CSC 242: Artificial Intelligence

Course website:

www.cs.rochester.edu/u/kyros/courses/csc242

Instructor: Kyros Kutulakos

Office: 623 CSB

Extension: x5-5860

Email: kyros@cs.rochester.edu

Hours: TR 3:30-4:30 (or by appointment)

TA: Joel Tetreault

Email: tetreaul@cs.rochester.edu

Hours: M 1:00-2:00

Recitations: TBA

Textbooks:

Russell & Norvig, *Artificial Intelligence*

Wilensky, *Common LISPcraft*

Another very good LISP textbook:
Winston & Horn, *LISP (Addison-Wesley)*

Common Lisp

- LISP is one of the most common “AI programming languages”

- LISP (=LISt Processing) is a language whose main power is in manipulating lists of symbols:

```
(a b c (d e f (g h)))
```

arithmetic operations are also included

```
(sqrt (+ (* 3 3) (* 4 4)))
```

- Lisp is a general-purpose, interpreter-based language

- All computation consists of expression evaluations:

```
lisp-prompt>(sqrt (+ (* 3 3) (* 4 4)))
lisp-prompt> 25
```

- Since data are list structures and programs are list structures, we can manipulate programs just like data

- Lisp is the second-oldest high-level programming language (after Fortran)

Getting Started

- Command-line invocation

  unix-prompt>cl

  system responds with loading & initialization messages followed by a Lisp prompt

  USER(1):

  whenever you have balanced parentheses & hit return, the value of the expression (or an error message) are returned

  ```
  USER(1): (+ 1 2)
  3
  ```

  saying “hello”

  ```
  USER(2): ‘hello
  HELLO
  USER(3): “Hello”
  “Hello”
  ```

- Exiting Lisp:

  ```
  USER(2):(exit)
  unix-prompt>
  ```

Getting Started (cont.)

- Reading a file f that is in the same directory from which you are running Lisp:

  `(load “f”)`

  system responds by reading & evaluating all expressions in the file & returning with a prompt for further interactive input; to read file from a different directory use

  `(load “/u/joe/lisp-prog/f”)`

  after all expressions in file are evaluated, you can use functions & constants that were defined in f

- Lisp file names conventionally have the **`.lisp`** extension

Basic Lisp Primitives

- Atom
  - Numeric
    - Fixed point
      `-17, 3400`
    - Fractions
      `1/2, 8/3`
    - Floating point
      `-17.0, 0.33333, 2.34`
  - Symbols (literal atoms)
    - Boolean values
      `T, NIL`
    - Other symbols
      `John, loves, Mary123, Book-1`
  - Characters
    `#\a, #\A, #\!`
  - Strings
    `“This is a string”`

- List: a left parenthesis, followed by 0 or more atoms, followed by a right parenthesis
  `(+ 1 2 3 4 5)`

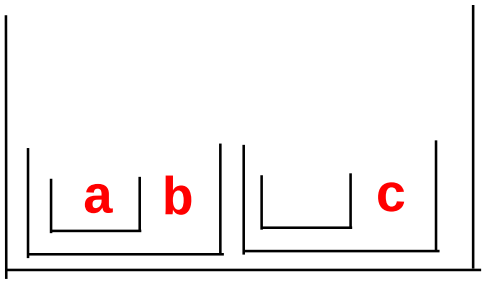
- Symbolic expression (s-expression)
  A list or an atom

Basic Lisp Primitives: Lists

- Lists provide an intuitive way to organize & represent concepts

```
(rochester (a-kind-of university)
           (location (rochester new-york)
           (phone 253-7000)
           (schools (computer-science
                    business
                    engineering)))
```

- Lists are like “bowls”




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

- Parentheses are extremely important when defining a Lisp list

- In Lisp, empty lists are significant

```
() = NIL
```

- In Lisp, “contains” means “directly contains”

Basic Lisp Primitives (cont.)

