A Lisp Primer for C and Java Programmers

1. Overview

To the CJ[1] programmer, Lisp appears *different*. While there are many important differences between CJ and Lisp, there are also some fundamental similarities. This primer presents the major features of Lisp, with examples that will help the CJ programmer sort out what is really new about Lisp and what is similar to or the same as CJ. The primer is a brief, but reasonably complete Lisp introduction. It covers all of Lisp's basic functions, and presents a number of examples on important Lisp programming idioms. If the reader is planning to do major programming in Lisp, a complete Lisp reference manual is in order.

Compared to CJ, the major differences in Lisp are the following:

- The syntax
- The interpretive environment
- The lack of explicit type declarations
- The functional, list-oriented style of programming.

The syntax is a profoundly *un*important difference between Lisp and CJ. However, since the syntax is the first feature a programmer sees, Lisp's "unusual" structure often leads CJ programmers to have an initially negative reaction. To avoid a prematurely negative impression, CJ programmers must be patient with Lisp's syntactic differences.

Lisp's interpretive environment is also different than most CJ programmer's are likely to be familiar with. In contrast to Lisp's syntax, its interpretive environment is typically received very well. This environment allows programs to be executed and tested very easily.

Lisp is a *weakly-typed* language. Lisp functions and variables do have types, however their types are not explicitly declared.[2] Rather, the type of an object is determined dynamically, as a program runs. This means, for example, that the same variable can hold an integer value at one point during execution, and a list value at some other point. While it is *possible* for variables to change types radically in a Lisp program, it is not typical. Generally, variables in Lisp are used in much the same way as they are in CJ. Viz., a particular variable is used to hold a particular type of data throughout the execution of a program. The difference in Lisp is that the programmer is not required to declare the type of usage explicitly.

The most significant difference between CJ and Lisp is the last of the four items listed above -- the functional, list-oriented style of Lisp. Recursive functions and lists are the "bread and butter" of Lisp programming, in much the same way that for-loops and arrays are the "bread and butter" of CJ. Also, Lisp programs are typically written as collections of many small functions. CJ programmers who use a programming style based on lengthy functions will find that this style is generally unsuited to Lisp.

The C/C++ programmer will notice the conspicuous absence of *pointers* in Lisp. In this area, the difference between Lisp and C/C++ is similar to the difference between Java and C/C++. Namely, pointers in Lisp and Java are "below the surface". Lisp's list data structure allows the programmer to define all of the structures that can be built with pointers. For the C/C++ programmer, it will take a while to adjust from the pointer mindset to the Lisp list mindset. It will probably take a little less adjustment for the Java programmer, since the below-the-surface treatment of references in Java is closer to the Lisp way of thinking than the explicit-pointer style of C/C++.

---

[1] The abbreviation "CJ" refers collectively to the family of programming languages consisting of C, C++, and Java. For the most part, these languages can be treated as a single class in comparison to Lisp. Where appropriate, distinctions are made between plain C, C++, or Java by referring to the languages individually instead of collectively as "CJ".

[2] Common Lisp does have constructs for variable declaration, but we can safely ignore these forms here.

Having considered the major differences in between CJ and Lisp, we should also point out some major similarities. These include:

- Overall program structure and scoping rules.
- Function invocation and conditional control constructs.
- An underlying similarity between Lisp lists and CJ's built-in data structures.

2. An Introductory Session with Lisp

This section of the primer presents an introductory scenario of Lisp usage. The point of the introduction is to provide the basic idea of how to use the Lisp interpretive environment. Additional scenario-style examples will be presented throughout the primer, to illustrate both the language and its environment.

All primer examples are based on GNU Common Lisp (GCL), which is the current distribution of the Austin dialect of Kyoto Common Lisp (AKCL). The GCL version is 2.4.0, compiled for UNIX. Three fonts are used in the scenarios, as well in the examples in the remainder of the primer:

Font	Usage
Bold	Information produced by GCL, such as prompting characters and the results of execution.
Plain	Information typed in by the user, such as commands to be executed.
Italic	Explanatory scenario remarks.

Here now is the introductory scenario.

```
% gcl
```

Run gcl from the UNIX prompt

```
GCL (GNU Common Lisp)  Version(2.4.0) Thu Mar 28 16:04:54 PST 2002
Licensed under GNU Library General Public License
Contains Enhancements by W. Schelter
```

```
>(+ 2 2)
```

GCL identifies itself, a prints a caret '>' as the prompt character. After the first prompt, the user types in a simple expression to be evaluated. The expression is `(+ 2 2)` *which is Lisp syntax for "2+2"*

```
4
```

GCL's response to "`(+ 2 2)`*" is to perform the addition and print the result. This is what Lisp always does at its top-level prompt. I.e., it reads what the user types in, evaluates it, prints the result, and then prints another prompt. This is called the "read-eval-print" loop.*

```
>(defun TwoPlusTwo () (+ 2 2))
```

Here the user types in a very simple function definition. The details of function declaration will appear later, but it should be clear what is being defined. Namely, the user is defining a function with no parameters that computes 2+2.

```
TWOPLUSTWO
```

GCL's response to a function definition is to print the name of the function that was just defined.

```
>(TwoPlusTwo)
```

This is a call to the function that was just defined. Lisp function calls have the general form `(function-name arg`$_1$ `... arg`$_n$`)`.

```
4
```

GCL's response to the function call is to print the value computed by the function.

```
>(defun TwoPlusXPlusY (x y) (+ 2 x y))
```

Here the user defines another function, this time with two parameters named x and y. The function computes the sum of 2 + x + y.

```
TWOPLUSXPLUSY
```

Again, GCL's response to the function definition is to print its name.

```
> (TwoPlusXPlusY 10 20)
```

This is a call to the function just defined.

```
32
```

Again, GCL's response to the function call is to print its return value.

```
>(load "avg.l")
```

Here the user decides to load a larger lisp program. Assume that the UNIX file `avg.l` *contains a lisp program to compute the average of a list of numbers.*

```
Loading avg.l
Finished loading avg.l

T
```

Lisp's response to loading a file consists of three lines, which inform the user when the loading starts and when it's finished. The last response line, consisting of the single **`T`** *is the final value of* `load`*. Even if a function does not need to return a value, it must do so in Lisp. Further, the top-level read-eval-*print *loop will always print the value of a function. Since the load function does not care about its return value, but needs one anyway, it chooses the value* `T`*, which stands for true in Lisp.*

```
>(avg '(1 2 3 4 5))
```

The user calls the just-loaded function with a list of five numbers.

```
3
```

GCL responds with the value computed by the `avg` *function.*

```
>(avg '(a b c))
```

The user attempts to call the average function with non-numeric arguments.

```
Error: C is not of type NUMBER.
      . . .
Broken at +.  Type :H for Help.
>>:q
```

GCL responds to the error by printing an appropriate message and entering the interactive debugger, signaled by the double caret prompt. The user responds immediately with the `:q` *debugger command, which quits the debugger, returning to the top-level.*

```
>(help)
```

The user calls the general help function.

```
Welcome to Kyoto Common Lisp (KCL for short).

Here are the few functions you should learn first.   ...
```

