3

Lisp: Functions, Recursion, and Lists

Lisp is the medium of choice for people who enjoy free style and flexibility.

Gerald J. Sussman

A Lisp programmer knows the value of everything, but the cost of nothing.

Alan Perlis

Lisp is a historically important language that is good for illustrating a number of general points about programming languages. Because Lisp is very different from procedure-oriented and object-oriented languages you may use more frequently, this chapter may help you think about programming in a different way. Lisp shows that many goals of programming language design can be met in a simple, elegant way.

3.1 LISP HISTORY

The Lisp programming language was developed at MIT in the late 1950s for research in artificial intelligence (AI) and symbolic computation. The name Lisp is an acronym for the *LIS*t *P*rocessor. Lists comprise the main data structure of Lisp.

The strength of Lisp is its simplicity and flexibility. It has been widely used for exploratory programming, a style of software development in which systems are built incrementally and may be changed radically as the result of experimental evaluation. Exploratory programming is often used in the development of AI programs, as a researcher may not know how the program should accomplish a task until several unsuccessful programs have been tested. The popular text editor emacs is written in Lisp, as is the linux graphical toolkit gtk and many other programs in current use in a variety of computing environments.

Many different Lisp implementations have been built over the years, leading to many different dialects of the language. One influential dialect was Maclisp,

JOHN MCCARTHY

A programming language designer and a central figure in the field of artificial intelligence, John McCarthy led the original Lisp effort at MIT in the late 1950s and early 1960s. Among other seminal contributions to the field, McCarthy participated in the design of Algol 60 and formulated the concept of time sharing in a 1959 memo to the director of the MIT Computation Center. McCarthy moved to Stanford in 1962, where he has been on the faculty ever since.

Throughout his career, John McCarthy has advocated using formal logic and mathematics to understand programming languages and systems, as well as common-sense reasoning and other topics in artificial intelligence. In the early 1960s, he wrote a series of papers on what he called a *Mathematical Theory of Computation.* These identified a number of important problems in understanding and reasoning about computer programs and systems. He supported political freedom for scientists abroad during the Cold War and has been an advocate of free speech in electronic media.

Now a lively person with graying hair and beard, McCarthy is an independent thinker who suggests creative solutions to bureaucratic as well as technical problems. He has won a number of important prizes and honors, including the ACM Turing Award in 1971.

developed in the 1960s at MIT's Project MAC. Another was Scheme, developed at MIT in the 1970s by Guy Steele and Gerald Sussman. Common Lisp is a modern Lisp with complex forms of object-oriented primitives.

McCarthy's 1960 paper on Lisp, called "Recursive functions of symbolic expressions and their computation by machine" [*Communications of the Association for Computing Machinery*, **3**(4), 184–195 (1960)] is an important historical document with many good ideas. In addition to the value of the programming language ideas it contains, the paper gives us some idea of the state of the art in 1960 and provides some useful insight into the language design process. You might enjoy reading the first few sections of the paper and skim the other parts briefly to see what they contain. The journal containing the article will be easy to find in many computer science libraries or you can find a retypeset version of the paper in electronic form on the Web.

3.2 GOOD LANGUAGE DESIGN

Most successful language design efforts share three important characteristics with the Lisp project:

- *Motivating Application:* The language was designed so that a specific kind of program could be written more easily.
- *Abstract Machine:* There is a simple and unambiguous program execution model.
- *Theoretical Foundations:* Theoretical understanding was the basis for including certain capabilities and omitting others.

These points are elaborated in the subsequent subsections.

Motivating Application

An important programming problem for McCarthy's group was a system called *Advice Taker*. This was a common-sense reasoning system based on logic. As the name implies, the program was supposed to read statements written in a specific input language, perform logical reasoning, and answer simple questions. Another important problem used in the design of Lisp was symbolic calculation. For example, McCarthy's group wanted to be able to write a program that could find a symbolic expression for the indefinite integral (as in calculus) for a function, given a symbolic description of the function as input.

Most good language designs start from some specific need. For comparison, here are some motivating problems that were influential in the design of other programming languages:

Lisp	Symbolic computation, logic, experimental programming
C	Unix operating system
Simula	Simulation
PL/1	Tried to solve all programming problems; not successful or influential

A specific purpose provides focus for language designers. It helps us to set criteria for making design decisions. A specific, motivating application also helps us to solve one of the hardest problems in programming language design: deciding which features to leave out.

Program Execution Model

A language design must be specific about how all basic operations are done. The language design may either be very concrete, prescribing exactly how the parts of the language must be implemented, or more abstract, specifying only certain properties that must be satisfied in any implementation. It is possible to err in either direction. A language that is too closely tied to one machine will lead to programs that are not portable. When new technology leads to faster machine architectures, programs written in the language may become obsolete. At the other extreme, it is possible to be too abstract. If a language design specifies only what the eventual value of an expression must be, without any information about how it is to be evaluated, it may be difficult for programmers to write efficient code. Most programmers find it important to have a good understanding of how programs will be executed, with enough detail to predict program running time. Lisp was designed for a specific machine, the IBM 704. However, if the designers had built the language around a lot of special features of a particular computer, the language would not have survived as well as it has. Instead, by luck or by design, they identified a useful set of simple concepts that map easily onto the IBM 704 architecture, and also onto other computers. The Lisp execution model is discussed in more detail in Subsection 3.4.3.

A systematic, predictable machine model is critical to the success of a programming language. For comparison, here are some execution models associated with the design of other programming languages:

Fortran	Flat register machine
	No stacks, no recursion
	Memory arranged as linear array
Algol family	Stack of activation records
	Heap storage
Smalltalk	Objects, communicating by messages

Theoretical Foundations

McCarthy described Lisp as a "scheme for representing the *partial recursive functions* of a certain class of symbolic expressions." We discussed computability and partial recursive functions in Chapter 2. Here are the main points about computability theory that are relevant to the design of Lisp:

- Lisp was designed to be Turing complete, meaning that all partial recursive functions may be written in Lisp. The phrase "Turing complete" refers to a characterization of computability proposed by the mathematician A.M. Turing; see Chapter 2.
- The use of function expressions and recursion in Lisp take direct advantage of a mathematical characterization of computable functions based on lambda calculus.

Today it is unlikely that a team of programming language designers would advertise that their language is sufficient to define all partial recursive functions. Most computer scientists nowadays know about computability theory and assume that most languages intended for general programming are Turing complete. However, computability theory and other theoretical frameworks such as type theory continue to have important consequences for programming language design.

The connection between Lisp and lambda calculus is important, and lambda calculus remains an important tool in the study of programming languages. A summary of lambda calculus appears in Section 4.2.

3.3 BRIEF LANGUAGE OVERVIEW

The topic of this chapter is a language that might be called *Historical Lisp*. This is essentially Lisp 1.5, from the early 1960s, with one or two minor changes. Because there are several different versions of Lisp in common use, it is likely that some function names used in this book will differ from those you may have used in previous Lisp programming.

An engaging book that captures some of the spirit of contemporary Lisp is the Scheme-based paperback by D.P. Friedman and M. Felleisen, titled *The Little Schemer* (MIT Press, Cambridge, MA, 1995). This is similar to an earlier book by the same authors entitled *The Little LISPer.* Lisp syntax is extremely simple. To make parsing (see Section 4.1) easy, all operations are written in prefix form, with the operator in front of all the operands. Here are some examples of Lisp expressions, with corresponding infix form for comparison.