- Expressions

  a list expression:

```
USER(1): (+ 1 2 3 4)
10
```

  an atom expression:

```
USER(2): 5
5
```

- Expressions involving lists always use the prefix notation convention:

```
(function-name arg-1 arg-2 … arg-n)
```

- The prefix notation ensures uniformity since the function name is always the first argument in a list

- When producing the value of a list expression, the first element of the list generally is the name of the function used to obtain the value

- The process of computing the value of an expression is called an evaluation

- Lisp programs are just sequences of expressions

Basic Lisp Primitives (cont.)

- Quotes stop expression evaluation

```
(+ 1 2 3)
6
'(+ 1 2 3)
(+ 1 2 3)
 '(A B C)
(A B C)
(A B C)
ERROR
```

- Lisp binding operations provide a mechanism for assigning values to symbols

```
(setf ab-list '(a b))
(A B)
```

- The side effect of **`setf`** is to associate the symbol **`ab-list`** with the list **`(A B)`**

```
ab-list
(A B)
(setf one 1)
1
(+ one 2 3)
6
```

Basic List Manipulation

- **`first`** returns the first element of a list:

```
(first '(a b c d))
A
(first ())
NIL
(first '((a b) (c d)))
(A B)
```

- **`rest`** returns a list with the first element removed

```
(rest '(a b c d))
(B C D)
(rest '(c))
NIL
(rest '())
NIL
```

- obtaining the second element of a list:

```
(first (rest '(a b c d)))
```

Basic List Manipulation

- old-fashioned expressions for list manipulation:

```
(car '(a b c))
A
(cdr '(a b c))
(B C)
(cadr '(a b c))
B
(car (cdr '(a b c)))
B
(caddr '(a b c))
C
```

- higher-order list manipulation:

```
(nthcdr 3 '(a b c d e f g))
(D E F G)
(butlast '(a b c d e f g) 3)
(A B C D)
(butlast '(a b c d e f g) 100)
NIL
```

- concatenating an element to a list:

```
(cons 'a '(a b))
(A A B)
(cons '(a b) '(a b))
((A B) A B)
```

- appending lists:

```
(append '(a b c) '(d e f))
(A B C D E F)
```

Basic List Manipulation

- getting the length of a list:

```
(length '(a b c))
3
```

- reversing a list:

```
(reverse '(a b c))
(C B A)
```

- getting the last element of a list:

```
(last '(a b c d))
(D)
```

- creating a new list:

```
(list '(a b c) '(d e f))
((A B C) (D E F))
```

Basic Expression Evaluation

- Expressions like

```
'hello
(load "f")
(setf n 5)
```

  are complete stand-alone programs, not simply statements (as in other programming languages)

- Numbers, characters, strings, boolean literals evaluate to themselves
- Quotes prevent evaluation
- Unquoted expressions are like "function calls"

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

1. First element of list is the function's name
2. All remaining elements of list are first evaluated & then used as arguments to the function

```
(A B C D)
```

- Some functions do not evaluate all their arguments

```
(setf ab-list '(a b))
```

First argument is not evaluated

Basic Storage Handling

- Unlike languages such as C, C++, Pascal, etc, memory allocation & storage handling in Lisp are transparent to the programmer

- Internal steps taken by Lisp interpreter upon execution of

```
(setf ab-list '(a b c))
(A B C)
```

1. Allocate memory to store the internal representation of list '(a b c)
2. Bind the name ab-list to the newly-allocated chunk of memory

- Internal steps taken by Lisp interpreter upon execution of

```
(append ab-list '(a b))
(A B C A B)
```

1. Allocate memory to store the internal representation of list '(a b)
2. Evaluate the symbol ab-list
3. Allocate memory to store the internal representation of the list (A B A B C)

- All list manipulation operations considered so far are <u>non-destructive</u>

```
ab-list
(A B C)
```

Defining New Functions

- Lisp programs are generally created by writing new functions that are composed of a sequence of expressions:

```
(defun hello () “Hello there!”)
HELLO
```

the side-effect of **defun** is to bind the name **hello** to a function that returns **“Hello there”**

- Function definitions are evaluated just like any other Lisp expression

```
(defun <function-name>
       (<arg-1> <arg-2> ...)
       <expression-1>
       <expression-2>
          ... )
```

- Executing a Lisp function:

```
(<function-name> <arg-1> <arg-2>...)

(hello)
“Hello there”
```

Data Structures

- Already seen:

  Atoms, lists

- Other data structures:

  Association lists, arrays, structures, dotted pairs, functions, streams, ...