GCL responds to the `(help)` *function by typing some general help information, not all of which is shown here. Further help is provided if the user follows through the help instructions. Basically, simple help descriptions are available for all GCL functions and environment variables.*

```
>(bye)
```

The function `bye` *exits the GCL environment, back to the UNIX shell.*

This concludes the initial scenario. Additional scenario-style examples will be used throughout the primer. Table 1 summarizes the significant points of this initial scenario, in particular, how to enter and leave the environment, and how to return to the top-level when an error occurs. For details on the use of the Lisp debugger, and other advanced

Command	**Meaning**
`% gcl`	Run gcl from the UNIX shell to enter the Lisp environment.
`>`*any legal Lisp expression*	Typing a legal Lisp expression at the top-level of Lisp results in the evaluation of the expression and the printing of its value.
`>`*any erroneous Lisp expression*	Typing an erroneous Lisp expression at the top-level of Lisp results in entry to the Lisp debugger.
`>>:q`	The command `:q` exits the debugger, returning to the top-level of Lisp.
`>(load "`*unix-file*`")`	The `load` function loads a Lisp file and evaluates its contents, as if they had been typed directly at the top level.
`>(help` *symbol*`)`	The `help` function provides information about built-in lisp *symbol*s, which include functions and environment variables.
`>(bye)`	The `bye` function exits gcl, back to the UNIX shell.

Table 1: Summary of Important Lisp Environment Commands.

environment commands, the reader should consult a manual.

3. Lexical and Syntactic Structure

Lisp has a very simple lexical and syntactic structure. Basically, there are two elemental forms in lisp -- the atom and the list. An atom is one of the following:

- an identifier, comprised of one or more printable characters
- an integer or real number
- a double-quoted string
- the constants t and nil

For example, the following are legal Lisp atoms:

```
10
2.5
abc
hi-there
"hi there"
t
nil
```

Notice the dash used in `hi-there`. In Lisp, identifiers may contain most any printable character. This is possible because of the different form of expression syntax used in Lisp, as will be described shortly. The constants `t` and `nil` represent the boolean values true and false, respectively. `nil` also represents the empty list, in addition to boolean false. This overloading of `nil` is similar to the overloading of `0` in C, where `0` represents both boolean false and the null pointer.

A Lisp list is comprised of zero or more elements, enclosed in matching parentheses, where an element is an atom or (recursively) a list. For example, the following are legal lists:

```
()
(a b c)
(10 20 30 40)
(a (b 10) (c (20 30)) "x")
(+ 2 2)
```

Note that list elements need not be the same type. That is, a list is a *heterogeneous* collection of elements. Note also the last of the examples. It is a three-element list that is also an executable expression. The commonality of expressions and data in Lisp is a noteworthy feature that we will examine further in upcoming examples.

3.1. Expression and Function Call Syntax

Lisp uses a uniform prefix notation for all expressions and function calls. In CJ, and most other programming languages, built-in expressions use infix notation. The purely prefix notation of Lisp may seem awkward at first to CJ programmers. The following examples compare expressions in Lisp and CJ:

Lisp	CJ	Remarks
(+ a b)	a + b	Call the built-in addition function with operands a and b.
(f 10 20)	f(10, 20)	Call the function named f, with arguments 10 and 20.
(< (+ a b) (- c d))	(a + b) < (c - d)	Evaluate (a+b)<(c-d).

The following is the general format of a Lisp function call, including any expression that uses a built-in operator:

(function-name arg_1 *...* arg_n*)*

A function call is evaluated as follows:

1. The *function-name* is checked to see that it is bound to a function value. Such binding is accomplished by

`defun`. If the name is so bound, its function value is retrieved.

2. Each of the arg_i is evaluated.
3. After argument evaluation, the value of each arg_i is bound to the corresponding formal function parameter. This binding is accomplished using a *call-by-value* parameter discipline.[3]
4. Finally, after parameter binding is complete, the body of the function is evaluated, and the resulting value is that of the last (only) expression within the function body.

This method of function evaluation is quite similar to CJ, including the fact that call-by-value is the only standard binding discipline[4].

3.2. The Quote Function

There is a potential problem with Lisp's ultra-simple syntax. Viz., there is no syntactic difference between a list as a data object and a list as a function call. Consider the following example

```
>(defun f (x) ... )                    Define a function f
F

> (defun g (x) ...)                    Define a function g
G
```

Given these definitions, what does the following Lisp form represent?

```
(f (g 10))
```

Is it

(a) A call to function f, with the two-element list argument (g 10)?
(b) A call to function f, with an argument that is the result of a call to function g with argument 10?

The answer is (b). That is, the default meaning for a plain list form in Lisp is a function call. To obtain the alternate meaning (a) above, we must use the Lisp *quote* function (usually abbreviated as a single quote character) to indicate that we want to treat a list as a literal datum. I.e., the following form produces meaning (a) above:

```
(f '(g 10))
```

which is equivalent to the spelled-out form

```
(f (quote (g 10)))
```

The `quote` function is somewhat more general than presented above. That is, `quote` is used for more than distinguishing a list data object from a function call. Specifically, `quote` is the general means to prevent evaluation in Lisp. There are three contexts in which Lisp performs evaluation:

1. The top-level read-eval-print loop performs evaluation.
2. When a function is called, each of its arguments is evaluated.
3. An explicit call to the `eval` function performs evaluation (see Section 9.1).

Upcoming examples will further clarify the use of the `quote` function.

3.3. No `main` Function Necessary

In CJ, a distinguished function named `main` must be supplied as the top-level of program execution. In Lisp, no `main` function is necessary. Rather, the programmer simply `defun`'s and/or `loads` as many functions as desired at

---

[3] Section 11.6 describes how call-by-reference parameter passing can be achieved for lists, using destructive list operations.

[4] In C++, as opposed to plain C and Java, there is call-by-reference, but it's generally seen as an application of the C++ reference mechanism as opposed to being a specific parameter-passing discipline.

the top-level of the Lisp interpreter. To start a Lisp program, any defined function can be called.

In terms of overall program structure, Lisp is similar to plain C in that a program is defined as a collection top-level function declarations. Lisp is different from C++ and Java in that there are no class definitions; all functions are declared as top-level entities.

4. Arithmetic, Logical, and Conditional Expressions

Lisp has the typical set of arithmetic and logical functions found in most programming languages. The following table summarize them:

Function	Meaning
(+ *numbers*)	Return the sum of zero or more numbers. If no arguments are given, return 0.
(1+ *number*)	Return *number* + 1.
(- *numbers*)	Return the difference of one or more numbers, obtained by subtracting the second and subsequent argument(s) from the first argument. If one argument is given, the argument is subtracted from 0 (i.e, the numeric negation of the argument is returned).
(1- *number*)	Return *number* - 1.
(* *numbers*)	Return the product of zero or more numbers. If no arguments are given, return 1.
(/ *numbers*)	Return the quotient of one or more numbers, obtained by dividing the second and subsequent argument(s) into the first argument. If one argument is given, the argument is divided into 1 (i.e, the reciprocal of the argument is returned).

There are a host of other arithmetic, logical, and string-oriented functions in Common Lisp. Consult a Common Lisp manual for details.

