Principles of Programming Language

[BE SE-6th Semester]

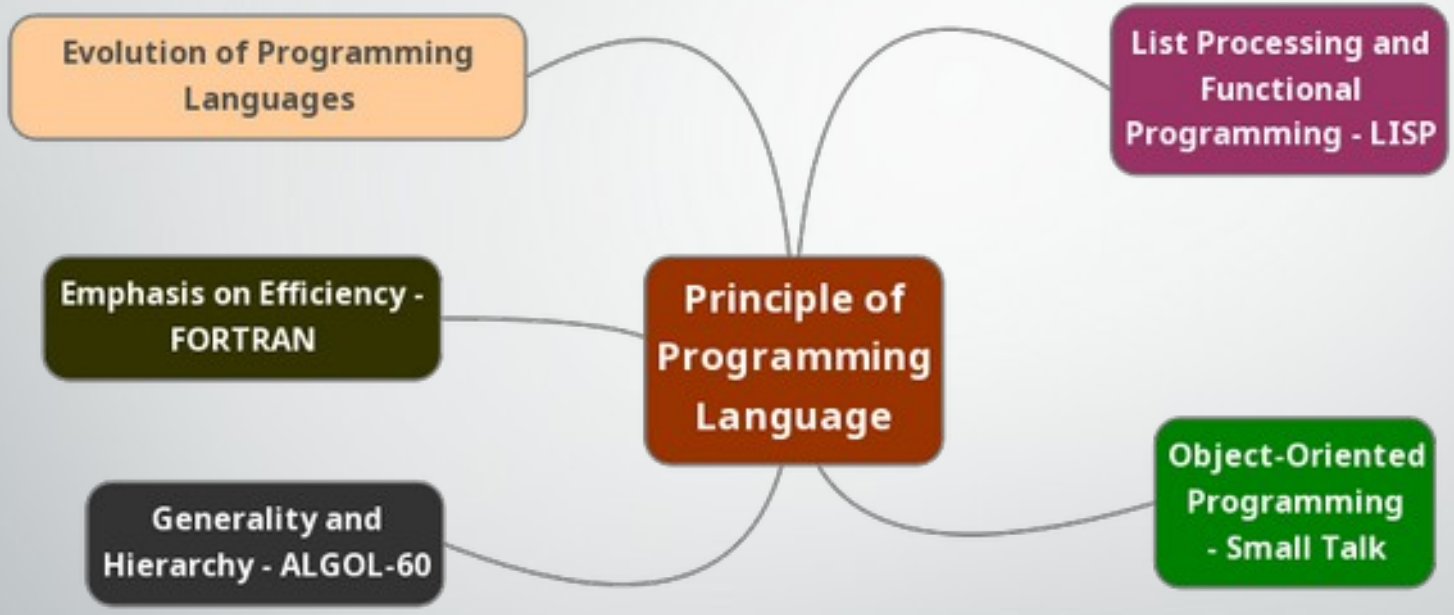
Rishi K. Marseni

Textbook:

Principles of programming languages: design, evaluation, and implementation.

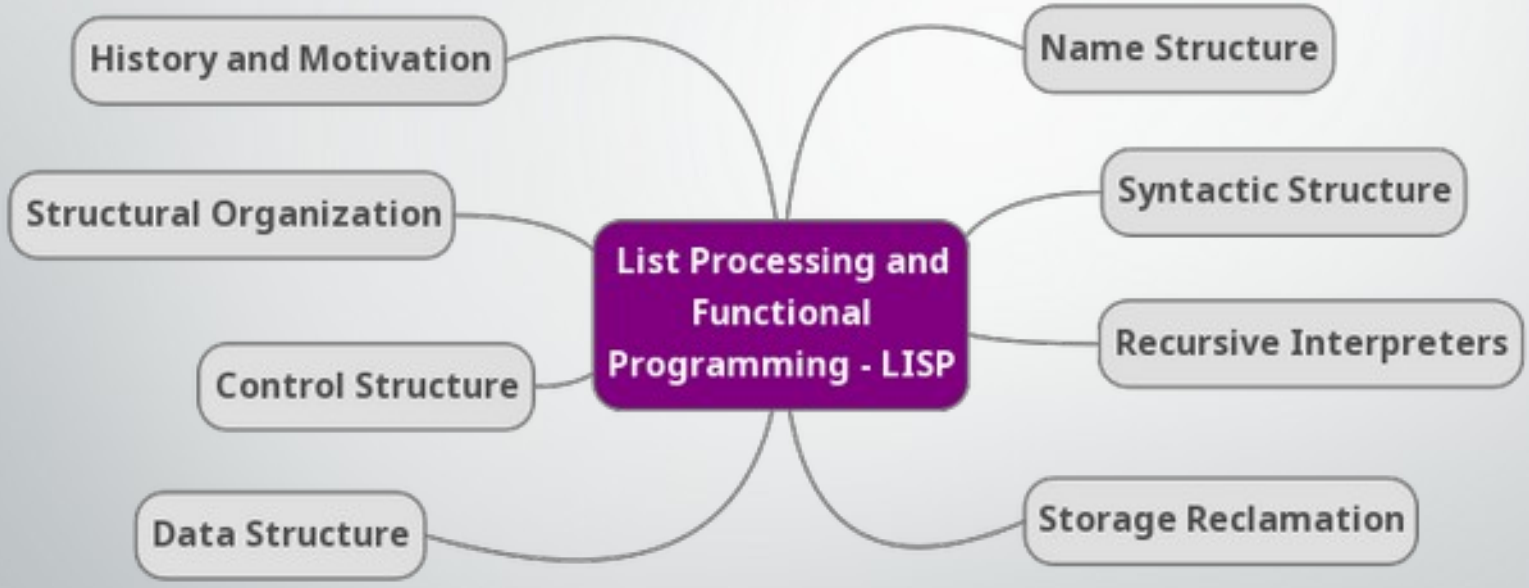
Author: Bruce J. MacLennan

Principle of Programming Language



List Processing and Functional Programming

LISP



Functional Programming Introduction

Functional Programming (FP)

- Although the imperative style of programming has been found acceptable by most programmers, its heavy reliance on the underlying architecture is thought by some to be an unnecessary restriction on the alternative approaches to software development.
- The functional programming paradigm, which is based on mathematical functions, is the design basis of the most important non-imperative styles of languages.
- In his Turing Award lecture, Backus (1978) made a case that purely functional programming languages are better than imperative languages because they result in programs that are more readable, more reliable, and more likely to be correct.
- Examples of FP languages are LISP, Scheme, ML, Haskell, OCaml, and F# etc.