Lisp prefix notation	Infix notation
`(+ 1 2 3 4 5)`	`(1 + 2 + 3 + 4 + 5)`
`(* (+ 2 3) (+ 4 5))`	`((2 + 3) * (4 + 5))`
`(f x y)`	`f(x, y)`

Atoms

Lisp programs compute with atoms and cells. Atoms include integers, floating-point numbers, and symbolic atoms. Symbolic atoms may have more than one letter. For example, the atom duck is printed with four letters, but it is *atomic* in the sense that there is no Lisp operation for taking the atom apart into four separate atoms.

In our discussion of *Historical Lisp*, we use only integers and symbolic atoms. Symbolic atoms are written with a sequence of characters and digits, beginning with a character. The atoms, symbols, and numbers are given by the following Backus normal form (BNF) grammar (see Section 4.1 if you are not familiar with grammars):

```
<atom> ::= <smbl> | <num>
<smbl> ::= <char> | <smbl><char> | <smbl><digit>
<num>  ::= <digit> | <num><digit>
```

An atom that is used for some special purposes is the atom nil.

S-Expressions and Lists

The basic data structures of Lisp are *dotted pairs*, which are pairs written with a dot between the two parts of the pair. Putting atoms or pairs together, we can write symbolic expressions in a form traditionally called S-expressions. The syntax of Lisp S-expressions is given by the following grammar:

```
<sexp> ::= <atom> | (<sexp> . <sexp>)
```

Although S-expressions are the basic data of Historical Lisp, most Lisp programs actually use lists. Lisp lists are built out of pairs in a particular way, as described in Subsection 3.4.3.

Functions and Special Forms

The basic functions of Historical Lisp are the operations

```
cons   car   cdr   eq   atom
```

on pairs and atoms, together with the general programming functions

```
cond   lambda   define   quote   eval
```

We also use numeric functions such as +, –, and ∗, writing these in the usual Lisp prefix notation. The function cons is used to combine two atoms or lists, and car and cdr take lists apart. The function eq is an equality test and atom tests whether its argument is an atom. These are discussed in more detail in Subsection 3.4.3 in connection with the machine representation of lists and pairs.

The general programming functions include cond for a conditional test (if... then...else...), lambda for defining functions, define for declarations, quote to delay or prevent evaluation, and eval to force evaluation of an expression.

The functions cond, lambda, define, and quote are technically called *special forms* since an expression beginning with one of these special functions is evaluated without evaluating all of the parts of the expression. More about this below.

The language summarized up to this point is called *pure Lisp*. A feature of pure Lisp is that expressions do not have *side effects*. This means that evaluating an expression only produces the value of that expression; it does not change the observable state of the machine. Some basic functions that *do* have side effects are

```
rplaca   rplacd   set   setq
```

We discuss these in Subsection 3.4.9. Lisp with one or more of these functions is sometimes called *impure Lisp*.

Evaluation of Expressions

The basic structure of the Lisp interpreter or compiler is the *read-eval-print* loop. This means that the basic action of the interpreter is to read an expression, evaluate it, and print the value. If the expression defines the meaning of some symbol, then the association between the symbol and its value is saved so that the symbol can be used in expressions that are typed in later.

In general, we evaluate a Lisp expression

(function $\text{arg}_1 \ldots \text{arg}_n$)

by evaluating each of the arguments in turn, then passing the list of argument values to the function. The exceptions to this rule are called special forms. For example, we evaluate a conditional expression

```
(cond (p₁ e₁)... (pₙ eₙ))
```

by proceeding from left to right, finding the first p_i with a value different from nil. This involves evaluating $p_1 \ldots p_n$ and one e_i if p_i is nonnil. We return to this below.

Lisp uses the atoms T and nil for true and false, respectively. In this book, true and false are often written in Lisp code, as these are more intuitive and more understandable if you are have not done a lot of Lisp programming. You may read Lisp examples that contain true and false as if they appear inside a program for which we have already defined true and false as synonyms for T and nil, respectively.

A slightly tricky point is that the Lisp evaluator needs to distinguish between a string that is used to name an atom and a string that is used for something else, such as the name of a function. The form quote is used to write atoms and lists directly:

(quote cons)	expression whose value is the atom "cons"
(cons a b)	expression whose value is the pair containing the values of a and b
(cons (quote A) (quote B))	expression whose value is the pair containing the atoms "A" and "B"

In most dialects of Lisp, it is common to write 'bozo instead of (quote bozo). You can see from the preceding brief description that quote must be a special form. Here are some additional examples of Lisp expressions and their values:

(+ 4 5)	expression with value 9
(+ (+ 1 2) (+ 4 5))	first evaluate 1+2, then 4+5, then 3+9 to get value 12
(quote (+ 1 2))	evaluates to a list (+ 1 2)
'(+ 1 2)	same as (quote (+ 1 2))

Example. Here is a slightly longer Lisp program example, the definition of a function that searches a list. The find function takes two arguments, x and y, and searches the list y for an occurrence of x. The declaration begins with define, which indicates that this is a declaration. Then follows the name find that is being defined, and the expression for the find function:

```
(define find (lambda (x y)
    (cond ((equal y nil) nil)
              ((equal x (car y)) x)
              (true (find x (cdr y)))
)))
```

Lisp function expressions begin with lambda. The function has two arguments, x and y, which appear in a list immediately following lambda. The return value of the function is given by the expression that follows the parameters. The function body is a conditional expression, which returns nil, the empty list, if y is the empty list. Otherwise, if x is the first element (car) of the list y, then the function returns the element x. Otherwise the function makes a recursive call to see if x is in the cdr of the list y. The cdr of a list is the list of all elements that occur after the first element. We can use this function to find 'apple in the list '(pear peach apple fig banana) by writing the Lisp expression

```
(find 'apple '(pear peach apple fig banana))
```

Static and Dynamic Scope

Historically, Lisp was a dynamically scoped language. This means that a variable inside an expression could refer to a different value if it is passed to a function that declared this variable differently. When Scheme was introduced in 1978, it was a statically scoped variant of Lisp. As discussed in Chapter 7, static scoping is common in most modern programming languages. Following the widespread acceptance of Scheme, most modern Lisps have become statically scoped. The difference between static and dynamic scope is not covered in this chapter.

Lisp and Scheme

If you want to try writing Lisp programs by using a Scheme compiler, you will want to know that the names of some functions and special forms differ in Scheme and Lisp. Here is a summary of some of the notational differences:

Lisp	Scheme	Lisp	Scheme
defun	define	rplacaset	car!
defvar	define	rplacdset	cdr!
car, cdr	car, cdr	mapcar	map
cons	cons	t	#t
null	null?	nil	#f
atom	atom?	nil	nil
eq, equal	eq?, equal?	nil	'()
Setq	set!	progn	begin
cond . . . t	cond . . . else		

3.4 INNOVATIONS IN THE DESIGN OF LISP

3.4.1 Statements and Expressions

Just as virtually all natural languages have certain basic parts of speech, such as *nouns*, *verbs*, and *adjectives*, there are programming language parts of speech that occur in most languages. The most basic programming language parts of speech are *expressions*, *statements*, and *declarations*. These may be summarized as follows:

Expression: a syntactic entity that may be evaluated to determine its value. In some cases, evaluation may not terminate, in which case the expression has no value. Evaluation of some expressions may change the state of the machine, causing a side effect in addition to producing a value for the expression.