4.1. Type Predicates

While there are no type declarations in Lisp, Lisp values do have types, and it is often necessary to determine the type of a value. The following table summarizes the important Lisp type predicates.

Function	Meaning
(atom *expr*)	Return `t` if *expr* is an atom, `nil` otherwise. `(atom nil)` returns `t`.
(listp *expr*)	Return `t` if *expr* is a list, `nil` otherwise. `(listp nil)` returns `t`.
(null *expr*)	Return `t` if *expr* is `nil`, `nil` otherwise.
(numberp *expr*)	Return `t` if *expr* is a number, `nil` otherwise.
(stringp *expr*)	Return `t` if *expr* is a string, `nil` otherwise.
(functionp *expr*)	Return `t` if *expr* is a function (defined with defun), `nil` otherwise.

There are other type predicates in Common Lisp. Consult a manual for details.

4.2. The `cond` Conditional Control Construct

Lisp has a flexible conditional expression called `cond`. It has features of both if-then-else and the switch statement in CJ. The general form of `cond` is the following:

(cond ((*test-expr*$_1$) *expr*$_1$ ... *expr*$_j$)
. . .
((*test-expr*$_n$) *expr*$_1$ ... *expr*$_k$)

The evaluation of cond proceeds as follows:

1. Evaluate each *test-expr* in turn.
2. When the first non-nil *test-expr* is found, the corresponding expression sequence is evaluated.
3. The value of `cond` is the value of the last $expr_i$ evaluated.
4. If none of the *test-expr*s is non-nil, then the value of the entire `cond` is nil.

Upcoming examples illustrate practical usages of `cond`.

4.3. Equality Functions

Lisp has a different type of equality for each type of atomic data, and two forms of equality for lists. The following table summarizes them

Function	Meaning
=	numeric equality
string=	string equality
equal	general expression equality (deep equality)
eq	same-object equality (shallow equality)

The difference between `eq` and `equal` is a subtle but important one. Viz., two lists are `eq` if they are bound to the same object, whereas they are `equal` if they have the same structure. This concept will be reconsidered after we have seen more about lists and the concept of binding.

5. Function Definitions

The introductory scenario illustrated two simple function declarations. The general format of function declaration in Lisp is:

(defun *function-name* (*formal-parameters*) $expr_1$ ... $expr_n$)

As an initial example, here is a side-by-side comparison of a simple function declaration in Lisp and CJ:

Lisp:	CJ:

`(defun f (x y)`
`    (plus x y)`
`)`
	`int f(int x,y) {`
	`    return x + y`
	`}`

As has been noted, there are no explicit type declarations in Lisp. Hence, where CJ declares the return type of the function and the types of the formal parameters, Lisp does not. Evidently, the parameters must be numeric (or a least addable), since the body of the function adds them together. However, the Lisp translator does not enforce any static type requirements on the formal parameters. Any addition errors will be caught at runtime.

Notice the lack of a return statement in the Lisp function definition. This owes to Lisp being an expression language -- i.e., every construct returns a value. In the case of a function definition, the value that the function returns is whatever value its expression body computes. No explicit "`return`" is necessary. This typically takes some getting used to for CJ programmers.

Further function definitions appear in forthcoming examples, wherein additional observations are made.

6. Lists and List Operations

As described earlier, the list is the basic data structure in Lisp. Common Lisp does support other structures, including arrays, sequences, and hash tables. These structures provide more efficiency than lists, but no fundamentally new expressive power over lists. Substantial Lisp applications can be and have been implemented using no data structure other than the list.

This primer describes the important list operations, and shows how lists can be easily used to represent the major built-in data structures of CJ -- arrays, structs, and pointer-based structures. The reader should consult a Lisp manual for discussion of the other Lisp data structures.

6.1. The Three Basic List Operations

There are only three fundamental list operations, from which all others can be derived:

Operation	Meaning
car	return the first element of a list
cdr	return everything except the first element of a list
cons	construct a new list, given an atom and another list

The following fundamental relationships exist between the three list primitives:

- (car (cons X Y)) = X
- (cdr (cons X Y)) = Y

The initial CJ-programmer reaction to these primitives may well be "Is that all?". In a formal sense, the answer is "Yes". That is, any list operation, including operations on complicated structures, can be built upon these three primitives. The practical answer to the question is of course "No". While it is theoretically possible to derive all list operations from these primitives, it would be silly for regular Lisp programmers to do so. Hence, Common Lisp provides a generous library of higher-level list functions, the important ones of which are described in the primer.

Despite the existence of higher-level functions, the primitive operations are not simply relics. They are still regularly used in even sophisticated programming. In particular, there is a fundamental idiom in Lisp called "tail recursion" that uses a combination of car, cdr, and recursion to iterate through a list. This tail recursion idiom is comparable to array iteration in CJ, using a for- or while-loop. Consider the following initial example:

```
(defun PrintListElems (l)
    (cond ( (not (null l))
            (print (car l)) (PrintListElems(cdr l))
          )
    )
)
```

This Lisp function is semantically comparable to the following CJ function:

```
void PrintArrayElems(int a[], int n) {
    int i;
    for (i=0; i<n; i++)
        printf("9d", a[i]);[5]
}
```

Using PrintListElems as an example, the tail recursion idiom goes like this:

1. Start by checking if the list being processed is nil.
2. If so, do nothing and return from the function. If any recursive calls have been made to this point, this "return-on-nil" check starts the recursive unwinding process, in effect terminating the list iteration.
3. If the list is non-nil, then do what needs to be done to the first element of the list (the `car`), and then recurse on the rest of the list (the `cdr`, a.k.a. the *tail*). In this case, what "needs to be done" is just printing, but in general it could be any processing.

---

[5] The `printf` function is C's version of `System.out.println` in Java. The first argument to `printf` is a formating string, the details of which are not important here.

A few other observations can be made about the CJ `PrintArrayElems` function as compared to its Lisp counterpart, `PrintListElems`. First, it is clear that the two functions are not semantically identical, in that the CJ function uses iteration to traverse the array, where the Lisp version uses recursion to traverse the list. While it is possible to use iteration in Lisp, and in turn possible to use tail recursion in CJ, each language has its most typically used idiomatic forms. In Lisp, recursive traversal of lists is the idiom of choice. In CJ, iterative traversal of arrays (and other linear collections) is more typically used than recursive traversal.

A second observation regards the need for the additional integer parameter to the CJ function. Since there is no automatic way in C/C++[6] to test for the end of an array, it is typical for array processing functions to be sent both an array, and the number of array elements to be processed. In the tail-recursive Lisp version, reaching the end of the list is a natural occurrence, and no additional list-length parameter is needed.

Finally, the CJ version of this example only works on arrays of integers, whereas the Lisp version works on lists of any type. The reason is that Lisp's weak typing provides a high degree of function polymorphism that is not available in plain C, and available in a more limited inheritance-based form in C++ and Java. A full discussion of polymorphic functions is beyond the scope of this primer, but its benefits are well known to Java and C++ programmers.

