Programming Languages

Lecturer: William W.Y. Hsu
Chapter 10a – LISP Basic Supplemental
The Common LISP Environment

- The Common LISP environment follows the algorithm below when interacting with users:

  loop read in an expression from the console;
  evaluate the expression;
  print the result of evaluation to the console;
  end loop.
The Common LISP Environment

• Common LISP reads in an expression, evaluates it, and then prints out the result.
• For example, if you want to compute the value of \((2 \times \cos(0) \times (4+6))\), you type in:

\[ (* 2 (\cos 0) (+ 4 6)) \]

• Common LISP replies: 20.0
The Common LISP Environment

- LISP expressions are composed of forms.
  - The most common LISP form is function application. LISP represents a function call $f(x)$ as (f x).
  - For example, $\cos(0)$ is written as (cos 0).
- LISP expressions are case-insensitive. It makes no difference whether we type (cos 0) or (COS 0).
- Similarly, "+" is the name of the addition function that returns the sum of its arguments.
- Some functions, like "+" and "*", could take an arbitrary number of arguments.
  - (* 2 3)  (* 2 3 4 5).
The Common LISP Environment

- In general, a function application form looks like \((function \ argument_1 \ argument_2 \ldots \ argument_n)\).
  - As in many programming languages (e.g. C/C++), LISP evaluates function calls in \textit{applicative order}, which means that all the argument forms are evaluated before the function is invoked.
  - The argument forms \((\cos 0)\) and \((+ 4 6)\) are respectively evaluated to the values \(1\) and \(10\) before they are passed as arguments to the * function.
  - Some other forms, like the \textit{conditionals} we will see later, are not evaluated in applicative order.
The Common LISP Environment

- Numeric values like 4 and 6 are called *self-evaluating forms*: they evaluate to themselves.
  - To evaluate (+ 4 6) in applicative order, the forms 4 and 6 are respectively evaluated to the values 4 and 6 before they are passed as arguments to the + function.
# Arithmetic Built-in Functions

<table>
<thead>
<tr>
<th>Numeric Functions</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ x₁ x₂ ... xₙ)</td>
<td>The sum of x₁, x₂, ..., xₙ</td>
</tr>
<tr>
<td>(* x₁ x₂ ... xₙ)</td>
<td>The product of x₁, x₂, ..., xₙ</td>
</tr>
<tr>
<td>(− x y)</td>
<td>Subtract y from x</td>
</tr>
<tr>
<td>(/ x y)</td>
<td>Divide x by y</td>
</tr>
<tr>
<td>(rem x y)</td>
<td>The remainder of dividing x by y</td>
</tr>
<tr>
<td>(abs x)</td>
<td>The absolute value of x</td>
</tr>
<tr>
<td>(max x₁ x₂ ... xₙ)</td>
<td>The maximum of x₁, x₂, ..., xₙ</td>
</tr>
<tr>
<td>(min x₁ x₂ ... xₙ)</td>
<td>The minimum of x₁, x₂, ..., xₙ</td>
</tr>
</tbody>
</table>
Defining Functions

- Evaluating expressions is not very interesting.
- We would like to build expression abstractions that could be reused in the future. For example, we could type in the following:
  
  \[
  \text{(defun double } \ (x) \ (* \ x \ 2))
  \]

- In the above, we define a function named `double`, which returns two times the value of its input argument \( x \).

- We can then test-drive the function as below:

\[
\begin{align*}
\text{>(double 3) } & \quad 6 \\
\text{>(double 7) } & \quad 14
\end{align*}
\]
Editing, Loading and Compiling LISP Programs

- You may use any text editor to edit LISP programs.
- Loading a LISP file:
  
  (load "testing.lisp")

- LISP code can also be compiled:
  
  (compile-file "testing.lisp")
Control Structures: Recursions and Conditionals

- We can implement a function to compute factorials using recursion:
  (defun factorial (N)
    "Compute the factorial of N."
    (if (= N 1) 1 (* N (factorial (- N 1))))
  )

- The if form checks if N is one, and returns one if that is the case, or else returns \(N \times (N-1)!\).

