Imperative, OO and Functional Languages

- A "C" program is …
  - a web of assignment statements, interconnected by control constructs which describe the time sequence in which they are to be executed.
- In Java programming,
  - "objects" are sent "messages".
- In "pure" LISP there is only …
  - the evaluation of an expression by function application
  - instead of "executing a program," LISP evaluates a symbolic expression (s-expr)

Why Learn LISP ???

- "Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot."
  - Eric Raymond, "How to Become a Hacker"

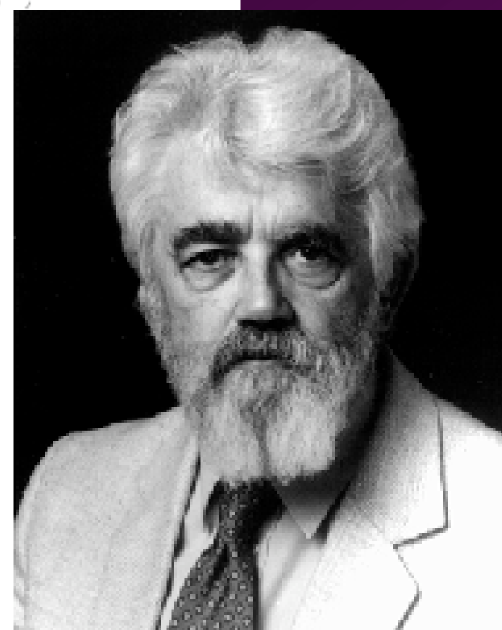
- For examples of some companies that use LISP, see http://www.paulgraham.com/apps.html

Where Does LISP Come From?

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge, Mass. *

April 1960



1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions

Introduction to LISP

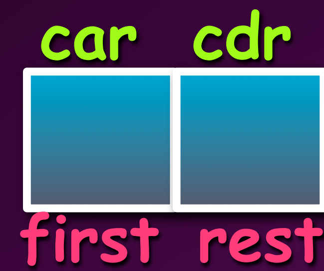
- LISP syntax: <u>s</u>ymbolic <u>expr</u>essions
  - atoms
  - lists of atoms
  - lists of s-exprs
- Grammar for symbolic expressions:
  - <s-expr> ::= <atom> | <list>
  - <list> ::= (<s-expr> *)
- **(name-or-description-of-a-function arg$_1$ arg$_2$ … arg$_n$)**

Evaluation of S-EXPRs

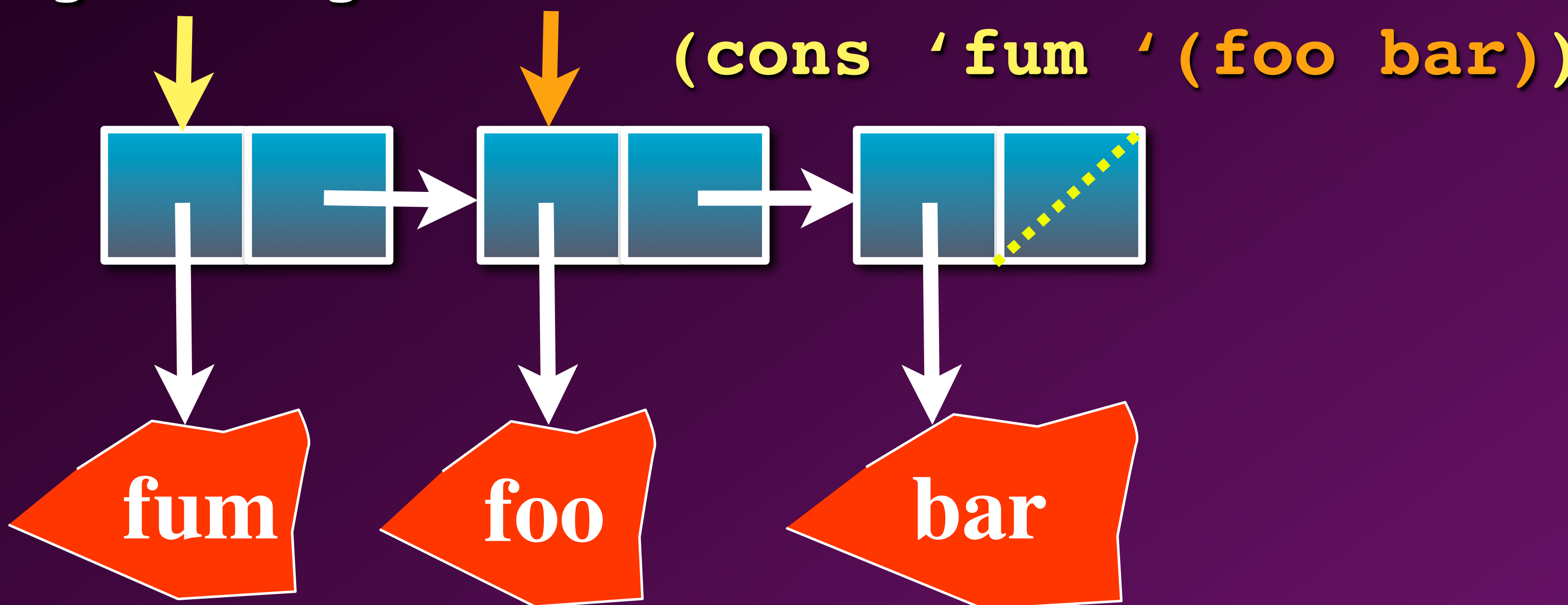
- Parentheses must be taken seriously!
- Quoting inhibits evaluation (EVAL does the opposite!)
- **NIL** is both a list and an atom
- Defining your own functions using DEFUN
  - area of a circle
- Other useful functions: IF, +, *, =
- Defining a recursive function: factorial

