Chapter 2

LISP Tutorial

Vasant Honavar
Artificial Intelligence Research Laboratory
Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, IA 50011. U.S.A.
http://www.cs.iastate.edu/~honavar/aigroup.html

This chapter presents a quick tutorial of LISP. It is not meant to be a substitute for courses in programming, data structures and algorithms, programming languages, or software engineering. It assumes a reasonable familiarity with key programming concepts and key programming language constructs (conditional evaluation, recursion, parameter passing, etc.). Our primary objective here is to take the reader through a quick guided tour of LISP. A reader who is proficient in designing and implementing programs in modern high-level programming languages, but may or may not have been exposed to LISP or one of its dialects (e.g., Scheme), can rapidly learn to program in LISP.

2.1 Introduction to LISP

`LISP`, short for **Lis**t **P**rocessor, was developed by John McCarthy and Steve Russel at MIT in the early 1960s. Modeled after Church's λ- calculus, it is a high-level general-purpose programming language. Because of its close tie to the precise mathematical notation of lambda calculus, programs written in `LISP` are often easy to analyze in terms of proofs of correctness. In contrast with many high-level languages such as `FORTRAN` and `C` whose design was motivated by the desire to simplify the task of writing compilers, LISP was motivated by the desire to simplify the task of programming.

The language has many distinctive features including:

- LISP itself is a LISP program
- LISP programs can be expressed as lists
- Users can easily extend the language

These features make LISP, in the words of John Foderaro, "a programmable programming language." This 'programmability' makes LISP an excellent paradigm for bottom-up, or experimental, programming. Instead of mapping a problem onto the language, programmers extend the language itself (by adding user-defined functions) so that it becomes the natural medium for expressing operations in the application domain of interest. Thus, if one is interested in writing a database program, one extends LISP to provide the primitives needed for storing, retrieving, and modifying information in a database; If one is interested in writing a scheduler, one extends LISP by defining functions that naturally express the scheduling operations of interest. As we will see, LISP contains many features that make it well-suited for writing easy-to-modify, modular, extensible, code as well as for object-oriented programming.

Many large software systems have been written in LISP. Some examples include **emacs** (a popular editor), **Autocad** (a computer-aided design program), and **CLIPS** (a widely used expert systems shell developed by NASA).

2.2 The LISP Language

2.2.1 LISP Interpreter

Interpreter reads and evaluates an S(symbolic)-expression and prints the result. S-expression is either an atom (number, symbol, string), a list, or a dotted pair.

2.2.2 The Basic Data Type – Lists

The basic data type in LISP is the list. Lists are constructed of elements called *cons cells* containing two pointers, one to the first element, or *car*, and one to the remaining elements, or *cdr*.

Many list manipulation functions are built into LISP. Here are a few simple examples:

```
> (length '(A B C))
   3

> (first '(A B C))
   A
```

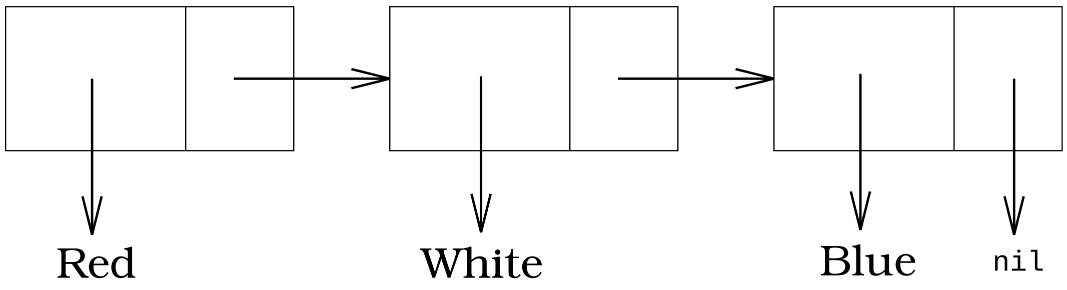

Figure 2.1: Internal Representation of Lists

```
> (rest '(A B C))
    (B C)

> (nth 1 '(A B C))
    B

> (reverse '(A B C))
    (C B A)
```