- Condition expression \((= N 1)\) is a relational expression. It returns boolean values T or NIL.
  - In fact, LISP treats NIL as false, and everything else as true.
Control Structures: Recursions and Conditionals

- Other relational operators include the following:

<table>
<thead>
<tr>
<th>Relational Operators</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(= x y)</td>
<td>x is equal to y</td>
</tr>
<tr>
<td>(/= x y)</td>
<td>x is not equal to y</td>
</tr>
<tr>
<td>(&lt; x y)</td>
<td>x is less than y</td>
</tr>
<tr>
<td>(&gt; x y)</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>(&lt;= x y)</td>
<td>x is no greater than y</td>
</tr>
<tr>
<td>(&gt;= x y)</td>
<td>x is no less than y</td>
</tr>
</tbody>
</table>
Control Structures: Recursions and Conditionals

• The if form is not a strict function (strict functions evaluate their arguments in applicative order).
• if form evaluates the condition (\(= N 1\)) before further evaluating the other two arguments.
  ▫ If the condition evaluates to true, then only the second argument is evaluated, and its value is returned as the value of the if form. Otherwise, the third argument is evaluated, and its value is returned.
• Forms that are not strict functions are called special forms.
Control Structures: Recursions and Conditionals

• The function is recursive.
• The definition of factorial involves invocation of itself.
• Recursion is, for now, our only mechanism for producing looping behavior.
  ▫ Specifically, the kind of recursion we are looking at is called linear recursion, in which the function may make at most one recursive call from any level of invocation.
Control Structures: Recursions and Conditionals

- To better understand the last point, we can make use of the debugging facility \texttt{trace} (do not compile your code if you want to use trace):

\[
\begin{align*}
&> \ (\text{trace factorial}) \\
&(\text{FACTORIAL}) \\
&> \ (\text{factorial 4}) \\
&\quad 0: \ (\text{FACTORIAL 4}) \\
&\quad \quad 1: \ (\text{FACTORIAL 3}) \\
&\quad \quad \quad 2: \ (\text{FACTORIAL 2}) \\
&\quad \quad \quad \quad 3: \ (\text{FACTORIAL 1}) \\
&\quad \quad \quad \quad \quad 3: \ \text{returned 1} \\
&\quad \quad \quad \quad \quad 2: \ \text{returned 2} \\
&\quad \quad \quad \quad \quad 1: \ \text{returned 6} \\
&\quad \quad \quad \quad 0: \ \text{returned 24} \\
&24
\end{align*}
\]
Multiple Recursions

- Recall the definition of Fibonacci numbers:
  \[
  \text{Fib}(n) = \begin{cases} 
  1, & \text{for } n = 0,1 \\
  \text{Fib}(n - 1) + \text{Fib}(n - 2), & \text{for } n > 1
  \end{cases}
  \]

- This definition can be directly translated to the following LISP code:

```lisp
(defun fibonacci (N)
  "Compute the N'th Fibonacci number."
  (if (or (zerop N) (= N 1)) 1
      (+ (fibonacci (- N 1)) (fibonacci (- N 2)))))
```
Multiple Recursions

- The function call `(zerop N)` tests if N is zero.
  - Shorthand for `(= N 0)`.
  - zerop returns either `T` or `NIL`.
  - We call such a boolean function a `predicate`, as indicated by the suffix `p`. Some other built-in shorthands and predicates are the following:
Multiple Recursions

Some other built-in shorthands and predicates are the following:

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1+ x)</td>
<td>(+ x 1)</td>
</tr>
<tr>
<td>(1- x)</td>
<td>(- x 1)</td>
</tr>
<tr>
<td>(zerop x)</td>
<td>(= x 0)</td>
</tr>
<tr>
<td>(plusp x)</td>
<td>(&gt; x 0)</td>
</tr>
<tr>
<td>(minusp x)</td>
<td>(&lt; x 0)</td>
</tr>
<tr>
<td>(evenp x)</td>
<td>(= (rem x 2) 0)</td>
</tr>
<tr>
<td>(oddp x)</td>
<td>(/= (rem x 2) 0)</td>
</tr>
</tbody>
</table>
Multiple Recursions

- \texttt{or} form is a logical operator.
- \texttt{or} is not a strict function.
  