- Association lists are just lists of sub-lists

```
(setf sarah '((height .54) (weight 4.4))
```

```
              key    value
```

- Retrieving elements from an association list

```
(assoc <key> <association-list>)

(assoc 'height sarah)
(HEIGHT .54)
```

Data Structures (cont.)

- Creating a Lisp array:

```
(setf a (make-array '(3 4)
                    :initial-contents
                '(((a b c d) 2 3 4)
                  ("this" 5 6 7)
                  (#\c 8 9 10))))
```

- Retrieving an array element:

```
(setf value (aref a 2 3))
10
```

- Changing a value:

```
(setf (aref a 2 3) 12)
```

- Creating & manipulating a Lisp structure:

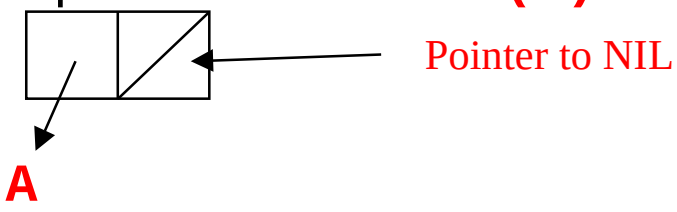
```
(defstruct course name time credits)
(setf csc242 (make-course
                    :name "AI"
                    :time 'TR300-415
                    :credits 4)
(setf (course-time csc242) 'TR200-315)
(course-time csc242)
TR200-315
```

Dotted Pairs

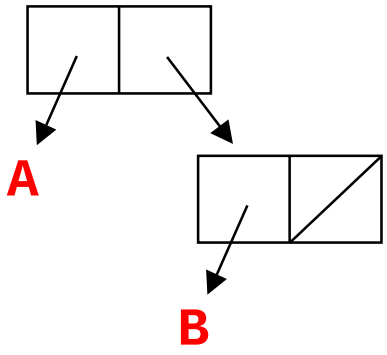
- We can usually think of one's data as atoms & lists; however, lists are actually built out of dotted pairs

- Dotted pairs provide a handle to Lisp's internal list representation

internal representation of list **(A)**



internal representation of list **(A B)**



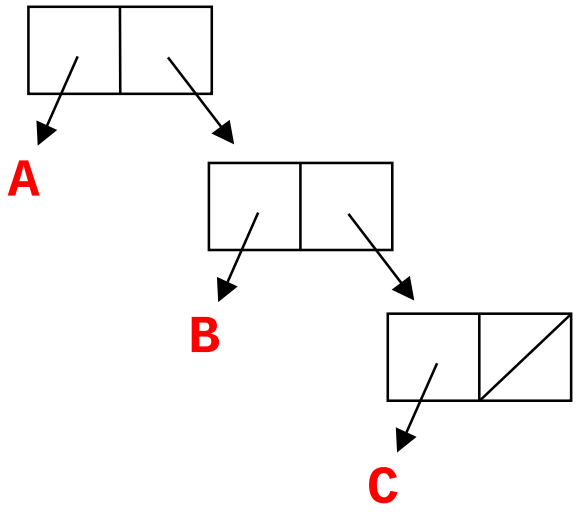
each "cell represents a dotted pair:

(A)	is	**(A . NIL)**
(A B)	is	**(A . (B . NIL))**

- Dotted pairs are especially useful for manipulating lists efficiently & for creating complex data structures (e.g., binary trees)

Dotted Pairs (cont.)