`Length` returns the number of elements in a list, `first` returns the first element, `rest` returns all elements beyond the first, and `nth` returns the nth element in a list. Notice, however, that the indexing begins with zero, not one. `First` and `rest` supplant `car` and `cdr` from previous versions of `LISP`. `LISP` also contains a number of functions that create or return modified versions of lists. These include `list`, `append`, and, as mentioned above, `cons`. Figure 2 (above) provides an example of data flow in a composite `LISP` expression. Here are some additional examples of (non-destructive) list modifiers. (Please consult the documentation for destructive counterparts of `conc` and `append`):

```
> (list 'A 'B 'C)
    (A B C)

> (append '(A B) '(C D))
    (A B C D)

> (cons 'A '(B C))
    (A B C)
```

`List` builds a list given an arbitrary number of elements, `append` concatenates two lists, and `cons` adds an element to the head of an existing list. In `LISP` it is generally true that a function that accepts two arguments will also accept more than two arguments. However, you are adviced to consult the documentation to confirm that this is indeed the case for any particular function that you may choose to use.

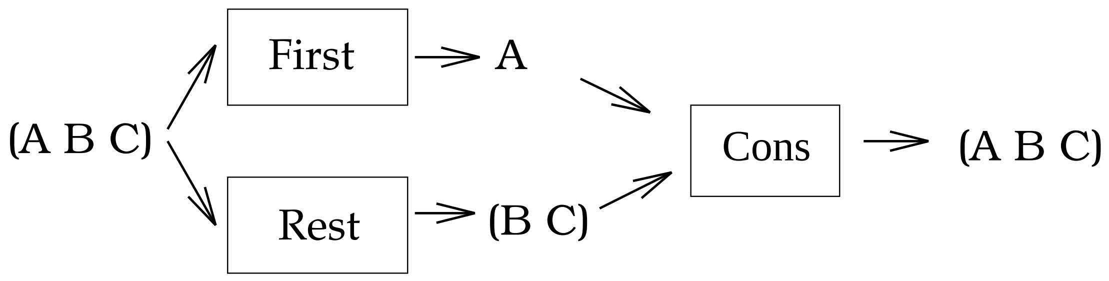

Figure 2.2: Data Flow for (cons (first '(A B C)) (rest '(A B C)))

Lists can also model sets, and `LISP` provides the necessary functionality to do so. This includes union, intersection, difference, and membership functions as well as symbol removal and equality. Some examples are:

```
> (union '(A B) '(B C))
    (A B C)

> (intersection '(A B) '(C D))
    nil

> (remove 'A '(A B A B C))
    (B B C)
```

Note that `remove` can remove any number of elements from the beginning or end of a list. Consult a `LISP` manual for details. To check for the empty list we use the predicate `null`. This returns **nil** for non-empty lists and **T** otherwise. In general, **nil** represents false, and **T** true – anything not **nil** is true. For example,

```
> (and 'JACK 'JILL)
    'JILL

> (or 'JACK 'JILL)
    'JACK
```

In the case of the `or` operator, the first operand evaluated to true and it was not necessary to evaluate the second operand. The equality of two values can be determined by one of four operators, depending upon what data type the operands are. Here is a listing of the operators and the data types that they are commonly used to compare. Please consult the documentation for details.

operator	domain
equal	all data types
eql	symbols and numbers
eq	symbols
=	numbers

It is possible to determine what data types you are operating upon by using the built-in predicates `atom`, `numberp`, `symbolp` and `listp`. These predicates enable a programmer to write functions that check the types of the arguments passed and do different things depending on the argument types.