  ▫ It evaluates its arguments from left to right, returning non-\texttt{NIL} immediately if it encounters an argument that evaluates to non-\texttt{NIL}.
  ▫ It evaluates to \texttt{NIL} if all tests fail.
  ▫ In the expression \texttt{(or \: \texttt{t} \: (= \: \texttt{1} \: \texttt{1}))}, the second argument \texttt{(= \: \texttt{1} \: \texttt{1})} will not be evaluated.
Multiple Recursions

Similar logical connectives are listed below:

<table>
<thead>
<tr>
<th>Logical Operators</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(or (x_1\ x_2\ ...\ x_n))</td>
<td>Logical or</td>
</tr>
<tr>
<td>(and (x_1\ x_2\ ...\ x_n))</td>
<td>Logical and</td>
</tr>
<tr>
<td>(not (x))</td>
<td>Logical negation</td>
</tr>
</tbody>
</table>
Multiple Recursions

- The function definition contains two self references.
- It first recursively evaluates \((\text{fibonacci } (- N 1))\) to compute \(Fib(N-1)\), then evaluates \((\text{fibonacci } (- N 2))\) to obtain \(Fib(N-2)\), and lastly return their sum.
- This kind of recursive definition is called *double recursion* (more generally, *multiple recursion*).
Multiple Recursions

- Tracing the function yields the following:

```lisp
> (fibonacci 3)
0: (FIBONACCI 3)
  1: (FIBONACCI 2)
  2: (FIBONACCI 1)
  2: returned 1
  2: (FIBONACCI 0)
  2: returned 1
  1: returned 2
  1: (FIBONACCI 1)
  1: returned 1
0: returned 3
```

3
**let Special Form**

- Some beginners might find nested function calls like the following very difficult to understand:
  
  \[
  (+ \text{(fibonacci}\ (-\ N\ 1))\ \text{(fibonacci}\ (-\ N\ 2)))\]

- Make such expressions easier to write and comprehend, one could define local name bindings to represent intermediate results:

  \[
  \text{(let}\ \begin{align*}
  &((F1\ \text{(fibonacci}\ (-\ N\ 1))) \\
  &\quad (F2\ \text{(fibonacci}\ (-\ N\ 2)))) \\
  &(+\ F1\ F2))
  \end{align*}
  \]
let Special Form

- The let special form above defines two local variables, F1 and F2, which binds to \( Fib(N-1) \) and \( Fib(N-2) \) respectively.
- Under these local bindings, let evaluates \((+ F1 F2)\).

(defun fibonacci (N)
  "Compute the N'th Fibonacci number."
  (if (or (zerop N) (= N 1)) 1
   (let ((F1 (fibonacci (- N 1)))
         (F2 (fibonacci (- N 2))))
     (+ F1 F2))))
let Special Form

- Notice that let creates all bindings in parallel.
- Both (fibonacci (- N 1)) and (fibonacci (- N 2)) are evaluated first, and then they are bound to F1 and F2.
- This means that the following LISP code will not work:

```lisp
(let
  ((x 1)
   (y (* x 2))
   (+ x y))
```
**let** Special Form

- LISP will attempt to evaluate the right hand sides first before the bindings are established.
- The expression \((* \ x \ 2)\) is evaluated before the binding of \(x\) is available.
- To perform sequential binding, use the **let**\(^*\) form instead:
  
  \[
  \text{(let* (}
  \begin{align*}
  (x & \ 1) \\
  (y & (* \ x \ 2)) \\
  (+ & x \ y)
  \end{align*}
  \text{)}
  \]

- LISP will bind 1 to \(x\), then evaluate \((* \ x \ 2)\) before the value is bound to \(y\).
Lists

- Numeric values are not the only type of data LISP supports.
- LISP is designed for symbolic computing.
- The fundamental LISP data structure for supporting symbolic manipulation are lists.
- LISP stands for "LIS\textit{t} Processing."
Lists

- Lists are containers that supports sequential traversal. List is also a *recursive data structure*: its definition is recursive.
- Most of its traversal algorithms are recursive functions.
- In order to better understand a recursive abstract data type and prepare oneself to develop recursive operations on the data type, one should present the data type in terms of its *constructors*, *selectors* and *recognizers*. 
Lists

- **Constructors** are forms that create new instances of a data type (possibly out of some simpler components). A list is obtained by evaluating one of the following constructors:
  - `nil`: Evaluating `nil` creates an *empty* list.
  - `(cons x L)`: Given a LISP object `x` and a list `L`. Evaluating `(cons x L)` creates a list containing `x` followed by the elements in `L`. 
Lists

• The above definition is inherently recursive.
  ▫ To construct a list containing 1 followed by 2, we could type in the expression:

> (cons 1 (cons 2 nil))
(1 2)

