$\boxed{3}$ LISP Symbolic Data

$\boxed{3.1}$ Symbolic Expressions (S-Expressions)

s-expression	::=	atom | list
atom	::=	number | symbol
list	::=	(s-expression*)

'::=' means 'is a' '<blank space>' means 'followed by a'

'|' means 'or a' '$\star$' means 'repeated 0 or more times'

Symbols: `A` `X` `A1` `1X` `HI-THERE` $\equiv$ `Hi-ThErE`

Lists: `(5)` `(A B C)` `(A (B C) D)`

`(+ (* 2 4) (/ 9 7) 8)`

`(HELLO THERE SPARKY)`

$\equiv$ means "equivalent when input to LISP"

$\boxed{3.2}$ Computer Representation of S-Expressions

s-expression	::=	atom | cons-cell
atom	::=	number | symbol
cons-cell	::=	(s-expression . s-expression)

non-empty-list	::=	(first-element . rest-of-list)	
empty-list	::=	`NIL` $\in$ symbol	`()` $\equiv$ `NIL`

`(A B)` $\equiv$ `(A . (B))` $\equiv$ `(A . (B . ()))` $\equiv$ `(A . (B . NIL))`

`(5 (A B) 7)` $\equiv$ `(5 . ((A . (B . NIL)) . (7 . NIL)))`

means ‘equivalent when input to LISP’

What if rest-of-list is a non-`NIL` atom?

`(A . (B . C))` $\equiv$ `(A B . C)` [dotted list]

$\boxed{3.3}$ Quote (')

LISP programs <u>are</u> s-expressions.
' suppresses evaluation of an s-expression.

`(* 2 5)` $\Longrightarrow$ `10`
but `'(* 2 5)` and `'M` <u>cannot</u> be rewritten

`5` $\Longrightarrow$ `'5`	`n` $\Longrightarrow$ `'n`	for `n` $\in$ number
`NIL` $\Longrightarrow$ `'NIL`	`T` $\Longrightarrow$ `'T`	special cases

`'x` $\equiv$ `(QUOTE x)`   `x` $\in$ s-expression
`(QUOTE (A (QUOTE B) C)` $\equiv$ `'(A 'B C)`
In this lecture "$\equiv$" means "equivalent when input to LISP".

Comments:

1. The "opposite" of the `QUOTE` function is the `EVAL` function that takes an s-expression and evaluates it. Thus:

   (eval '(+ 9 5)) $\Longrightarrow$ 14
   '(+ 9 5) $\Longrightarrow$ '(+ 9 5)
   (+ 9 5) $\Longrightarrow$ 14

   When you type an s-expression into a LISP system, what it does is read the expression, call `EVAL` to evaluate it, and then print the resulting value.

2. In this lecture '$\equiv$' means that two different expressions produce the same s-expression when read by a LISP system. In other lectures '$\equiv$' will mean other things (e.g. equivalence of propositional calculus formulae).

3. An s-expression typically has many different input forms. For example, `01` $\equiv$ `1`. But an s-expression has only one form when printed; for example `1` is printed and not `01`.

   However there are several different printer options that control how an s-expression is printed. None of these cause dot notation to be used, unfortunately for educators.

$\boxed{3.4}$ Manipulation of List Data

`(CONS 'x 'y)`	$\Longrightarrow$	`'(x . y)`	`x`, `y` $\in$ s-expression
`(CAR '(x . y))`	$\Longrightarrow$	`'x`	`FIRST` does same thing
`(CAR '())`	$\Longrightarrow$	`NIL`	as `CAR`.
`(CDR '(x . y))`	$\Longrightarrow$	`'y`	`REST` does same thing
`(CDR '())`	$\Longrightarrow$	`NIL`	as `CDR`.

