Introduction

CSC 358/458
4.1.2003
Outline

- Course goals
- Course organization
- History of Lisp
- Lisp syntax
- Examples
- Lisp IDE
Symbolic Programming

- Programming as manipulation of symbols
Math without Symbols

... a square and 10 roots are equal to 39 units. The question therefore in this type of equation is about as follows: what is the square which combined with ten of its roots will give a sum total of 39? The manner of solving this type of equation is to take one-half of the roots just mentioned. Now the roots in the problem before us are 10. Therefore take 5, which multiplied by itself gives 25, an amount which you add to 39 giving 64. Having taken then the square root of this which is 8, subtract from it half the roots, 5 leaving 3. The number three therefore represents one root of this square, which itself, of course is 9. Nine therefore gives the square.
Math with Symbols

\[ x^2 + bx = c \]

\[ x = \left( \sqrt{\left( \frac{b}{2} \right)^2 - c} \right) - \frac{b}{2} \]
Symbolic Programming

- Symbolic representation of a problem
- Formal operations to manipulate the representations
- Closely related to AI
Topics

- Introduction
- Lists and Trees
- Functions
- Strings and I/O
- Control structures
- Midterm
- AI application: pattern matching
- Macros
- AI application: search
- Structures and objects
Lisp origins

- 2\textsuperscript{nd} Oldest Programming Language
  - John McCarthy, MIT 1960s
- Originally intended as an internal language for working with FORTRAN programs
  - developed a life of its own
- LISt Processing
Lisp History

- Early implementations
  - very slow
  - purely interpreted
  - widely varying semantics
  - different function libraries

- 80s
  - AI boom
  - need for portable Lisp
  - CommonLisp initiative
Lisp Today

- Many Lisp ideas become mainstream
  - prototyping
  - dynamic memory management
- Modern Lisp
  - very fast
  - multi-paradigm programming
  - extensible
Syntax

- Basic elements of the language
- Relationships between them
public class Multiplier implements NumberSequence {
    private NumberSequence m_sequence;
    private int m_multiplier;

    public Multiplier (int multiplier, NumberSequence baseSequence) {
        m_sequence = baseSequence;
        m_multiplier = multiplier;
    }

    public boolean hasMoreNumbers () {
        return m_sequence.hasMoreNumbers();
    }

    public int nextNumber () {
        return m_multiplier * m_sequence.nextNumber();
    }
}

Lisp Syntax

(defun multiplier (n seq)
  (values (function (lambda () (* n (pop seq))))
  (function (lambda () (not (null seq))))))
Lisp Syntax

- Every expression is either
  - a list
    - (a b c)
  - an atom
    - 5

- Atoms are evaluated
  - some self-evaluate
  - symbols look for bindings
Lisp syntax

- Lists
  - the first element is evaluated as a function
  - each other element is evaluated
  - then the function is called with the other elements as arguments
  - a value is returned
Exception

- If first element
  - is the name of a special forms or macro
  - then all arguments may not be evaluated

- Example
  - (if (over-target) (drop-bomb) (wait-longer))
Evaluation

- a
- pi
- "a"
- #\a
- (cos pi)
- `(cos pi)"
Lisp environment

- Allegro Common Lisp
  - free download www.franz.com
  - labs
  - terminal server
Quote

- Prevents evaluation
  - crucial for symbols
- Obviously a special form
- `(quote a) => a`
- Abbreviation
  - '
- `'a => a`
Functional values

- Different kind of quoting
  - (function tan) => #<Function TAN>
- Abbreviation
  - #'
  - #'tan => #<Function TAN>
- We can pass functions as arguments
  - (funcall #'tan 1) equivalent to
  - (tan 1)
Lists

- Basic Lisp data type
  - (a b c)

- Front
  - a
  - "car"

- Rest
  - (b c)
  - "cdr"

- nil = empty list
Lists, cont'd

- Assembling lists
  - (list 'a 'b 'c)
  - (cons 'a '(b c))
  - (append '(a) '(b c))

- Lists can contain lists
  - (a (b c) (d e (f)))

- Lisp programs are lists!
Assignment

- Bind a symbol to a value
  - (setf a '(1 2 3))

- Also
  - (set 'a '(1 2 3))
  - then (setq a '(1 2 3))

- Setf generalizes setq
  - first argument can be a function call
  - (setf (car a) 0)
Defining functions

- `(defun name (args) body)`
- Examples
  - `(defun square (x) (* x x))`
  - `(defun sum-sq (x y) (+ (* x x) (* y y)))`
- Recursive function
  - factorial
Truth values

- T/nil
  - actually any non-nil value is assumed to be T
Conditions

- **Syntax**
  - `(if exp1 exp2 [exp3])`
- **If exp1 evaluates to T**
  - evaluate exp2 and return the value
  - otherwise evaluate exp3 and return it
  - if no exp3, return nil
And / Or / Not

- (and exp1 exp2 ... )
  - keep evaluating while T
- (or exp1 exp2 ... )
  - keep evaluating while nil
- (not exp1)
  - reverse truth value
Cond

- Generalized conditional
  - (cond clause1 ... clausen)
  - clause = (condition value)

- Example
  
  (defun interest-rate (money)
    (cond ((< money 0) 0)
          ((< money 1000) 2)
          ((< money 10000) 5)
          ((< money 100000) 7)
          (t 10)))

  (defun interest-rate (money)
    (cond ((< money 0) 0)
          ((< money 1000) 2)
          ((< money 10000) 5)
          ((< money 100000) 7)
          (t 10)))
Let

- Defining local variables
  
  \[
  \text{(let ((var1 val1) \ldots (varn valn)) body)}
  \]

- Example
  
  \[
  \text{(defun fielding (put-outs assists errors)}
  
  \text{(let ((numerator (+ put-outs assists))}
  
  \text{(denom (+ put-outs assists errors)))}
  
  (\text{/ numerator denom))))}
  \]
Let*

- Same syntax
- but assignments made sequentially
- Example

(defun fielding (put-outs assists errors)
  (let* ((numerator (+ put-outs assists))
         (denom (+ numerator errors)))
    (/ numerator denom)))
Extended Example

- polynomials