Mathematical Functions

A mathematical function is a **mapping** of members of one set, called the **domain set**, to another set, called the **range set**.

A **function definition** specifies the domain and range sets along with the mapping. The mapping is described by an **expression** or a **table**.

The evaluation order of mapping expressions is controlled by

recursion and **conditional** expressions.

They always map a **particular element** of the domain to the **same element** of the range.

In mathematics, there is **no such thing as a variable** that models a memory location; there is **no concept of the state of a function**.

A mathematical function **maps its parameter(s) to a value (or values)**, rather than specifying a sequence of operations on values in memory to produce a value.

Functions in FP

Function definitions are written as a **function name**, followed by a **list of parameters** in parentheses, followed by the **mapping expression**.

cube(x) Ξ x * x * x, where x is a real number (Ξ is called “defined as”)

The parameter **x is bound to a value** (say 2.0) during the evaluation and there are **no unbound parameters**. Furthermore, **x is a constant** (its value cannot be changed) during the evaluation.

Lambda notation, as devised by Alonzo Church (1941), provides a method for defining **nameless functions**.

λ(x)x * x * x

Church defined a formal computation model (a formal system for function definition, function application, and recursion) using lambda expressions. This is called **lambda calculus**. **Untyped lambda calculus** serves as the inspiration for the functional programming languages.

Function is said to be applied to a value(s) when it is evaluated: -

(λ(x)x * x * x)(2)

Functional Forms

A **higher-order function**, or functional form, is one that either **takes one or more functions as parameters** or **yields a function** as its result, or both.

A **function composition** has **two functional parameters** and yields a function whose value is the first actual parameter function applied to the result of the second.

Function composition is written as an expression, using ° as an operator, as in **h Ξ f ° g**

if **f(x) Ξ x + 2** and **g(x) Ξ 3 * x** then **h** is defined as

h(x) Ξ f(g(x)), or **h(x) Ξ (3 * x) + 2**

Apply-to-all is a functional form that takes a single function as a parameter.

History and Motivation

The Fifth Generation Comprises Three Overlapping Paradigms

- functional programming
- object-oriented programming
- logic programming

Lisp is for functional programming.

The Desire for an Algebraic List-Processing Language

- Lisp is developed in the 1950s for artificial intelligence programming.
- In these applications complex interrelationships among data must be represented.
- Natural data structures are:
  - Pointers
  - Linked lists

The Desire for an Algebraic List-Processing Language

In the 1950s Newell, Shaw, and Simon (at Carnegie institute of technology and the Rand Corporation) developed many of the ideas of list processing in the IPL family of programming languages.

these ideas included the linked representation of list structures and the use of a stack (specifically, a push-down list) to implement recursion.

The Desire for an Algebraic List-Processing Language

- In the summer of 1956, the first major workshop on artificial intelligence was held at Dartmouth.
- At this workshop john McCarthy, then at MIT, heard a description of the ipl2 programming language.
- which had a low-level pseudo-code, or assembly-language-like syntax.
- McCarthy realized that an algebraic list-processing language, on the style of the recently announced FORTRAN I system, would be very useful.

FLPL Was Based on Fortran

- That summer Gerlernter and Gerberich of IBM were working, with the advice of McCarthy, on a geometry program.
- As a tool they developed FLPL, the Fortran list-processing language, by writing a set of list-processing subprograms for use with Fortran programs.
- One result of this work was the development of the basic list-processing primitives that were later incorporated into lisp.

McCarthy Developed the Central Ideas of LISP

- FORTRAN I had only one conditional construct—the arithmetic IF-statement.
- This construct was very inconvenient for list processing.
- which led McCarthy, in 1957, to write an IF function with three arguments:

X = IF (N .EQ. 0, ICAR(Y) , ICDR(Y))

If N were zero, then X = ICAR(Y) else X = ICDR(Y)

McCarthy Developed the Central Ideas of LISP

- An important consequence of this invention was that it made it feasible to compose IF function and the list-processing functions to achieve more complicated actions.
- Since FORTRAN does not permit recursive definitions, it became apparent that a new language was needed.
- In 1958 McCarthy began using recursion in conjunction with conditional expression in his definition of list-processing functions.

The Lisp List-Handling Routines Were Developed First

- in the fall of 1958, implementation of a LISP system began.
- One important component of this was a set of primitive list-handling subroutines for the lisp run-time environment.
- These were the first parts of the system that were implemented.
- The original intention was to develop a compiler like FORTRAN.
- Therefore, to gain experience in code generation, a number of LISP programs were hand compiled into assembly language.

A LISP Universal Function Resulted in an Interpreter

- McCarthy became convinced that recursive list-processing functions with conditional expressions formed an easier-to-understand basis for the theory of computation than other formalisms such as Turing machines.
- Recursive Functions of Symbolic Expressions and Their Computation by Machine.

A LISP Universal Function Resulted in an Interpreter

- He defined a universal LISP function that could interpret any other lisp function.
- he wrote a LISP interpreter in LISP.
- Since manipulates only lists, writing a universal function required developing a way of representing LISP program as list structures.

A LISP Universal Function Resulted in an Interpreter

- For example the function call

f [x+y; u*z]

- Would be represented by a list whose first element is ‘f’ and whose second and third element are the lists representing ’x+y’ and ‘u*z’. in LISP this list is written as

(f (plus x y) (times u z))

A LISP Universal Function Resulted in an Interpreter

- The algol-like notation (e,g., f[x+y; u*z]) is called M-*expressions* (M for meta-language).
- The list notation is called S-*expressions* (S for symbolic language)

LISP Became Widely Used in Artificial Intelligence

- The first implementation of LISP was on IBM 704
- LISP systems rapidly spread to other computers, and they now exist on virtually all machines, including microcomputers.
- LISP has become the most widely used programming language for artificial intelligence and other symbolic applications.
- LISP was standardized after a period of divergent evolution

DESIGN: STRUCTURAL ORGANIZATION

- **An Example LISP Program (in the S-expression)**

```
(defun make-table (text table)
    (if (null text)
          table
             (make-table (cdr text)
                (update-entry table (car text))
             )
       )
 )
```

Function Application Is the Central Idea

programming language are often divided into two classes:

- ***Imperative language***

  Depend heavily on an assignment statement and a changeable memory for accomplishing a programming task.