Examples:

`(CONS 'A '(B C))`	$\Rightarrow$	`'(A . (B C))` $\Rightarrow$ `'(A B C)`
`(CAR '(A B C))`	$\Rightarrow$	`'A`
`(CDR '(A B C))`	$\Rightarrow$	`'(B C)`
`(CONS 'A NIL)`	$\Rightarrow$	`'(A . NIL)` $\equiv$ `'(A)`

$\boxed{3.5}$ NTH and APPEND

Examples: (NTH 0 '(A B C)) $\Longrightarrow$ 'A [0 Based]

(NTH 2 '(A B C)) $\Longrightarrow$ 'C

(NTH 3 '(A B C)) $\Longrightarrow$ NIL [off the end]

(APPEND '(A B) '(1 2)) $\Longrightarrow$ '(A B 1 2)

RR: (NTH 0 'x) $\Longrightarrow$ (CAR 'x) $x \in$ s-expression

(NTH n 'x) $\Longrightarrow$ (NTH (- n 1) (CDR 'x))

if n $>$ 0, n $\in$ integer

(APPEND '() 'z) $\Longrightarrow$ 'z x, y, z $\in$ s-expression

(APPEND '(x . y) 'z) $\Longrightarrow$ (CONS 'x (APPEND 'y 'z))

$\boxed{3.6}$ REVERSE

E.g.: `(REVERSE '(A B C))` $\Longrightarrow$ `'(C B A)`

`(REVERSE '(A B) '(1 2))` $\Longrightarrow$ `'(B A 1 2)`

`(REVERSE 'x 'y)`: Add the elements of `x` in reverse order to the front of `y`.

RR: `(REVERSE 'x)` $\Longrightarrow$ `(REVERSE 'x '())`

`(REVERSE '() 'z)` $\Longrightarrow$ `'z`

`(REVERSE '(x . y) 'z)` $\Longrightarrow$ `(REVERSE 'y '(x . z))`

`x`, `y`, `z` $\in$ s-expression

3.7 Symbolic Data Predicates

```
(CONSP 'a) ==> NIL                    a, b ∈ atom
(CONSP '(x . y)) ==> T                w, x, y, z ∈ s-expression
(EQL 'a 'b) ==> T          if a and b print the same
            ==> NIL        otherwise
(EQL 'a '(x . y)) ==> NIL
(EQL '(x . y) 'b) ==> NIL
(EQL '(w . x) '(y . z)) ==> undefined for now

(ATOM 'x) => (NOT (CONSP 'x))        (NULL 'x) => (EQL 'x NIL)
(LISTP 'x) => (OR (NULL 'x) (CONSP 'x))
```

Comments:

4. In the code for `REVERSE` the symbol `&OPTIONAL` is used in the argument list of the `DEFUN`. This symbol means that the following arguments are optional, and can be omitted for any actual call.

   Furthermore, the arguments following `&OPTIONAL` in the `DEFUN` argument list are given as pairs consisting of an argument name and a default value expression. The default value expression is an expression whose value is given to the argument if the argument is <u>not</u> given in the call.

   The default value expression can contain the names of previous arguments in the argument list: arguments are bound left to right.

5. LISP has functions for performing logic on truth values: `AND`, `OR`, and `NOT`.

   `AND` evaluates its arguments left to right until it finds one that is false, i.e. `NIL`. Then it returns the value of that argument, i.e. `NIL`. If no argument is false, `AND` returns the value of the last argument.

   `OR` evaluates its arguments left to right until it finds one that is true, i.e. non-`NIL`. Then it returns the value of that argument. If no argument is true, `OR` returns `NIL`.

   `NOT` evaluates its one argument and returns `NIL` if the argument value is true and `T` if the argument value is false.

Comments:

6. In general, the arguments of a LISP function are evaluated before the function is called. This is called eager evaluation.

   However, some functions are an exception, and these are called special functions. Most of these functions evaluate their arguments only when they need the values of the arguments. This last is called lazy evaluation. With lazy evaluation, arguments whose values are never needed are never evaluated.