Statement : a command that alters the state of the machine in some explicit way. For example, the machine language statement `load 4094 r1` alters the state of the machine by placing the contents of location 4094 into register r1. The programming language statement `x := y + 3` alters the state of the machine by adding 3 to the value of variable y and storing the result in the location associated with variable x.

Declaration: a syntactic entity that introduces a new identifier, often specifying one or more attributes. For example, a declaration may introduce a variable i and specify that it is intended to have only integer values.

Errors and termination may depend on the order in which parts of expressions are evaluated. For example, consider the expression

```
if f(2)=2 or f(3)=3 then 4 else 4
```

where f is a function that halts on even arguments but runs forever on odd arguments. In many programming languages, a Boolean expression A or B would be evaluated from left to right, with B evaluated only if A is false. In this case, the value of the preceding expression would be 4. However, if we evaluate the test A or B from right to left or evaluate both A and B regardless of the value of A, then the value of the expression is undefined.

Traditional machine languages and assembly languages are based on statements. Lisp is an expression-based language, meaning that the basic constructs of the language are expressions, not statements. In fact, pure Lisp has no statements and no expressions with side effects. Although it was known from computability theory that it was possible to define all computable functions without using statements or side effects, Lisp was the first programming language to try to put this theoretical possibility into practice.

3.4.2 Conditional Expressions

Fortran and assembly languages used before Lisp had conditional statements. A typical statement might have the form

```
if (condition) go to 112
```

If the condition is true when this command is executed, then the program jumps to the statement with the label 112. However, conditional expressions that produce a value instead of causing a jump were new in Lisp. They also appeared in Algol 60, but this seems to have been the result of a proposal by McCarthy, modified by a syntactic suggestion of Backus.

The Lisp conditional expression

```
(cond (p₁ e₁) ... (pₙ eₙ))
```

could be written as

```
if p₁ then e₁
    else if p₂ then e₂
        ...
            else if pₙ then eₙ
                else no_value
```

in an Algol-like notation, except that most programming languages do not have a direct way of specifying the absence of a value. In brief, the value of (cond (p_1 e_1) ... (p_n e_n)) is the first e_i, proceeding from left to right, with p_i nonnil and p_j nil (representing false) for all $j < i$. If there is no such e_i then the conditional expression has no value. If any of the expressions $p_1 \ldots p_n$ have side effects, then these will occur from left to right as the conditional expression is evaluated.

The Lisp conditional expression would now be called a *sequential conditional expression*. The reason it is called sequential is that the parts of this expression are evaluated in sequence from left to right, with evaluation terminating as soon as a value for the expression can be determined.

It is worth noting that (cond (p_1 e_1) ... (p_n e_n)) is undefined if

$p_1,\ldots,p_n$ are all nil
$p_1,\ldots,p_i$ false and p_{i+1} undefined
$p_1,\ldots,p_i$ false, p_{i+1} true, and e_{i+1} undefined

Here are some example conditional expressions and their values:

```
(cond ((< 2 1) 2) ((< 1 2) 1))   has value 1
(cond ((< 2 1) 2) ((< 3 2) 3))   is undefined
(cond (diverge 1) (true 0))      is undefined, if diverge does not terminate
(cond (true 0) (diverge 1))      has value 0
```

Strictness. An important part of the Lisp cond expression is that a conditional expression may have a value even if one or more subexpressions do not. For example, (cond (true e_1) (false e_2)) may be defined even if e_2 is undefined. In contrast, $e_1 + e_2$ is undefined if either e_1 or e_2 is undefined. In standard programming language terminology, an operator or expression form is *strict* if all operands or subexpressions are evaluated. Lisp cond is not strict, but addition is. (Some operators from C that are not strict are && and ||.)

3.4.3 The Lisp Abstract Machine

What is an Abstract Machine?

The phrase *abstract machine* is generally used to refer to an idealized computing device that can execute a specific programming language directly. Typically an abstract machine may not be fully implementable: An abstract machine may provide infinitely many memory locations or infinite-precision arithmetic. However, as we use the phrase in this book, an abstract machine should be sufficiently realistic to provide useful information about the real execution of real programs on real hardware. Our goal in discussing abstract machines is to identify the mental model of the computer that a programmer uses to write and debug programs. For this reason, there is a tendency to refer to *the* abstract machine associated with a specific programming language.

The Abstract Machine for Lisp

The abstract machine for Pure Lisp has four parts:

- A *Lisp expression* to be evaluated.
- A *continuation*, which is a function representing the remaining program to evaluate when done with the current expression.
- An *association list*, commonly called the *A-list* in much of the literature on Lisp and called the *run-time stack* in the literature on Algol-based languages. The purpose of the A-list is to store the values of variables that may occur either in the current expression to be evaluated or in the remaining expressions in the program.
- A *heap*, which is a set of cons cells (pairs stored in memory) that might be pointed to by pointers in the A-list.

The structure of this machine is not investigated in detail. The main idea is that when a Lisp expression is evaluated some bindings between identifiers and values may be created. These are stored on the A-list. Some of these values may involve cons cells that are placed in the heap. When the evaluation of an expression is completed, the value of that expression is passed to the continuation, which represents the work to be done by the program after that expression is evaluated.

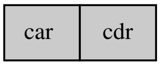
This abstract machine is similar to a standard register machine with a stack, if we think of the current expression as representing the program counter and the continuation as representing the remainder of the program.

There are four main equality functions in Lisp: eq, eql, equal, and =. The function eq tests whether its arguments are represented by the same sequence of memory locations, and = is numeric equality. The function eql tests whether its arguments are the same symbol or number, and equal is a recursive equality test on lists or atoms that is implemented by use of eq and =. For simplicity, we generally use equal in sample code.

Cons Cells

Cons cells (or dotted pairs) are the basic data structure of the Lisp abstract machine. Cons cells have two parts, historically called the *address* part and the *decrement* part. The words address and decrement come from the IBM 704 computer and are hardly ever used today. Only the letters a and d remain in the acronyms `car` (for "contents of the address register") and `cdr` (for "contents of the decrement register").

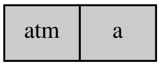
We draw cons cells as follows:



Cons cells

- provide a simple model of memory in the machine,
- are efficiently implementable, and
- are not tightly linked to particular computer architecture.

We may represent an atom may be represented with a cons cell by putting a "tag" that tells what kind of atom it is in the address part and the actual atom value in the decrement part. For example, the letter "a" could be represented as a Lisp atom as



where atm indicates that the cell represents an atom and a indicates that the atom is the letter a.

Because putting a pointer in one or both parts of a cons cell represents lists, the bit pattern used to indicate an atom must be different from every pointer value. There are five basic functions on cons cells, which are evaluated as follows:

`atom`, a function with one argument: If a value is an atom, then the word storing the value has a special bit pattern in its address part that flags the value as being an atom. The `atom` function returns true if this pattern indicates that the function argument is an atom. (In Scheme, the function `atom` is written `atom?`, which reminds us that the function value will be `true` or `false`.)

`eq`, a function with two arguments: compares two arguments for equality by checking to see if they are stored in the same location. This is meaningful for atoms as well as for cons cells because conceptually the Lisp compiler behaves as if each atom (including every number) is stored once in a unique location.