- ***Applicative language***

  The central idea is function application, that is, applying a function to its argument .

Function Application Is the Central Idea

- In the LISP almost everything is a function application

(f a1 a2…an)

- where f is the function and a1, a2, …, an are the arguments. This notation is called *Combridge Polish (prefix notation)*.

Example:

(plus 2 3)

(plus 10 5 8 64)

- LISP is *fully* parenthesized
- Functions cannot be mixed because of the list structure

Example

(set ‘Freq (make-table text nil))

- is a nested function application : set is applied to two arguments:
- Freq
- the result of applying make-table to the arguments text and nil.
- In an Algol-like language this would be written

  set (Freq, make-table (text, nil))

Example

```
(cond
   ((null  x)  0)
   ((eq  x  y)  (f  x))
   (t  (g  y))  )
```

- In an Algol-like language this would be written

```
If        null(x)   then  0
elseif   x = y  then   f(x)
else     g(y)    endif
```

function definition is accomplished by calling a function, *defun*, with three arguments: the name of the function, its formal parameter list, and its body.

```
(defun make-table (text table)
    (if (null text)
          table
             (make-table (cdr text)
                 (update-entry table (car text))
              )
        )
 )
```

Why is everything a function application in LISP?

Simplicity Principle

- If there is only one basic mechanism in a language, the language is easier to learn, understand, and implement.

The List is the Primary Data Structure Constructor

- We said that one of LISP's goals was to allow computation with Symbolic data.
- This is accomplished by allowing the programmer to manipulate lists of data.
- **Example:**

  (set ‘text ‘ (to be or not to be))

  the second argument to set is the list

  (to be or not to be)
- This list above is composed of four distinct atoms:

  to be or not

The List is the Primary Data Structure Constructor

- LISP manipulates list just like other languages manipulate number;
- they can be compared,
- passed to functions,
- put together,
- and taken apart.

Programs Are Represented as Lists

- Function application and lists look the same.

  (make-table text nil)

- could either be a three-element list whose elements are the make-table, text, nil; or it could be an application of the function make-table to the arguments named text and nil.
- Because a LISP program is itself a list.
- If the list is quoted, then it is treated as data; that is, it is unevaluated.

Programs Are Represented as Lists

- If the list is quoted, then it is treated as data; that is, it is unevaluated.
- Example:

  (set ‘text ‘ (to be or not to be))

  (set ‘text (to be or not to be))

What happens?

Unquoted ones are interpreted

(i.e. to is also treated as a function)

the fact that LISP represents both programs and data in the same way is of the utmost importance:

- it makes it very easy to write a LISP interpreter in LISP.
- it makes it convenient to have one LISP program generate and call for the execution of another LISP program.
- It also simplifies writing LISP programs that transform and manipulate other LISP programs.
- These capabilities are important in artificial intelligence and other advanced program-development environments.
- These amplification come with a reduction of readability.

Implications?

- If programs are lists
  - and data is also list
  - then we can generate a list that can be interpreted as a program
- In other words
  - We can write a program to write and execute another program
  - Useful in artificial intelligence
- Reductive aspects?

LISP is often Interpreted

- Most LISP systems provide interactive interpreters. We interact with the LISP interpreter by typing in function applications. The LISP system then interprets them and point out the result.
- Example:

  (plus 2 3)

  the system will respond

  5
- Similarly if we type

  (eq (plus 2 3) (difference 9 4))

  the system will respond

  t

Pure Function

- functions like *eq* and *plus* are pure functions (or simply functions) because they have no effect other than the computation of a value.
- Pure functions obey the Manifest Interface Principle because their interfaces are apparent (manifest).

The Manifest Interface Principle

All interfaces should be apparent (manifest) in the syntax.

Procedures or Pseudo-Functions

- Some function in LISP are pseudo-function (or procedure). These are functions that have a side effect on the state of computer in addition to computing a result.
- Example:

  (set ‘text ‘(to be or not to be))
- binds the name (atom) text to the (to be or not to be) that it is a list and return this list.
- The atom n can now be used as a name for this list in any expression, for example

  (set ‘Freq (make-table text nil))

Function

- another important pseudo-function is defun, which define a function.
- Example:

  (defun f (x1,x2, …, xn) b)

- defines a function with the name f, formal parameters x1, x2, … , xn; and body b
- **The function application used to define functions is different in many dialects.**
- **in LISP the binding process is dynamic, that is, it takes place at run-time.**

DESIGN: DATA STRUCTURE

The Primitives Include Numeric Atoms

- We classify LISP data structure (like other languages) into ***primitive*** and ***constructors***.
- The principal ***constructor*** is the list: it permits more complicated structures to be built from simpler structures.
- The primitive data structures are the starting points to building process. Thus, the primitives are those data structure that are not build from any other: they have no parts. It is for this reason that they are called atoms.

Primitive Data Structures

There are at least two types of atoms in all LISP systems:

Numeric

- Operations: plus, minus, times, eq, etc.

Nonnumeric atoms

- Strings of characters used as symbols
  - Much like enumerated types in Pascal
  - Not used as strings
- Operations: eq
- Special atoms
  - t: true
  - nil: false; non-existent atom; empty list

Primitive Data Structures

- LISP provides a very large set of primitive functions for manipulating numeric atoms:
- Arithmetic operations (plus, difference, etc.)
- Predecessor and successor function (sub1, add1)
- Maximum and minimum functions
- Relation tests (equal, lessp, greaterp)
- Predicates (i.e., tests, such as zerop, onep, minusp)

Primitive Data Structures

- All of these functions take both integer and floating-point (and in some systems, multiple-precision) arguments and return results of the appropriate type.

Example

- LISP's use of Combridge Polish limits its numerical applications.
- Example:

  $(-b + \sqrt{(b^2 - 4ac)}) / 2a$

- must be written

```
(quotient  (plus  (minus  b)
                  (sqrt  (difference  (expt  b  2)
                                      (times  4  a  c))))
           (times  2  a)
```

Nonnumeric Atoms Are Also Provided

- These atoms are strings of characters that were originally intended to represent words or symbols.

Example

- With few exceptions, the only operations that can be performed on nonnumeric atoms are comparisions for equality and inequality. This is done with function **eq:**
- Example:

(**eq** x y)

t if x and y are the same atom,

nil if x and y are not the same atom,

Example

- the atom nil has many uses in LISP.
- This operation is often used as a base for recursive definitions.
- (eq x nil)
- (null x)
- nil is the noun and null is the corresponding adjective.