7. `COND`, `AND`, and `OR` are special functions that use lazy evaluation. They evaluate arguments from left to right, skipping completely the evaluation of arguments that are not needed. This is important because `COND`, `AND`, and `OR` are often used in a way such that unneeded arguments would cause an error if evaluated.

$\boxed{3.8}$ MEMBER

Examples:
```
(MEMBER 'A '(A B C)) ==>  '(A B C)
(MEMBER 'B '(A B C)) ==>  '(B C)
(MEMBER 'C '(A B C)) ==>  '(C)
(MEMBER 'X '(A B C)) ==>  NIL
```

RR:
```
(MEMBER 'x '()) ==> NIL
(MEMBER 'x '(y . z)) ==> '(y . z)   if (EQL 'x 'y)
                     ==> (MEMBER 'x 'z) otherwise
```

x, y, z $\in$ s-expression

$\boxed{3.9}$ EQUAL

Examples: `(EQUAL 'A 'A)` $\Longrightarrow$ `T`

`(EQUAL '(A B) '(A B))` $\Longrightarrow$ `T`

`(EQUAL '(A (B C) 5) '(A (B C) 5))` $\Longrightarrow$ `T`

`(EQUAL '(A (B C) 5) '(A (B X) 5))` $\Longrightarrow$ `NIL`

RR: `(EQUAL 'x 'y)`

$\Longrightarrow$ `(EQL 'x 'y)` if either `x` or `y` is an atom

`(EQUAL '(w . x) '(y . z))`

$\Longrightarrow$ `(AND (EQUAL 'w 'y) (EQUAL 'x 'z))`

`w, x, y, z` $\in$ s-expression

$\boxed{3.10}$ LET

variable value

$$\begin{array}{l}\texttt{(LET ((}v_1\ e_1\texttt{)}\\ \qquad\quad\texttt{(}v_2\ e_2\texttt{)}\\ \qquad\quad . \ . \ . \quad \texttt{)}\\ \quad\texttt{answer)}\end{array} \quad\Longrightarrow\quad \begin{pmatrix}v_1\\v_2\\\cdots\end{pmatrix} \longleftarrow \begin{pmatrix}e_1\\e_2\\\cdots\end{pmatrix}$$

and then return `answer`

```
(LET ((X 'A)
      (Y 'A))
  (EQL X Y))
   ==> T
```

```
(LET ((X (CONS 'A NIL))
      (Y (CONS 'A NIL)))
  (EQL X Y))
   ==> NIL
```

[Why?]

Comments:

8. The LET special function, which binds names to values of expressions, evaluates all the expressions before binding any names to values. Thus if there are several values, they cannot refer to each other: the names cannot be used in an expression of a LET to refer to the values of previous expressions in the same LET.

   There is a different and very useful special function, LET*, which binds the name to each expression value before evaluating any subsequent expression in the LET*, so each expression can refer to previous expressions by name.