`cons`, a function with two arguments: The expression (`cons x y`) is evaluated as follows:

1. Allocate new cell `c`.
2. Set the address part of `c` to point to the value of `x`.
3. Set the decrement part of `c` to point to the value of `y`.
4. Return a pointer to `c`.

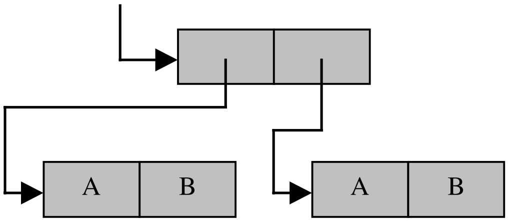
`car`, a function with one argument: If the argument is a cons cell `c`, then return the *c*ontents of the *a*ddress *r*egister of `c`. Otherwise the application of `car` results in an error.

`cdr`, a function with one argument: If the argument is a cons cell `c`, then return the *c*ontents of the *d*ecrement *r*egister of `c`. Otherwise the application of `cdr` results in an error.

In drawing cons cells, we draw the contents of a cell as either a pointer to another cell or as an atom. For our purposes, it is not important how an atom is represented inside a cons cell. (It could be represented as either a specific bit pattern inside a cell or as a pointer to a location that contains the representation of an atom.)

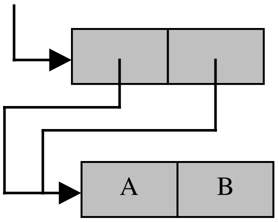
Example 3.1
We evaluate the expression (cons 'A 'B) by creating a new cons cell and then setting the car of this cell to the atom 'A and the cdr of the cell to 'B. The expression '(A . B) would have the same effect, as this is the syntax for a dotted pair of atom A and atom B. Although this dotted-pair notation was a common part of early Lisp, Scheme and later Lisps emphasize lists over pairs.

Example 3.2
When the expression (cons (cons 'A 'B) (cons 'A 'B)) is evaluated, a new structure of the following form is created:



The reason that there are two cons cells with the same parts is that each evaluation of (cons 'A 'B) creates a new cell. This structure is printed as ((A . B) . (A . B)).

Example 3.3
It is also possible to write a Lisp expression that creates the following structure, which is also printed ((A . B) . (A . B)):

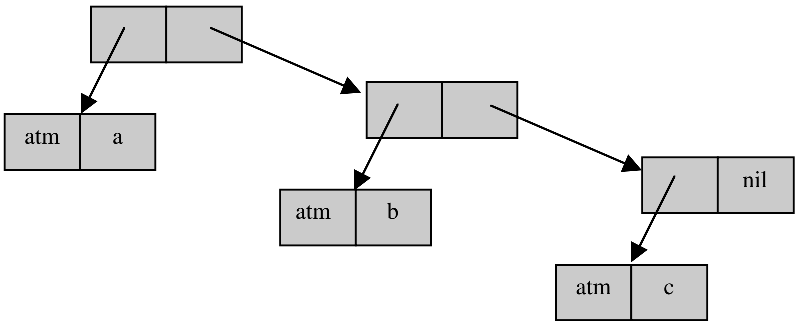


One expression whose evaluation produces this structure is ((lambda (x) (cons x x)) (cons 'A 'B)). We proceed with the evaluation of this expression by first evaluating the function argument (cons 'A 'B) to produce the cons cell drawn here, then passing the cell to the function (lambda (x) (cons x x)) that creates the upper cell with two pointers to the (A.B) cell. Lisp lambda expressions are described in this chapter in Subsection 3.4.5.

Representation of Lists by Cons Cells
Because a cons cell may contain two pointers, cons cells may be used to construct trees. Because Lisp programs often use lists, there are conventions for representing

lists as a certain form of trees. Specifically, the list $a_1, a_2, \ldots a_n$ is represented by a cons cell whose car is a_1 and whose cdr points to the cells representing list $a_2, \ldots a_n$. For the empty list, we use a pointer set to NIL.

For example, here is representation for the list (A B C), also written as (A . (B . (C . NIL))):



In this illustration, atoms are shown as cons cells, with a bit pattern indicating atom in the first part of the cell and the actual atom value in the second. For simplicity, we often draw lists by placing an atom inside half a cons cells. For example, we could write A in the address part of the top-left cell instead of drawing a pointer to the cell representing atom A, and similarly for list cells pointing to atoms B and C. In the rest of the book, we use the simpler form of drawing; the illustration here is just to remind us that atoms as well as lists must be represented inside the machine in some way.

3.4.4 Programs as Data

Lisp data and Lisp programs have the same syntax and internal representation. This makes it easy to manipulate Lisp programs as data. One feature that sets Lisp apart from many other languages is that it is possible for a program to build a data structure that represents an expression and then evaluates the expression as if it were written as part of the program. This is done with the function eval.

Example. We can write a Lisp function to substitute expression x for all occurrences of y in expression z and then evaluate the resulting expression. To make the logical structure of the substitute-and-eval function clear, we define substitute first and then use it in substitute-and-eval. The substitute function has three arguments, exp1, var, and exp2. The result of (substitute exp1 var exp2) is the expression obtained from exp2 when all occurrences of var are replaced with exp1:

```
(define substitute (lambda (exp1 var exp2)
     (cond   ((atom exp2) (cond ((eq exp2 var) exp1) (true exp2)))
             (true (cons (substitute exp1 var (car exp2))
                   (substitute exp1 var (cdr exp2)))))))
(define substitute-and-eval (lambda (x y z) (eval (substitute x y z))))
```

The ability to use list operations to build programs at run time and then execute them is a distinctive feature of Lisp. You can appreciate the value of this feature if you think about how you would write a C program to read in a few C expressions and execute them. To do this, you would have to write some kind of C interpreter or compiler. In Lisp, you can just read in an expression and apply the built-in function eval to it.

3.4.5 Function Expressions

Lisp computation is based on functions and recursive calls instead of on assignment and iterative loops. This was radically different from other languages in 1960, and is fundamentally different from many languages in common use now.

A Lisp function expression has the form

```
(lambda ( ⟨parameters⟩ ) ⟨function_body⟩)
```

where ⟨parameters⟩ is a list of identifiers and ⟨function_body⟩ is an expression. For example, a function that places its argument in a list followed by atoms A and B may be written as

```
(lambda (x) (cons x ′(A B)))
```

Another example is this function that, with a primitive function used for addition, adds its two arguments:

```
(lambda (x y) (+ x y))
```

The idea of writing expressions for functions may be traced to lambda calculus, which was developed by Alonzo Church and others, beginning in the 1930s. One fact about lambda calculus that was known in the 1930s was that every function computable by a Turing machine could also be written in the lambda calculus and conversely.

In lambda calculus, function expressions are written with the Greek lowercase letter lambda (λ), rather than with the word lambda, as in Lisp. Lambda calculus also uses different conventions about parenthesization. For example, the function that squares its argument and adds the result to y is written as

$$\lambda x.(x^2 + y)$$

in lambda calculus, but as

```
(lambda (x) (+ (square x) y))
```

in Lisp. In this function, x is called the *formal parameter*; this means that x is a placeholder in the function definition that will refer to the actual parameter when the function is applied. More specifically, consider the expression

```
((lambda (x) (+ (square x) y)) 4)
```

that applies a function to the integer 4. This expression can be evaluated only in a context in which y already has a value. The value of this expression will be 16 plus the value of y. This will be computed by the evaluation of (plus (square x) y) with x set to 4. The identifier y is a said to be a *global variable* in this function expression because its value must be given a value by some definition outside the function expression. More information about variables, binding, and lambda calculus may be found in Section 4.2.

