Introduction to LISP

York University
Department of Computer Science and Engineering

Overview

- Introduction to LISP
- Evaluation and arguments
- S- expressions
  - Lists
  - Numbers
  - Symbols
- setq, quote, set
- List processing functions
  - car, cdr, cons, list, append

[ref.: Wilensky, Chap1-2]

LISP vs. Prolog

- Prolog
  - Logic Programming
  - Based on Predicate Logic
  - Working with predicates
  - Computation is reasoning, initiated by a query
  - Popular in natural language processing
  - More research on/with Prolog in University of Edinburgh
- LISP
  - Functional Programming
  - Based on Lambda Calculus
  - Working with functions
  - Computation is evaluation
  - Used in Artificial Intelligence
  - More research on/with LISP in MIT

LISP

- Designed by John McCarthy in 1958 in MIT
- Second-oldest high-level programming language (after Fortran) [Wikipedia]
- Popular dialects: Common LISP, Scheme
  - We use Common Lisp (www.clisp.org)
  - Execute the command *clisp*, execute *(exit)* to exit (or ctrl + D)
- Theory based on Lambda Calculus by Alonzo Church (1930)
  - λ-calculus: theory of functions as formulas
  - Easier manipulation of functions using expressions

LISP

- LISP: acronym for LISt Processing
  - Joke: LISP is acronym for “Lots of Irritating Single Parentheses”
- Primary data structure:
  - Symbolic expression (s-expression):
    - Lists
    - Atoms
- LISP interpreter: An interactive environment, always **evaluating** input
- Example:

```
> (+ 2 3)
5
```

Evaluation

- (+ 2 3)
  - A list is an s-expression, defined by a pair of parentheses
  - First element is assumed to be a function
  - The rest are arguments to the function
  - Arguments are evaluated as s-expressions themselves

- Example:

  ```
  > (+ 2 (* 3 4))
  14
  ```

- LISP evaluation rule:

  Look at the outermost list first. Evaluate each of its arguments. Use the results as arguments to the outermost function.

Evaluation (cont.)

- LISP evaluates everything!
  - Even when the arguments are simple numbers, they are evaluated!
  - Numbers evaluate to themselves

    > 8

    8
  - A value is returned from the evaluation of an expression

- Nested Lists

  > (+ (* 8 9) (- 8 10))

  70

- "Lots of Irritating Single Parentheses"!!!

Arguments

- Number of arguments
  - Supply the correct number of arguments

  > (**1+** 5)

  6
  - Otherwise error! It enters debugger, use *quit* or Ctrl+D to exit debugger
  - + is defined to allow more than 2 arguments

  >(+ 1 2 3)

  6

- <u>Supplied</u> arguments vs. <u>actual</u> arguments

  > (+ 2 (* 3 4))

  Supplied args: 2 and (* 3 4)

  Actual args: 2 and 12

Symbols

- Symbols can serve the role of variables

- Can be assigned values:

  > (setq x 8)

  8

  > x

  8

LISP is not case-sensitive!

Symbols evaluate to the last value assigned to them

- Note **setq** is a special function,
  - First argument is not evaluated
  - Second argument is evaluated and assigned to first argument
  - The value is returned

