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, DeNero’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 Fundamentals

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

```
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (quotient (+ 2 4) (+ 3 5))
10 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>)
  > (if (< x 0) (- x) x)
  > (if (< x 0) (- x) x)

- and or: (and <e1> ... <en>) (or <e1> ... <en>)

- Binding symbols: (define <symbol> <expression>)

- New procedures: (define (<symbol> <formal parameters>) <body>)

```
> (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
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

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

An operator can be a call expression too:

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

Evaluates to the \(x+y+z\) procedure

Pairs and Lists

In the late 1950s, computer scientists used confusing names
- cons: Two-argument procedure that creates a pair
- car: Procedure that returns the first element of a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list
- A (non-empty) list in Scheme is a pair in which the second element is nil or a Scheme list
- Important! Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

\[
\frac{(\text{define \ x \ (\text{cons} \ 1 \ 2))}}{(\text{define \ x \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ nil)))))}}
\]

Pairs and Lists

In the late 1950s, computer scientists used confusing names
- cons: Two-argument procedure that creates a pair
- car: Procedure that returns the first element of a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list
- A (non-empty) list in Scheme is a pair in which the second element is nil or a Scheme list
- Important! Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

\[
\frac{(\text{define \ x \ (\text{cons} \ 1 \ 2))}}{(\text{define \ x \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ nil)))))}}
\]

Pairs and Lists

In the late 1950s, computer scientists used confusing names
- cons: Two-argument procedure that creates a pair
- car: Procedure that returns the first element of a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list
- A (non-empty) list in Scheme is a pair in which the second element is nil or a Scheme list
- Important! Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

\[
\frac{(\text{define \ x \ (\text{cons} \ 1 \ 2))}}{(\text{define \ x \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ nil)))))}}
\]

Pairs and Lists

In the late 1950s, computer scientists used confusing names
- cons: Two-argument procedure that creates a pair
- car: Procedure that returns the first element of a pair
- cdr: Procedure that returns the second element of a pair
- nil: The empty list
- A (non-empty) list in Scheme is a pair in which the second element is nil or a Scheme list
- Important! Scheme lists are written in parentheses separated by spaces
- A dotted list has some value for the second element of the last pair that is not a list

\[
\frac{(\text{define \ x \ (\text{cons} \ 1 \ 2))}}{(\text{define \ x \ (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ nil)))))}}
\]

Symbolic Programming

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

\[
(\text{define \ a \ 1})
(\text{define \ b \ 2})
(\text{define \ a b \ 1})
\]

No sign of "a" and "b" in the resulting value

Quotation is used to refer to symbols directly in Lisp.

\[
(\text{list} \ 'a \ b)
(\text{list} \ 'a \ b)
\]

Symbols are now values

Quotation can also be applied to combinations to form lists.

\[
(\text{car} \ '(a \ b \ c))
(\text{cdr} \ '(a \ b \ c))
\]

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair.

\[
(\text{cdr} \ (\text{cdr} \ '(1 \ 2 \ . \ 3)))
\]

However, dots appear in the output only of ill-formed lists.

\[
'(1 \ 2 \ . \ 3)
'(1 \ 2 \ . \ (3 \ 4))
'(1 \ 2 \ 3 \ . \ nil)
\]

What is the printed result of evaluating this expression?

\[
(\text{cdr} \ '((1 \ 2) \ . \ (3 \ 4) \ . \ (5)))
\]

\[
\text{nil} \ . \ (3 \ 4) \ . \ (5)
\]