3.4.6 Recursion

Recursive functions were new at the time Lisp appeared. McCarthy, in addition to including them in Lisp, was the main advocate for allowing recursive functions in Algol 60. Fortran, by comparison, did not allow a function to call itself.

Members of the Algol 60 committee later wrote that they had no idea what they were in for when they agreed to include recursive functions in Algol 60. (They might have been unhappy for a few years, but they would probably agree now that it was a visionary decision.)

Lisp lambda makes it possible to write *anonymous functions*, which are functions that do not have a declared name. However, it is difficult to write recursive functions with this notation. For example, suppose we want to write an expression for the function f such that

```
(f x) = (cond      ((eq x 0) 0)
                   (true (+ x (f ( - x 1))))
          )
```

A first attempt might be

```
(lambda (x) (cond ((eq x 0) 0) (true (+ x (f ( - x 1))))))
```

However, this does not make any sense because the f inside the function expression is not defined anywhere. McCarthy's solution in 1960 was to add an operator called label so that

```
(label f (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))
```

defines the recursive f suggested previously. In later Lisps, it became accepted style just to declare a function by use of the define declaration form. In particular, a recursive function f can be declared by

```
(define f (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))
```

Another notation in some versions of Lisps is defun for "define function," which eliminates the need for lambda:

```
(defun f (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1))))))
```

McCarthy's 1960 paper comments that the lambda notation is inadequate for expression recursive functions. This statement is false. Lambda calculus, and therefore Lisp, is capable of expressing recursive functions without any additional operator such as label. This was known to experts in lambda calculus in the 1930s, but apparently not known to McCarthy and his group in the 1950s. (See Subsection 4.2.3 for an explanation of how to do it.)

3.4.7 Higher-Order Functions

The phrase *higher-order function* means a function that either takes a function as an argument or returns a function as a result (or both). This use of higher-order comes from a convention that calls a function whose arguments and results are not functions a *first-order function*. A function that takes a first-order function as an argument is called a *second-order function*, functions on second-order functions are called *third-order functions*, and so on.

Example 3.4
If f and g are mathematical functions, say functions on the integers, then their composition $f \circ g$ is the function such that for every integer x, we have

$$(f \circ g)(x) = f(g(x)).$$

We can write composition as a Lisp function compose that takes two functions as arguments and returns their composition:

```
(define compose (lambda (f g) (lambda (x) (f (g x)))))
```

The first lambda is used to identify the arguments to compose. The second lambda is used to define the return value of the function, which is a function. You might enjoy calculating the value of the expression

```
(compose (lambda (x) (+ x x)) (lambda (x) (* x x)))
```

Example 3.5
A maplist is a function that takes a function and list and applies the function to every element in the list. The result is a list that contains all the results of function application. Using define to define a recursive function, we can write the maplist as follows:

```
(define maplist (lambda (f x)
   (cond ((eq x nil) nil) (true (cons (f (car x)) (maplist f (cdr x)))))))
```

We cannot say whether the maplist is a second-order or third-order function, as the elements of the list might be atoms, functions, or higher-order functions. As an example of the use of this function, we have

```
(maplist square '(1 2 3 4 5)) ⇒ (1 4 9 16 25),
```

where the symbol ⇒ means "evaluates to."

Higher-order functions require more run-time support than first-order functions, as discussed in some detail in Chapter 7.

3.4.8 Garbage Collection

In computing, *garbage* refers to memory locations that are not accessible to a program. More specifically, we define garbage as follows:

> At a given point in the execution of a program P, a memory location m is *garbage* if no completed execution of P from this point can access location m. In other words, replacing the contents of m or making this location inaccessible to P cannot affect any further execution of the program.

Note that this definition does not give an algorithm for finding garbage. However, if we could find all locations that are garbage (by this definition), at some point in the suspended execution of a program, it would be safe to deallocate these locations or use them for some other purpose.

In Lisp, the memory locations that are accessible to a program are cons cells. Therefore the garbage associated with a running Lisp program will be a set of cons cells that are not needed to complete the execution of the program. Garbage collection is the process of detecting garbage during the execution of a program and making it available for other uses. In garbage-collected languages, the run-time system receives requests for memory (as when Lisp cons cells are created) and allocates memory from some list of available space. The list of available memory locations is called the free list. When the run-time system detects that the available space is below some threshold, the program may be suspended and the garbage collector invoked. In Lisp and other garbage-collected languages, it is generally not necessary for the program to invoke the garbage collector explicitly. (In some modern implementations, the garbage collector may run in parallel with the program. However, because concurrent garbage collection raises some additional considerations, we will assume that the program is suspended when the garbage collector is running.)

The idea and implementation of automatic garbage collection appear to have originated with Lisp.

Here is an example of garbage. After the expression

```
(car (cons e₁ e₂ ))
```

is evaluated, any cons cells created by evaluation of e_2 will typically be garbage. However, it is not always correct to deallocate the locations used in a list after applying car to the list. For example, consider the expression

```
((lambda (x) (car (cons x x))) '(A B))
```

When this expression is evaluated, the function car will be applied to a cons cell whose "a" and "d" parts both point to the same list.

Many algorithms for garbage collection have been developed over the years. Here is a simple example called mark-and-sweep. The name comes from the fact that the algorithm first marks all of the locations reachable from the program, then "sweeps" up all the unmarked locations as garbage. This algorithm assumes that we can tell which bit sequences in memory are pointers and which are atoms, and it also assumes that there is a tag bit in each location that can be switched to 0 or 1 without destroying the data in that location.

Mark-and-Sweep Garbage Collection

1. Set all tag bits to 0.
2. Start from each location used directly in the program. Follow all links, changing the tag bit of each cell visited to 1.
3. Place all cells with tags still equal to 0 on the free list.

Garbage collection is a *very* useful feature, at least as far as programmer convenience goes. There is some debate about the efficiency of garbage-collected languages, however. Some researchers have experimental evidence showing that garbage collection adds of the order of 5% overhead to program execution time. However, this sort of measurement depends heavily on program design. Some simple programs could be written in C without the use of any user-allocated memory, but when translated into Lisp could create many cons cells during expression evaluation and therefore involve a lot of garbage-collection overhead. On the other hand, explicit memory management in C and C++ (in place of garbage collection) can be cumbersome and error prone, so that for certain programs it is highly advantageous to have automatic garbage collection. One phenomenon that indicates the importance and difficulty of memory management in C programs is the success of program analysis tools that are aimed specifically at detecting memory management errors.

