LISP
LISP is the premier language for Artificial Intelligence applications.

It is a dynamic language: editing changes take effect immediately, without the need for recompilation.

It is primarily a functional language: all work can be done via function composition and recursion.

There is no "main program:" the programmer can call any function from the input prompt.
Introduction

- When you learn LISP, you will grasp the essential nature of the language better if you first learn to write programs using only function composition and recursion.

- This approach is harder, of course, because it requires you to master a new way of thinking.

- ``thinking in LISP''
- LISP is an old language, and has been updated many times.

- The most recent standard, Common LISP, is a huge language.

- This introduction covers only the core of LISP, and should be accurate for virtually any LISP system.
Lisp Data Types and Structures

- **Data object types**: originally only atoms and lists
- **List form**: parenthesized collections of sublists and/or atoms
e.g., `(A B (C D) E)`
- Originally, Lisp was a typeless language
- Lisp lists are stored internally as single-linked lists
Lisp Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.
  e.g., If the list `(A B C)` is interpreted as data it is a simple list of three atoms, A, B, and C.
  If it is interpreted as a function application, it means that the function named A is applied to the two parameters, B and C.

- The first Lisp interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation.
Common Lisp

- A combination of many of the features of the popular dialects of Lisp around in the early 1980s
- A large and complex language--the opposite of Scheme
- Features include:
  - records
  - arrays
  - complex numbers
  - character strings
  - powerful I/O capabilities
  - packages with access control
  - iterative control statements
Macros

Create their effect in two steps:
- Expand the macro
- Evaluate the expanded macro

Some of the predefined functions of Common Lisp are actually macros

Users can define their own macros with DEFMACRO
Common Lisp (continued)

- Backquote operator (``)
  - Similar to the Scheme’s `QUOTE`, except that some parts of the parameter can be unquoted by preceding them with commas
    
    `(a (* 3 4) c) evaluates to (a (* 3 4) c)

    `(a ,(* 3 4) c) evaluates to (a 12 c)
Common Lisp (continued)

- Common Lisp has a symbol data type (similar to that of Ruby)
  - The reserved words are symbols that evaluate to themselves
  - Symbols are either bound or unbound
    - Parameter symbols are bound while the function is being evaluated
    - Symbols that are the names of imperative style variables that have been assigned values are bound
    - All other symbols are unbound
Functions Expressions

- Primitive Arithmetic Functions: +, −, *, /, ABS, SQRT, REMAINDER, MIN, MAX

  e.g., (+ 5 2) yields 7

  (+ (* a x x) (* b x))
Special Form Function: DEFUN

DEFUN:
To bind names to expressions

e.g., (DEFUN square (x) (* x x))
Example use: (square 5)
(defun sum-greater (x y z) (> (+ x y) z))
Numeric Predicate Functions

- \( t \) is true and \( \text{nil} \) is false
- \( =, >, <, \geq, \leq \)
- \( \text{EVENP}, \text{ODDP}, \text{ZEROP} \)

- The \textsc{not} function inverts the logic of a Boolean expression
Control Flow

- Selection: the special form, IF

(IF predicate then_exp else_exp)

(IF (<> count 0)
   (/ sum count)
)

- (if t 5 6)
- (if nil 5 6)
- (if 4 5 6)
(setq a 7)
(setq b 0)
(setq c 5)
(if (> a 5)
  (  
    (setq a (+ b 7))  
    (setq b (+ c 8)))  
  (setq b 4))
Control Flow

- the **COND** function:

```
(cond ((evenp a) a)
     ((> a 7) (/ a 2))
     ((< a 5) (- a 1))
     (t 17))
```
Control Flow

- the \texttt{COND} function:

\[
\text{cond (evenp a) a) } \quad ; \text{if a is even return a} \\
\text{((> a 7) (/ a 2)) } \quad ; \text{else if a is bigger than 7 return a/2} \\
\text{((< a 5) (- a 1)) } \quad ; \text{else if a is smaller than 5 return a-1} \\
\text{t 17)) } \quad ; \text{else return 17}
\]
List Functions

- **QUOTE** - takes one parameter; returns the parameter without evaluation

  **QUOTE** is required because the Scheme interpreter, named **EVAL**, always evaluates parameters to function applications before applying the function. **QUOTE** is used to avoid parameter evaluation when it is not appropriate.

- **QUOTE** can be abbreviated with the apostrophe prefix operator

  '(A B) is equivalent to (**QUOTE** (A B))
List Functions (continued)

- **LIST** is a function for building a list from any number of parameters

  \[(\text{LIST } '\text{apple } '\text{orange } '\text{grape}) \text{ returns}\]

  (apple orange grape)
'(I have 3 books)

'(I have (+ 1 2) books)
'(I have 3 books)

'(I have (+ 1 2) books)

(list `I `have (+ 1 2) `books)
(list '+ 1 2) (+ 1 2))

Here the first argument is quoted, and so yields a list. The second argument is not quoted, and is treated as a function call, yielding a number.
The function cons builds lists. If its second argument is a list, it returns a new list with the first argument added to the front:

- `(cons 'a '(b c d))`

- `(cons 'a (cons 'b nil))`
- `(list 'a 'b)`
car / cdr

- The car of a list is the first element,
- The cdr is everything after the first element.

- (car '(a b c))
- (cdr '(a b c))
List Functions (continued)

- Examples:

  (CAR '((A B) C D)) returns ????

  (CAR 'A) returns ???

  (CDR '((A B) C D)) returns ???

  (CDR 'A) returns ???

  (CDR '(A)) returns ???

  (CONS () '(A B)) returns ???

  (CONS '(A B) '(C D)) returns ???

  (CONS 'A 'B) returns ???
List Functions (continued)

Examples:

```lisp
(CAR ′((A B) C D)) returns (A B)

(CAR ′A) is an error

(CDR ′((A B) C D)) returns (C D)

(CDR ′A) is an error

(CDR ′(A)) returns ()

(CONS ′() ′(A B)) returns (() A B)

(CONS ′(A B) ′(C D)) returns ((A B) C D)

(CONS ′A ′B) returns (A . B) (a dotted pair)
```
(car (cdr (cdr '(a b c d)))))