

IS2450

AI Programming Tools

[www.sis.pitt.edu/~is2450](http://www.sis.pitt.edu/~is2450)

Description/Objectives

LISP programming language

- unique features

- functional programming

  - macros, closures,

- program abstraction

Programming paradigms

- pattern matching, search, indexing.

AI programming paradigms

- rule-based programming

  - logic programming

  - production systems

- object programming

- knowledge representation

- natural language parsing

## LISP

### Reasons for LISP

- rapid prototyping vs. conventional software engineering
  - problems that aren't understood
  - incremental development
    - writing new languages
    - delay making decisions
- AI problems involve uncertainty, search, dynamic memory

### Features of Lisp

- Functional Programming
  - applicative programming
  - first class user defined functions
    - vs. state-oriented, implicative programming (assignment statements)

- Recursion

- Dynamic Storage Allocation
  - and garbage collection

- Simple Uniform Syntax

- Programs = Data
  - functions are first class data structures
  - macros

- Dynamic Data Typing – Manifest Typing

- Generic Functions

- Dynamic Programming Environment

- Interpreted Code and Incremental Compiling

- Extensibility

  - see “On Lisp” by Grahm

```
; Lisp
(defun sum (n)
  (let ((s 0))
    (dotimes (i n s)
      (incf s i))))

/* C */
int sum(int n) {
  int i, s = 0;
  for(i = 0; i < n; i++)
    s += i;
  return(s);
}
```

```
; Lisp
(defun addn (n)
  #'(lambda (x)
      (+ x n)))
```

```
(mapcan (addn 5) '(3 4 5))
```

## 1. INTRODUCTION

### Syntax

atoms: symbols, numbers, strings

lists: (+ 2 2)

evaluation rule

nesting (+ (\* 5 5) 2)

quoting (append '(Pat Kim) '(Robin Sandy))

(append (quote (Pat Kim)) (quote (Robin Sandy)))

(length '(a b (c d) e))

no distinction between statements (effects) and expressions (values)

pure lisp no effects just function evaluation

(append '(Pat Kim) '(Robin Sandy))

vs.

(setf freinds (append '(Pat Kim) '(Robin Sandy)))

symbols as variables

### 1.3 Special Forms

similar to statements in other languages  
but always lists, return a value  
refers to both symbols and expressions  
(s. f. operators and s. f. expressions)

setf  
quote (')  
defun  
defparameter  
let  
case  
if  
function (#)

macros can also alter order of evaluation  
and  
or

### 1.4 Lists

append, length  
**first**, (car), second, third, fourth,  
**cons**, **rest**, (cdr)  
last

(list 'my (+ 2 1) "Sons")

(cons `a (cons `b (cons `c nil )))

nil –two meanings: (), false  
true = anything else  
so predicates return false vs. true + useful information

## 1.5 Defining New Functions

```
(defun function-name (parameter...)  
  "documentation string"  
  function-body.....)
```

use of renamed functions- **<abstraction>**

```
(defun first-name (name)  
  "Return the first name from a name represented as list."  
  (first name))
```

```
(defun last-name (name)  
  "Return the last name from a name represented as list."  
  (first (last name)))
```

## 1.6 Using Functions

```
(setf names '((John Q Public) (Malcom X)
             (Admiral Grace Murry Hopper)
             (Madam Major General Paula Jones)))
```

```
> (mapcar #'last-name names)
(public x hopper jones)
```

n-ary functions

```
> (mapcar #'+ '(1 2 3) '(10 20 30))
```

```
(defparameter *titles*
  '(Mr Mrs Miss Ms Admiral Major General Madam)
  "A list of titles .....")
```

```
(defun first-name (name)
  "Return the first name from a name represented as list."
  (if (member (first name) *titles*)
      (first-name (rest name))
      (first name)))
```

member  
if, trace, untrace

function  
special forms

```
(trace first-name)
```

### recursion

requirement: return correct value =  
              must return value  
              not return any incorrect values  
stopping case(s)  
recursive case(s)

```
(defun our-member (obj lst)
  (if (null lst)
      nil
      (if (eql (car lst) obj)
          lst
          (our-member obj (cdr lst ))))))
```