Example. In Lisp, we can write a function that takes a list lst and an entry x, returning the part of the list that follows x, if any. This function, which we call select, can be written as follows:

```
(define select (lambda (x lst)
   (cond ((equal lst nil) nil)
        ((equal x (car lst)) (cdr lst))
        (true (select x (cdr lst)))
)))
```

Here are two analogous C programs that have different effects on the list they are passed. The first one leaves the list alone, returning a pointer to the cdr of the first cell that has its car equal to x:

```
typedef struct cell cell;
struct cell {
      cell * car, * cdr;
} ;
cell * select (cell *x, cell *lst) {
     cell *ptr;
     for (ptr=lst; ptr != 0; ) {
          if (ptr->car == x) return(ptr->cdr);
          else ptr = ptr->cdr;
     };
};
```

A second C program might be more appropriate if only the part of the list that follows x will be used in the rest of the program. In this case, it makes sense to free the cells that will no longer be used. Here is a C function that does just this:

```
cell * select1 (cell *x; cell *lst) {
     cell *ptr, *previous;
     for (ptr=lst; ptr != 0; ) {
          if (ptr->car == x) return(ptr->cdr);
          else previous = ptr;
          ptr = ptr->cdr;
          free(previous);
     }
}
```

An advantage of Lisp garbage collection is that the programmer does not have to decide which of these two functions to call. In Lisp, it is possible to just return a pointer to the part of the list that you want and let the garbage collector figure out whether you may need the rest of the list ever again. In C, on the other hand, the programmer must decide, while traversing the list, whether this is the last time that these cells will be referenced by the program.

Question to Ponder. It is interesting to observe that programming languages such as Lisp, in which most computation is expressed by recursive functions and linked data structures, provide automatic garbage collection. In contrast, simple imperative languages such as C require the programmer to free locations that are no longer needed. Is it just a coincidence that function-oriented languages have garbage collection and assignment-oriented languages do not? Or is there something intrinsic to function-oriented programming that makes garbage collection more appropriate for these languages? Part of the answer lies in the preceding example. Another part of the answer seems to lie in the problems associated with storage management for higher-order functions, studied in Section 7.4.

3.4.9 Pure Lisp and Side Effects

Pure Lisp expressions do not have side effects, which are visible changes in the state of the machine as the result of evaluating an expression. However, for efficiency, even early Lisp had expressions with side effects. Two historical functions with side effects are rplaca and rplacd:

(rplaca x y) – replace the address field of cons cell x with y,

(rplacd x y) – replace the decrement field of cons cell x with y.

In both cases, the value of the expression is the cell that has been modified. For example, the value of

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

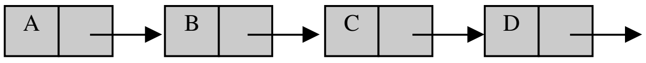
is the cons cell with car 'C and cdr 'B produced when a new cons cell is allocated in the evaluation of (cons 'A 'B) and then the car 'A is replaced with 'C.

With these constructs, two occurrences of the same expression may have different values. (This is really what side effect means.) For example, consider the following code:

```
(lambda (x) (cons (car x) (cons (rplaca x c) (car x)))) (cons a b)
```

The expression (car x) occurs twice within the function expression, but there will be two different values in the two places this expression is evaluated. When rplaca and rplacd are used, it is possible to create circular list structures, something that is not possible in pure Lisp.

One situation in which rplaca and rplacd may increase efficiency is when a program modifies a cell in the middle of a list. For example, consider the following list of four elements:



Suppose we call this list x and we want to change the third element of list x to '. In pure Lisp, we cannot change any of the cells of this list, but we can define a new list with elements A, B, y, D. The new list can be defined with the following expression, where cadr x means "car of the cdr of x" and cdddr x means "cdr of cdr of cdr of x":

```
(cons (car x) (cons (cadr x) (cons y (cdddr x))))
```

Note that evaluating this expression will involve creating new cons cells for the first three elements of the list and, if there is no further use for them, eventually garbage collecting the old cells used for the first three elements of x. In contrast, in impure Lisp we can change the third cell directly by using the expression

```
(rplaca (cddr x) y)
```

If all we need is the list we obtained by replacing the third element of x with y, then this expression gets the result we want far more efficiently. In particular, there is no need to allocate new memory or free memory used for the original list.

Although this example may suggest that side effects lead to efficiency, the larger picture is more complicated. In general, it is difficult to compare the efficiency of different languages if the same problem would be best solved in very different ways. For example, if we write a program by using pure Lisp, we might be inclined to use different algorithms than those for impure Lisp. Once we begin to compare the efficiency of different solutions for the same problem, we should also take into account the amount of effort a programmer must spend writing the program, the ease of debugging, and so on. These are complex properties of programming languages that are difficult to quantify.

3.5 CHAPTER SUMMARY: CONTRIBUTIONS OF LISP

Lisp is an elegant programming language designed around a few simple ideas. The language was intended for symbolic computation, as opposed to the kind of numeric computation that was dominant in most programming outside of artificial intelligence research in 1960. This innovative orientation can be seen in the basic data structure, lists, and in the basic control structures, recursion and conditionals. Lists can be used to store sequences of symbols or represent trees or other structures. Recursion is a natural way to proceed through lists that may contain atomic data or other lists.

Three important aspects of programming language design contributed to the success of Lisp: a specific motivation application, an unambiguous program execution model, and attention to theoretical considerations. Among the main theoretical considerations, Lisp was designed with concern for the mathematical class of partial recursive functions. Lisp syntax for function expressions is based on lambda calculus.

The following contributions are some that are important to the field of programming languages:

- *Recursive functions.* Lisp programming is based on functions and recursion instead of assignment and while loops. Lisp introduces recursive functions and supports functions with function arguments and functions that return functions as results.

- *Lists.* The basic data structure in early Lisp was the cons cell. The main use of cons cells in modern forms of Lisp is for building lists, and lists are used for everything. The list data structure is extremely useful. In addition, the Lisp presentation of memory as an unlimited supply of cons cells provides a more useful abstract machine for nonnumerical programming than do arrays, which were primary data structures in other languages of the early days of computing.
- *Programs as data.* This is still a revolutionary concept 40 years after its introduction in Lisp. In Lisp, a program can build the list representation of a function or other forms of expression and then use the `eval` function to evaluate the expression.
- *Garbage collection.* Lisp was the first language to manage memory for the programmer automatically. Garbage collection is a useful feature that eliminates the program error of using a memory location after freeing it.

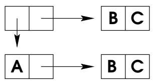
In the years since 1960, Lisp has continued to be successful for symbolic mathematics and exploratory programming, as in AI research projects and other applications of symbolic computation or logical reasoning. It has also been widely used for teaching because of the simplicity of the language.

EXERCISES

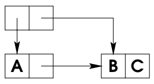
3.1 Cons Cell Representations

(a) Draw the list structure created by evaluating `(cons 'A (cons 'B 'C))`.

(b) Write a pure Lisp expression that will result in this representation, with no sharing of the `(B . C)` cell. Explain why your expression produces this structure.



(c) Write a pure Lisp expression that will result in this representation, with sharing of the `(B . C)` cell. Explain why your expression produces this structure.



While writing your expressions, use only these Lisp constructs: lambda abstraction, function application, the atoms 'A 'B 'C, and the basic list functions (cons, car, cdr, atom, eq). Assume a simple-minded Lisp implementation that does not try to do any clever detection of common subexpressions or advanced memory allocation optimizations.

3.2 Conditional Expressions in Lisp

The semantics of the Lisp conditional expression

$$(\texttt{cond}\ (p_1\ e_1)\ \ldots\ (p_n\ e_n))$$

is explained in the text. This expression does not have a value if $p_1, \ldots, p_k$ are false and p_{k+1} does not have a value, regardless of the values of $p_{k+2}, \ldots, p_n$.

