme’s built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn’t matter)

(Demo)
Special Forms
A combination that is not a call expression is a special form:

- **if** expression:  
  \[(\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\]

- **and** and **or**:  
  \[(\text{and } \langle\text{e1}\rangle \ldots \langle\text{en}\rangle), \ (\text{or } \langle\text{e1}\rangle \ldots \langle\text{en}\rangle)\]

- Binding symbols: 
  \[(\text{define } \langle\text{symbol}\rangle \ \langle\text{expression}\rangle)\]

- New procedures: 
  \[(\text{define } (\langle\text{symbol}\rangle \ \langle\text{formal parameters}\rangle) \ \langle\text{body}\rangle)\]

The symbol “pi” is bound to 3.14 in the global frame.

A procedure is created and bound to the symbol “abs.”

The evaluation process:
1. Evaluate the predicate expression.
2. Evaluate either the consequent or the alternative.
Booleans in Scheme

False values:

    #f

True values:

    Everything else:

    1, 2, #t, -1.2, +

    0, nil – are both true values now! This is different from Python!
Defining in Scheme

```python
>>> x = 4

>>> def f(x, y):
    return x + y

>>> f(1, 2)
3

scm> (define x 4)

scm> (define (f x y) (+ x y))

scm> (f 1 2)
3
```
Scheme Interpreters

(Demo)
Lambda Expressions
Lambda expressions evaluate to anonymous procedures

\[(\text{lambda } (<\text{formal-parameters}>))\ (<\text{body})\]\n
Two equivalent expressions:

\[
\text{(define (plus4 x) (+ x 4))}
\]

\[
\text{(define plus4 (lambda (x) (+ x 4)))}
\]

An operator can be a call expression too:

\[
((\text{lambda } (x\ y\ z)\ (+\ x\ y\ (\text{square } z)))\ 1\ 2\ 3)\ \rightarrow\ 12
\]

Evaluates to the \(x+y+z^2\) procedure
Sierpinski's Triangle

(Demo)
More Special Forms
The \texttt{cond} special form that behaves like if–elif–else statements in Python

\begin{verbatim}
if x > 10:
   print('big')
elif x > 5:
   print('medium')
else:
   print('small')
\end{verbatim}

The \texttt{begin} special form combines multiple expressions into one expression

\begin{verbatim}
if x > 10:
   print('big')
   print('guy')
else:
   print('small')
   print('fry')
\end{verbatim}

\begin{verbatim}
(cond ((> x 10) (print 'big))
 (cond ((> x 5) (print 'medium))
 (else (print 'small))))
\end{verbatim}

\begin{verbatim}
if x > 10:
   (begin (print 'big)   (print 'guy))
else:
   (begin (print 'small) (print 'fry))))
\end{verbatim}
Let Expressions

The let special form binds symbols to values temporarily; just for one expression

\[
\begin{align*}
  a &= 3 \\
  b &= 2 + 2 \\
  c &= \text{math.sqrt}(a \ast a + b \ast b) \\
  \text{define } c &= (\text{let } ((a 3) (b (+ 2 2))) (\text{sqrt } (+ (* a a)(* b b)))) \\
\end{align*}
\]

\textit{a and b are still bound down here} \hspace{2cm} \textit{a and b are not bound down here}
Lists
Scheme Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a linked list.
- **car**: Procedure that returns the first element of a list.
- **cdr**: Procedure that returns the rest of a list.
- **nil**: The empty list.

Important! Scheme lists are written in parentheses with elements separated by spaces.

```scheme
> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil)))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

(Demo)
### Scheme Lists vs. Linked Lists

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(cons a b)</code></td>
<td><code>Link(a, b)</code></td>
</tr>
<tr>
<td><code>(car a)</code></td>
<td><code>a.first</code></td>
</tr>
<tr>
<td><code>(cdr a)</code></td>
<td><code>a.rest</code></td>
</tr>
<tr>
<td><code>(list a b c)</code></td>
<td><code>Link(a, Link(b, Link(c)))</code></td>
</tr>
</tbody>
</table>
Symbolic Programming
Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

Quotation can also be applied to combinations to form lists.

> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)