Lisp I

Versions of LISP
- LISP is an acronym for List Processing language
- Lisp is an old language with many variants
  - Fortran is the only older language still in wide use
  - Lisp is alive and well today
- Most modern versions are based on Common Lisp
- Scheme is one of the major variants
  - We will use Lisp, not Scheme, in this class
  - Scheme is used for CS 101 in quite a few Universities
- The essentials haven't changed much

LISP Features
- An expression as the universal data type
  - Atoms are similar to identifiers, but can also be numeric constants
  - Lists can be lists of atoms, lists, or any combination of the two
- Functional Programming Style – all computation is performed by applying functions to arguments. Variable declarations are rarely used.
- Uniform Representation of Data & Code – eg, (A B C D) is
  - A list of four elements (interpreted as data)
  - An application of the function 'A' to the three parameters B, C, and D (interpreted as code)
- Reliance on Recursion – a strong reliance on recursion has allowed Lisp to be successful in many areas, including Artificial Intelligence.
- Garbage Collection – Lisp has built-in garbage collection – programmers need not explicitly free dynamically allocated memory.

Pure Lisp and Common Lisp
- Lisp has a small and elegant conceptual core that has not changed much in over 40 years.
- McCarthy’s original Lisp paper defined all of Lisp using just seven primitive functions
- Common Lisp is large (> 800 built-in functions), has all the modern data-types, good programming environments, and good compilers.
Informal Syntax

- An \textit{atom} is either an integer or an identifier.
- A \textit{list} is a left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis.
- An \textit{S-expression} is an atom or a list.
- Example: \((A \ (B \ 3) \ (C) \ (\ () \ ) )\)

Hello World

\begin{verbatim}
(defun helloWorld ()
  ;; prints and returns the message.
  (print "Hello World")
\end{verbatim}

T and NIL

- \texttt{NIL} is the name of the empty list, ( )
- As a test, \texttt{NIL} means "false"
- \texttt{T} is usually used to mean "true," but...
- ...anything that isn't \texttt{NIL} is "true"
- \texttt{NIL} is both an atom and a list
  - it's defined this way, so just accept it

Predicates

- A predicate (in any computer language) is a function that returns either "true" or "false"
- In Lisp,
  - "false" is represented by \texttt{NIL}, or ()
  - "true" is represented by \texttt{anything} that isn't \texttt{NIL}
- Hence, a Lisp predicate returns either \texttt{NIL} or non- \texttt{NIL}
  - Predicates often return "true" values other than \texttt{T}, especially if the returned value might be useful
  - E.g. (member 'c '(a b c d e f)) returns '(d e f))
**Function calls and data**

- A function call is written as a list
  - the first element is the name of the function
  - remaining elements are the arguments
- Example: \((F \ A \ B)\)
  - calls function \(F\) with arguments \(A\) and \(B\)
- Data is written as atoms or lists
- Example: \((F \ A \ B)\) is a list of three elements
  - Do you see a problem here?

**Quoting**

- Is \((F \ A \ B)\) a call to \(F\), or is it just data?
- All *literal data* must be quoted (atoms, too)
  - Except NIL, which does not need to be quoted
- `(QUOTE (F A B))` is the list \((F \ A \ B)\)
  - `QUOTE` is not a function, but a *special form*
  - The arguments to a special form are not evaluated or evaluated in some special manner
- `(QUOTE (F A B))` is another way to quote data
  - There is just one single quote at the beginning
  - It quotes an `S-expression`

**Basic Functions**

- **CAR** returns the head of a list
  - `(car '(1 2 3)) => 1`
- **CDR** returns the tail of a list
  - `(cdr '(1 2 3)) => (2 3)`
- **CONS** inserts a new head into a list
  - `(cons 1 '(2 3)) => (1 2 3)`

**More Basic Functions**

- **EQ** compares two atoms for equality
  - `(eq 'foo 'foo) => T, (eq 'foo 'bar) => NIL`
- **ATOM** tests if its argument is an atom
  - `(atom 'foo) => T, (atom '(1 2)) => NIL`
Other useful Functions

- (NULL S) tests if S is the empty list
  - (NULL '(1 2 3)) => NIL
  - (NULL NIL) => T
- (LISTP S) tests if S is a list
  - (listp '(1 2 3)) => T
  - (listp '3) => NIL