6.2. cXr forms

Programming with lists frequently requires composition of basic list operations. For example, `(car (cdr L))` is a commonly-used composition. For notational convenience, Lisp provides short-hand compositions, of the form

`c`*X*`r`

where the *X* can be replaced by two, three, or four *a*'s and/or *d*'s. For example, `(cadr L)` is the short-hand for `(car (cdr L))`.

6.3. Other Useful List Operations

The following table summarizes particularly useful built-in list operations.

Function	Meaning
(append *lists*)	Return the concatenation of zero or more lists. Similar to `cons`, but `cons` requires exactly two arguments.
(list *elements*)	Return the concatenation of zero or more items, where items can include atoms. Similar to `append`, but `append` requires all but the last argument to be a list.
(member *element list*)	Return the sublist of *list* beginning with the first element of *list* `eq` to *element*.
(length *list*)	Return the integer length of *list*. Note that `length` works on the uppermost level of a list, and does not recursively count the elements of nested lists. E.g., `(length '( (a b c) (d ( e f)) ))` = 2.
(reverse *list*)	Return a list consisting of the elements in reverse order of the given list.
(nth *n list*)	Return the *n*th element of *list*, with elements numbered from 0 (as in CJ arrays).
(nthcdr *n list*)	Return the result of `cdr`ing down *list n* times.
(assoc *key alist*)	Return the first pair P in *alist* such (car P) = *key*. The general form of an alist is *((* key_1 $value_1$*) ... (* key_n $value_n$*))*. See section 7.2 for further discussion of alists.
(sort *list*)	Return the *list* sorted.

[6] Java, of course, provides the built-in array length operator.

6.4. Dot Notation and the Internal Representation of Lists

The internal representation of lists within the Lisp translator is a linked binary tree. For example, the list `(a b c d)` has the internal representation shown in Figure 1. In the figure, notice that the right pointer in the rightmost element points to `nil`. By convention, this is the standard internal representation of a list, and it corresponds to how a CJ programmer might think of implementing a Lisp-style list. Among other things, the rightmost nil pointer allows any list-based function to find the end of a list reliably. (This is akin to how strings in C/C++ are always terminated with a null character.)

The rationale for the binary representation is that it is a generally efficient low-level representation on most standard computer architectures. In addition, the binary-tree representation has a long history in Lisp, and it has become the internal standard.

There is an important question related to this internal representation. Viz., is it possible to create a binary tree that violates the convention of the rightmost pointer being nil? The answer is yes, and it is done quite easily. Specifically, we use the standard `cons` function. For example, Figure 1 is the internal representation of `(cons 'a 'b)`.

The next question is, what is the *external* representation of such a value. I.e., if `(cons 'a 'b)` is entered at the top-level of Lisp, what is printed as the value? The answer is `(a . b)`. That is, rather than a space separating the elements, a dot is used. Hence, the name for this form of item is a *dotted pair*. In general, the only way that a dotted pair can be constructed is if we provide an atomic value as the second argument to `cons`, or to some function that returns the value of a `cons`.

So, at the internal representation level, what cons actually does is allocate a single binary-tree cell that can be part of a linked tree structure. For this reason, these cells are called *cons cells* in Lisp terminology, or just *cons*'s for short.

Most of the time, we can get along just fine in Lisp without ever worrying about the dotted-pair level of list representation. Occasionally, we may unintentionally create a dotted pair instead of a list, in which case it is useful to know what Lisp is doing. For example, suppose we want to append an atom onto the end of a list. If we use the cons operation like this:

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

we obtain the potentially unexpected result

```
((A B C) . D)
```

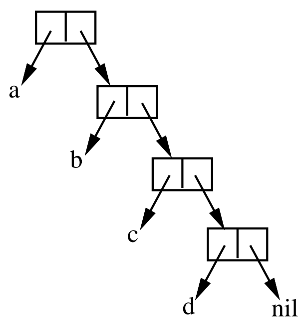

Figure 1: Internal representation of the list`(a b c d)`.

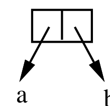

Figure 2: Internal representatino of`(cons 'a 'b)`.

The reason, again, relates to the precise meaning of `cons`. Viz., `(cons x y)` produces the value `(x . y)`. If `y` is a list, the value is `(x . (y))`, which is written in normal list notation as `(x y)`, and hence the dot is not a visible part of the value. On the other hand, if `y` is an atom in `(cons x y)`, then there is no high-level list notation for the result, and Lisp must use dot notation to accurately describe the value.

In practice, accidental and even intentional creation of dotted pairs is typically rare in Lisp. In the circumstances where such creations do arise, it is necessary to understand dot notation in order to be clear about the values that Lisp is manipulating.

There is another area in which an understanding of low-level list structure is necessary. This is in the use of the so-called *destructive* list operations, which are covered in Section 11.5 below.

7. Building CJ-Like Data Structures with Lists

This section of the primer demonstrates how Lisp's simple list structure can be used to represent all of the common built-in data structures found in modern programming languages. Specifically, we show how to use lists to represent CJ arrays, structs, linked lists, and trees.

7.1. Arrays

Given the built-in `nth` function, CJ arrays are trivially represented as lists. While `nth` is a built-in library function, it is instructive to consider its implementation in terms of the list primitives. Here it is:

```
(defun my-nth (n l)
    (cond ( (< n 0) nil )
          ( (eq n 0) (car l) )
          ( t (my-nth (- n 1) (cdr l)) )
    )
)
```

The name has been changed, so as not to replace the built-in Lisp library function named `nth`. The algorithm uses a tail recursive idiom to move over the first *n-1* `cars`, returning the *nth* if it exists, `nil` otherwise. Note the last alternative of the `cond` in the body of `my-nth`. Using `t` as the test-expression is the standard way to make the last element of a `cond` an "otherwise" (a.k.a. "else") clause.

7.2. Structs[7]

A CJ-style struct can be represented by a list with the following general format:

((field-name$_1$ value$_1$) ... (field-name$_1$ value$_1$))

That is, we use a list of pairs, where each pair represents a struct field.

To make such structures useful, we need a couple functions to access and modify them, called getfield and setfield. Here are their implementations:

```
; Return the first pair in a struct with the given field-name,
; or nil if there is no such field-name.
;
(defun getfield (field-name struct)
   (cond
        ( (eq struct nil)  nil )
        ( (eq field-name (caar struct)) (car struct) )
        ( t (getfield field-name (cdr struct)) )
    )
```

[7] For the Java-only programmer, consider a struct to be a class with all public data fields and no methods.

```
)

; Change the value of the first pair with the given field-name
; to the given value.  No effect if no such field-name.  Return
; the changed struct, without affecting the given struct.
;
(defun setfield (field-name value struct)
    (cond ( (null struct) nil )
          ( (eq field-name (caar struct) )
            (cons (cons (caar struct) (list value)) (cdr struct)) )
          ( t (cons (car struct) (setfield field-name value (cdr struct))) )
    )
)
```

The `getfield` function uses a straightforward tail recursion, quite similar to the `my-nth` function we saw earlier. The `setfield` function is worthy of some additional study, since it is representative of an important Lisp idiom -- non-destructive list modification. The basic strategy of the `setfield` function is as follows.

a. First, make a copy of all fields up to, but not including the field to be changed.

b. Then make a new field, consisting of the old field name, and the new value.

c. Then copy the rest of the fields following the one that was changed.

d. Assemble the collection of copied and new fields into a list, and return the list as the value of the function.

