Scheme

Scheme is a Dialect of Lisp

What are people saying about Lisp?

• "If you don’t know Lisp, you don’t know what it means for a programming language to be powerful and elegant."
  – Richard Stallman, created Emacs & the first free variant of UNIX

• "The only computer language that is beautiful."
  – Neal Stephenson, DeMers’s favorite sci-fi author

• "The greatest single programming language ever designed."
  – Alan Kay, co-inventor of Smalltalk and OOP (from the user interface video)

Scheme Expressions

Scheme programs consist of expressions, which can be:

- Primitive expressions: 2, 3.3, true, quotient
- Combinations: (quotient 10 2), (not true)

Numbers are self-evaluating; symbols are bound to values

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

(Demo)

> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3

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

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

Special Forms

A combination that is not a call expression is a special form:

- if expression: (if <predicate> <consequent> <alternative>)
- and or or: (and <e1> ... <en>), (or <e1> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

(Demo)

> (define pi 3.14)
> (* pi 2)
6.28
> (define (abs x)
  (if (< x 0)
      (- x)
      x))
> (abs -3)
3

The symbol "pi" is bound to 3.14 in the global frame

A procedure is created and bound to the symbol "abs"

Evaluation:
(1) Evaluate the predicate expression
(2) Evaluate either the consequent or alternative

Scheme Interpreters

(Demo)
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[ \lambda (\text{formal-parameters}) \text{ body} \]

Two equivalent expressions:

\[
\begin{align*}
\text{(define (plus4 x) (+ x 4))} \\
\text{(define plus4 (lambda (x) (+ x 4)))}
\end{align*}
\]

An operator can be a call expression too:

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

Lists

In the late 1950s, computer scientists used confusing names:

- \text{cons}: Two-argument procedure that creates a linked list
- \text{car}: Procedure that returns the first element of a list
- \text{cdr}: Procedure that returns the rest of a list
- \text{nil}: The empty list

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

\[
\begin{align*}
\text{>(cons 1 (cons 2 nil))} & \rightarrow \text{(1 2)} \\
\text{(define x (cons 1 (cons 2 nil)))} & \rightarrow x \\
\text{>(car x)} & \rightarrow 1 \\
\text{>(cdr x)} & \rightarrow (2) \\
\text{(cons 1 (cons 2 (cons 3 (cons 4 nil)))))} & \rightarrow \text{(1 2 3 4)}
\end{align*}
\]

Symbolic Programming

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

\[
\begin{align*}
\text{>(define a 1)} \\
\text{>(define b 2)} \\
\text{>(list a b)} & \rightarrow \text{(1 2)} \\
\text{>(car (list a b))} & \rightarrow a \\
\text{>(cdr (list a b))} & \rightarrow b
\end{align*}
\]

Quotation is used to refer to symbols directly in Lisp.

\[
\begin{align*}
\text{>(list 'a 'b)} & \rightarrow \text{(a b)} \\
\text{>(car (list 'a 'b))} & \rightarrow a \\
\text{>(cdr (list 'a 'b))} & \rightarrow b
\end{align*}
\]

Quotation can also be applied to combinations to form lists.

\[
\begin{align*}
\text{>(list 'a b c)} & \rightarrow \text{([a b c])} \\
\text{>(list 'a b c)} & \rightarrow \text{([a b c])}
\end{align*}
\]