Artificial Intelligence Programming
CSC 521 Spring 2010

Rob St. Amant

Department of Computer Science
North Carolina State University

Lisp basics
Why Lisp?

Some recent Lisp success stories include Paul Graham’s Viaweb, which became Yahoo Store when Yahoo bought his company; ITA Software’s airfare pricing and shopping system, QPX, used by the online ticket seller Orbitz and others; Naughty Dog’s game for the PlayStation 2, Jak and Daxter, which is largely written in a domain-specific Lisp dialect Naughty Dog invented called GOAL, whose compiler is itself. [Seibel, Practical Common Lisp]
What advantages does Lisp offer?

The nearest thing Common Lisp has to a motto is the koan-like description, “the programmable programming language.” [Seibel]

- High-level concepts can often be mapped directly onto Lisp constructs.
- Lisp is extensible at the syntactic level.
- Lisp data structures and programs are expressed in a common form.
Why Lisp and AI?

John McCarthy conceived of Lisp at the 1956 Dartmouth Summer Research Project on Artificial Intelligence.

Symbolic processing remains a core area in AI, important for search, pattern matching, planning, natural language, . . . Lisp is very good for this. (Lisp is also good for number crunching, though it takes some skill.)
Lecture topics

- Atoms and lists
- Evaluation in the REPL
- Basic functions
- Variables
- Control
- Functions
- Our own tiny REPL
- Miscellany
Source material

*Practical Common Lisp* [PCL], Peter Seibel, Acorn Press, 2003–5 (online version, Chapters 1–2).

Atoms

One of the basic divisions in Lisp is between atoms and lists (collectively called s-expressions). Examples of atoms include

- Numbers
- Strings
- Symbols
- Vectors
- Hash tables...
Lists

Internally, lists are constructed from cons cells. A cons has two parts: a car and a cdr.

A single cons in which the car and cdr are both atoms is sometimes called a dotted pair: (car . cdr).

The empty list is a special case: it is a list but not a cons, and it is also an atom. Its printed representation is () or NIL.
A sample cons structure
Proper lists

A proper list is a chain of cons cells terminated by NIL. (An improper list is terminated by a non-NIL atom.)

Printed representation of the example: (A B)
Equivalently: (A . (B . NIL))

In general we won’t worry too much about the cons-level representation of lists; our examples will all deal with proper lists.
Because lists are represented internally as cons structures, the car of a cons cell can be interpreted as the first element of a list beginning with that cons cell; the cdr analogously being the remainder.

While we have used a symbol such as A as an example of the car of a cons cell (or the first element of a list), we might have used any object, including another list.
Arithmetic

Basic arithmetic examples; note the prefix form.

(+ 1 2 3) → 6 ; 1 + 2 + 3

(sqrt -50) → #C(0.0 7.071068) ; \sqrt{-50}

(/
(*
8
10)
6) → 40/3 ; (8 \times 10)/6

(/
(*
8
10.0)
6) → 13.333333 ; (8 \times 10.0)/6

(floor 2.5) → 2 0.5 ; \lfloor 2.5 \rfloor

(expt 2 200) ; 2^{100}

→ 1267650600228229401496703205376
The REPL

The REPL is the *read-eval-print-loop*, an environment (sometimes called a Lisp listener) that interactively interprets (or compiles) Lisp expressions.

CL-USER> 100
100
CL-USER> (+ 1 2 3)
6
CL-USER> "hello, world"
"hello, world"
The REPL: read

When you type or paste text into the REPL, the *Lisp reader* translates the text into a Lisp object, with the read function. The Lisp reader is a procedure “that parses character representations of objects from a stream, producing objects.” [CLHS]

I/O in Lisp works on *streams*: objects that act as a source or sink for characters (or bytes).

By default, interactive input in the REPL is read from the stream *standard-input*. 
The REPL: eval

The s-expression that has been read in is evaluated by the function eval.

Numbers in Lisp are self-evaluating: if eval is given a number, it simply returns that number.

Some other objects are also self-evaluating:: T, NIL, strings, keyword symbols, etc.

More to come.
The REPL: print

The result of the evaluation is a Lisp object; objects have a default printed representation. This result is output on the stream *standard-output* using the print function.

More flexible (and complex) output is possible with format.
The REPL: loop

The *read-eval-print* sequence is wrapped in an infinite loop.