More useful Functions

- LIST makes a list of its arguments
  - (LIST 'A 'B C 'D) => (A B C D)
  - (LIST (CDR 'A B) 'C) => ((B) C)

  Note that the parenthesized prefix notation makes it easy to define functions that take a varying number of arguments.
  - (LIST 'A) => (A)
  - (LIST) => NIL

More useful Functions

- APPEND concatenates two lists
  - (APPEND '(1 2) '(3 4)) => (1 2 3 4)
  - (APPEND '(A B) '((X) Y)) => (A B X Y)
  - (APPEND NIL '(1 2 3)) => (1 2 3)
  - (APPEND NIL NIL NIL) => NIL

Dotted Pairs

- The second argument to CONS can be:
  - A list: the result is always another list
  - An atom: the result is a dotted pair
- CONS of A and B is (A . B)
  - (CAR '(A . B)) => A
  - (CDR '(A . B)) => B
**EQUAL and EQ**

- **EQUAL** tests whether two s-expressions are “the same”.
  - `(equal '(a b (c)) '(a b (c)))` => **T**
  - `(equal '(a (b) c) '(a b (c)))` => **NIL**
- **EQ** tests whether two symbols are equal
  - `(eq 'foo 'foo)` => **T**
  - `(eq 'foo 'bar)` => **NIL**
- **EQ** is just a pointer test, like Java’s ‘=’
- **EQUAL** compares two complex objects, like a Java object’s equal method

---

**ATOM**

- **ATOM** takes any S-expression as an argument
- **ATOM** returns “true” if the argument you gave it is an atom
- As with any predicate, **ATOM** returns either **NIL** or something that isn’t **NIL**

---

**COND**

- **COND** implements the if...then...elseif...then...elseif...then... control structure
- The arguments to a function are evaluated before the function is called
  - This isn’t what you want for **COND**
- **COND** is a **special form**, not a function

---

**Special forms**

- A special form is like a function, but it evaluates the arguments as it needs them
- **COND**, **QUOTE** and **DEFUN** are special forms
- Lisp lets you define your own special forms
- We won’t be defining special forms in this course
Form of the COND

(COND
  (condition1  result1 )
  (condition2  result2 )
  ...
  (T resultN ))

Cond Example

(cond ((not (numberp x)) 0)
  ((< x 0) 0)
  ((< x 10) x)
  (T 10))

NOT is an alias for NULL

IF

- In addition to COND, Lisp has an IF special form that does much the same thing
- Note: IF is a function that returns a value.
- (IF <test> <then> <else>)
  - (IF (< 4 6) 'foo 'bar) => foo
  - (IF (< 4 2) 'foo 'bar) => bar
  - (IF <test> <then> )
  - (IF (= 1 (+ 2 1)) 'foo) => NIL

Defining Functions

- (DEFUN function_name  parameter_list  function_body )
- Example: Test if the argument is the empty list
  (DEFUN NULL  (X) (IF X NIL T) )
Example: MEMBER

| Here's how MEMBER is actually defined: |
|---|---|
| (DEFUN MEMBER (X LIST) |
| (COND ((NULL LIST) NIL) |
| ((EQUAL X (CAR LIST)) T) |
| (T (MEMBER X (CDR LIST))) |)

Note that it returns NIL if the 1st arg is not found. Why?

MEMBER is typically a built-in function.

Here are two versions, using if and cond:

```lisp
(defun append (l1 l2)
  (if (null l1) l2
    (cons (car l1) (append (cdr l1) l2))))
```

```
(defun append (l1 l2)
  (cond ((null l1) l2)
        (t (cons (car l1) (append (cdr l1) l2))))
)
```

(append '(1 2 3) '(a b)) => (1 2 3 a b)

Example: UNION

| Here are two examples, using if and cond: |
|---|---|
| (DEFUN UNION (SET1 SET2) |
| (COND ((NULL SET1) SET2) |
| ((MEMBER (CAR SET1) SET2) UNION (CDR SET1) SET2) |
| (T UNION (CDR SET1) SET2)) |)}

