LISP Characteristics

1.1

- Programs consist of (recursive) functions opposed to statements
- Programs and data are in the form of symbolic expressions called

  S-expressions

- There are no explicit type declarations or storage models in pure LISP
- LISP manipulates whole structures not just "a-word-at-a-time". This allows short, clear programs
- LISP programs can generate and execute new LISP programs, dynamically
- LISP comes from the lambda calculus; the mathematical foundations promotes resoning about programs.

1.2

LISP: A LITTLE HISTORY

- **Developed by McCarthy in the late 1960s**
- **First functional language**
- **Motivated by artificial intelligence work and its special need for symbolic processing**
- **Linked lists and associated operations**
- **Programs & data are syntactically the same (S-expressions)**
- **Pure Lisp isn't "practical"; impure Lisps are used extensively in academia and industry**
- **There are many Lisp dialects, including Lisp 1.5, MAC Lisp, Zeta-Lisp, Franz Lisp, UCI Lisp, Interlisp, Scheme, etc.**
- **We use Common Lisp, a standard Lisp with several implementations.**
- **Common Lisp has some significant differences from other Lisps.**

1.3

S-Expressions

- S-expressions are **atoms** or **lists.**
- Atoms can be
  numbers: 6, -27, 89.523
  or
  symbols: A, B, C, smurf, T, NIL
- A list is a (possibly empty) set of S-expressions inside matching left or right parentheses.
- **(A B C)**
- **(THIS IS A SENTENCE)**
- **(+ (* 2 3) (* 5 6))**
- **(f X1 X2)**
- **( ) or NIL**

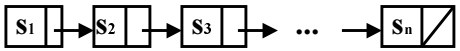
1.4

Relationship to Data Structures

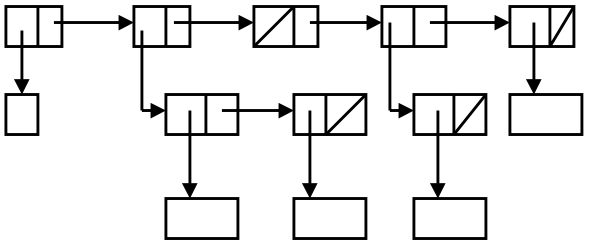
The list

(s_1 s_2 s_3 … s_n)

is represented by the data structure



Note: the s_i's are usually implemented as pointers to atoms or to **S**-expressions.



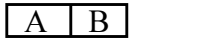
1.5

DOTTED PAIRS

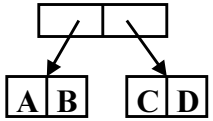
- Actually, the list is just a special case of the more general structure: dotted pair.

- In "dotted pair notation"
  An S-expression is
  1) an atom
  or 2) a dotted pair $(s_1 \bullet s_2)$ of s-expressions s_1 and s_2 .

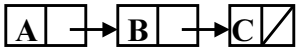
(A • B)



((A • B) • (C • D))



(A • (B • (C • NIL)))



Corresponds to list (A B C) !

1.6

Any S-expression with dot-notation form

(s_1 • (s_2 • (• • • (s_n • NIL) • • •)))

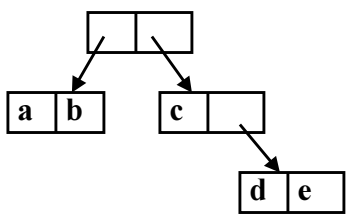
has list-notation form

(s_1 s_2 • • • s_n).

Not all S-expressions have a list-notation form.

((a • b) • (c • (d • e))

But they all have an internal representation.



List notation is preferred, whenever possible.

1.7

The Lisp Interpreter

Lisp is an interactive system that responds to your commands;

The prompt for Allegro Common Lisp for Windows is **>**.

When you type in an expressions, it is read, evaluated by the Lisp interpreter, and the result is printed to the screen.

Examples:

```
> 185
185

> abc
Error:  the variable ABC is unbound

> (setf abc 185)
185

> abc
185
```

1.8

Functional Forms

((<function> $<arg_1>$ $<arg_2>$ … $<arg_n>$)

Lists represent function invocations.

The first element in the list is the function.

The remaining elements are its arguments.

Arguments are evaluated from left to right, and then the function is applied.

```
>   ( + 3 5 )
  8

>   ( * 95 ( + 3 5 ) )
760

>   ( equal  t  nil )
NIL

>   (setf  a 1 b 3 c 2 )
  2
```

1.9

Elementary Functions

- **Quote**
  Quote is a special function that does not evaluate its argument.

(1 + 5)	**evaluates to 6**
(Quote (1+ 5))	**evaluates to (1+ 5)**
(Quote A)	**evaluates to A**
(Quote Quote)	**evaluates to Quote**

Quote is usually abbreviated by '.

'(1+ 5)
'A
'Quote
'(I AM A LIST)

1.10

List

(list a1 a2 . . . an) constructs and returns a list with elements a1 , a2 , … , an , in that order.