2.2.3 Conditionals

Conditional expressions allow the program to dynamically change the flow of control during execution. The result of a Boolean expression determines which code to evaluate. Two examples of this are `if` and `cond` expressions. Their syntax is as follows:

expression	syntax
if	(if <condition> <true expr> <false expr>)
cond	(cond (<test1> <conseq1> <test2> <conseq2> . . .) ⋮ (<default>))

Here are some examples of how they might be used:

```
> (if (< 3 2) (+ 3 7) 'A)
    A

> (cond ((eq 'A 'B) (print 3)) ((> 2 7) (print 56)) (22))
    22
```

Other conditional expressions exist, such as `case`. Consult a LISP reference manual for more information.

2.2.4 Defining and Calling Functions

Functions are at the heart of good `LISP` programming. Solving a problem typically involves creating a number of functions and then composing them. Since functions are first-class objects in `LISP` (meaning they can be passed as arguments), general-purpose functions such as `mapcar` can be written. These can be applied to an infinite variety of data types, thus promoting code reuse (another hallmark of OOP). Functions are defined in `LISP` using the `defun` operator. The syntax is

```
    (defun <function name> <formal parameters> <body>)
```

As an example, let us define a function that checks if a given list is a palindrome:

```
    (defun palindrome (lst) (equal lst (reverse lst)))
```

Note that the scope of the formal parameters is limited to the function body. (`Common Lisp`, unlike some of its predecessors, uses *lexical-scoping* as default. It is possible to have *dynamic scoping* by declaring a variable to be special).

Avoid using non-local variables in functions because they might be undefined or contain incorrect values. Locals variables can be created using the `let` and `let*` expressions. An example of `let` follows:

```
>   (defun average (x y)
    (let ((sum (+ x y)))
    (/ sum 2.0)))
```

In this example `sum` is initialized to the value of $x + y$. Any number of local variables can be created this way. `Let*` behaves exactly like `let` except that the variables are initialized sequentially thereby allowing one to use a variable initialized earlier in an expression that defines the initial value of a variable that appears later in the scope of `let*`. `Let` on the other hand, initializes all the local variables within its scope in parallel. As mentioned above, creation of local variables in this fashion is preferable to using `setf`, which has side-effects.

Because functions can be passed as arguments to other functions, `LISP` provides a number of *applicative operators* to support *applicative programming*. `Apply` invokes a function and its list of arguments while `funcall` invokes a function and an arbitrary number of arguments (not contained in a list).

It is easy to define applicative operators using one of the built-in functions such as `funcall` or `apply`. To illustrate this, here are some definitions of applicative operators that use one or the other:

```
> (defun garble (fn text) (funcall fn text))

> (garble #'reverse '(BILL LIKES FRIES))
    (FRIES LIKES BILL)

> (defun garble (fn text) (apply fn (list text)))
```

Both definitions of `garble` are equivalent. It is not necessary, however, to define a function with `defun` each time you need to pass a function to another function. You can use the `lambda` expression to create anonymous functions directly within the body of an expression.

```
> (mapcar #'(lambda (x) (+ x 2)) '(1 3 5 7))
    (3 5 7 9)

> (defun make-adder (n) #'(lambda (x) (+ x n)))


> (setf add2 (make-adder 2))


> (add2 7)
    9
```

In the first example a lambda function is passed to `mapcar`. The use of `mapcar` with lambda functions obviates the need for iteration in most cases. The second example includes a function `make-adder` that returns a function. This is feasible because in `LISP` functions are 'first-class' objects, just like symbols or numbers. To illustrate, here are some examples of applicative functions, i.e., functions that take other functions as arguments:

Here are some more examples of applicative functions.

```
> (defun sq (x)
    (* x x))

> (mapcar #'sq '(1 2 3 4))
    (1 4 9 16)

> (every #'> '(10 12 6) '(5 9 2))
    T

> (reduce #'+ '(1 2 3 4))
    10

> (find-if #'oddp '(2 4 3 6))
    3
```

`Mapcar` applies a function to all elements in a list. Similarly, and`every` applies a predicate to all elements in a list. If the predicate holds true for all values in the list, `every` returns `T`, else `nil`. `Reduce` iteratively reduces a list by continually applying a function to pairs of elements. Notice that `#'` must be used to *quote* names of functions that are being passed as arguments. This prevents `LISP` from attempting to evaluate the function name. Similarly, a `'` signifies that a value (except functions) is a literal, not to be evaluated.

