

# Lisp: a history

- Developed by John McCarthy in the 1950's.
- Only Fortran has higher “seniority”
- LISP: List Processing language
  - (i) Symbolic language: its execution effects can be represented by processing symbols or strings of data
  - (ii) Functional language: its basic components are denotable by mathematical functions
  - (iii) general purpose language: it has evolved to incorporate many “real-world” features
  - (iv) a language for Artificial Intelligence: symbol  $\leftrightarrow$  concept, word
- Many dialects exist; Common Lisp is the modern “standardized” Lisp

– fig 1.1 Harrison

# Lisp: history

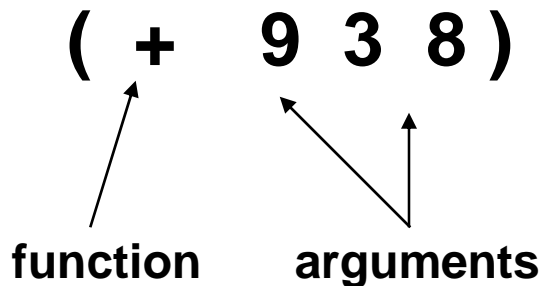
- **Lisp design borrows from:**
  - **Programming languages: Some Fortran, IPL syntax**
  - **Math: recursive function theory, lambda calculus**
  - **McCarthy's list representation of programs, data**
- **Some implementation characteristics:**
  - **often interpreted, which makes it slow (compilers exist)**
  - **memory intensive: extensive dynamic allocation/deallocation of memory**
    - » **lots of research in “garbage collection” techniques**
  - **there exist special Lisp computers (Lisp Machine); however, RISC architecture has superceded these efforts**
- **Program characteristics:**
  - **symbolic nature encourages experimentation, fast prototyping**
    - » **a major reason it's popular in AI**
  - **very recursive**
  - **very easy to write “read-once” programs**
    - » **must be doubly diligent to write clean, politically correct programs!**

# Getting started with Xlisp

- **Startup:**
  1. **put /usr/local/public/xlisp in your path**
  2. **copy /usr/local/public/xlisp/init.lsp to your working directory**
  3. **“xlisp”**
- **Xlisp program files have “.LSP” extension**
- **Because its interpreted, you can create, edit, and debug programs during execution.**
  - **recommended that you use a text editor to enter programs into files explicitly**
- **To load your file:**
  - > **(load “/usr/people/mccarthy/myfile.lsp”)**
- **To save a copy of your interactive session:**
  - > **(dribble “session\_file\_name”)**
- **To exit Xlisp:**
  - > **(exit)**

# Lisp syntax: List

LISP: “Lost in Stupid Parentheses”



- you can replace function or argument by another list; arbitrarily deep

- examples

(5)                      (dog (cat mouse))  
(4 7 3 5)              (+ 3 (\* 3 6) (/ 3 5))

- You will spend much time balancing parentheses
  - try vi’s “%” key

# Lisp functions, Atoms

- **Basic rule of Lisp:**
  - when combining Lisp functions, the embedded list arguments are always executed before the function is applied to them (“eager evaluation”)
  - compare to: lazy evaluation - evaluate only when needed

- **Atom: a basic data value (vs. lists, which are compound values)**

eg. `george`            `3.14159`  
    `56`  
    `+`  
    `two-words`

- **Special list: `NIL` or `()`**
  - considered to be both a list and an atom

## Simple function calls

**> (+ 8 12 3)**  
**23**

**> (\* 5 6 7)**  
**210**

**> (+ (- 4 5) (\* 2 2))**  
**3**

**> (/ 2 3 4)**  
**1/6**

**> (/ (/ 2 3) 4)**  
**1/6**

**> (/ 2 (/ 3 4))**  
**8/3**

**> (/ 2.0 3 4)**  
**0.1666667**

# Lisp

- There are lots of builtin Lisp functions (see Xlisp doc.)
- There are builtin data types as well (float, bool, string,...)
- One key point: program code and data are both symbolic expressions in Lisp
  - the interpreter doesn't distinguish between them
- Lisp interpreter:
  - 1. evaluate each argument --> obtains a value
  - 2. evaluate the function applied to the argument values
- sometimes we don't want a list to be evaluated: use the quote function to create a *literal expression*

' (a b c)

- differs from: (a b c) <-- expects a function called "a"

## List functions: CAR, CDR

- **CAR:** returns first element in a list
  - result may be an atom or list
- **CDR:** returns tail of the list (rest of list minus first element)
  - - result is always a list

eg. `(car '(a b c d))` --> returns atom "a"

`(car '((a b) (c d) (e (f g))))` --> returns list (a b)

note: `(car (a b c d))` will try to evaluate (a b c d) first!