## I/O

```
(format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3 ))  
2 plus 3 equals 5.  
NIL
```

side effect  
value

```
(defun askem (string)  
  (format t "~A" string)  
  (read))
```

```
(askem "how old are you")
```

## Variables

local variables (local scope)

```
(let ((x 1) (y 2))  
  (+ x y))
```

```
(defun ask-number ()  
  (format t "Please enter a number. ")  
  (let ((val (read)))  
    (if (numberp val)  
        val  
        (ask-number))))
```

global variables

```
(defparameter *globe* 99)
```

avoid clobbering with \* convention

```
(defconstant limit (+ *glob* 1))
```

```
(boundp `*glob*)
```

## Assignment

(setf \*glob\* 98)

on global

(let ((n 10))  
 (setf n 2)  
 n)

on local

(setf x (list `a `b `c))

creates global implicitly

setting the value at a (settable) place

(setf (car x) `n)

(multiple settings

(setf a b c d e f)

## 1.7 Higher-Order Functions

functions that take functions as arguments

```
(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```

append	function
apply	function

```
(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (if (null the-list)
      nil
      (append (funcall fn (first the-list))
              (mappend fn (rest the-list)))))
```

### anonymous functions

lambda expressions

(lambda (*parameters....*) *body....*) is a function name

so ((lambda (x) (+ x)) 4) ⇒ 6

(funcall #'(lambda (x) (+ x 2)) 4) ⇒ 6

1. avoid unnecessary names
2. create new functions at run time- closures
3. efficiency- avoid overhead of function name and storage

symbols

value slot  
function slot

Lisp vs. Scheme

## 1.8 Other Data Types

(about 25)

so far: numbers, symbols, lists, functions

also: vectors, arrays, structures, characters, streams, hash tables, etc.

now: strings

- evaluate to selves

- surround by double quotes

- sequence operators (lists, strings etc.) overloading/complex function

  - length on strings

## 2. A Simple Lisp Program

### 2.1 Grammar for Subset of English

Sentence  $\Rightarrow$  Noun-Phrase + Verb-Phrase

Noun-Phrase  $\Rightarrow$  Article + Noun

Verb-Phrase  $\Rightarrow$  Verb + Noun-Phrase

Article  $\Rightarrow$  the, a, ....

Noun  $\Rightarrow$  man, ball, woman, table ...

Verb  $\Rightarrow$  hit, took, saw, liked .....

### 2.2 Generating Sentences

grammar rules as Lisp functions

```
(defun sentence ()      (append (noun-phrase)(verb-phrase)))
(defun noun-phrase ()  (append (Article) (Noun)))
(defun verb-phrase ()  (append (Verb) (noun-phrase)))
(defun Article ()      (one-of '(the a)))
(defun Noun ()         (one-of '(man ball woman table)))
(defun Verb ()         (one-of '(hit took saw liked)))
```

```
(defun one-of (set)
  "Pick one element of set, and make a list of it."
  (list (random-elt set)))
```

```
(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))
```

```
(trace sentence noun-phrase verb-phrase Article Noun Verb)
```

but recursive definitions?

```
(defun Adj* ()
  (if (= (random 2) 0)
      nil
      (append (Adj) (Adj*))))
```

```
(defun PP* ()
  (if (random-elt '(t nil))
      (append (PP) (PP*))
      nil))
```

```
(defun noun-phrase () (append (Article) (Adj*) (Noun) (PP*)))
(defun PP () (append (Prep) (noun-phrase)))
(defun Adj () (one-of '(big little blue green adiabatic)))
(defun Prep () (one-of '(to in by with on)))
```

## 2.3 A Rule-Based Solution

```
(defparameter *simple-grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defvar *grammar* *simple-grammar*
  "The grammar used by generate. Initially, this is
  *simple-grammar*, but we can switch to other grammars.")

(defun rule-lhs (rule)
  "The left hand side of a rule."
  (first rule))

(defun rule-rhs (rule)
  "The right hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))

(defun generate (phrase)
  "Generate a random sentence of phrase."
  (cond ((listp phrase) ; if rewrite produced list of symbols
        (mappend #'generate phrase))
        ((rewrites phrase) ; choose random rewrite for symbol
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))) ; no rewrite must be a terminal symbol

(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))
```