It was mentioned above that most built-in LISP functions that accept 2 arguments typically accept an arbitrary number of arguments whenever it makes sense to do so. Thus, if it makes sense to add two numbers, it makes sense to add 10 numbers. By adding `&optional` or `&rest` to the formal parameter list of a function, programmers can include this flexibility into their own functions. `&Optional` allows programmers to define functions that can be invoked with or without an optional argument. When the optional argument is not provided, its default value is `nil`. Alternatively, its default value can be specified in the function definition.

`&Rest` allows for an arbitrary number of arguments when a function is called. The arguments are collected into a list, which is processed by the invoked function. Defaults are handled in an analogous manner to `&optional`.

```
> (defun average (&rest args) (if (= (length args) 0) 0
    (/ (reduce #'+ args) (length args) 1.0 )))

> (average 1 2 3 4)
    2.5

> (defun philosophize (thing &optional prop)
    (list thing 'is prop))
```

```
> (philosophize 'DEATH)
    (DEATH is NIL)

> (philosophize 'DEATH 'FINAL)
    (DEATH is FINAL)

> (defun philosophize (thing &optional (prop 'temporary))
    (list thing 'is prop))

> (philosophize 'DEATH)
    (DEATH is temporary)

> (philosophize 'DEATH 'FINAL)
    (DEATH is FINAL)
```

In the case of **average** function defined above, note that the use of **&rest** causes the arguments to be collected in a list. Consequently, we use **reduce** in the body of the function to add up the elements of the list.

Keyword arguments provide a mechanism for supplying a large number of arguments to a function without needing to remember the order of the arguments in the function call. This is especially useful when a majority of the argumentstake default vaules and only a small number

For example, the function **remove** has a keyword **:count** that specifies how many elements of a certain kind to remove.

```
> (remove 'A '(A B A B A B))
    (B B B)

> (remove 'A '(A B A B A B) :count 2)
    (B B A B)
```

The first call to **remove** has no **:count** keyword and therefore removes all A's from the list. The second invocation, however, uses the keyword to limit the number of A's removed to two. The **keywordp** predicate can be used to see if a certain keyword exists for a given system. Programmers can include keywords in their function declarations by using the **&key** modifier. Here is an example:

```
> (defun make-coffee (name &key
    (size 'regular) (flavor 'amaretto)) (list size flavor 'coffee
    'for name))

> (make-coffee 'john :size 'large)
    (large amaretto coffee for john)
```

Make-coffee can take up to three arguments: the name, the size of coffee to be made, and the flavor of the coffee. However all that is strictly required is the name of the person for whom the coffee is being made. In this call the size of the coffee was included and since the flavor was not included, the flavor was set to its default value.

2.2.5 Property Lists

Long before Object-Oriented programming (OOP) languages became popular, LISP contained features that support OOP. LISP uses the notion of property lists to attach data (analogous to instance variables) and functions (analogous to methods) to atoms. This following series of expressions serves as an example of how to create a property list and retrieve its data:

```
> (setf (get 'JOE 'PARENTS '(BOB JANE)))

> (get 'JOE 'PARENTS)
    (BOB JANE)

> (remprop 'JOE 'PARENTS)
    T

> (get 'JOE 'PARENTS)
    nil
```

The first expression makes the parents property of Joe equal to Bob and Jane. The `get` returns a given property of an atom. `Remprop` removes a property from the property list. Here is an example which incorporates functions into the property list:

```
> (defun behave (animal) (funcall (get animal 'behavior)))

> (setf (get 'dog 'behavior) #'(lambda () ...))

> (behave 'dog)
```

The function `behave` invokes a given animal's behavior. We use the `setf` function and a lambda expression to attach a behavior to the atom `dog`. Calling `behave` then executes the dog's behavior. These examples show how `LISP` (even without `CLOS`, the `Common Lisp Object System`, provides the necessary machinery for writing object-oriented programs. Paul Graham's text illustrates how easy it is to implement object-oriented systems in Common `LISP`.