• LISP replies by printing (1 2), which is a more readable representation of a list containing 1 followed by 2.
  ▫ Notice that nil is a list (an empty one), and thus (cons 2 nil) is also a list (a list containing 1 followed by nothing).
  ▫ Applying the second constructor again, we see that (cons 1 (cons 2 nil)) is also a list (a list containing 1 followed by 2 followed by nothing).
Lists

- If we already know all the elements in a list, we could enter our list as *list literals*.
- To enter a list containing all prime numbers less than 20, we could type in the following expression:

  ```lisp
  > (quote (2 3 5 7 11 13 17 19))
  (2 3 5 7 11 13 17 19)
  ```

- We have quoted the list using the `quote` special form.
  - This is necessary because, without the quote, LISP would interpret the expression `(2 3 5 7 11 13 17 19)` as a function call to a function with name "2" and arguments 3, 5, ..., 19.
Lists

• The quote is just a syntactic device that instructs LISP not to evaluate the a form in applicative order, but rather treat it as a literal.

• Since quoting is used frequently in LISP programs, there is a shorthand for quote:

```
> ' (2 3 5 7 11 13 17 19)
(2 3 5 7 11 13 17 19)
```

• The quote symbol ' is nothing but a syntactic shorthand for (quote ...).
Lists

- The second ingredient of an abstract data type are its selectors.
- Given a composite object constructed out of several components, a selector form returns one of its components.
- Suppose a list $L_1$ is constructed by evaluating $(\text{cons } x \ L_2)$, where $x$ is a LISP object and $L_2$ is a list.
Lists

- The selector forms `(first \(L_1\))` and `(rest \(L_1\))` evaluate to \(x\) and \(L_2\) respectively.

```lisp
> (first '(2 4 8))
2
> (rest '(2 4 8))
(4 8)
> (first (rest '(2 4 8)))
4
> (rest (rest '(2 4 8)))
(8)
> (rest (rest (rest '(8))))
NIL
```
Lists

- Recognizers are expressions that test how an object is constructed.
- Corresponding to each constructor of a data type is a recognizer.
- In the case of list, they are null for nil and consp for cons.
- Given a list \( L \), \((\text{null } L)\) returns \( t \) iff \( L \) is nil, and \((\text{consp } L)\) returns \( t \) iff \( L \) is constructed from \( \text{cons} \).
Lists

> (null nil)
T
> (null '(1 2 3))
NIL
> (consp nil)
NIL
> (consp '(1 2 3))
T
Structural Recursion with Lists

• Understanding how the constructors, selectors and recognizers of lists work helps us to develop recursive functions that traverse a list.
• The LISP built-in function `list-length` counts the number of elements in a list.

> (list-length '(2 3 5 7 11 13 17 19))
8
Structural Recursion with Lists

- Let us try to see how such a function can be implemented recursively.
- A given list \( L \) is created by either one of the two constructors, namely nil or a cons:
  - \textit{Case 1:} \( L \) is nil.
    - The length of an empty list is zero.
  - \textit{Case 2:} \( L \) is constructed by \texttt{cons}.
    - Then \( L \) is composed of two parts, namely, (first \( L \)) and (rest \( L \)). In such case, the length of \( L \) can be obtained inductively by adding 1 to the length of (rest \( L \)).
Structural Recursion with Lists

- Formally, we could implement our own version of list-length as follows:
  (defun recursive-list-length (L)
    "A recursive implementation of list-length."
    (if (null L) 0
      (1+ (recursive-list-length (rest L)))))

- We use the recognizer null to differentiate how $L$ is constructed.
  - In case $L$ is nil, we return 0 as its length.
  - Otherwise, $L$ is a cons, and we return 1 plus the length of (rest $L$).
  - Recall that $(1+n)$ is simply a shorthand for $(+ n 1)$. 
> (trace recursive-list-length)
(RECURSIVE-LIST-LENGTH)
> (recursive-list-length '(2 3 5 7 11 13 17 19))
  0: (RECURSIVE-LIST-LENGTH (2 3 5 7 11 13 17 19))
    1: (RECURSIVE-LIST-LENGTH (3 5 7 11 13 17 19))
      2: (RECURSIVE-LIST-LENGTH (5 7 11 13 17 19))
        3: (RECURSIVE-LIST-LENGTH (7 11 13 17 19))
          4: (RECURSIVE-LIST-LENGTH (11 13 17 19))
            5: (RECURSIVE-LIST-LENGTH (13 17 19))
              6: (RECURSIVE-LIST-LENGTH (17 19))
                7: (RECURSIVE-LIST-LENGTH (19))
                  8: (RECURSIVE-LIST-LENGTH NIL)
                    8: returned 0
                      7: returned 1
                        6: returned 2
                          5: returned 3
                            4: returned 4
                              3: returned 5
                                2: returned 6
                                  1: returned 7
                                    0: returned 8

8
Structural Recursion with Lists

• The kind of recursion we see here is called *structural recursion*.

