Lisp and Scheme I

Versions of LISP

- LISP is an acronym for LISt Processing language
- Lisp is an old language with many variants
  - Fortran is the only older language still in wide(?) use
  - Lisp is alive and well today
- Most modern versions are based on Common Lisp
- Scheme is one of the major variants
  - We'll use Scheme, *not* Lisp, in this class
- The essentials haven't changed much

Why Study Lisp?

- It's a simple, elegant yet powerful language
- You will learn a lot about PLs from studying it
- We'll look at how to implement a Scheme interpreter in Scheme
- Many features, once unique to Lisp, are now in "mainstream" PLs: python, javascript, perl, R …
- It will expand your notion of what a PL can be
- Lisp is considered hip and esoteric by some, but not all, computer scientists

LISP Features

- **S-expression as the universal data type** – either as atom (e.g., number, symbol) or a list of atoms or sublists
- **Functional Programming Style** – computation done by applying functions to arguments, functions are first class objects, minimal use of side-effects
- **Uniform Representation of Data and Code** – (A B C D) can be interpreted as data (i.e., a list of four elements) or code (calling function 'A' to the three parameters B, C, and D)
- **Reliance on Recursion** – iteration is provided too, but recursion is much more natural
- **Garbage Collection** – frees programmer from explicit memory management

What’s Functional Programming?

- FP: computation is applying functions to data
- Imperative or procedural programming: a program is a set of steps to be done in order
- FP eliminates or minimizes side effects and mutable objects that create/modify state
- FP treats functions as objects that can stored, passed as arguments, composed, etc.

Pure Lisp and Common Lisp

- Lisp has a small and elegant conceptual core that has not changed much in almost 50 years.
- McCarthy's original Lisp paper http://www-formal.stanford.edu/jmc/recursive.pdf defined all of Lisp using just **seven** primitive functions
- Common Lisp is large (> 800 built-in functions), has all the modern data-types, good programming environments, and good compilers.

Scheme

- Scheme is a dialect of Lisp that is favored by people who teach and study programming languages
- Why?
  - It's simpler and more elegant than Lisp
  - It's pioneered many new programming language ideas (e.g., continuations, call/cc)
  - It's influenced Lisp (e.g., lexical scoping of variables)
  - It's still evolving, so it's a good vehicle for new ideas

But I want to learn Lisp!

- Lisp is used in many practical systems, but Scheme is not
- Learning Scheme is a good introduction to Lisp
- We can only give you a brief introduction to either language, and at the core, Scheme and Lisp are the same
- We'll point out some differences along the way

But I want to learn Clojure!

- Clojure is a new Lisp dialect that compiles to the Java Virtual Machine
- It offers advantages of both Lisp (dynamic typing, functional programming, closures, etc.) and Java (multi-threading, fast execution)
- We might look at Clojure briefly later

DrScheme and MzScheme

- The PLT Scheme system was developed by a group of academics (Brown, Northeastern, Chicago, Utah)
- It's most used for teaching introductory CS courses
- MzScheme is the basic Scheme engine and can be called from the command line and assumes a terminal style interface
- DrScheme is a graphical programming environment for Scheme

mzscheme on gl

```
linux1.gl.umbc.edu> pwd
/afs/umbc.edu/users/n/i/nicholas/home/courses/331
linux1.gl.umbc.edu> cat test.ss
(define (add2 x) (+ x 2))

(define (square x) (* x x))
linux1.gl.umbc.edu> mzscheme
Welcome to MzScheme v4.1.1 [3m], Copyright (c) 2004-2008 PLT Scheme Inc.
> (load "test.ss")
> (add2 3)
5
> (square 1.41421)
1.9999899240999999
> (square (add2 1))
9
> (exit)
linux1.gl.umbc.edu>
```