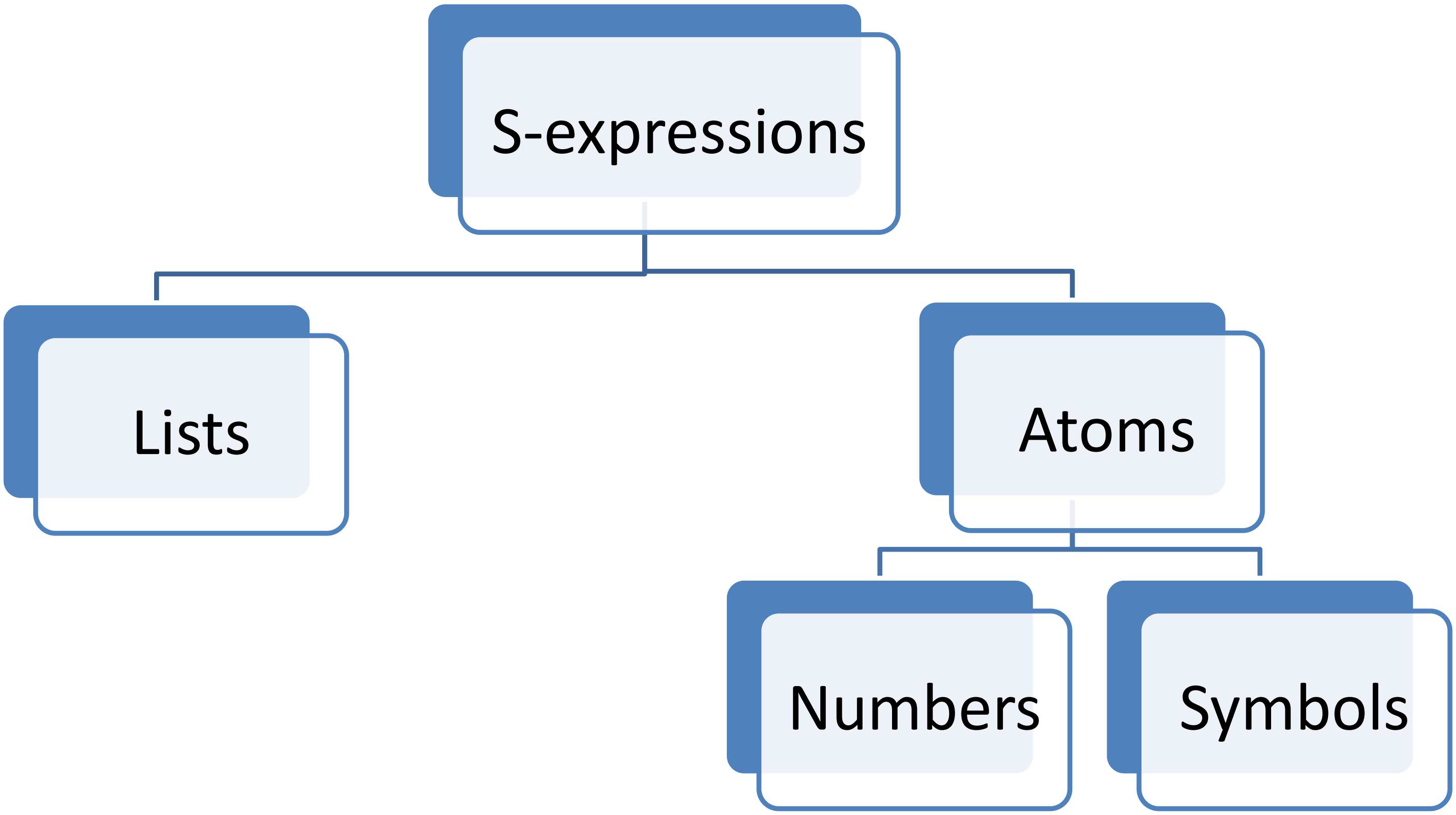
Symbols (cont.)

- Symbols can also serve the role of function identifiers
  - For example +, 1+, setq are all symbols

- Can have both roles simultaneously!

```
> (setq 1+ 5)
5
> (1+ 7)
8
> 1+
5
```

S-expressions



More on numbers

- Integers: 1, 10, ...
- Ratios: 1/2, 2/3, ...
  - > (+ 1/2 1/3)
  - 5/6
- Floating point numbers: 1.2, 0.25, 3.33**E**23
  - Can specify precision by using **S, F, D, L** for *short*, *single*, *double*, *long* precision respectively instead of E
  - For example 1.2D10, 2S0
- Arithmetic functions on page 429, 434-437 Wilensky

Lists

- Use parentheses to denote lists in LISP, no commas
  - e.g. (a b c)

  >(setq x (a b c))
  Error: Undefined function A!
  - Evaluation of lists: first element is assumed to be a function
- Use *quote* (short form is ‘) to prevent evaluation

  > (setq x ‘(a b c))
  (A B C)

  > (setq x (quote (a b c)))   (same as previous)
  (A B C)

Set

- setq is actually **set quote**

  > (setq x 5) is same as > (**set** ‘x 5)

  – Reminder: setq does not evaluate its first argument

- More examples:

  > (set ‘x (+ 2 3))
  5
  > x
  5

  > (set ‘x ‘(+ 2 3))
  (+ 2 3)
  > x
  (+ 2 3)

Values are S-expressions

- Assigning a value that is itself a symbol

> (setq x ‘y)
Y

> x
Y

> (set x (+ 2 3))
5

> x
Y

> y
5

Supplied arguments:
x and (+ 2 3)
Actual arguments:
y and 5

Lists as binary trees

- A list is actually a binary tree, consisting of the head and the tail
- List notation vs. dot notation