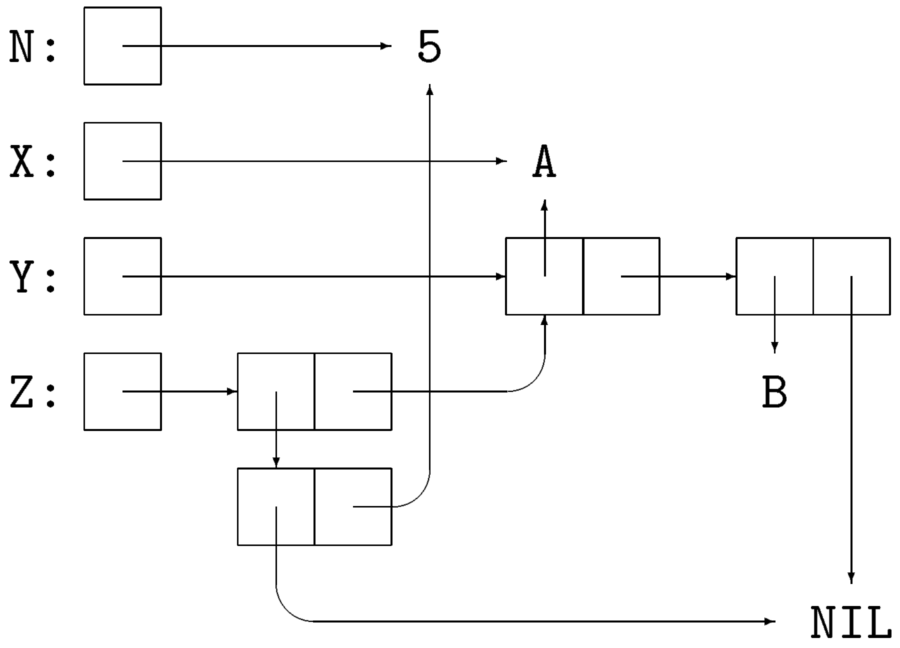
   I.e.

```
(LET* ((v1 e1)                    (LET ((v1 e1))
       (v2 e2)          ==>         (LET ((v2 e2)
       . . . )                              . . .)
  . . . )                           . . .)
```

3.11 Box Notation: The LISP Memory Model

```
(LET ((N 5) (X 'A) (Y '(A B)))
  (LET ((Z (CONS '(NIL . 5) Y)))
    . . .  )
```

(A B) ≡
(A . (B . NIL))

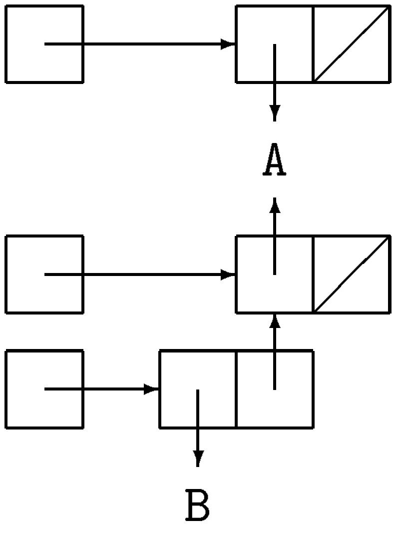


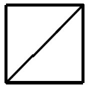
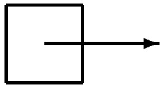
Represent:
Atom by
  printed form.
Variable by
  box with pointer.
Cons-cell by
  pair of boxes
  with 2 pointers.

3.12 EQL

(EQL ’x ’y) $\Longrightarrow$ T   if x = y as pointers

$\Longrightarrow$ NIL otherwise

Given	Then
X:  (CONS ’A NIL) Y: (CONS ’A NIL) Z: (CONS ’B Y)	(EQL X Y) $\Longrightarrow$ NIL (EQL X X) $\Longrightarrow$ T (EQL (CAR X) (CAR Y)) $\Longrightarrow$ T (EQL X (CDR Z)) $\Longrightarrow$ NIL (EQL Y (CDR Z)) $\Longrightarrow$ T

where  means  NIL

Comments:

9. Every programming language has a memory model. It is very important to understand this memory model.

   The LISP memory model we have given is not exactly how LISP is really implemented. For example, actual systems often duplicate large numbers in memory, instead of having only one copy of each number. `EQL` on such a system must test equality of the numbers themselves to get the semantics (meaning) required by the LISP memory model.

10. The `EQ` function is an efficiency hack on `EQL`; `EQ` just tests pointer equality, and does not look at numeric values. Since most LISP systems store duplicates of some types of numbers, and different systems duplicate different types of numbers, `EQ` gives very implementation dependent results on numbers.

    Thus use of `EQ` is prohibited in this course as bad style.

    However, professional LISP programmers will use `EQ` whenever testing for equality with a non-number, because they know it compiles to much more efficient code than `EQL`, since the latter has to check to see if numbers are being compared.