Imagine you are an MIT student in 1958 and you and McCarthy are considering alternative interpretations for conditionals in Lisp:

(a) Suppose McCarthy suggests that the value of $(\texttt{cond}\ (p_1\ e_1) \ldots (p_n\ e_n))$ should be the value of e_k if p_k is true and if, for every $i<k$, the value of expression p_i is either false or undefined. Is it possible to implement this interpretation? Why or why not? (*Hint:* Remember the halting problem.)

(b) Another design for conditional might allow any of several values if more than one of the guards $(p_1, \ldots, p_n)$ is true. More specifically (and be sure to read carefully), suppose someone suggests the following meaning for conditional:

i. The conditional's value is undefined if none of the p_k is true.

ii. If some p_k are true, then the implementation *must* return the value of e_j for *some* j with p_j true. However, it need not be the first such e_j.

Note that in (cond (*a b*) (*c d*) (*e f*)), for example, if a runs forever, c evaluates to true, and e halts in error, the value of this expression should be the value of d, if it has one. Briefly describe a way to implement conditional so that properties i and ii are true. You need to write only two or three sentences to explain the main idea.

(c) Under the original interpretation, the function

```
(defun odd (x) (cond ((eq x 0) nil)
                     ((eq x 1) t)
                     ((> x 0) (odd (- x 2)))
                     (t (odd (+ x 2)))))
```

would give us t for odd numbers and nil for even numbers. Modify this expression so that it would always give us t for odd numbers and nil for even numbers under the alternative interpretation described in part (b).

(d) The normal implementation of Boolean OR is designed not to evaluate a subexpression unless it is necesary. This is called the *short-circuiting* OR, and it may be defined as follows:

$$\textsc{Scor}(e_1, e_2) = \begin{cases} \text{true} & \text{if } e_1 = \text{true} \\ \text{true} & \text{if } e_1 = \text{false and e}_2 = \text{true} \\ \text{false} & \text{if } e_1 = e_2 = \text{false} \\ \textit{undefined} & \text{otherwise} \end{cases} .$$

It allows e_2 to be undefined if e_1 is true.

The *parallel* OR is a related construct that gives an answer whenever possible (possibly doing some unnecessary subexpression evaluation). It is defined similarly:

$$\textsc{Por}(e_1, e_2) = \begin{cases} \text{true} & \text{if } e_1 = \text{true} \\ \text{true} & \text{if } e_2 = \text{true} \\ \text{false} & \text{if } e_1 = e_2 = \text{false} \\ \textit{undefined} & \text{otherwise} \end{cases} .$$

It allows e_2 to be undefined if e_1 is true and also allows e_1 to be undefined if e_2 is true. You may assume that e_1 and e_2 do not have side effects.

Of the original interpretation, the interpretation in part (a), and the interpretation in part (b), which ones would allow us to implement SCOR most easily? What about POR? Which interpretation would make implementations of short-circuiting OR difficult? Which interpretation would make implementation of parallel OR difficult? Why?

3.3 Detecting Errors

Evaluation of a Lisp expression can either terminate normally (and return a value), terminate abnormally with an error, or run forever. Some examples of expressions that terminate with an error are (/ 3 0), division by 0; (car 'a), taking the car of an atom; and (+ 3 "a"), adding a string to a number. The Lisp system detects these errors, terminates evaluation, and prints a message to the screen. Your boss wants to handle errors in Lisp programs without terminating the computation, but doesn't know how, so your boss asks you to...

(a) ...implement a Lisp construct (error? E) that detects whether an expression E will cause an error. More specifically, your boss wants the evaluation of (error? E) to halt with the value *true* if the evaluation of E terminates in error and to halt with the value *false* otherwise. Explain why it is not possible to implement the error? construct as part of the Lisp environment.

(b) ...implement a Lisp construct (guarded E) that either executes E and returns its value, or, if E halts with an error, returns 0 without performing any side effects. This could be used to try to evaluate E and, if an error occurs, just use 0 instead. For example,

```
(+ (guarded E) E′)    ; just E′ if E halts with an error; E+E′ otherwise
```

will have the value of E′ if the evaluation of E halts in error and the value of E + E′ otherwise. How might you implement the guarded construct? What difficulties might you encounter? Note that, unlike that of (error? E), evaluation of (guarded E) does not need to halt if evaluation of E does not halt.

3.4 Lisp and Higher-Order Functions

Lisp functions compose, mapcar, and maplist are defined as follows, with #t written for *true* and () for the empty list. Text beginning with ;; and continuing to the end of a line is a comment.

```
(define compose
    (lambda (f g)   (lambda (x) (f (g x)))))

(define mapcar
  (lambda (f xs)
    (cond
      ((eq? xs ()) ())   ;; If the list is empty, return the empty list
      (#t               ;; Otherwise, apply f to the first element...
          (cons (f (car xs))
                           ;; and map f on the rest of the list
            (mapcar f (cdr xs))
)))))

(define maplist
  (lambda (f xs)
    (cond
      ((eq? xs ()) ())   ;; If the list is empty, return the empty list
      (#t               ;; Otherwise, apply f to the list...
          (cons (f xs)
                           ;; and map f on the rest of the list
```

```
                    (maplist f (cdr xs))
        )))))
```

The difference between maplist and mapcar is that maplist applies f to every sublist, whereas mapcar applies f to every element. (The two function expressions differ in only the sixth line.) For example, if inc is a function that adds one to any number, then

```
(mapcar inc  '(1 2 3 4)) = (2 3 4 5)
```

whereas

```
(maplist (lambda (xs) (mapcar inc xs)) '(1 2 3 4))
= ((2 3 4 5) (3 4 5) (4 5) (5))
```

However, you can almost get mapcar from maplist by composing with the car function. In particular, note that

```
(mapcar f  '(1 2 3 4))
= ((f (car (1 2 3 4))) (f (car (2 3 4))) (f (car (3 4))) (f (car (4))))
```

Write a version of compose that lets us define mapcar from maplist. More specifically, write a definition of compose2 so that

```
((compose2 maplist car) f xs) = (mapcar f xs)
```

for any function f and list xs.

(a) Fill in the missing code in the following definition. The correct answer is short and fits here easily. You may also want to answer parts (b) and (c) first.

```
(define compose2
    (lambda (g h)
        (lambda (f xs)
            (g (lambda (xs) (________ )) xs )

)))
```

(b) When (compose2 maplist car) is evaluated, the result is a function defined by (lambda (f xs) (g . . .)) above, with

i. which function replacing g?

ii. and which function replacing h?

(c) We could also write the subexpression (lambda (xs) (. . .)) as (compose (. . .) (. . .)) for two functions. Which two functions are these? (Write them in the correct order.)

3.5 Definition of Garbage

This question asks you to think about garbage collection in Lisp and compare our definition of garbage in the text to the one given in McCarthy's 1960 paper on Lisp. McCarthy's definition is written for Lisp specifically, whereas our definition is stated generally for any programming language. Answer the question by comparing the definitions as they apply to Lisp only. Here are the two definitions.

Garbage, our definition: At a given point in the execution of a program P, a memory location m is garbage if no continued execution of P from this point can access location m.

Garbage, McCarthy's definition: "Each register that is accessible to the program is accessible because it can be reached from one or more of the base registers by a chain of car and cdr operations. When the contents of a base register are changed, it may happen that the register to which the base register formerly pointed cannot be reached by a car–cdr chain from any base register. Such a register may be considered abandoned by the program because its contents can no longer be found by any possible program."