2.2.6 Recursion

`LISP`, because of its close relation with λ-calculus and recursive functions, is a natural language for writing recursive function definitions. Recall that a recursive function is one that calls itself to solve a problem. You have seen recursion in other languages like `C` and `PASCAL`. In `LISP`, recursion is preferred to iteration. The following example demonstrates the use of recursion in `LISP`.

```
> (defun factorial (n)
    (cond ((zerop n) 1)
    (t (* n (factorial (- n 1))))))
```

`Factorial` takes a single argument which is the number used to compute the factorial. The factorial function is defined to be $n \times (n-1) \times (n-2) \ldots \times 2 \times 1$. Another way to look at the factorial definition is as follows: $factorial(n) = n \times factorial(n-1)$ where $factorial(0) = 1$. This is a recursive definition. It uses $factorial(n-1)$ to solve for $factorial(n)$. There are two important things to remember about recursion:

- A base case is needed which has a direct solution to stop the recursion. Be sure that all possible base cases are well defined. Otherwise, the recursion will not terminate (or you will get a stack overflow).
- Work towards the base case in the recursive calls.

`Problem:` Write a recursive function **`any-oddp`** that returns `T` if any number in a list of numbers is odd and returns `NIL` otherwise.

```
> (any-oddp '(2 4 3 5))
    T


> (any-oddp '(4 6))
    NIL
```

`Solution:`

```
> (defun any-oddp (x)
    (cond ((null x) nil)
    ((oddp (first x)) T)
    (t (any-oddp (rest x)))))
```

The base case of **`any-oddp`** is the case that the list is empty. If this is true, then there are not any odd numbers. If the list is not empty, then we test the first element. If it is odd, we return `T`. Otherwise, we then need to test the rest of the list. This is accomplished by passing all but the first element of the list back to **`any-oddp`**.

`Problem:` Given the following mystery function, determine the value returned by the function when called with 4 as its argument.

```
> (defun mystery-function (n)
    (cond ((zerop n) nil)
    (t (cons 'ha (mystery-function (- n 1))))))


> (mystery-function 4)
    (ha ha ha ha)
```

Note that the base case in this recursive function is ill-defined. Thus, if the function is called with -1 (or for that matter any negative number as argument), it would not terminate.

```
> (mystery-function −1)
    Infinite Recursion
```

The base case should have tested for less than or equal to zero. This would have been a better definition of the base case. In many of the illustrative examples given in this tutorial, we have not taken sufficient care to check for the validity of arguments passed to a function and handle any resulting errors. This was done to keep the reader's attention focused on the essential features of `LISP` that were being introduced. Error-checking and robustness that comes with it, are as important in writing reliable and error-free LISP programs as they are in writing robust programs in any other language.

As yet another example of recursive function definition, consider the computation of Fibonacci numbers:
$fib(0) = 1$
$fib(1) = 1$
$fib(n) = fib(n-1) + fib(n-2)$

Given this definition it is simple to write a `LISP` function to compute the nth fibonacci number.

```
> (defun fib (n)
    (cond ((equal n 0) 1)
    ((equal n 1) 1)
    (t (+ (fib (- n 1)) (fib (- n 2))))))
```

This concludes our brief tutorial on `LISP`. In summary, we have introduced the three most useful styles of `LISP` programming: functional programming (using function composition), applicative programming (using functions that accept other functions as arguments), and recursive programming. The reader is refered to one of the standard texts on `LISP` programming and the `LISP` documentation for details of the language that were not covered in this tutorial. Readers are strongly encouraged at this time to start working with a `LISP` programming environment and write some simple programs in `LISP`. After all, the best way to learn a language is to start using the language.