• To process an instance $X$ of a recursive data type:
  ▫ Use the recognizers to determine how $X$ is created (i.e. which constructor creates it).
  ▫ We use null to decide if a list is created by nil or cons.

• For instances that are atomic (i.e. those created by constructors with no components), return a trivial value.
  ▫ In the case when a list is nil, we return zero as its length.

• If the instance is composite, then use the selectors to extract its components.
  ▫ We use first and rest to extract the two components of a nonempty list.
Structural Recursion with Lists

- We apply recursion on one or more components of $X$.
  - We recursively invoked recursive-list-length on (rest L).
- We use either the constructors or some other functions to combine the result of the recursive calls, yielding the value of the function.
  - In the case of recursive-list-length, we return one plus the result of the recursive call.
Structural Recursion with Lists

- Sometimes, long traces like the one for list-length may be difficult to read on a terminal screen.
- Common LISP allows you to capture screen I/O into a file so that you can produce a hard copy for more comfortable reading.
- To capture the trace of executing `(recursive-list-length '(2 3 5 7 11 13 17 19))`, we use the dribble command:
  > (dribble "output.txt")
  dribbling to file "output.txt"
  ...
  > (drible)
Structural Recursion with Lists

- The form `(dribble "output.txt")` instructs Common LISP to begin capturing all terminal I/O into a file called `output.txt`.
- The trailing `(dribble)` form instructs Common LISP to stop I/O capturing, and closes the file `output.txt`.

Symbols

- Another data type of LISP is *symbols*.
  - A symbol is simply a sequence of characters:

    ```lisp
    > 'a ; LISP is case-insensitive.
    A
    > 'A ; 'a and 'A evaluate to the same symbol.
    A
    > 'apple2 ; Both alphanumeric characters ...
    APPLE2
    > 'an-apple ; ... and symbolic characters are allowed.
    AN-APPLE
    > t ; Our familiar t is also a symbol.
    T
    > 't ; In addition, quoting is redundant for t.
    T
    > nil ; Our familiar nil is also a symbol.
    NIL
    > 'nil ; Again, it is self-evaluating.
    NIL
    ```
Symbols

- With symbols, we can build more interesting lists:

  > '(how are you today ?) ; A list of symbols.
  (HOW ARE YOU TODAY ?)

  > '(1 + 2 * x) ; A list of symbols and numbers.
  (1 + 2 * X)

  > '(pair (2 3)) ; A list containing 'pair and '(2 3).
  (pair (2 3))
Symbols

• Notice that the list (pair (2 3)) has length 2
> (recursive-list-length '(pair (2 3)))
2

• Notice also the result of applying accessors
> (first '(pair (2 3)))
PAIR
> (rest '(pair (2 3)))
((2 3))

• Lists containing other lists as members are difficult to understand for beginners.
  ▪ Make sure you understand the above example.
Example: \textit{nth}

- LISP defines a function (\textit{nth} \textit{N} \textit{L}) that returns the \textit{N}'th member of list \textit{L} (assuming that the elements are numbered from zero onwards):

\begin{verbatim}
> (nth 0 '(a b c d))
A
> (nth 2 '(a b c d))
C
\end{verbatim}
Example: \( nth \)

- We could implement our own version of \( nth \) by linear recursion.
- Given \( N \) and \( L \), either \( L \) is nil or it is constructed by cons.
  - \textbf{Case 1:} \( L \) is nil.
    - Accessing the \( N \)'th element is an undefined operation, and our implementation should arbitrarily return nil to indicate this.
  - \textbf{Case 2:} \( L \) is constructed by a cons.
    - Then \( L \) has two components: (first \( L \)) and (rest \( L \)). There are two subcases: either \( N = 0 \) or \( N > 0 \):
      - \textbf{Case 2.1:} \( N = 0 \).
        - The zeroth element of \( L \) is simply (first \( L \)).
      - \textbf{Case 2.2:} \( N > 0 \).
        - The \( N \)'th member of \( L \) is exactly the \((N-1)\)'th member of (rest \( L \)).
Example: \textit{nth}

