61A Lecture 24

Announcements

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, 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
> (+ (* 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

Special Forms

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

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

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

```
> (define pi 3.14)
> (* pi 2)
6.28
```

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

```
> (define (abs x)
    (if (< x 0)
        (- x)
        x))
> (abs -3)
3
```

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

(Demo)

Scheme Interpreters

(Demo)

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to anonymous procedures

```
(lambda (<formal-parameters>) <body>)
```

λ

Two equivalent expressions:

```
(define (plus4 x) (+ x 4))

(define plus4 (lambda (x) (+ x 4)))
```

An operator can be a call expression too:

```
((lambda (x y z) (+ x y (square z))) 1 2 3)  ▶  12
```

Evaluates to the $x+y+z^2$ procedure

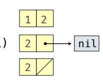
Pairs and Lists

Pairs and Lists

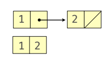
In the late 1950s, computer scientists used confusing names

- **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

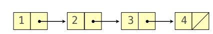
```
(cons 1 2)
(cons 2 nil)
```




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



Not a well-formed list!



(Demo)

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)
```

Symbols are now values

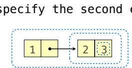
Quotation can also be applied to combinations to form lists.

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

Scheme Lists and Quotation

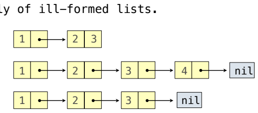
Dots can be used in a quoted list to specify the second element of the final pair.

```
> (cdr (cdr '(1 2 . 3)))
3
```



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

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



What is the printed result of evaluating this expression?

```
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```