`(cdr '(a b c d))` --> returns (b c d)

`(cdr '(a))` --> returns NIL or ()

`(car (cdr '((a b) (c d) (e (f g))))))` --> returns (c d)

`(car '(cdr '((a b) (c d) (e (f g))))))` --> returns CDR ! (programs are data)

# List functions: CONS, LIST

- **cons**: inserts an item in front of a list
  - **cons** takes **elem1** (atom or list), and stuffs it into first position of 2nd argument, which must be a list

```
( cons 'a '( b c d )  
      ↓      ↓  
      ( a b c d )
```

- **list**: creates a list from a set of arguments
  - **list** takes arguments, and wraps parentheses around them

```
( list 'a 'b 'c 'd )
```

```
( ( a b c d ) )
```

# List functions

- more examples

`(list 'a '(b c)) --> returns (a (b c))`

`(list '(a b) '(c d)) --> returns ((a b) (c d))`

`(cons 'a '(b c)) --> returns (a b c)`

`(cons '(a b) '(c d)) --> returns ((a b) c d)`

`(list (cons 'a '(b c)) (cdr '(d e f)))`

`= (list (a b c) (e f)) --> returns ((a b c) (e f))`

`(cons '(* 3 4) '(5 6)) --> returns ((* 3 4) 5 6)`

`(cons (* 3 4) '(5 6)) --> returns (12 5 6)`

# Global variables

- **setq** takes a variable name (same rules as Pascal etc) and a value, and assigns variable that value

```
> (setq f 25)
```

```
25
```

```
> (setq g '(a b c))
```

```
(a b c)
```

```
> (cons f g)
```

```
(25 a b c)
```

```
> (cons 'f g)
```

```
(f a b c)
```

```
> (list (cdr g) (list (car (cdr g)) (car g)))
```

```
(( b c ) (b a) )
```

```
> (setq g 'dog)
```

```
dog
```

# Miscellaneous

- Important reminder:

You must put the quote ' before any constants (atoms) in your program.

eg. ( list 'cat 'dog 'bat )

Otherwise, if the quote is missing, Lisp thinks you are referring to a variable:

```
> ( setq cat 5 )  
> ( list cat 'dog 'bat )  
( 5 dog bat )
```

## More List primitive functions

- **append**: merges two or more lists into a single list
  - all arguments must be lists

eg. (append '(a b) '(c)) --> returns (a b c)

(append '(a (b c)) '(d e) '((f))) --> returns ( a (b c) d e (f) )

- note: unlike list, append presumes one level of list structure, and puts all the list arguments into this list

- **reverse**: reverse order of elements in the single argument
  - note: does not reverse argument sublists!

eg. (reverse (cdr '(a b c d))) --> returns (d c b)

- **last**: returns a list consisting of last element in its argument list

eg. (last '( a (b c) d e (f g) )) --> returns ((f g))

# CONS, LIST, APPEND...

- These functions are “similar”, but shouldn’t be confused.

**( cons '(a) '(b c d e) ) --> ( (a) b c d e )**

- only two arguments; second must be a list
- inserts 1st arg as 1st element in 2nd list

**( list '(a) 'b '(c(d)) 'e 'f ) --> ( (a) b (c(d)) e f )**

- any number of arguments
- wraps parentheses around them, creating a list

**( append '(a) '((b)) '(c(d)) ) --> (a (b) c (d) )**

- any number of args, all must be lists
- takes parentheses off of each argument, and then wraps parentheses around stripped results

# Conditional processing

- **conditional processing (“tests”)** is crucial to any programming language
- **Predicate:** a function that returns information about its arguments
  - usually interpreted by program as “true” or “false”
  - “false”: usually NIL
  - “true”: either the constant ‘t’ or any value other than NIL
- **atom:** ‘t’ if arg is an atom; otherwise NIL
  - ( atom ‘dog ) --> t
  - ( atom ‘(dog) ) --> NIL
  - ( atom NIL ) --> t (NIL is both an atom and a list)
- **listp:** ‘t’ if arg is a list; otherwise NIL
  - ( listp ‘dog ) --> NIL
  - ( listp ‘(dog)) --> t
  - ( listp NIL ) --> t