An important property of the list operations we have seen thus far is that they are *non-destructive*. That is, these operations never change the value of any list parameter. Rather, they only access existing lists, or create new lists. The `setfield` function, as written above, is similarly non-destructive. Rather than change its list (i.e., struct) parameter, it constructs a new list, according to the strategy outlined above.

Lisp does have *destructive* list operations, that do change list parameters. These operations are discussed in Section 11.5 below.

While this development of CJ-like structs has been instructive, it is in fact largely unnecessary. As might be expected, Lisp has a number of built-in functions to operate on struct-like data structures. The standard terminology used in Lisp for such structures is the *alist*, which stands for *association list*. The general form of an alist is:

$((key_1\ value_1) \ldots (key_n\ value_n))$

That is, alists and our CJ-like structs have the same structure. The built-in `assoc` function performs precisely the same function as `getfield`. A destructive, i.e, in-place, form of `setfield` is considered in Section 12.6.

7.3. Linked Lists and Trees

CJ programmers generally earn their keep by building pointer-based structures[8]. In Lisp, any pointer-based structure can be represented using a list, owing fundamentally to the fact that lists may recursively contain other lists, to an arbitrary level of recursion.

This primer will not present a formal and exhaustive comparison of recursive lists and pointer-based structures. Rather, we will look at two representative examples of pointer-based structures, by which the reader should be convinced that lists can at least go quite a way towards handling pointer-based structures.

[8] *reference-based* structures in Java terminology

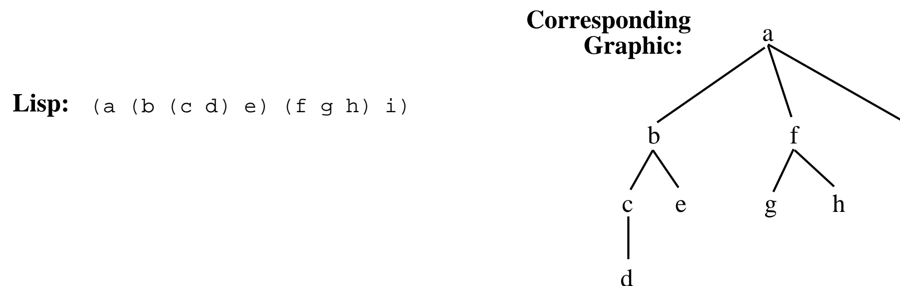

Figure 3: Sample Lisp Representationof an N-Ary Tree.

7.3.1. Linked Lists

CJ-style singly-linked lists are trivial in Lisp. Viz., they are just plain lists. And, if we consider the underlying dot-notation level, Lisp lists are in fact implemented using pointers, in a way that pointers are typically used to implement lists in CJ.

7.3.2. N-Ary Trees

An n-ary tree can be represented using a recursive list of the following form:

$(root\ subtree_1 \ldots subtree_n)$

For example, Figure 3 shows an example n-ary tree, and its corresponding graphic representation. It is left as an exercise for the reader to design some basic tree access and manipulation functions. Since tree data structures are one place in CJ where recursion is used regularly, the Lisp implementation of tree functions will have some noticeable similarity to tree implementations in CJ, except in Lisp things are a bit simpler.

While it is possible to generalize the n-ary tree structure to cover a wide range of pointer-based structures, there is one important class that cannot be covered easily -- cyclic structures. While it is possible to represent cyclic structures without non-destructive list processing, it is rather cumbersome to do so. Hence, the complete coverage of pointer-based structures relies on the list operations discussed in Section 11.5 below.

8. A Multi-Function List-Processing Example

As an illustration of typical list programming style, a multi-function example is shown if Figure 4. The collection of functions performs a merge sort on a list that contains orderable elements, where orderable means they can be compared with the < operator. The lines of the program are numbered for reference in the discussion that follows.

Line 1 defines the topmost function of the collection, `merge-sort`, that takes one list argument. Lines 2 through 5 are the recursive implementation of `merge-sort`. Line 3 is a typical end-of-recursion check for an empty list. Line 4 is the base case, where we are trivially sorting a list of length 1, which means we return the list itself. Line 5 is the crux of the merge sort algorithm. Here a two-way recursion is used, in contrast to the one-way tail recursion we have seen in previous list-processing examples. The logic of line 5 should read reasonably easily. Viz., the recursive task is to subdivide the input list into two halves, sort each, and them merge them back together using the auxiliary function `merge-lists`.

`merge-lists` is also a fundamental part of the algorithm. It is a tail-recursive function that merges two sorted lists by taking each element of the first list and inserting it in its proper ordered position in the second list. The tail recursion, on line 11, is used to iterate through the elements of the first list. As the iteration takes place, the auxiliary function `ordered-insert` is called.

`ordered-insert` is another simple tail-recursive function. Its task is to insert an element in the its proper ordinal position within a list. Its processing is similar to the `setfield` function discussed earlier. Viz., `ordered-`

```
 1   (defun merge-sort (l)
 2       (cond
 3           ((null l) nil)
 4           ((eq (length l) 1) l)
 5           (t (merge-lists (merge-sort (1st-half l)) (merge-sort (2nd-half l))))
 6       )
 7   )
 8
 9   (defun merge-lists (l1 l2)
10       (cond ((null l1) l2)
11             (t (merge-lists (cdr l1) (ordered-insert (car l1) l2)))
12       )
13   )
14
15   (defun ordered-insert (i l)
16       (cond ((null l) (list i))
17             ((< i (car l)) (cons i l))
18             (t (cons (car l) (ordered-insert i (cdr l))))
19       )
20   )
21
22   (defun 1st-half (l)
23       (1st-half1 l (floor (/ (length l) 2)))
24   )
25
26   (defun 1st-half1 (l n)
27       (cond ((eq n 0) '())
28             (t (cons (car l) (1st-half1 (cdr l) (- n 1))))
29       )
30   )
31
32   (defun 2nd-half (l)
33       (nthcdr (floor (/ (length l) 2)) l)
34   )
35
36
```

Figure 4: Collection of Functions to Perform Merge Sort of a List.

`insert` takes in a single element and a sorted list as inputs. For its return value, it constructs a new list composed of all elements less than the input element, `cons`'d with the input element, `cons`'d with the rest of the input list.

Lines 22 through 30 illustrate a typical *function/function1* idiom. This form of function pairing is used in cases where a single property is needed to help with further recursive processing. For example, in this case it is easier to to compute the first half of a list if we know how many elements there are in it. Hence, the `1st-half` function on lines 22 through 23 computes how many elements are in the half, and passes the work on to `1st-half1`. `1st-half1` uses simple recursion to cons up the the first *n* elements of its given list, where *n* is provided by `1st-half`.

Finally, the one-liner `2nd-half` uses the library `nthcdr` function directly (line 33).