- The following code implements our algorithm:

\begin{verbatim}
(defun list-nth (N L)
  "Return the N'th member of a list L."
  (if (null L)
      nil
      (if (zerop N)
          (first L)
          (list-nth (1- N) (rest L))))
\end{verbatim}

- \(1 - N\) is merely a shorthand for \((- N 1)\).
Example: *nth*

> (list-nth 2 '(a b c d))
  0: (LIST-NTH 2 (A B C D))
    1: (LIST-NTH 1 (B C D))
      2: (LIST-NTH 0 (C D))
      2: returned C
    1: returned C
  0: returned C
C
Example: \textit{nth}

- LISP has a built-in function (last \(L\)) that returns a the last cons structure in a given list \(L\).

\[
\text{> (last '(a b c d))}
\]
\[
(d)
\]
\[
\text{> (last '(1 2 3))}
\]
\[
(3)
\]
Example: nth

- Alternatively, it can be implemented using the \texttt{cond} special form.

\begin{verbatim}
(defun list-nth (n L)
  "Return the n'th member of a list L."
  (cond ((null L) nil)
        ((zerop n) (first L))
        (t (list-nth (1- n) (rest L)))))
\end{verbatim}
Example: **member**

- LISP defines a function (member $E L$) that returns non-NIL if $E$ is a member of $L$.

```lisp
> (member 'b '(perhaps today is a good day to die))
   ; test fails
NIL
> (member 'a '(perhaps today is a good day to die))
   ; returns non-NIL
'(a good day to die)
```
Example: member

• We implement our own recursive version as follows.

> (defun list-member (E L)
"Test if E is a member of L."
(cond ((null L) nil)
((eq E (first L)) t)
(t (list-member E (rest L))))"
Example: member

- The correctness of the above implementation is easy to justify. The list $L$ is either constructed by $nil$ or by a call to $cons$:
  - **Case 1**: $L$ is nil.
    - $L$ is empty, and there is no way $E$ is in $L$.
  - **Case 2**: $L$ is constructed by $cons$
    - Then it has two components: (first $L$) and (rest $L$). There are two cases, either (first $L$) is $E$ itself, or it is not.
      - **Case 2.1**: $E$ equals (first $L$).
        - This means that $E$ is a member of $L$,
      - **Case 2.2**: $E$ does not equal (first $L$).
        - Then $E$ is a member of $L$ iff $E$ is a member of (rest $L$).
Example: member

- Tracing the execution of list-member, we get the following:
  > (list-member 'a '(perhaps today is a good day to die))
    0: (LIST-MEMBER A (PERHAPS TODAY IS A GOOD DAY TO DIE))
    1: (LIST-MEMBER A (TODAY IS A GOOD DAY TO DIE))
    2: (LIST-MEMBER A (IS A GOOD DAY TO DIE))
    3: (LIST-MEMBER A (A GOOD DAY TO DIE))
    3: returned T
    2: returned T
    1: returned T
    0: returned T
    T

- In the implementation of list-member, the function call \((\texttt{eq } x y)\) tests if two symbols are the same.
Example: member

> (list-member '(a b) '((a a) (a b) (a c)))
  0: (LIST-MEMBER (A B) ((A A) (A B) (A C)))
  1: (LIST-MEMBER (A B) ((A B) (A C)))
  2: (LIST-MEMBER (A B) (A C))
  3: (LIST-MEMBER (A B) NIL)
  3: returned NIL
  2: returned NIL
  1: returned NIL
  0: returned NIL
NIL

• In the example above, we would have expected a result of t.
  ▫ However, since '(a b) does not eq another copy of '(a b) (they are not the same symbol), list-member returns nil.
Example: `member`

- If we want to account for list equivalence, we could have used the LISP built-in function `equal` instead of `eq`. Common LISP defines the following set of predicates for testing equality:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code> (= x y)</code></td>
<td>True if <code>x</code> and <code>y</code> evaluate to the same number.</td>
</tr>
<tr>
<td><code>(eq x y)</code></td>
<td>True if <code>x</code> and <code>y</code> evaluate to the same symbol.</td>
</tr>
<tr>
<td><code>(eql x y)</code></td>
<td>True if <code>x</code> and <code>y</code> are either <code>=</code> or <code>eq</code>.</td>
</tr>
<tr>
<td><code>(equal x y)</code></td>
<td>True if <code>x</code> and <code>y</code> are <code>eql</code> or if they evaluate to the same list.</td>
</tr>
<tr>
<td><code>(equalp x y)</code></td>
<td>To be discussed in Tutorial 4.</td>
</tr>
</tbody>
</table>
Example: append