- internal representation of list `(A B C)`

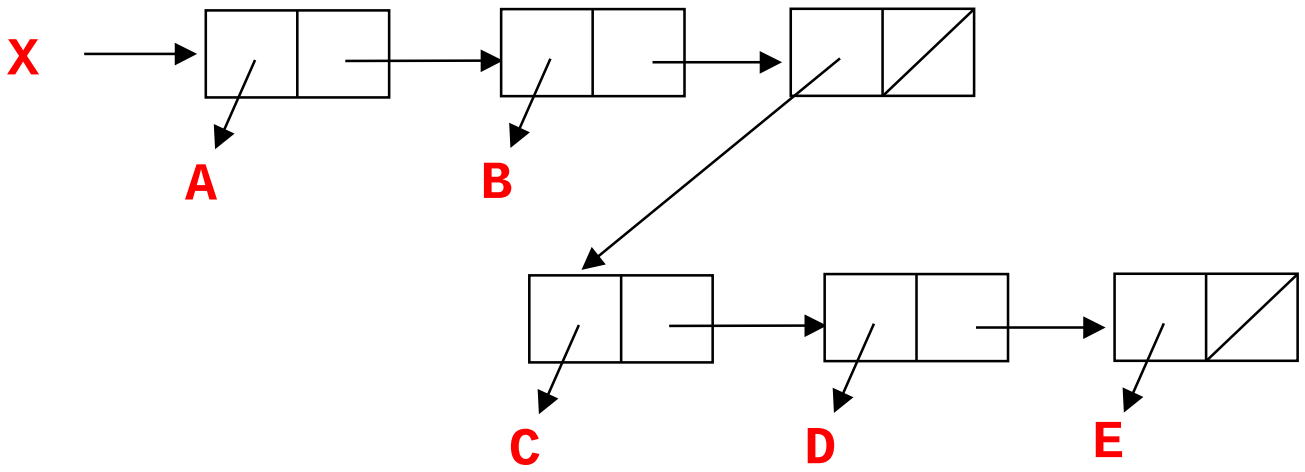


dotted pair representation:

```
(A . (B . (C . NIL)))
```

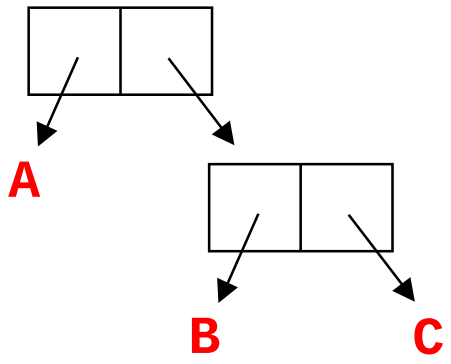
- effect of expression

```
(setf x '(a b (c d e)))
```



Dotted Pairs (cont.)

- Dotted pairs can be used to represent binary trees that are not lists:



dotted pair representation:

```
(A . (B . C))

(A B . C)
```

Dotted Pairs: CAR & CDR

- The left part of a dotted pair (the destination of the left pointer) is called its **`car`** and the right part its **`cdr`** (pronounced “could’r”)

- Examples:

```
(car ‘(a)) ⇒ (car ‘(a . NIL))
A
(crd ‘(a)) ⇒ (cdr ‘(a . NIL))
NIL
(car ‘(a b)) ⇒ (car ‘(a . (b .
NIL)))
A
(cdr ‘(a b)) ⇒ (cdr ‘(a . (b .
NIL)))
⇒(B.NIL)
(B)
```

- In reality, car and cdr simply manipulate pointers to lists

```
(car ‘(a b)) equiv. to (first ‘(a b))
(crd ‘(a b)) equiv. to (rest ‘(a b))
```

Dotted Pairs (cont.)

- Constructing a dotted pair:

```
(cons 'a 'b)
(A.B)
(cons 'a NIL)
(A)
```

- When the second argument of cons is a list, we can think of its operation as inserting the first argument at the head of the list given as second argument:

```
(cons 'a '(b c))
(A B C)
(cons '(a b) '(c (d e)))
((A B) C (D E))
(cons () ())
(())
(NIL)
```

Summary

- Basic Lisp primitives
- Manipulating lists in Lisp
- Expressions in Lisp & their evaluation
- Defining simple functions
- Basic Lisp data structures
- Dotted pairs

Next time:

- Joel will be talking about functions, iteration, etc
- First programming assignment