9. Basic Input and Output

The following table summarizes the basic Lisp I/O functions:

Function	Meaning
(read [*stream*])	Read from the given *stream*, if specified, or from stdin otherwise.
(print *expr* [*stream*])	Print a newline followed by the value of *expr* to the given *stream*, if specified, or to stdout otherwise.
(princ *expr* [*stream*])	Like print, but without the leading newline.
(terpri [*stream*])	Print a single newline to *stream* or stdout; (highly anachronistic name).
(pprint *expr* [*stream*])	"Pretty" print the value of *expr* to the give *stream*; "pretty" means format the parenthesized structure of a large value in human-readable form.
(open *filename*)	Return a stream (usable in functions above) open on the given *filename*, which is specified as a string.

10. Programs as Data

As noted earlier, Lisp lists and function calls are syntactically identical. For example, the expression "`(+ 2 2)`" can be regarded in two equally valid ways: a three-element data list, containing the atoms '+', 2, and 2; a function call that applies the operator '+' to the arguments 2 and 2. In the first form, the list is "data", whereas in the second form it is a "program".

Not only do programs and data look alike in Lisp, they can be manipulated interchangeably. To this end, Lisp provides functions that evaluate a list data object as a program.

A limited form of the programs-as-data concept is available in C/C++ through the use of function pointers, and in Java through the use of reflection. In C/C++, the value of a function can be assigned to a variable or passed as an actual parameter to another function. In Java, functions (a.k.a., methods) can be treated as first-class objects using the facilities of the `java.lang.reflect` package. If we think of a function pointer (method value) as a "program", then assigning it to a variable is treating it as "data".

Lisp takes the concept of programs-as-data to its logical conclusion. In Lisp, *any* expression can be treated equally well as program *or* data. Lisp provides two functions to explicitly evaluate a data object -- `eval` and `apply`. These functions take data objects that look like programs and evaluate them as such.

10.1. Eval

The callable `eval` function is exactly the same as the *eval* in Lisp's top-level *read-eval-print* loop. `eval` is an extremely powerful function in Lisp, since it effectively puts the power of the full Lisp translator at the disposal of the programmer. We can construct any legal Lisp expression, and then give it to eval to be executed. If we had the equivalent power in CJ, we would be able to build a piece of CJ program at runtime, call the CJ compiler (as a function) to compile it, and then execute the result of the compilation, all while the original program is still running.

As an illustration of the power of eval, suppose we would like to build a simple desk calculator program, where the user could input the name of a calculator operation (such as + or -), followed by the values to be operated on. Here is a function `calc` that illustrates how eval could be used in such a calculator program.

```
>(defun calc ()
    (print "Input the name of an operation and its two operands:")(terpri)
    (eval (list (read) (read) (read))))

>(calc)
Input the name of an operation and its two operands:
+                                        User input for first read
2                                        User input for second read
2                                        User input for third read

4                                        Result of the call to calc
```

What happens here is that a list is constructed out of the three inputs that are read in. Since this list is the legal expression `(+ 2 2)`, it can be given to eval for evaluation.

The reader should consider how the function calc could be written in CJ. It is considerably longer than three lines.

10.2. Apply

When the full power of `eval` is not needed, there is a "junior" function named `apply`. `Apply` is slightly less powerful than `eval`, in that `apply` takes the name of a function and a list of its arguments, and applies the function to them. E.g.,

```
(apply '+ '(2 2))
```

produces 4.

`apply` provides a capability similar to that available in C/C++ with invocation through function pointers, and in Java via `Method.invoke`. However, Lisp's `apply` is a good deal more powerful than function invocation in CJ, since built-in as well as user-defined functions can be given to `apply` in the first argument. In CJ, only user-defined function names or function pointer (method) variables can be applied to arguments.

11. Scoping with Let

Common Lisp provides two scoping constructs similar to the curly-brace scoping block in CJ. The constructs are `let` and `prog`. The `let` block is described here and `prog` is defined in the next section on imperative features. The `let` expression has the following general format:

(`let` ([[(]var_1 [val_1)] ... [(]var_n [val_n)]]) $expr_1$ ... $expr_n$)

The square brackets indicate optional constructs, so that the elements of the first let subexpression can be either single *var*s or (*var val*) pairs.

The evaluation of `let` proceeds as follows:

1. Each of the val_i is evaluated and bound to the corresponding var_i.
2. These bindings are done "in parallel", not in left-to-right sequential order.
3. If any val_i is missing, then the var_i is bound to nil.
4. After the bindings are are completed, the $expr_i$ are evaluated sequentially.
5. The value of the let is the value of $expr_n$.

As an example, here is a side-bye-side comparison of a Lisp `let` expression and a comparable CJ block:

Lisp:	**CJ:**

<pre>(let
    (i
      (j 10)
      (k 20))
 	
    expr₁
       ...
    exprₙ
)</pre>
	<pre>{
	    int i;
	    int j = 10;
	    int k = 20;
	
	    stmt₁
	       ...
	    stmtₙ
	}</pre>

The placement of `let` in Lisp is essentially the same as the placement of curly-brace blocks in CJ. That is, `let` is most typically used as a function body, but is not limited to this usage. A `let` may appear anywhere an expression is legal in Lisp, in the same way that a block may appear anywhere that a statement is legal in CJ.

As explained above, the binding of `let` variables is carried out in parallel. In particular, this means that no *val* expression can use the result of a preceding binding in the same `let`. For example, if variable `x` has no previous global binding, then the following `let` is erroneous, since the binding of *x* is not available for use in binding `y`:

```
(let ( (x 10) (y x) )  ...  )
```

In some circumstances, it might well be useful to have `let` evaluate the bindings sequentially, rather than in parallel, so that this example would in fact work (i.e., `y` would be bound to 10). This behavior is available in the `let*` expression. Specifically, `let*` evaluates the same as `let` except that var/val bindings are performed sequentially rather than in parallel.

In the terminology of functional programming, the `let` expression is a *single assignment* construct. Without the use of assignment statements within the body of the `let` expression, the `let` variables are only bound (i.e., assigned to) a single time, at the beginning of the `let`. This means that the `let` construct itself is side-effect free, in contrast to the imperative features of Lisp we are about to discuss.

12. Imperative Features

The Lisp features presented thus far provide all the power that is necessary to write serious Lisp programs. These features comprise the what is called the *functional* or *pure* subset of lisp. A detailed discussion on the difference between functional and imperative languages is beyond the scope of this primer. What can be said is that there are a number of compelling advantages to using only, or primarily, Lisp's functional constructs.

Since CJ is an imperative language, the imperative features of Lisp make it a much more "CJ-like" language. In fact, using these imperative features, it is possible to write programs in a style very much like CJ. However, such a style may be counter-productive in many cases. For example, it is frequently more natural to use recursion in Lisp to iterate through the elements of a list, rather than the more CJ-like do loop described below. In any case, it is a mistake for the CJ programmer to cling tightly to the CJ-like imperative features of Lisp, since much of the power and elegance of Lisp will be lost in doing so.

12.1. Assignment Statements

The *setq* function is the Lisp assignment statement. For example, the following Lisp expression

```
(setq x (+ 2 2))
```

is equivalent to the CJ assignment statement:

```
x = 2 + 2;
```

Since variables are not declared in Lisp, the first assignment to a variable serves to declare it, as well as initialize it. In Lisp terminology, the assignment of a value to a variable is called *binding*. Hence, `setq` is said to *bind* the value of its second argument to the variable in its first argument.