(a) If a memory location is garbage according to our definition, is it necessarily garbage according to McCarthy's definition? Explain why or why not.

(b) If a location is garbage according to McCarthy's definition, is it garbage by our definition? Explain why or why not.

(c) There are garbage collectors that collect everything that is garbage according to McCarthy's definition. Would it be possible to write a garbage collector to collect everything that is garbage according to our definition? Explain why or why not.

3.6 Reference Counting

This question is about a possible implementation of garbage collection for Lisp. Both impure and pure Lisp have lambda abstraction, function application, and elementary functions atom, eq, car, cdr, and cons. Impure Lisp also has rplaca, rplacd, and other functions that have side effects on memory cells.

Reference counting is a simple garbage-collection scheme that associates a reference count with each datum in memory. When memory is allocated, the associated reference count is set to 0. When a pointer is set to point to a location, the count for that location is incremented. If a pointer to a location is reset or destroyed, the count for the location is decremented. Consequently, the reference count always tells how many pointers there are to a given datum. When a count reaches 0, the datum is considered garbage and is returned to the free-storage list. For example, after evaluation of (cdr (cons (cons 'A 'B) (cons 'C 'D))), the cell created for (cons 'A 'B) is garbage, but the cell for (cons 'C 'D) is not.

(a) Describe how reference counting could be used for garbage collection in evaluating the following expression:

```
(car (cdr (cons (cons a b) (cons c d))))
```

where a, b, c, and d are previously defined names for cells. Assume that the reference counts for a, b, c, and d are initially set to some numbers greater than 0, so that these do not become garbage. Assume that the result of the entire expression is not garbage. How many of the three cons cells generated by the evaluation of this expression can be returned to the free-storage list?

(b) The "impure" Lisp function rplac<u>a</u> takes as arguments a cons cell *c* and a value *v* and modifies *c*'s <u>a</u>ddress field to point to *v*. Note that this operation does *not* produce a new cons cell; it modifies the one it receives as an argument. The function rplacd performs the same function with respect the decrement portion of its argument cons cell.

Lisp programs that use rplaca or rplacd may create memory structures that cannot be garbage collected properly by reference counting. Describe a configuration of cons cells that can be created by use of operations of pure Lisp and rplaca and rplacd. Explain why the reference-counting algorithm deos not work properly on this structure.

3.7 Regions and Memory Management

There are a wide variety of algorithms to chose from when implementing garbage collection for a specific language. In this problem, we examine one algorithm for finding garbage in pure Lisp (Lisp without side effects) based on the concept of *regions*. Generally speaking, a region is a section of the program text. To make things simple, we consider each function as a separate region. Region-based collection reclaims garbage each time program execution leaves a region. Because we are treating functions as regions in this problems, our version of region-based collection will try to find garbage each time a program returns from a function call.

(a) Here is a simple idea for region-based garbage collection:

> When a function exits, free all the memory that was allocated during execution of the function.

However, this is not correct as some memory locations that are freed may still be accessible to the program. Explain the flaw by describing a program that could possibly access a previously freed piece of memory. You do not need to write more than four or five sentences; just explain the aspects of an example program that are relevant to the question.

(b) Fix the method in part (a) to work correctly. It is not necessary for your method to find all garbage, but the locations that are freed should really be garbage. Your answer should be in the following form:

> When a function exits, free all memory allocated by the function except. . . .

Justify your answer. (*Hint:* Your statement should not be more than a sentence or two. Your justification should be a short paragraph.)

(c) Now assume that you have an correctly functioning region-based garbage collector. Does your region-based collector have any advantages or disadvantages over a simple mark-and-sweep collector?

(d) Could a region-based collector like the one described in this problem work for impure Lisp? If you think the problem is more complicated for impure Lisp, briefly explain why. You may consider the problem for C instead of for impure Lisp if you like, but do not give an answer that depends on specific properties of C such as pointer arithmetic. The point of this question is to explore the relationship between side effects and a simple form of region-based collection.

3.8 Concurrency in Lisp

The concept of *future* was popularized by R. Halstead's work on the language Multilisp for concurrent Lisp programming. Operationally, a future consists of a location in memory (part of a cons cell) and a process that is intended to place a value in this location at some time "in the future." More specifically, the evaluation of (future e) proceeds as follows:

(i) The location ℓ that will contain the value of (future e) is identified (if the value is going to go into an existing cons cell) or created if needed.
(ii) A process is created to evaluate e.
(iii) When the process evaluating e completes, the value of e is placed in the location ℓ.
(iv) The process that invoked (future e) continues in parallel with the new process. If the originating process tries to read the contents of location ℓ while it is still empty, then the process blocks until the location has been filled with the value of e.

Other than this construct, all other operations in this problem are defined as in pure Lisp. For example, if expression e evaluates to the list (1 2 3), then the expression

```
(cons 'a (future e))
```

produces a list whose first element is the atom 'a and whose tail becomes (1 2 3) when the process evaluating e terminates. The value of the future construct is that the program can operate on the car of this list while the value of the cdr is being computed in parallel. However, if the program tries to examine the cdr of the list before the value has been placed in the empty location, then the computation will block (wait) until the data is available.

(a) Assuming an unbounded number of processors, how much time would you expect the evaluation of the following fib function to take on positive-integer argument n?

```
(defun fib (n)
        (cond ( (eq n 0) 1)
              ((eq n 1) 1)
              (T (plus (future (fib (minus n 1)))
                       (future (fib (minus n 2)))))))
```

We are interested only in time up to a multiplicative constant; you may use "big Oh" notation if you wish. If two instructions are done at the same time by two processors, count that as one unit of time.

(b) At first glance, we might expect that two expressions

```
( ... e ... )
( ... (future e) ... )
```

which differ only because an occurrence of a subexpression e is replaced with (future e), would be equivalent. However, there are some circumstances in which the result of evaluating one might differ from the other. More specifically, side effects may cause problems. To demonstrate this, write an expression of the form (... e ...) so that when the e is changed to (future e), the expression's value or behavior might be different because of side effects, and explain why. Do not be concerned with the efficiency of either computation or the degree of parallelism.

(c) Side effects are not the only cause for different evaluation results. Write a pure Lisp expression of the form (... e′ ...) so that when the e′ is changed to (future e′), the expression's value or behavior might be different, and explain why.

(d) Suppose you are part of a language design team that has adopted futures as an approach to concurrency. The head of your team suggests an error-handling feature called a *try block*. The syntactic form of a try block is

```
(try e
    (error-1 handler-1)
    (error-2 handler-2)
    . . .
    (error-n handler-n))
```

This construct would have the following characteristics:

i. Errors are programmer defined and occur when an expression of the form `(raise error-i)` is evaluated inside `e`, the main expression of the try block.

ii. If no errors occur, then `(try e (error-1 handler-1) . . . )` is equivalent to `e`.

iii. If the error named `error-i` occurs during the evaluation of `e`, the rest of the computation of `e` is aborted, the expression `handler-i` is evaluated, and this becomes the value of the try block.

The other members of the team think this is a great idea and, claiming that it is a completely straightforward construct, ask you to go ahead and implement it. You think the construct might raise some tricky issues. Name two problems or important interactions between error handling and concurrency that you think need to be considered. Give short code examples or sketches to illustrate your point(s). (*Note:* You are not being asked to solve any problems associated with futures and try blocks; just identify the issues.) Assume for this part that you are using pure Lisp (no side effects).