Non-primitive or Abstract Data Strucutes

- Some LISP systems provide additional types of atoms, such as strings.
- In these cases special operations for manipulating these values are also provided

Abstract Data Type

- **An abstract data type is a set of data values together with a set of operations on those values.**

The Principal Constructor is the List

- The characteristic method of data structuring provided by LISP is called the *list*.
- Lists are written in the S-expression notation by surrounding with parentheses the list's elements, which are separated by blanks.
- Lists can have none, one, or more elements, so they satisfy the Zero-One-Infinity Principle. The elements of lists can themselves be lists, so the Zero-One-Infinity Principle is also satisfied by the nesting level of lists.

The Principal Constructor is the List

- The empty list, ( ) is considered equivalent to the atom *nil:*
- (eq ‘( ) nil)

  or
- (null ‘( ))
- Except the null list all lists are non-atomic, they are sometimes called composite data values.

- We can find out whether or not something is an atom by using the atom predicate.
- Example:
- (atom ‘to)

  t
- (atom (plus 2 3))

  t
- (atom nil)

  t
- (atom ‘(to be))

  nil
- (atom ‘(( )))

  nil

Notice neither (( )) nor (nil) is the null list

Constructor and Selector

- Operations that build a structure are called constructors.
- Operation that extract their parts are called selectors.

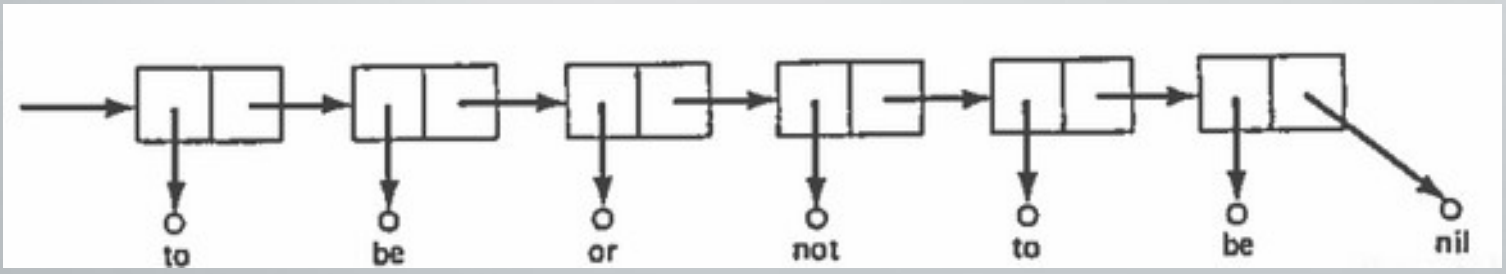
Car and Cdr Access the Parts of Lists

- Car : return the first element of the list.
- Cdr : returns all of a list except its first element.
- Example: (car ‘(to be or not to be))

  returns the atom -> to

  (cdr ‘(to be or not to be))

  returns the list -> (be or not to be)



Car and Cdr Access the Parts of Lists

- Notice that the argument to car and cdr is always a nonnull list.
- Unlike car, cdr always returns a list.
- Car and Cdr are pure functions.
- The easiest way to think of the way they work is that they make a new copy of the list.

Car and Cdr can be used in combination to access the components of a list

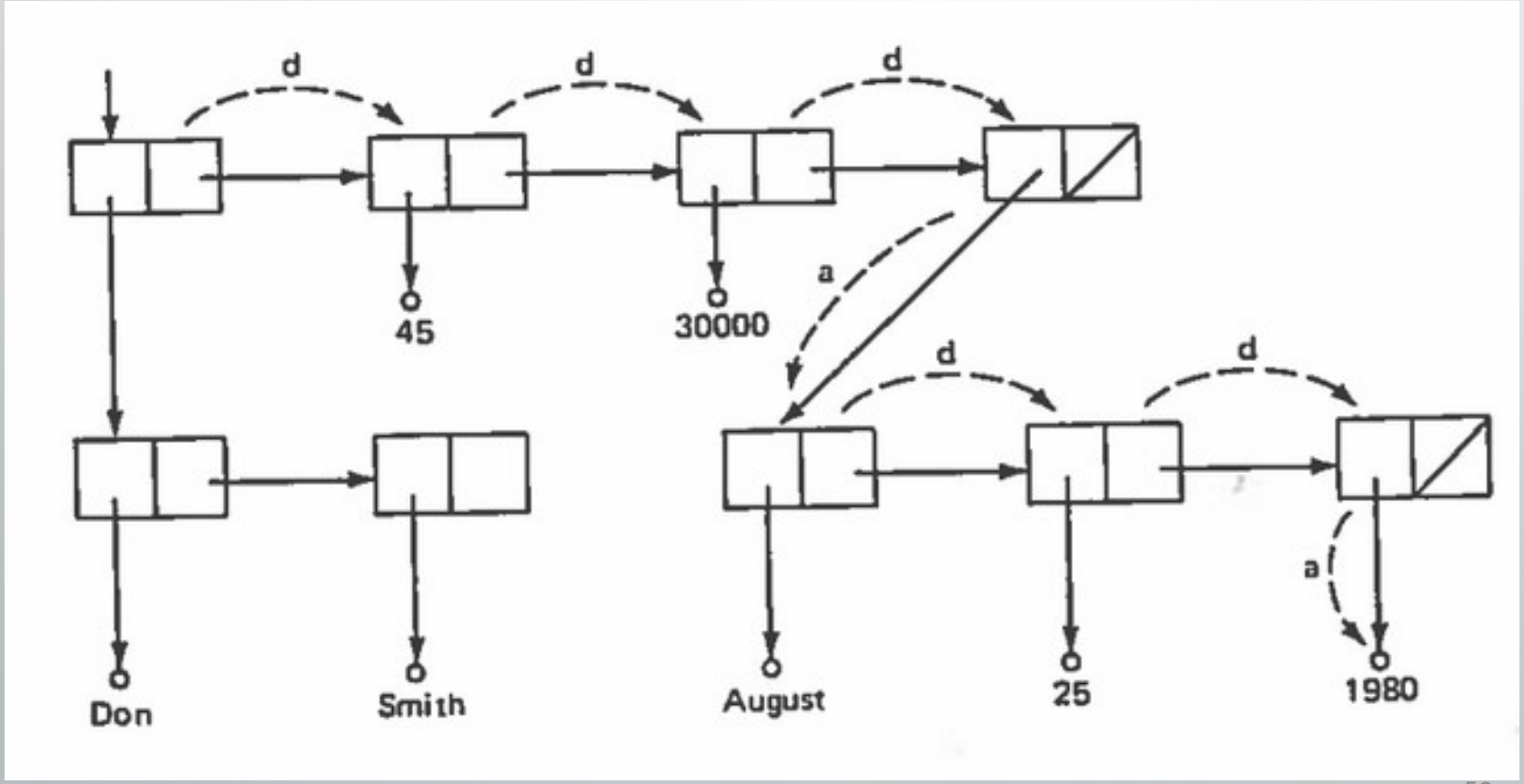
- Example:

  (set ‘DS ‘(Don Smith) 45 30000 (august 25 1980)))

- (car (cdr (cdr (cdr DS))))
- returns the list
- (august 25 1980)

Car and Cdr can be used in combination to access the components of a list

(set 'DS '(Don Smith) 45 30000 (august 25 1980)))



Car and Cdr can be used in combination to access the components of a list

- In general, the nth element of a list can be access by n-1 cdrs followed by a car.
- The composition of cars and cdrs is represented by the sequence of ‘a’s and ‘d’s between the initial ‘c’ and the final ‘r’.
- This can be seen more clearly if the list is written as a linked data structure; then a ‘d’ moves to the right and an ‘a’ moves down.

Car and Cdr can be used in combination to access the components of a list

- Clearly, these sequence of ‘a’s and ‘d’s can become quite complicated to read. Writing them is also error-prone.
- One solution to this is to write a set of special-purpose functions for accessing the part of a record. For example a function for accessing the hire date could be defined as
- **(defun hire-date (r) (cadddr (r))**
- Then (hire-date DS) returns Don Smith hire date

Information Can Be Represented by Property Lists

- A personnel record would not probably be represented in LISP in the way we have just described: it is too inflexible.
- A better arrangement is to precede each information with an indicator identifying the property.
- Example:
- (name (Don Smith) age 45 salary 30000 hire-date (august 25 1980))
- this method of representing information is called a *property list* or p-list.

  (P1 v1 P2 v2 … Pn vn)
- each Pi is the indicator for a property and each vi is the corresponding property value

How can the properties of a p-list be accessed?

```
(defun  getprop  (p   x)
    (if  (eq  (car   x)  p)
     (cadr   x)
     (getprop    p  (cddr   x))   ))
```

- Example:

  (getprop ‘name DS)

- Return (Don Smith)

- What will this function do if we ask for a property that is not in the property list?
- (getprop ‘weight DS)
- Error: car of nil

- One way to do this is to have getprop return a distinguished value if the property does not exist.
- An obvious choice is nil,
  - but this would not be a good choice since it would then be impossible to distinguish an undefined property from one that is defined but whose value is nil.

- A better decision is to pick some atom, such as undefined-property, which is unlikely to be used for any other purpose.
- (getprop ‘weight DS)

  undefined-property

Information Can Be Represented in Association List

- The property list data structure works best when exactly one value is to be associated with each property.
- Some properties are flags that have no associated value for example retired flag in our personnel record.
- if a property has several associated values, for example the *manages* property, might be associated with the names of everyone managed by Don Smith.

- These problems are solved by another common LISP data structure, the *association list*, or *a-list*.
- An *a-list* is a list of pairs.
- The general form of an *a-list* is a list of attribute-value pairs:

  ((a1 v1) (a2   v2)…(an   vn))

((name (Don Smith)

  (age 45)

  (salary 30000)

  (hire-date (August 25 1980)))

Assoc

- The function that does the forward association.
- For example

  (assoc ‘hire-date DS)
- Return **(August 25 1980)**

Cons Constructs Lists

- If the constructors undo what the selectors do and vice versa, we will have more regular data types.
- cons adds a new element to the beginning of a list.
- cons is the inverse of car and cdr

- Example:

  (cons ‘to ‘(be or not to be))
- return the list

  (to be or not to be)
- (car ‘(to be or not to be)) = to
- (cdr ‘(to be or not to be)) = (be or not to be))
- (cons ‘to ‘(be or not to be)) = (to be or not to be)

Quiz:

(cons ‘(a b) ‘(c d))

- Notice that the second argument of cons must be a list.
- Cons is a pure function.

Lists Are Usually Constructed Recursively

Append

- (append M L) will return the concatenation of the lists L and M.
- (append ‘( ) L) = L
- (append L ‘( )) = L
- (append ‘(b c) ‘(d e f)) = (b c d e f)

Atoms Have Properties

- LISP was originally developed for artificial intelligence application.
- AI applications often must deal with
  - the properties of objects and
  - the relationship among objects.

- In LISP,
  - objects are represented by atoms,
  - each atom has an associated p-list
    - represents the properties of the atom and
    - the relationship in which it participates.

- (set ‘Europe ‘(England France Spain Germany …))
- >> (England France Spain Germany …)

- (car Europe)
- >> England

- (eq ‘England (car Europe))
- >> t

- (eq ‘France (car Europe))
- >> nil

How are other properties attached to an atom?

- Example:

  (putprop ‘France ‘Paris ‘capital)