A more general form of assignment in Common Lisp is available through the `setf` function. It is more general since `setq` must have an atom as its first argument, whereas the first argument to `setf` can be an arbitrary l-value. That is, `setf` can be used to assign to any place in a cons cell. Consider the following example:

```
>(setq x '(a b c))
(A B C)

>(setf (cadr x) 10)
10

>X
(A 10 C)
```

As can be seen in the resulting value of x, the `setf` functions results in a permanent (a.k.a., destructive) change to the `cadr` of x. To accomplish change, `setf` has accessed the pointer-level representation of the value of x, and changed (destructively) the second element. Given this behavior, `setf` belongs in the category of *destructive* list operations, defined below.

12.2. Details of Scope and Binding

There are no explicit declarations for global variables is Lisp. Whenever a variable is bound at the top-level of the Lisp interpreter, it becomes a global variable. Consider the following example.

```
>(defun f (x y) (+ x y z))
F

>(setq x 1)                          Bind x at top-level, making it global.
1

>(setq y 2)                          y is global
2

>(setq z 3)                          z is global
3

>(f 10 20)
33
```

In function `f`, `x` and `y` are the formal parameters. Just as in CJ, all function parameters have *local* scope. Hence, as in CJ, the references to `x` and `y` within the body of function `f` are to the parameters, not to the globally bound `x` and `y`. Also as in CJ, references within a function to non-locals are references to globally-bound variables[9]. Thus, when function `f` is called in the above example, `x` is locally bound to 10, `y` is locally bound to 20, and `z` is globally bound to 3.

In Lisp terminology, the variable `z` is said to be *free* in function `f`. Because there are no explicit declarations in Lisp, there is a subtle but important difference between a free variable in a Lisp and a global variable in CJ. Namely, in Lisp, a free variable need not exist when a function that references it is declared. For example, the function `f` above is declared before its free variable `z` is ever bound. Free variables must be bound before a referring function is *evaluated*. If not, then an "unbound variable" error occurs.

12.3. Prog

The Lisp `prog` expression is a more imperative version of the `let` expression described earlier. Using `prog` in Lisp, it is possible to write very CJ-like sequences of expressions, including the use of `goto`s as in C/C++. There are no actual "statements" in Lisp, only expressions, the difference being that an expression always returns a value whereas a statement does not. Given the imperative nature of expressions like `setq` and `go`, they can be used effectively like statements within a `prog`. The general format of a Lisp `prog` is the following:

(prog ((var_1 val_1)) ... (var_n val_n) $expr_1$ ... $expr_k$)

The first argument to `prog` is a possibly empty list of local variable bindings. Before the `prog` is executed, each val_i is bound to the corresponding var_i. The scope of these variables is entirely local to the prog. Once the locals are bound, each $expr_i$ is evaluated.

The CJ analog of Lisp's prog is the compound statement. Here is a side-by-side example that illustrates the similarity between Lisp prog and CJ compound block:

[9] Java programmers may delude themselves that Java does not have truly global variables in the raw sense of C or C++; such is indeed a delusion, given that a static public data field is just a "dot away" from being as global as it gets.

Lisp:	**CJ:**

```
(prog                           {
    ((i 10)                         int i = 10;
     (j 20.5)                       float j = 20.5;
     (k "xyz"))                     char* k = "xyz"

  (setq i (+ i 1))                  i = i + 1;
  (setq j (1+ j))                   j += 1;
  (print (+ i j))                   printf("20d", i + j);
)                               }
```

The default value of a `prog` expression is `nil`. It is possible to have a prog return a value other than `nil` using the Lisp `return` function. It is important for CJ programmers not to confuse Lisp's return with CJ's. The two are similar, but not identical. Specifically, a `return` in Lisp can only appear within an enclosing `prog`, not in a general function body. When Lisp evaluates `(return` *expr*`)`, it terminates the `prog` within which it is contained, and the entire `prog` returns the value of the `return` *expr*. Lisp `return`'s will look much like CJ return's, if a function body consists of just a single `prog`. However, the Lisp `return` returns from a `prog` *NOT* from a function.

The final `prog`-related function is `go`, which acts like a statement in C/C++. A simple example easily illustrates the use of go:

```
(defun read-eval-print-loop ()
    (prog ()
        loop
            (princ ">")
            (print (eval (read)))
            (terpri)
            (go loop)
    )
)
```

This is *the* read-eval-print loop for Lisp's top-level. Nasty, but slick.

12.4. Iterative Control Constructs

There is a very CJ-like iteration function in the imperative Lisp trick bag. It is called `do`, and it is a lot like CJ's for loop. The general format of `do` is:

(do ((var_1 val_1 rep_1) ... (var_n val_n rep_n)) *exit-clause* $expr_1$ ... $expr_k$)

Each (var_i val_i rep_i) triple is similar to the semi-colon separated control expression of a CJ for loop. When the `do` is initially started, each var_i is initialized to the corresponding val_i.

After the var_i initialization, the *exit-clause* is examined. It is of the general form:

([*test* [$exit_1$ ... $exit_m$]])

If the entire exit clause is `nil`, then the `do` terminates immediately. Otherwise, the *test* expression is evaluated, as an *until* test. That is, if *test* is non-nil, then the `do` is ready to terminate. Just prior to termination, the $exit_p$ expressions are executed in order, and the value of the last one of these is the return value of the `do`.

If the until *test* is `nil`, then the body of the `do` is executed, where the body consists of the $expr_j$. When the body completes one cycle of execution, each var_i is rebound to the corresponding rep_i, and the test cycle of the preceding paragraph is repeated.

Here is a side-by-side comparison of a Lisp `do` and a comparable CJ `for` loop:

Lisp:	**CJ:**
```(do```	```for```

```
Lisp:                          CJ:
(do                            for
    ((i 1                          (i = 1;
     (1+ i)))                      i <= 10;
    ((> i 10))                     i++) {
  (print i)                      printf("\n%d", i);
)                              }
```

Note that the use of the parentheses in the Lisp `do` may seem a bit tricky at first. It is easy to forget the extra set of parentheses around the binding triples and exit clause. For example, the following is a reasonable looking but erroneous version of the preceding example:

```
(do (i 1 (1+ i)) (> i 10) (print i))
```

missing parens *missing parens*

12.5. Destructive List Operations

Several sections above alluded to the so-called "destructive" list operations. Simply put, these operations provide direct access to the internal pointer-based implementation of Lisp lists. The two primitive list destructors are `rplaca` and `rplacd`, which stand for "replace car" and "replace cdr", respectively. Here is a summary of these two, as well as some higher-level destructive operations:

Function	**Meaning**
(rplaca *cons-cell expr*)	Destructively replace the car of the *cons-cell* with *expr* and return the result.
(rplacd *cons-cell expr*)	Destructively replace the cdr of the *cons-cell* with *expr* and return he result.
(nconc *lists*)	Destructively change the last cons-cell of each of the given lists to point to the next of the given lists. `nconc` is the destructive version of `append`.
(setf *cons-cell expr*)	Destructively change the contents of the given *cons-cell* to the given *expr*, and return the value of *expr*.

Here is a telling scenario of what can be done with these destructive operations:

```
>(setq x-safe '(a b c))
(A B C)

>(setq y-safe x-safe)
(A B C)

>(setq x-safe (cons 'x (cdr x-safe)))
(X B C)

>y-safe                         Non-destructive change to x-safe does not affect y-safe.
(A B C)

>(setq x-unsafe '(a b c))
(A B C)

>(setq y-unsafe x-unsafe)
(A B C)

>(rplaca x-unsafe 'x)
(X B C)

>y-unsafe                       Destructive change to x-unsafe does effect y-unsafe
(X B C)
```

```
>(setf (cadr y-unsafe) 'y)                  Another destructive change
Y

>x-unsafe
(X Y C)

>y-unsafe
(X Y C)
```

12.6. Call-by-Reference Parameters

Destructive list operations make it possible to achieve call-by-reference parameter passing, even though the Lisp function evaluation mechanism only supports call-by-value. Any list passed to a function can be effectively treated as a call-by-reference parameter by performing a destructive operation on it.

As an illustrative example, here is an alternative implementation of the `setfield` function, defined earlier in the primer. Here it is called dsetfield, to indicate its destructive semantics.

```
(defun dsetfield (field-name value struct)
    (setf (cdr (assoc field-name struct)) (list value)))
```

12.7. Pointers Fully Revealed

In order to cope effectively with the power of destructive list operations, we must fully understand how Lisp manages its list structures at the pointer level. Specifically, we must understand:

a. Exactly when new cons cells are allocated.

b. How pointers are used during variable and parameter binding.

c. How pointers are used in both destructive and non-destructive list functions.

New cons cells are allocated in one of two ways: (1) explicit application of `cons` or a non-destructive constructor function derived from cons; (2) the appearance of a literal list value anywhere in a program.

Whenever a list value is bound to a variable or formal parameter, the binding is via pointer copy. The process of binding itself never creates new cons cells. For example, in an assignment such as

```
(setq x '(a b c))
```

it is the creation of the literal list value that allocates the cons cells. What is assigned to `x` is a pointer to the list containing `a`, `b`, and `c`. Whenever the list value of one variable is assigned to another variable, it is the pointer value that is assigned, not a copy of the list. For example, variables `x`, `y`, and `z` all point to the same list after the following sequence of assignments is evaluated:

```
(setq x '(a b c))
(setq y x)
(setq z y)
```

All list access functions, whether destructive or non-destructive, access lists via pointer manipulation, and they do not allocate new cons cells. The critical distinction between destructive versus non-destructive functions is that a non-destructive function never changes the value of a pointer *inside* a cons cell, whereas a destructive function does.

To illustrate the above points in a succinct example, consider the following scenario:

```
>(setq x '(a b c))
(A B C)

>(defun f (i j k)
    (setf (cdddr k) (cdr j))
    (setf (cadr j) i)
    (setf (cdr j) k)
    (setq z k)
```

```
        nil
)
F

>(f x (cdr x) '(a b c))
NIL
```

The three parts of Figure 5 show the following three snapshots of the list space as this scenario proceeds:

a. after the assignment to x, but before function f has been called;

b. after f has been called, its parameters bound, but before its body has been executed;

c. after the execution of f has finished.

It should be clear from this example that it is possible to make arbitrary spaghetti out of the list space using destructive list operations. One of the more unhappy aspects of building cyclic structures such as the one shown in Figure 4c is that all standard print functions will infinitely traverse them. This is in fact why the last line in the body of function f is nil. If that line is omitted, then the output of f is k, because k is the value of the last expression in the body of f. Since k is an entry point into the cyclic structure, the read-eval-print loop will choke when attempting to print the return value of f. As noted earlier, all standard list functions rely on the nil-termination structure of lists in order to behave sensibly. When this structure is violated via destructive list manipulation, the programmer must be keenly aware of the consequences.

An important consequence of Lisp's internal pointer manipulation is that non-destructive operations are more efficient than they might appear. For example, one invocation of cons only takes enough time to allocate a single two-element memory block, and copy two pointers into it. Hence, a cons operation such as this

```
(cons huge-list1 huge-list2)
```

takes the same (small) amount of time as any other cons operation. This is because cons, and its non-destructive derivatives, do not copy all of their arguments to make new lists, just pointers to the arguments.

12.8. Caveat CJ Hack

While some Lisp programmers find destructive list operations indispensable, many others utterly decry their use. For example, Robert Wilensky (author of the widely-used *Common LISPcraft*) does not introduce destructive list operations until page 265 of his book, and he does so under the subheading "The Evil of rplaca and rplacd".

Having seen Lisp's imperative constructs, and the destructive list operations in particular, the die-hard CJ hack may well be thinking "At last, something in Lisp I can sink my teeth into!". The most appropriate response to this is

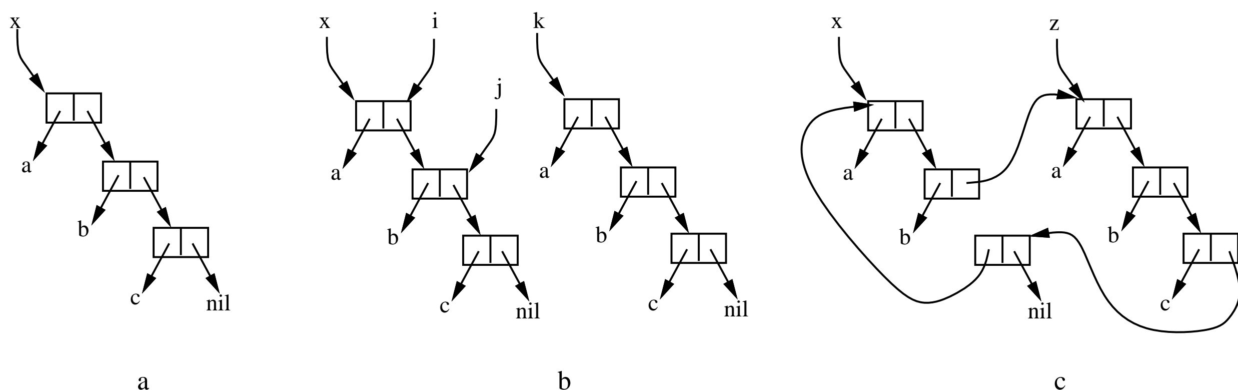

Figure 5: Three Snapshots of the List Space.

"You better have strong teeth". C/C++ die-hards are well acquainted with the havoc that the misuse of pointers can wreak. In Java, reference misuse wreaks less havoc than is possible in C, but Java programmer's must still be mindful of the side effects caused by changing values through references. Even though Java's pointer manipulation is less syntactically explicit than in C/C++, Java's pointer-based processing is still fundamentally destructive in the Lisp sense.

The destructive list operations in Lisp totally expose the pointer-level representation of Lisp lists. The indiscriminant use of these operations can easily reduce the design, implementation, and debugging of Lisp programs to the awfulest levels of CJ. This is indeed a sad way to use an otherwise lovely functional language.