Useful variables in the REPL:

- *, **, ***: the three most recent results.
- +, ++, +++: the three most recently evaluated forms.
Evaluation

To understand what’s happening during evaluation, we’ll need to talk about two concepts:

- Symbols.
- Operators and arguments in list form.
Symbols

A symbol is a data structure with these attributes (slots):

- **Name**: A string used for identification.
- **Value**: Optionally, an object.
- **Function**: Optionally, a function definition.
- **Package**: Optionally, an object for organization.
- **Property list**: Mainly for historical compatibility.
List evaluation

Let’s consider how Lisp evaluates (+ 1 2). In simplified form, eval works as follows:

1. Get the car of the list, the operator +.
2. Determine that it is a symbol.
3. Determine that its function slot is bound.¹
4. Walk through the cdr of the list, left to right; eval each argument. (Note the recursion.)
5. Apply the function + to 1 and 2.

¹More on what “bound” means in a bit.
Symbol evaluation

Evaluation of symbols is more straightforward:

1. Determine that the s-expression is a symbol.
2. Determine that its value slot is bound.
3. Return the object in the value slot.

For example, the built-in symbol *print-base* is used to control the printing of rational numbers:

```
CL-USER> *print-base*
10
```
Too much information?

Why do we care about Lisp’s internal evaluation model? Because we have access to all of its functionality when we’re writing programs.

We’ll shortly be able to write a simplified form of the REPL for ourselves. To understand how, we’ll need a few more concepts:

- List construction and traversal functions.
- Variable binding.
- Functions.
List functions

The components of cons and list structures we have named are also functions: \texttt{cons} constructs a cons cell; \texttt{car} and \texttt{cdr} return the relevant substructures.

\[
\text{(cons 1 (cons 2 nil))} \rightarrow (1 \ 2)
\]

\[
\text{(car (cons 1 (cons 2 nil)))} \rightarrow 1
\]

\[
\text{(cdr (cons 1 (cons 2 nil)))} \rightarrow (2)
\]
For convenience, we have other functions to work with. The functions first and rest are modern equivalents of car and cdr; second through tenth access remaining elements. To build a list we can use list instead of cons. The function last returns the last cons of a list.

\[
\begin{align*}
(\text{list } 1 & \ 2 \ 3) \rightarrow (1 \ 2 \ 3) \\
(\text{second } (\text{rest } (\text{list } 1 & \ 2 \ 3))) & \rightarrow 3 \\
(\text{last } (\text{list } 1 & \ 2 \ 3)) & \rightarrow (3)
\end{align*}
\]
Yet more basic list functions:

- **nth and nthcdr**: the nth element and “tail” of a list.
- **append**: list concatenation.
- **reverse**: list reversal.
- **butlast**: list subsequence—all but the last \( n \) (with \( n = 1 \) by default.)
- **listp and atom**: testing whether an s-expression is a list or not.
- **length**: number of “top-level” elements in a list.
Variable assignment

Syntax: setf pair* → result*
        psetf pair* → result*

(setf *p1* 10) → 10
(setf *p1* 5 *p2* 3) → 3
(psetf *p1* *p2* *p2* *p1*) → 3 ; a swap, in parallel

The symbol *p1* is said to be bound; it has a number in its value slot. Trying to access the value of an unbound symbol results in an error.
Variable declaration

A “global” variable like *p1* can be declared before it is used, though Lisp implementations generally do the right thing when a new variable is set to a value in the top level REPL.

(defvar *p1*) → *P1*
(defparameter *p2* 20) → *P2*
(defconstant *p3* 30) → *P3*
Local variables

Syntax: \texttt{let (\{var \mid (var [init-form])\})*) form* → result*}

\texttt{let} performs bindings in parallel, \texttt{let*} in sequence.

\begin{verbatim}
(let ((a 2)
     (b 3))
  (+ a b))
→ 5

declares bindings

(let* ((a 2)
       (b (* a a)))
  b)
→ 4
\end{verbatim}
Wait a minute. . .

Variable assignments and declarations don’t seem to follow the simple evaluation model we described above.

What would happen if `setf` were defined in the same way as a function such as `+`?

`(setf *p1* 10) → ?`
Special operators and macros

Some symbols are defined as *special operators*; lists with a special operator as the first element are *special forms*. Special forms have different evaluation rules than function symbols, with `let` being just one example.

Other symbols are defined as *macros*, which perform source-level transformations on code; `setf` is an example. Macros also have special evaluation rules.
Binding variables

`setf` treats its first argument(s) in a special way; as with other programming languages, symbols are treated as *places* when they’re on “the left hand side” of an assignment.
A control issue

Suppose that I want a branch in my code. If I define a function `branch` (not a built-in operator) to handle conditional statements, what will `eval` do with it?

```
(branch (= 1 1)
  (print "Yes!"))
```

```
 (print "No!"
))
```
“Stopping” evaluation

It’s sometimes (often?) necessary to get at an s-expression itself, without evaluation:

CL-USER> (setf *p1* 10)
10
CL-USER> *p1*
10
CL-USER> (quote *p1*)
*p1*
CL-USER> (symbol-value ’*p1*)
10

quote (or ’) is a special operator that returns its argument without evaluating it.
Similarly, lists can be quoted to prevent their evaluation. Quotes can be used within a list structure, to prevent evaluation at any point. If \(*p1*\) is bound to 10, then

\[
(+ \*p1* (+ \*p1* \*p1*)) \rightarrow 30
\]

\[
'(+ \*p1* \*p1*) \rightarrow (+ \*p1* \*p1*)
\]

\[
(list \*p1* (+ \*p1* \*p1*)) \rightarrow (10 20)
\]

\[
(list \*p1* '(+ \*p1* \*p1*)) \rightarrow (10 (+ \*p1* \*p1*))
\]
Control: conditionals

Syntax: if test-form then-form [else-form] → result*

(let ((n 7))
  (if (> n 0)
    (setf n (- n 1))
    (setf n (+ n 1)))))
→ 6 ; for a meaningless example

As discussed, if if were an ordinary function, this wouldn’t work: test-form, then-form, and else-form would all be evaluated.
Control: sequential evaluation

Syntax: \texttt{progn form* → result*}

\texttt{progn} evaluates forms in order and returns the value(s) of the last.

\begin{verbatim}
(progn
  (print "First!")
  (print "Second!")
  t)
"First!"
"Second!"
→ t
\end{verbatim}

Many operators (but not if) support implicit progs.
Control: iteration

Syntax: **dotimes** (var count-form [result-form])

tag | statement* → result*

(let ((sum 0))
  (dotimes (j 20)
    ;; Note implicit progn
    (print j)
    (setf sum (+ sum (random 1.0))))
  (/ sum 20))

0

... 19

→ 0.5631143
Syntax: **dolist** (var list-form [result-form])
tag | statement* → result*

(let ((list nil))
  (dolist (x (list 1 2 3 4 5))
    (setf list (cons x list)))
list)
→ (5 4 3 2 1) ; Hey, reverse!

(Other iteration operators (**do**, **do***, and **loop**) are much more powerful than the ones we’ve seen so far.)
Functions

We can only go so far typing expressions into the REPL. Let’s define a function:

```
(defun factorial (n)
  (if (> n 0)
      (* n (factorial (- n 1)))
      1))

(factorial 5) → 120
```
Functions and symbols

Where does factorial “live”? In the function slot of the symbol factorial.

(symbol-function 'factorial)
→ #<Compiled-function FACTORIAL #x84BCD36>

Notice that a symbol can have an independent value and function; you’ll often see the symbol list used as a variable as well as being called as a function, within the same function.
Applying functions

Given a function, we can call it by one of two operators: `apply` or `funcall`. These behave identically but expect their arguments differently structured.

```
(funcall (symbol-function '+) 1 2 3) → 6
```

```
(apply (symbol-function '+) '(1 2 3)) → 6
```

```
(funcall (symbol-function '+) '(1 2 3))
Error: value (1 2 3) is not of the expected type
NUMBER...
```
Why do we need apply and funcall? Why don’t we just call + directly?

(funcall (symbol-function '+) 1 2 3) → 6

(+ 1 2 3) → 6
Why do we need apply and funcall? Why don’t we just call + directly?

\[(\text{funcall (symbol-function ' +)} 1 2 3) \rightarrow 6\]

\[(+ 1 2 3) \rightarrow 6\]

apply and funcall give us abstraction: We can pass functions around and apply them, without knowing their names or even what they do.
A quick and dirty desk calculator

(defun dc ()
  (dotimes (j 20)
    (print (dc-eval (read))))
)

(defun dc-eval (sexp) ; no error-checking
  (if (atom sexp)
      sexp
    (let ((results nil))
      (dolist (arg (rest sexp))
        (setf results
          (cons (dc-eval arg) results)))
      (apply (symbol-function (first sexp))
        (reverse results))))

We’ll see a more elegant implementation in the next lecture.
There are a few things essential to know, but it’s hard to place them appropriately in a short presentation.

- “Historical” constructs based on car and cdr.
- Comparisons.
- Loading files.
- Writing and editing Lisp.
Car, cdr, etc.

caaaar, caaadr, caaar, caadar, caaddr, caadr, caar, cadaar, cadadr, cadar, caddar, cadddr, caddr, cdr, car, cdaaar, cdaadr, cdaar, cdadar, cdaddr, cdadr, cdr, cdar, cddaar, cddadr, cddar, cdddar, cdddar, cddddr, cdddr, cddr, and cdr are all Lisp functions.

(cadr arg) = (car (cdr arg)) = (first (rest arg)).
(caar arg) = (car (car arg)) = (first (first arg)).
(cddr arg) = (cdr (cdr arg)) = (rest (rest arg)).