Return ***Paris***

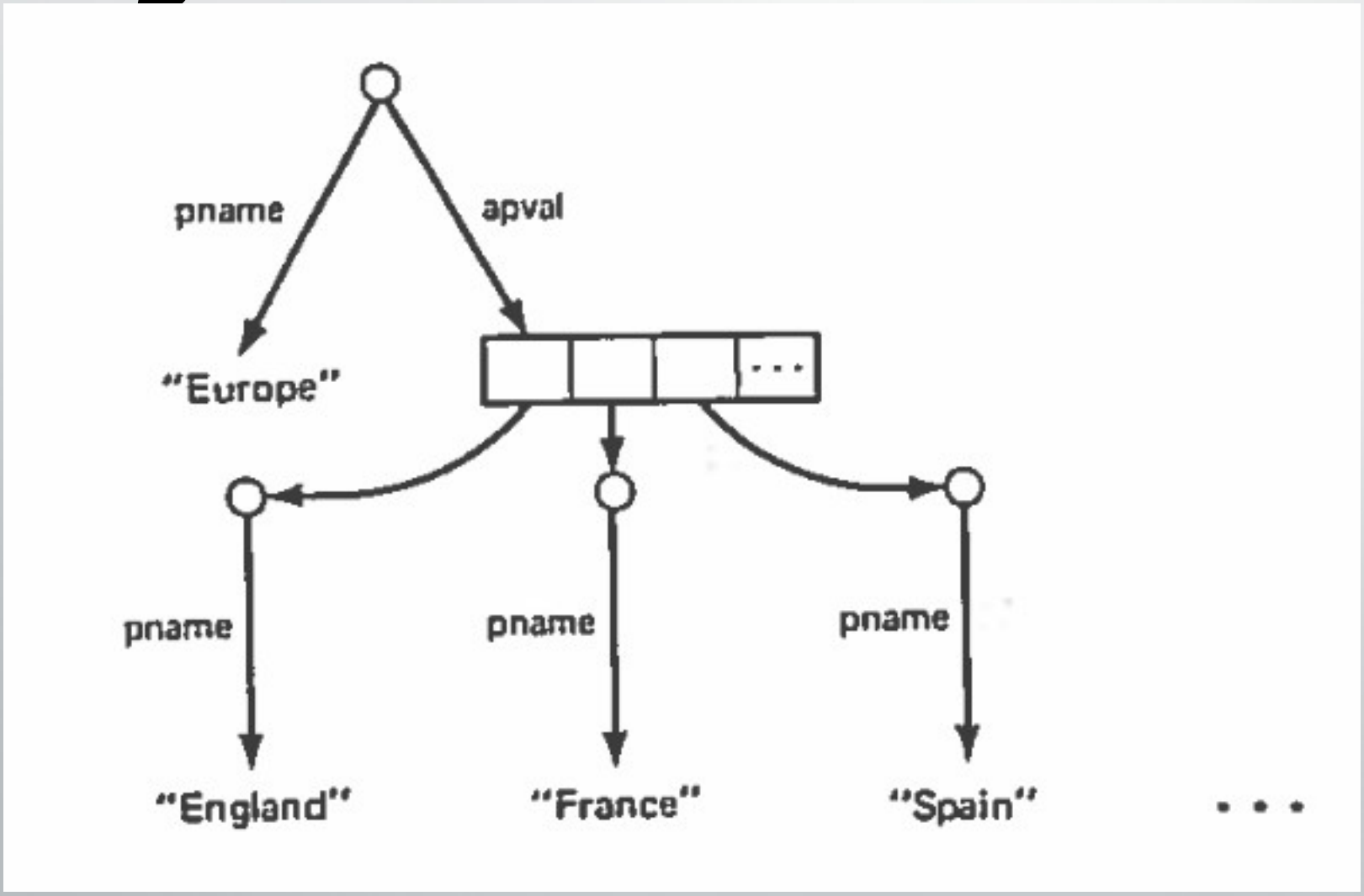
- We can find out the value of a property with the *get* function:

  (get ‘France ‘capital)

Return ***Paris***

- ***Figure 9.7***

Figure 9.7



- Each object comes complete with a property, its *print name*, which is a character string tagged by pname.

Atoms are pointers to their property lists

- In implementation atoms are equivalent to memory locations

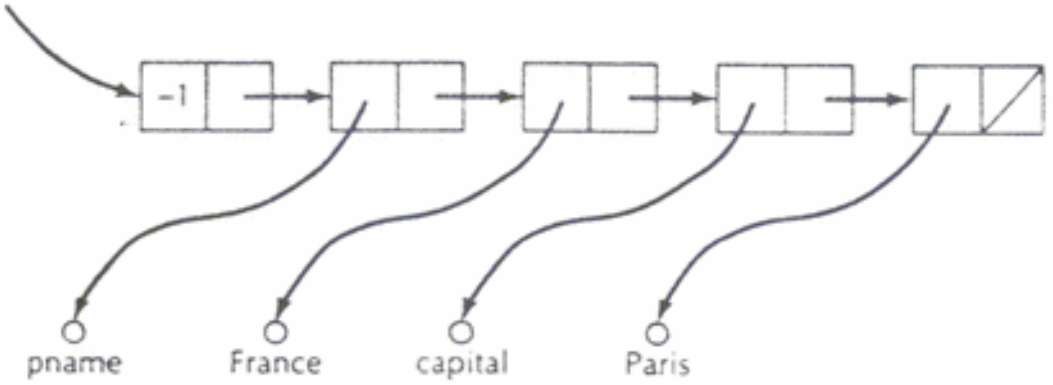


Figure 9.13 Representation of the Atom `France`

- Binding atoms to lists:

(Set ‘Europe ‘(England France …)

(get ‘Europe ‘apval)

(England France Spain …)

- Binding of atoms to functions is represented by the *expr* property

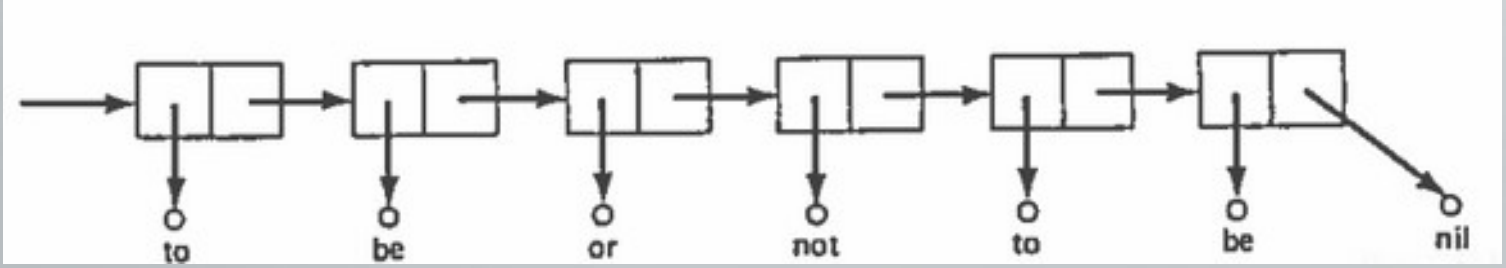
(Get ‘getprop ‘expr)

{function expr)