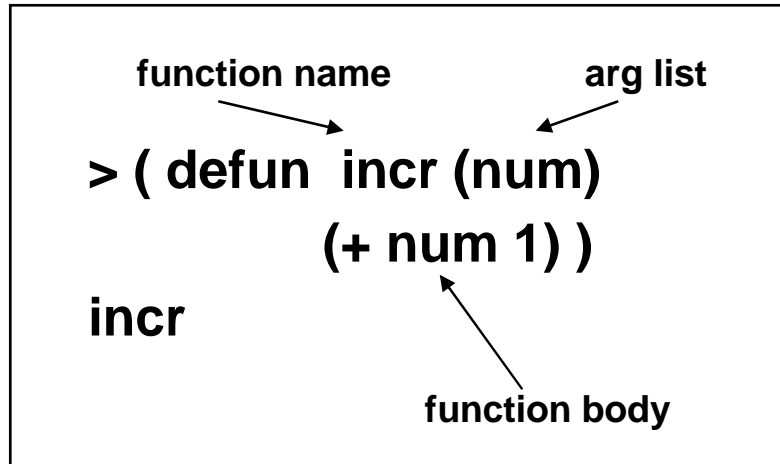
# Conditional processing

- **numberp: 't' if arg is a number (integer, float), otherwise NIL**
  - ( numberp 10 ) --> t
  - ( numberp 10.5 ) --> t
  - ( numberp 'chris ) --> NIL
- **zerop: 't' if arg evaluates to 0**
  - ( zerop (/ 0 5) ) --> t
  - ( zerop (/ 1 5)) --> NIL
- **null: 't' if argument is NIL, otherwise NIL**
  - ( null nil ) --> t
  - ( null '(a b) ) --> NIL
- **equal: 't' if its 2 args are equal, otherwise NIL**
  - ( equal 2 4 ) --> NIL
  - ( equal '(b c) (cdr '(a b c)) ) --> t

# Conditional processing

- **member:** if first arg is found as element in 2nd list arg. it returns tail of list starting with that first arg value; otherwise NIL
  - ( member 'a '(b c a e f) ) --> (a e f)
  - ( member 'a '( b (a e) d ) ) --> NIL
- **logical functions:**
  - **OR:** evaluates args from left to right; returns first non-NIL value found
  - **NOT:** if arg evaluates to NIL, returns t; otherwise, non-NIL args return NIL
  - **AND:** evaluates left-to right, and returns NIL at first arg that evals to NIL;  
if all args are non-NIL, returns value of last argumenteg. ( seq x 25 )
  - ( or (> x 0) (< x -30) ) --> t
  - ( not (> x 0) ) --> NIL
  - ( and (> x 0) (< x 30) ) --> t
  - ( and 'a 'b 'c ) --> c
- **There are many other predicates:**
  - relational tests:< > \=
  - precise tests on lists, constructs... see documentation for more

# Defining Lisp Functions



- **Function name must be an atom**
- **Parameters are call by value: changes to them affect local memory**
  - **think of them as local variables into which the argument values are copied; they disappear after function is executed**
- **function body is also a list**
  1. **parameter values are substituted into all parameter references**
  2. **body is executed**
  3. **final value is returned as value for function**

# Function definitions

- (Table 2.1 text)

# Conditional processing

- **cond: Lisp's if-then-else statement**
  - evaluates argument lists one after another
  - the first list with a non-NIL test (“true”) has its body evaluated and returned as answer
  - common to put a default “otherwise” case at the end; otherwise NIL is returned if no cases are true

```
( defun abc (val)
  ( cond
    ( ( equal val 'a) 'first )
    ( ( equal val 'b) 'second )
    ( t 'rest) ) )
```

optional 'default' case



# Example

- write a function that takes two values and returns the greater value, presuming they are numeric. Do error recovery.

## Version 1:

```
(defun bigger ( a b )  
  ( cond ((not (numberp a)) NIL)  
        ((not (numberp b)) NIL)  
        ((> a b) a)  
        ((> b a) b)  
        (t a) ) )
```

(bigger 1 2) --> 2

(bigger 2 'c) --> NIL

# Example

## Version 2

```
(defun bigger ( a b )  
  ( cond ((not (numberp a)) NIL)  
         ((not (numberp b)) NIL)  
         (> a b) a  
         (t b) ) )
```

## Version 3

```
(defun bigger ( a b )  
  ( and (numberp a)  
        (numberp b)  
        (or (and (> a b) a)  
            b) ) )
```

: very cryptic function!

# Example

- (text, 3.16) `safediv` - divides 2 numbers, but does error checking for proper data types, div by zero...

eg. `(safediv 6 3)` --> 2

`(safediv 6 0)` --> NIL

`(safediv 'w 4)` --> NIL

```
( defun safediv ( num1 num2 )
```

```
  ( and ( numberp num1)
```

```
        ( numberp num2)
```

```
        ( not (zerop num2) )
```

```
        (/ num1 num2) ) )
```

# Miscellaneous

- **Comments in programs:** start line with `;`
- **functions with no arguments:**
  - either: `(defun f () .....`
  - or `(defun f NIL ....`
- **Interpreter: evaluates multiple expressions from left to right**
  - eg. `(list (setq x 1) (setq x 2))` --> returns `(1 2)`, but variable `x = 2`
  - eg. `(defun f (a b)`
    - `(action 1)`
    - `(action 2)`
    - `...`
    - `(action k) (more later)`
- **returning a constant value:**
  - `(defun five()`
    - `5)`