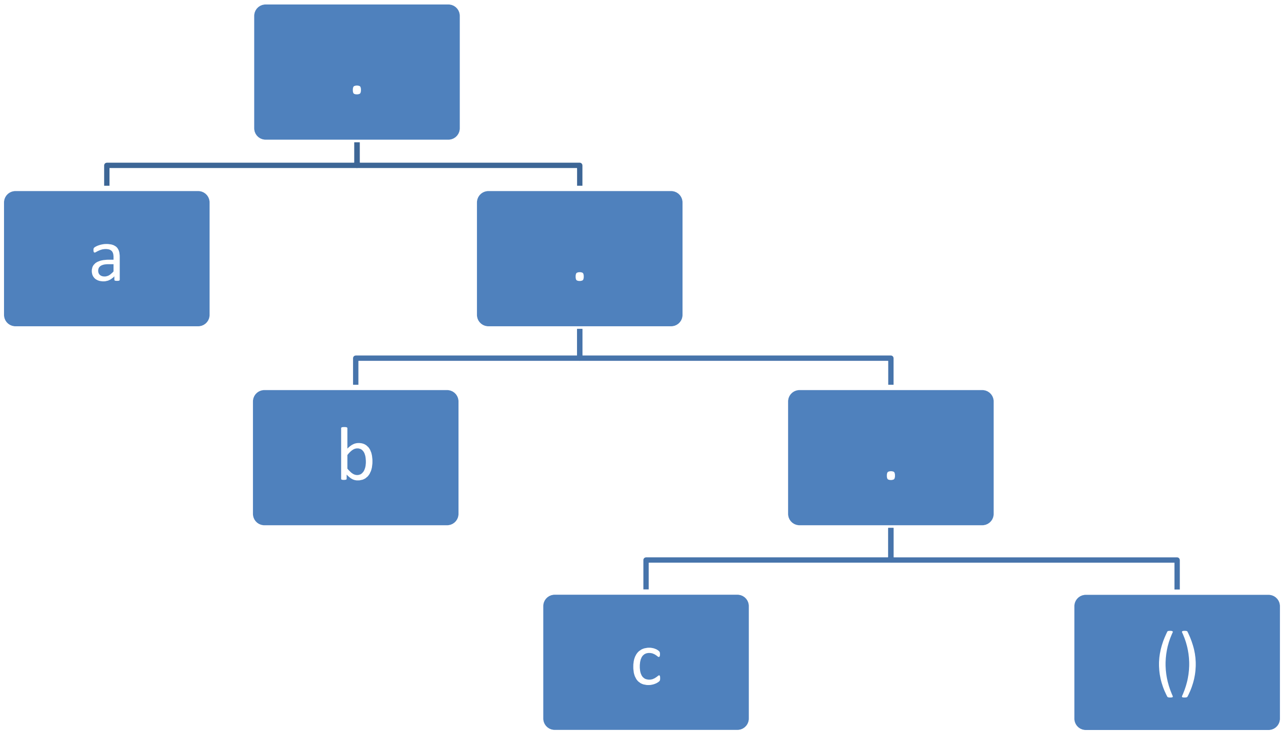
List notation	**Dot notation**
(a)	(a . ()) or (a . nil)
(a b)	(a . (b . nil))
(a b c)	(a . (b . (c . nil)))
((a b) c)	((a . (b . nil)) . (c . nil))
N/A	(a . b)

Nil is a constant. Its value can not be changed.

Numbers and quoted expressions are also constants.

Lists as binary trees

- (a b c) is (a . (b . (c . ())))



Heads and Tails

- car: returns the first element of the list (head)
  - Originating from a CPU instruction: Copy Address Register
- cdr: returns the list with first element missing (tail)
  - Originating from : Copy Decrement Register
- Examples:

  > (car ‘(a b c))
  A

  > (cdr ‘(a b c))
  (B C)

  > (car (cdr (car ‘((a b)))))
  B

More predefined functions

- cadr = car (cdr
- cadar = car (cdr (car
- cddaar, cadadr, ...
- Examples:

  > (cadr ‘(a b c))
  B

  > (cadar ‘((a b c)))
  B

Cons

- Construct a list using its head and tail
  - second argument must be a list

  > (cons ‘a ‘(b c))
  (A B C)

  > (cons ‘(a b) ‘(c d))
  ((A B) C D)

- Somehow an inverse for car and cdr pair
  > (setq x ‘(a b c))
  (A B C)

  > (cons (car x) (cdr x))
  (A B C)

- Cons is expensive, due to memory allocation and garbage collection

More list construction functions

- List: constructs a list of its arguments
  - any number of arguments

  > (list ‘a ‘b ‘c)
  (A B C)

  > (list ‘(1 2) ‘(3 4))
  ((1 2) (3 4))

- Append: constructs a list by appending its arguments
  - Any number of arguments
  - Arguments must be lists

  > (append ‘(a) ‘(b) ‘(c))
  (A B C)

  > (append ‘(1 2) ‘(3 4))
  (1 2 3 4)

Examples

- Use car and cdr to return x when applied to

  (a (b (x d)))

  (cdr ‘(a (b (x d)))) → ((b (x d)))

  (car (cdr ‘(a (b (x d))))) → (b (x d))

  (car (cdr (car (cdr ‘(a (b (x d))))))) →(x d)

  (car (car (cdr (car (cdr ‘(a (b (x d)))))))) →x

- What is the difference between these expressions?

  (car (setq x ‘(a b c))) → A

  (car ‘(setq x ‘(a b c))) → SETQ