Internal Representation

- LISP lists are stored as linked-lists of records with CAR and CDR fields



- Diagramming list structure:



Other LISP Functions

- Assignment (side-effects) using SETF, SET
- Lisp Manipulation Functions
  - CAR, CDR, CONS, LIST, APPEND
- Predicates
  - EQ, EQL, EQUAL, ATOM, LISTP, CONSP, NULL, ZEROP, PLUSP, MINUSP, EVENP, ODDP, NUMBERP, SYMBOLP, BOUNDP, >, <, <=, >=, =, /=
  - Define a function to recursively compute the length of a list

Predicates

- Summary of commonly confused primitive predicates:

	'A	'(A)	nil
atom	t	nil	t
lisp	nil	t	t
consp	nil	t	nil
null	nil	nil	t
symbolp	t	nil	t

Defining 2 More Functions

- Swap the first and second elements of a list.
  - e.g., `(swap '(A B C D))` → `(B A C D)`
  - ```
    (defun swap (lst)
        (cons '(cadr lst)
              (cons (car l) (cddr l))))
    ```
  - Note: this builds a new list with some structure shared with L
- Compute the "next even integer" following n
  - ```
    (defun next-even (n)
        (if (evenp n) n (1+ n)))
    ```

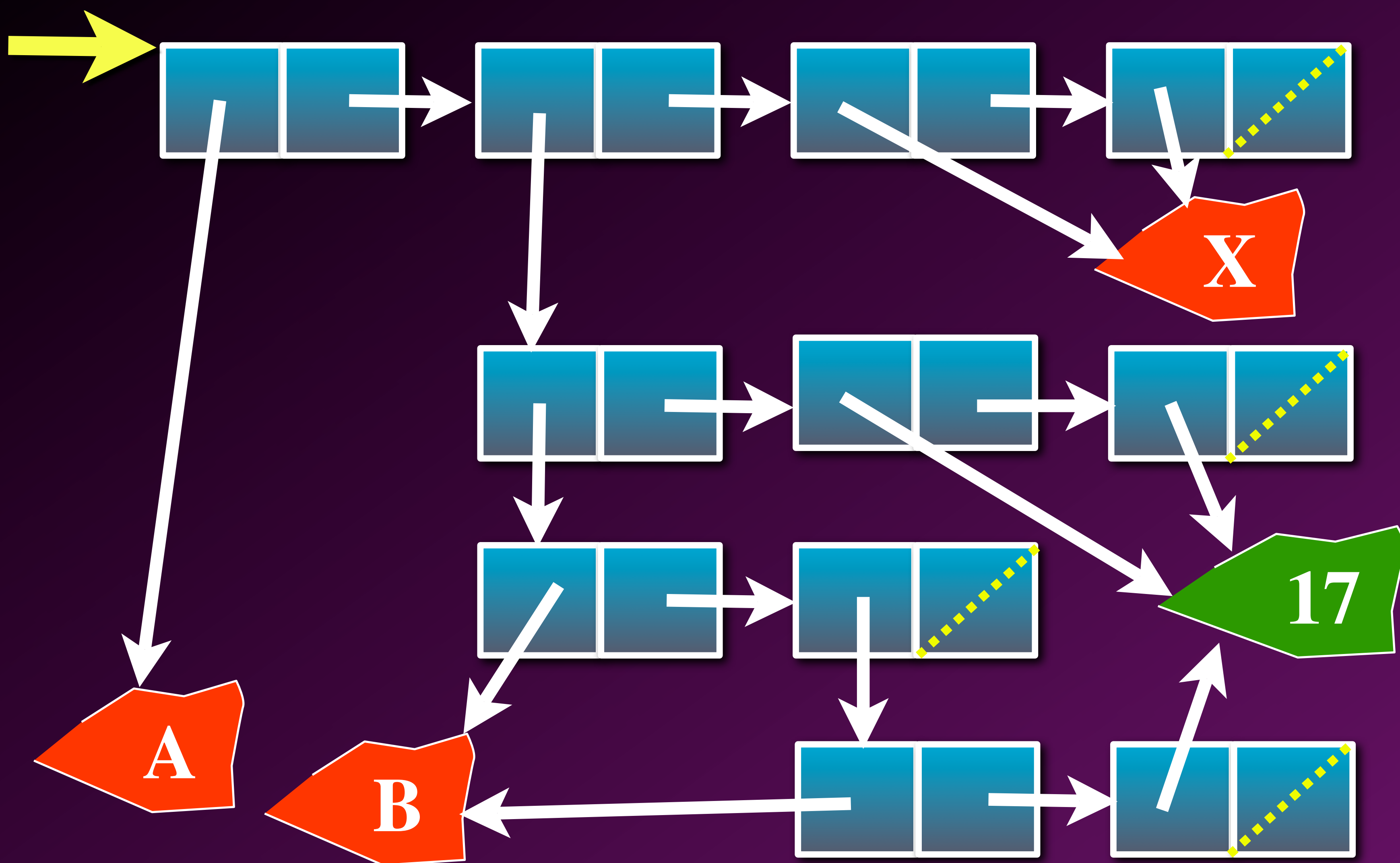
The Most General Conditional

$$
\begin{array}{ll}
(cond & (test_1 \;\; s\text{-}expr_{11} \; s\text{-}expr_{12} \; \ldots \; s\text{-}expr_{1n_1}) \\