- Defun equivalent to putprop

Lists Have a Simple Representation

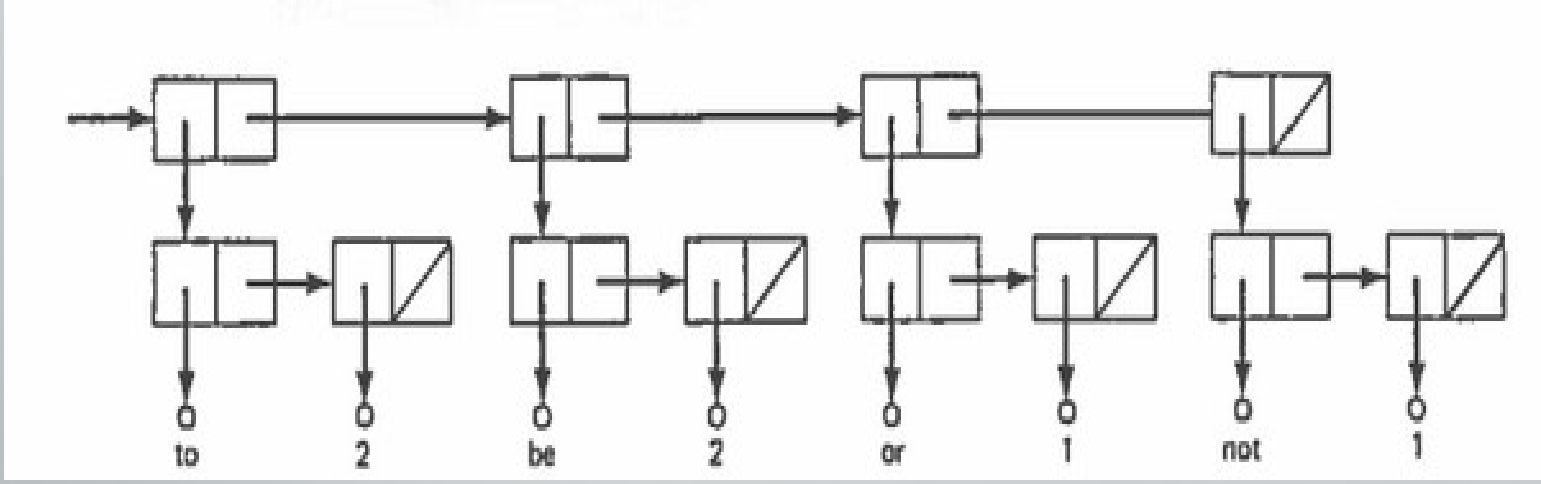
- LISP lists are usually implemented as linked structure. For instance, the list
- (to be or not to be)
- is represented as
  - a sequence of six cells in memory,
  - each containing a pointer to the next.
  - Each cell also contains a pointer to the element of the list it represents.



- List containing other lists are represented in the same way. For example, the list

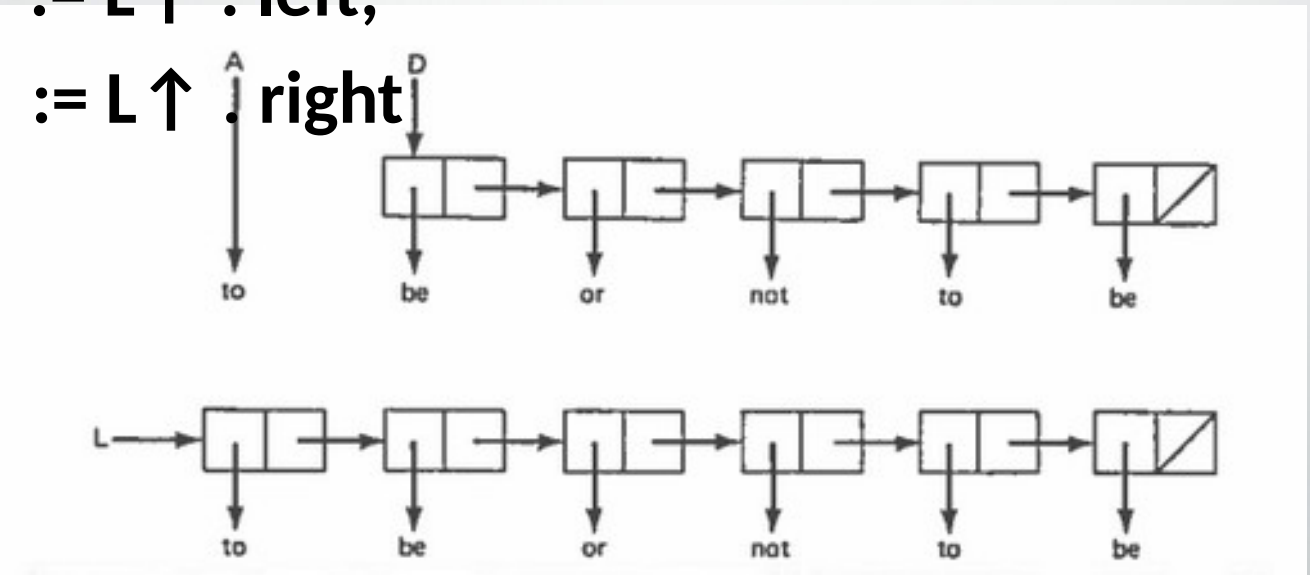
  ((to 2) (be 2) (or 1) (not 1))

- would be represented in storage as shown in below.



The List Primitive Are Simple and Efficient

- **Car A := L↑ . left;**
- **Cdr D := L↑ . right**



- Cons

  new(L);

  L↑ . left : = A;

  L↑ . right := D;

Shared sub-lists (Quiz)

(set ‘L ‘(or not to be))

>> (or not to be)

(set ‘M ‘(to be))

>> (to be)

(cadr M)

>> be

(set ‘N (cons(cadr M) L))

>>…

(car M)

>…

(set ‘O(cons(car M) N))

>…

(cons (car M) (cons (cadr M)L))

>…

Sublists can be shared

- A lot of the substructures can be shared.
- Memory efficiency
- Danger of Aliasing (and sharing of data structures) only when
  - ***There is the ability to change the content of memory***
- Because car,cdr,cons are pure functions, they have no side effect on memory, so they are danger free

Lists can be modified

- Pseudo-function
  - rplaca : replace address part
  - rplacd : replace decrement part

  - (rplaca L A)
    - L ^ .left := A;
  - (rplacd L D)
    - L^ .right := D;

Figure 9.12

(set ‘text ‘(to be or not to be))
>> (to be or not to be)
((rplacd (cdr text) ‘(is all))