(list 'a 'b 'c) produces (a b c)
(list '+ 8 9) produces (+ 8 9)
(list (list 'a 'b)(list 'c 'd)) produces ((a b)(c d))

Note: most Lisps, capitalize everything, so
(list 'a 'b 'c) = (list 'A 'B 'C)
and actually produces **(A B C)** !

1.11

CONS

Cons stands for construct.

It produces a new dotted pair or "cons cell".

(cons x y)
where
x is an S-expression
y is an S-expression
produces the dotted pair (x • y).

(cons 'a 'b) produces (A • B)
which has the data structure 

(cons x l)
where
x is an S-expression
l is a list
produces a new list with x as first element
followed by l .

(cons 'a (list 'b 'c)) produces (A B C)

1.12

First (car)

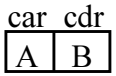
First extracts the first element of a list

(first (list 'a 'b 'c)) returns A

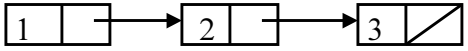
(first (list (list 'a 'b) 'c)) returns (A B)

First is the modern ? name for the function car which merely extracts and returns the car (first) field of any dotted pair.

(car (cons 'a 'b)) returns A.



(car (list 1 2 3)) returns 1



(car (car (list (list 2 3) 1))) returns 2

1.13

rest (cdr)

Rest extracts the remainder of the list, the part after the first element.

(rest (list 'a 'b 'c)) returns (B C)

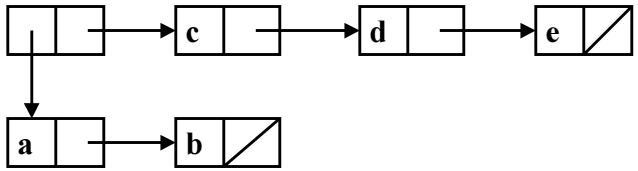
(rest (list (list 'a 'b 'c)) returns (C)

Rest is the modern name for the function cdr which extracts the cdr (second) field of any dotted pair.

(cdr (cons 'a 'b)) returns B.

(cdr (list 1 2 3)) returns (2 3)

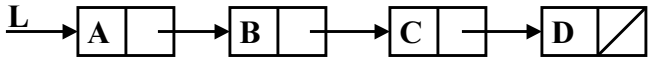
(cdr '((a b) c d e)) returns (C D E)



1.14

Suppose L is the list (A B C D).

(cons 'A (cons 'B (cons 'C (cons 'D NIL))))



(car L) = A

(cdr L) = (B C D)

(car (cdr L)) = B

(cdr (cdr L)) = (C D)

(car (cdr (cdr L))) = C

(cdr (cdr (cdr L))) = (D)

(car (cdr (cdr (cdr L)))) = D

(cdr (cdr (cdr (cdr L)))) = NIL

or

(rest (rest (rest (rest L)))) = NIL

1.15

DEFINING FUNCTIONS

defun

The built-in function defun allows us to define new functions.

(defun <name> <arglist> <body>)

defines a new function.

<name> **is a symbol, naming it**

<arglist> **is a list of its formal parameters**

<body> **is an expression representing its body**

```
( defun 2+ ( x ) ( 1+ ( 1+ x ) ) )
( defun cadr ( L ) ( car ( cdr L ) ) )
```

1.16

```
( defun myfun ( x y )
        ( * ( + x 2 ) ( + y 3 ) ) )

( defun xor ( conscell )
    ( not ( equal ( car conscell )
                  ( cdr conscell ) ) ) )

( defun push ( x s )
    ( cons x s ) )

( defun pop ( s )
    ( cdr s ) )

( defun exists ( x ) ( not ( null x ) ) )

( defun le ( a b ) ( not ( > a b ) ) )
```

1.17

MORE UTILITIES

cons constructs structures.

first and **rest** pull them apart.

Note: first and rest are nondestructive. They merely return a structure that is a piece of the original.

There are some more such functions.

second, third, etc. can pull out other specific elements.

last returns a list consisting of the last element in its argument list.

nthcdr returns the list that would be obtained by removing the first **n** elements from the argument list.

butlast returns a list consisting of all but the **n** last elements of the argument list.

1.18

(second	**' (1 2 3 4 5))**	**return 2**
(third	**' (1 2 3 4 5))**	**return 3**
(last	**' (1 2 3 4 5))**	**returns (5)**
(butlast	**' (1 2 3 4 5))**	**returns (1 2 3 4)**
(butlast	**' (1 2 3 4 5) 2)**	**returns (1 2 3)**
(nthcdr	**1 '(1 2 3 4 5))**	**returns (2 3 4 5)**
(nthcdr	**2 '(1 2 3 4 5))**	**returns (3 4 5)**
(nthcdr	**5 '(1 2 3 4 5))**	**?**

1.19

append

Append gives us another way of putting lists together by appending one (or more) lists to the end of the first one specified.

(append LIST**1** LIST**2** … LISTn)

constructs and returns a new list containing the elements of LIST1, followed by those of LIST2, … , up to LISTn.

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

Which is more efficient?

-- consing a new element to the front of a list or

-- appending a new element to the end?