& (test_2 \;\; s\text{-}expr_{21} \; s\text{-}expr_{22} \; \ldots \; s\text{-}expr_{2n_2}) \\
& \quad \ldots \\
& \quad \ldots \\
& (test_k \;\; s\text{-}expr_{k1} \; s\text{-}expr_{k2} \; \ldots \; s\text{-}expr_{kn_k}))
\end{array}
$$

Recursion on List Structures

- CAR and CDR take lists apart
- the analog of "subtract 1" (when doing induction on a number) is taking the CDR of a list
- Our version of the built-in MEMBER function:
  - ```
    (defun memb (element lis)
        (cond    ((null lis) nil)
                 ((eq (car lis) element) lis)
                 (t (memb element (cdr lis)))))
    ```
- Recursion in "two directions" — function OCCURS
  - ```
    (defun occurs (element lis)
        (cond  …
    ```

Example of Nested List



Nameless Functions via LAMBDA

- LISP programs are conceived and written with a mathematical rigor, based upon the formalisms of "recursive function theory" and the "lambda calculus."
  - Consider $y + (x * y)$ for the values 3 and 4
  - Clarify using the "Lambda notation" of Alonzo Church.
  - In LISP, we use a similar notation
- (MAPCAR F L)
  - F is a function of one parameter
  - L is a list $(x_1 \ x_2 \ \ldots \ x_n)$
  - Produces a list $(y_1 \ y_2 \ldots y_n)$ where $y_i = (F \ x_i)$