(append '(1 2 3) '(a b)) => (1 2 3 a b)
Naïve Reverse

(defun reverse1 (l)
  (if (null l)
      nil
      (append (reverse1 (cdr l))
              (list (car l)))))

Tail Recursive Reverse

(defun reverse2 (l)
    (reverse2sub l nil))
(defun reverse2sub (l answer)
    (if (null l)
        answer
        (reverse2sub (cdr l)
                    (cons (car l) answer))))

Still more useful functions

- (LENGTH L) returns the length of list L
  - The “length” is the number of top-level elements in the list
- (RANDOM N), where N is an integer, returns a random integer \( \geq 0 \) and \(< N\)
- EQUAL tests if two S-expressions are equal
  - If you know both arguments are atoms, use EQ instead

Programs on file

- Use any text editor to create your program
- Save your program on a file with the extension .lsp
  - (Load ‘foo) loads foo.lsp
  - (load “foo.bar”) loads foo.bar
- Each s-expression in the file is read and evaluated.
Comments

- In Lisp, a comment begins with a semicolon (;) and continues to the end of the line
- Conventions for ;; and ;; and ;
- Function document strings:
  (defun square (x)
   "(square x) returns x*x"
   (* x x))

Read – eval - print

Lisp’s interpreter essentially does:
(\(\text{loop (print (eval (read)))}\))
\(\text{i.e.,}\)
1. Read an expression
2. Evaluate it
3. Print the resulting value
4. Go to 1

Understanding the rules for evaluating an expression is key to understanding lisp.
Reading and printing, while a bit complicated, are conceptually simple.

Eval(S)

- If S is an atom, then call evalatom(A)
- If S is a list, then call evallist(S)

EvalAtom(S)

- Numbers eval to themselves
- T evals to T
- NIL evals to NIL
- Atomic symbol: look up current value of symbol
**EvalList(S)**

- Assume S is (s1 s2 ... sn)
  - If s1 is an atom representing a special form (e.g., quote, defun) handle it as a special case
  - If s1 is an atom naming a regular function
    - Evaluate s2 .. Sn
    - Apply the function named by S1 to the resulting values
  - If 11 is a list... more on this later ...

---

**Iteration - Loop**

- (loop <s1><s2>...<sn>) executes the <si>'s until an explicit return is done.
  - (defun echo ()
    - (loop (if (null (print (read)))(return t)))

---

**Iteration - DO**

- (do ((x 1 (1+ x))
  (y 100 (1- y)))
  ((> x y)(+ x y))
  (princ “Doing “)
  (princ (list x y))
  (terpri))

---

**Let**

- (let <vars><s1><s2>...<sn>)
  - <vars> = (<var1>...<varn>)
  - <var1> = <name> or (<name> <value>)

- Creates environment with local variables v1..vn, initializes them in parallel & evaluates the <si>.
  - Example:
    - >(let (x (y)(z (+ 1 2))) (print (list x y z)))
      (NIL NIL 3)
      (NIL NIL 3)
Output

> (print '(foo bar))
(FOO BAR)
> (setq *print-length* 3)
3
> (print '(1 2.3 4 5 6.7 8))
(1 2.3 ...)
> (format t "The sum of one and one is \( + 1 1 \)\)
The sum of one and one is 2.
NIL

Getting help: apropos and describe

> (defun foo (x) "foo is my function" (plus x x))
> (apropos 'foofoo)
:FOO                                       constant:
FOO                                        function:
:FOOTER                                    constant:
> (describe 'foofoo)
foofoo is the symbol FOO, lies in #<PACKAGE COMMON--LISPLISP--USER>, is accessible in 1
package COMMON--LISPLISP--USER, names a function, has 2 properties
SYSTEM::DEFINITION, SYSTEM::DOCUMENTATION--STRINGS.
Documentation as a FUNCTION:
foofoo is my function
For more information, evaluate (SYMBOL--PLIST 'FOO).
#<CLOSURE FOO (X) (DECLARE (SYSTEM::IN--DEFUN FOO)) (BLOCK FOO (PLUS X X))>
is an interpreted function.
argument list: (X)

Input

- (read) reads and returns one s-expression
  from the current open input stream.
- (read)
- foo
- FOO
- (read)
- (a b (1 2))
- (A B (1 2))