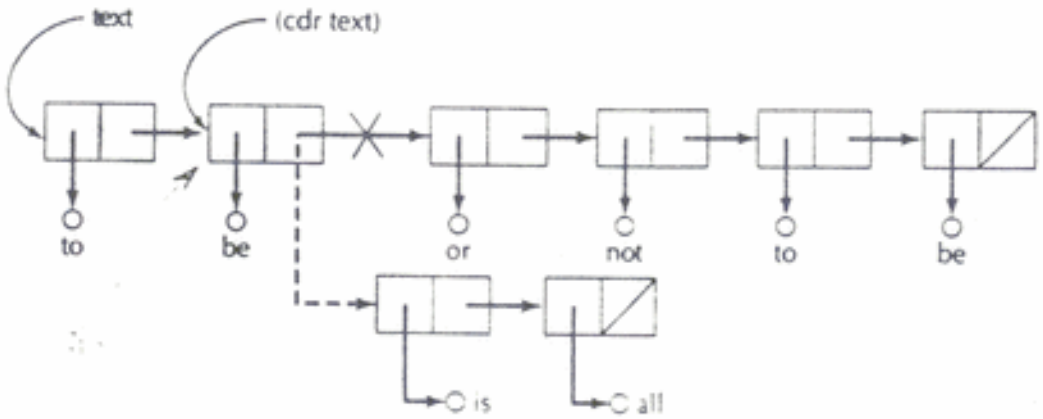


Figure 9.12 Execution of `(rplacd (cdr text) '(is all))`

List Structures Can Be Modified

- rplaca (replace address part)
- rplacd (replace decrement part)

```
(set 'text '(to be or not to be))
(to be or not to be)
( set 'x (cdr text))
      (be or not to be)
(rplacd x '(is all))
      (be is all)
text
      (to be is all)
```

- Danger of unwanted memory change

Iteration by Recursion

- Iteration is done by recursion
- Iteration is mostly needed to perform an operation on every element of a list
- This can be done using combination of

  testing for end of list, operating on first element, and

  recursing on rest of the list

  (defun plus-red (a)

  (if (null a) nil

  (plus (car a) (plus-red (cdr a)))))!
- Notice: No array bounds are needed! Function is very genera

Iteration = Recursion

- Theoretically, recursion and iteration have the same power, and are equivalent
- One can be translated to the other (although may not be practical)
- Recursion → iteration(Use iteration and keep track of auxiliary information in an explicit stack)
- Iteration → recursion(Need to pass control information (variables)

Storage Reclamation

- What happens to cons'd pointers that are no longer in use?
- Explicit reclamation is the obvious / traditional way
  - C: malloc, calloc, realloc, free
  - C++: new, delete
  - Pascal: new, dispose
- Issues
  - Complicates programming(Requires the programmer to keep track of pointers)
  - Violates security of the environment(Memory freed, but still referenced (dangling pointers))

Automatic Storage Reclamation

- It would be nice for the system to automatically ‘reclaim’ storage no longer used
- System can keep track of number of references to storage

  – When references decrease to 0, storage is returned to ‘free-list’
- Advantage:

  – Storage reclaimed immediately as last reference is destroyed
- Disadvantage:

  – Cyclic structures (points to itself) cannot be reclaimed

Garbage Collection

- A different approach is garbage collection
  - Do not keep track of references to location
  - When last reference is destroyed, we still do not do anything, and leave the memory as garbage

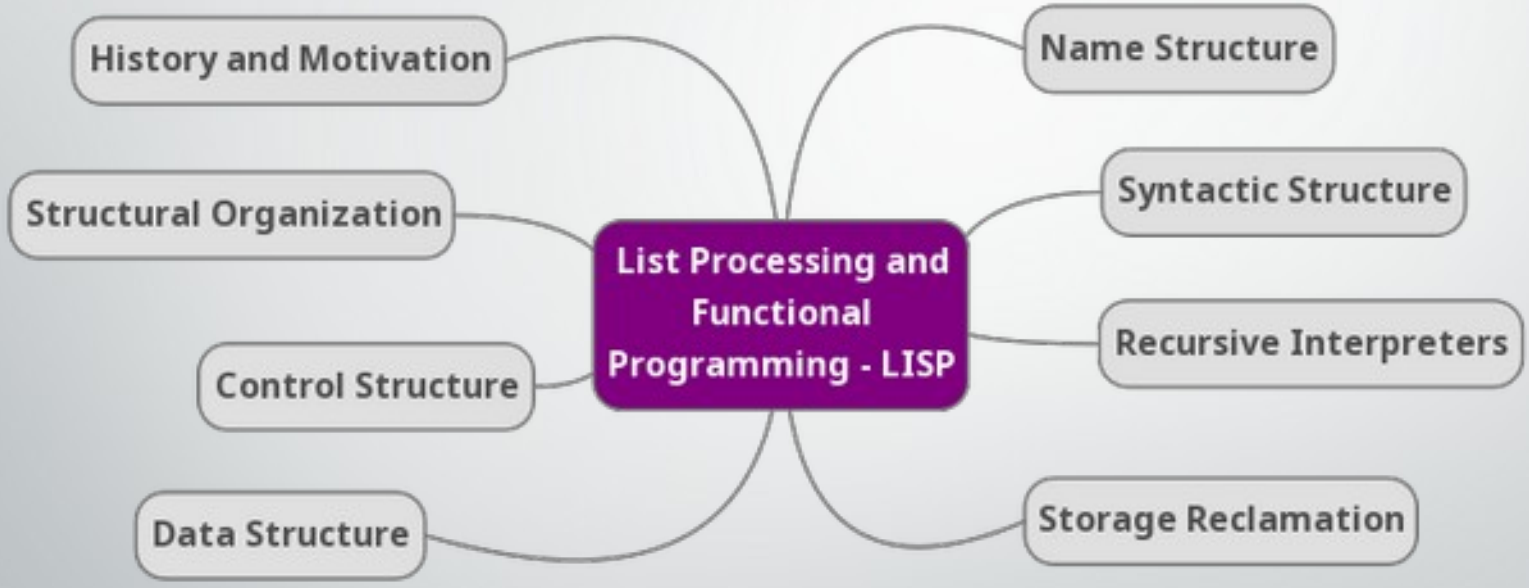
    (unused, non-reusable storage, littering the memory)
  - Collect garbage if system runs out of storage
- Mark all areas unused
- Then examine all visible pointers and mark storage they point to as ‘used’
- Leftover is garbage, and can be put on free-list
- This is called the mark-and-sweep method

Garbage Collection

- Advantages
  - Fast until runs out of memory
  - No additional memory is needed for tracking references
- Disadvantages
  - Garbage collection itself can be slow
- If memory is large, and have many references
- Must halt entire system, since all dynamic memory must be marked as unused first
- Java uses this approach

List Processing and Functional Programming

LISP



Principle of Programming Language