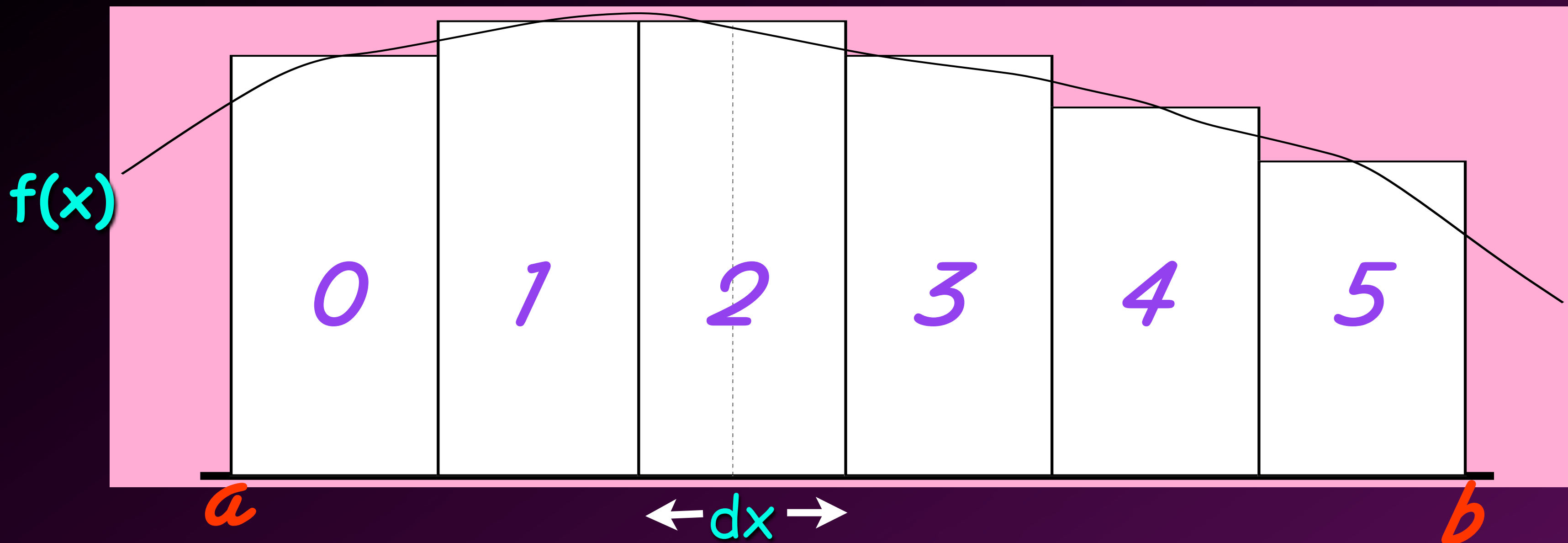
Procedural Abstraction

- SUM-INTEGERS computes $\sum_{n=\text{first}}^{\text{last}} n$
- SUM-SQUARES computes $\sum_{n=\text{first}}^{\text{last}} n^2$
- Now make the function-of-n itself a third parameter:
  - SUM-TERMS computes $\sum_{n=\text{first}}^{\text{last}} \textit{term-fn}(n)$
- Consider now the infinite series

$$\frac{\pi}{8} = \frac{1}{1 * 3} + \frac{1}{5 * 7} + \frac{1}{9 * 11} + \ldots$$

and define PI-TERM(n) to compute the above

Integration by Summation



- $$\sum_{n=0}^{(b-a)/dx\,-1} f(a + n.dx + dx/2)\ .\ dx$$
- $$dx\ .\sum_{n=0}^{(b-a)/dx\,-1} f(a + n.dx + dx/2)$$

Symbolic Pattern Matching

- Another kind of search problem: in a linear list of words (symbols, whatever) to discover specified patterns.
  - Although LISP itself has no built-in pattern-matching, it's a good implementation language for such a function.
  - (match pattern data) will return T or NIL
  - The pattern may contain "wildcard" variables such as ? (stand for one symbol) and * (stands for a sequence of 0 or more symbols)

Pattern Matching, part 2

- "Wildcard" examples
  - A ? B matches A A B
    but not A B
    nor A B C
  - A * B matches A A B
    and A B
    and A X Y Z B
  - * X * Y matches any sequence containing
    both X and Y in that order

Additional "Wildcards"

- ?variable   matches a single atom, and assigns that atom to variable
- *variable   matches a sequence of ≥ 0 atoms, and assigns a list of that sequence to variable
- Example from Doctor program:
  - (cond ((match '(I am worried *blah-blah) userInput) (princ (append '(How long have you been worried) blah-blah)))