- LISP defines a function append that appends one list by another:

  \[ \texttt{(append ' (a b c) ' (c d e))} \]
  \[ \texttt{(A B C C D E)} \]

- We implement a recursive version of append.
- Suppose we are given two lists \( L_1 \) and \( L_2 \). \( L_1 \) is either nil or constructed by cons.
  - \textit{Case 1:} \( L_1 \) is nil.
    - Appending \( L_2 \) to \( L_1 \) simply results in \( L_2 \).
  - \textit{Case 2:} \( L_1 \) is composed of two parts: (first \( L_1 \)) and (rest \( L_1 \)).
    - If we know the result of appending \( L_2 \) to (rest \( L_1 \)), then we can take this result, insert (first \( L_1 \)) to the front, and we then have the list we want.
Example: append

- Formally, we define the following function:

```lisp
(defun list-append (L1 L2)
  "Append L1 by L2."
  (if (null L1) L2
      (cons (first L1)
            (list-append (rest L1) L2))))
```
Example: append

- An execution trace is the following:

```lisp
> (list-append '(a b c) '(c d e))
0: (LIST-APPEND (A B C) (C D E))
  1: (LIST-APPEND (B C) (C D E))
    2: (LIST-APPEND (C) (C D E))
      3: (LIST-APPEND NIL (C D E))
        3: returned (C D E)
    2: returned (C C D E)
  1: returned (B C C D E)
0: returned (A B C C D E)
(A B C C D E)
```
Using Lists as Sets

- Formally, lists are ordered sequences. They differ with sets in two ways:
  - Sets are unordered, but lists are. (a b c) and (c b a) are two different lists.
  - An element either belong to a set or it does not.
    - There is no notion of multiple occurrences. Yet, a list may contain multiple occurrences of the same element. (a b b c) and (a b c) are two different lists.
- However, one may use lists to approximate sets, although the performance of such implementation is not the greatest.
- We have already seen how we can use the built-in function member to test set membership.
- LISP also defines functions like (intersection L1 L2), (union L1 L2) and (difference L1 L2) for boolean operations on sets.
Using Lists as Sets

> (defun list-intersection (L1 L2)
  "Return a list containing elements belonging to both L1 and L2."
  (cond ((null L1) nil)
        ((member (first L1) L2)
         (cons (first L1)
               (list-intersection (rest L1) L2)))
        (t (list-intersection (rest L1) L2)))))
Using Lists as Sets

- The correctness of the implementation is easy to see. $L1$ is either an empty set (nil) or it is not:
  - **Case 1**: $L1$ is an empty set.
    Then its intersection with $L2$ is obviously empty.
  - **Case 2**: $L1$ is not empty.
    $L1$ has both a first component and a rest component. There are two cases:
    either (first $L1$) is a member of $L2$ or it is not.
      - **Case 2.1**: (first $L1$) is a member of $L2$.
        (first $L1$) belongs to both $L1$ and $L2$, and thus belong to their intersection.
        Therefore, the intersection of $L1$ and $L2$ is simply (first $L1$) plus the
        intersection of (rest $L1$) and $L2$.
      - **Case 2.2**: (first $L1$) is not a member of $L2$.
        Since (first $L1$) does not belong to $L2$, it does not belong to the intersection
        of $L1$ and $L2$. As a result, the intersection of $L1$ and $L2$ is exactly the
        intersection of (rest $L1$) and $L2$.
Using Lists as Sets

- A trace of executing the function is given below:

```
> (trace list-intersection)
(LIST-INTERSECTION)
> (list-intersection '(1 3 5 7) '(1 2 3 4))
  0: (LIST-INTERSECTION (1 3 5 7) (1 2 3 4))
    1: (LIST-INTERSECTION (3 5 7) (1 2 3 4))
      2: (LIST-INTERSECTION (5 7) (1 2 3 4))
        3: (LIST-INTERSECTION (7) (1 2 3 4))
          4: (LIST-INTERSECTION NIL (1 2 3 4))
            4: returned NIL
          3: returned NIL
        2: returned NIL
      1: returned (3)
    0: returned (1 3)
)(1 3)
```