... Avoid the longer versions.
Comparisons

Some predicates apply to specific types of objects:

- Numbers: =, <, <=, >, >=, etc. Numerical type conversion is implicit and automatic.
- Characters: char=, char>, char>=, char-equal, char-lessp, char-not-lessp, etc.
- Strings: string=, string>, string>=, string-equal, string-lessp, string-not-lessp, etc.

General equality of objects is more subtle.
Equality: eq

\(eq\) is the most discriminating: It returns \(T\) if its two arguments are the *identical object*.

\[
\begin{align*}
(eq\ 'a\ 'a) & \rightarrow T \\
(eq\ 1\ 1) & \rightarrow T \text{ or } NIL \ ; \ \text{ Why?} \\
(eq\ #\a\ #\a) & \rightarrow T \text{ or } NIL \\
(eq\ 1\ 1.0) & \rightarrow NIL \\
(eq\ '(1)\ '(1)) & \rightarrow T \text{ or } NIL \ ; \ \text{ Why?} \\
(eq\ (\text{cons}\ 1\ 2)\ (\text{cons}\ 1\ 2)) & \rightarrow NIL \\
(eq\ "abc"\ "abc") & \rightarrow NIL \\
(eq\ '(1.0\ 1)\ '(1\ 1.0)) & \rightarrow NIL
\end{align*}
\]
Equality: eql

eql relaxes eq to encompass numbers and characters, if they are of the same type.

(eql 'a 'a) → T
(eql 1 1) → T
(eql #\a #\a) → T
(eql 1 1.0) → NIL
(eql '(1) '(1)) → T or NIL
(eql (cons 1 2) (cons 1 2)) → NIL
(eql "abc" "abc") → NIL
(eql '(1.0 1) '(1 1.0)) → NIL
Equality: equal

equal relaxing eql to encompass lists and strings (among other objects) with the same structure and contents.

\[
\begin{align*}
\text{(equal 'a 'a)} & \rightarrow T \\
\text{(equal 1 1)} & \rightarrow T \\
\text{(equal #\a #\a)} & \rightarrow T \\
\text{(equal 1 1.0)} & \rightarrow \text{NIL} \\
\text{(equal '(1) '(1))} & \rightarrow T \\
\text{(equal (cons 1 2) (cons 1 2))} & \rightarrow T \\
\text{(equal "abc" "abc")} & \rightarrow T \\
\text{(equal '(1.0 1) '(1 1.0))} & \rightarrow \text{NIL}
\end{align*}
\]
Equality: equalp

equalp relaxes equal to handle character and string comparisons without case sensitivity, as well as numbers regardless of type.

\[(\text{equalp} \ 'a\ 'a) \rightarrow T\]
\[(\text{equalp} \ 1 \ 1) \rightarrow T\]
\[(\text{equalp} \ #\a \ #\a) \rightarrow T\]
\[(\text{equalp} \ #\A \ #\a) \rightarrow T\]
\[(\text{equalp} \ #C(1.0 \ 0.0) \ 1) \rightarrow T\]
\[(\text{equalp} \ (\text{cons} \ 1 \ 2) \ (\text{cons} \ 1 \ 2)) \rightarrow T\]
\[(\text{equalp} \ "abc" \ "abc") \rightarrow T\]
\[(\text{equalp} \ "AbC" \ "aBc") \rightarrow T\]
\[(\text{equalp} \ '(1.0 \ 1) \ '(1 \ 1.0)) \rightarrow T\]
Loading files

You can save and load Lisp files, instead of typing (and retyping) everything into the REPL.

CL-USER> (load "test-file.lisp")
; Loading ...
T

Forms in a file are evaluated in order. (Comments, by the way, begin with a semicolon and end at the end of a line.)
Editing files

Using a Lisp-aware editor is essential. It should provide at least

- Parenthesis matching.
- Automatic indentation. (Experienced programmers ignore parentheses when reading Lisp, relying on indentation instead.)
- Incremental compilation.
Reading and writing Lisp

A few hints for novices:

- **Whitespace**: `my-var` is a single symbol.
- **Case insensitivity**: `FOO`, `foo`, `Foo`, and `fOo` all name the same symbol.
- **Naming**: Use `my-name`, not `my_name` or `MyName`; don’t worry about long names.
- **Parentheses** are meaningful, not just for grouping: `(if ((atom x))...)` won’t work.
- **Lisp style**: Use it.
Don’t do this, for example:

(defun dc-eval (sexp)
  (if ((atom sexp)) ; broken
    sexp
    (let ( results = nil ) ; broken
      (dolist (arg (rest sexp))
        (setf results
          (cons (dc-eval arg) results)
        )
      )
      (apply (symbol-function (first sexp))
        (reverse results)
      )
    )
  ) ; Do we really need all these line breaks? Not really.
)

(dc-eval(4)) → Error: 4 is not a function...