Untitled - DrScheme
Untitled
(define ...)
Save
Debug
Check Syntax
Run
Stop
(define (add2 x) (+ x 2))
(define (square x) (* x x))
drscheme
Welcome to DrScheme, version 4.1 [3m].
Language: Advanced Student custom; memory limit: 128 megabytes.
Teachpack: matrix.ss.
This program should be tested.
> (add2 100)
102
> square
square
> (square 200)
40000
>
Advanced Student custom
11:2

Informal Syntax

- An *atom* is either an integer or an identifier
- A *list* is a left parenthesis, followed by zero or more S-expressions, followed by a right parenthesis
- An ***S-expression*** is an atom or a list
- Example: ()
- (A (B 3) (C) (( )))

Hello World

```
(define (helloWorld)
  ;; prints and returns the message.
  (printf "Hello World\n"))
```

Square

```
> (define (square n)
      ;; returns square of a numeric argument
      (* n  n))
> (square 10)
100
```

REPL

- Lisp and Scheme are interactive and use what is known as the "read, eval, print loop"
  - **While true**
    - **Read** one expression from the open input
    - **Evaluate** the expression
    - **Print** its returned value
- (define (repl) (print (eval (read))) (repl))

What is evaluation?

- We evaluate an expression producing a value
  - Evaluating "2 + sqrt(100)" produces 12
- Scheme has a set of rules specifying how to evaluate an s-expression
- We will get to these very soon
  - There are only a few rules
  - Creating an interpreter for scheme means writing a program to
    - read scheme expressions,
    - apply the evaluation rules, and
    - print the result

Built-in Scheme Datatypes

Basic Datatypes

- Booleans
- Numbers
- Strings
- Procedures
- Symbols
- Pairs and Lists

The Rest

- Bytes & Byte Strings
- Keywords
- Characters
- Vectors
- Hash Tables
- Boxes
- Void and Undefined

Lisp: T and NIL

- NIL is the name of the empty list, ( )
- As a boolean, NIL means “false”
- T is usually used to mean “true,” but…
- …anything that isn’t NIL is “true”
- NIL is both an atom and a list
  - it’s defined this way, so just accept it

Scheme: #t, #f, and ‘()

- Scheme’s boolean datatype includes #t and #f
- #t is a special symbol that represents true
- #f represents false
- In practice, anything that’s not #f is true
- Booleans evaluate to themselves
- Scheme represents empty lists as the literal ( ) which is also the value of the symbol *null*

Numbers

- Numbers evaluate to themselves
- Scheme has a rich collection of number types including the following
  - Integers (42)
  - Floats (3.14)
  - Rationals: (/ 1 3) => 1/3
  - Complex numbers: (* 2+2i -2-2i) => 0-8i
  - Infinite precision integers: (expt 99 99) => 369...99 *(contains 198 digits!)*
  - And more…

Strings

- Strings are fixed length arrays of characters
  - "foo"
  - "foo bar\n"
  - "foo \"bar\"”
- Strings are immutable
- Strings evaluate to themselves

Predicates

- A predicate (in any computer language) is a function that returns a boolean value
- In Lisp and Scheme predicates return either #f or often something else that might be useful as a true value
  - The member function returns true iff its first argument is in the list that is its second argument
  - (member 3 (list 1 2 3 4 5 6)) => (3 4 5 6))

Function calls and data

- A function call is written as a list
  - the first element is the name of the function
  - remaining elements are the arguments
- Example: (F A B)
  - calls function F with arguments A and B
- Data is written as atoms or lists
- Example: (F A B) is a list of three elements
  - Do you see a problem here?

Simple evaluation rules

- Numbers evaluate to themselves
- #t and #f evaluate to themselves
- Any other atoms (e.g., foo) represents variables and evaluate to their values
- A list of n elements represents a function call
  - e.g., (add1 a)
  - Evaluate each of the n elements (e.g., add1->a procedure, a->100)
  - Apply function to arguments and return value

Example

```
(define a 100)
> a
100
> add1
#<procedure:add1>
> (add1 (add1 a))
102
> (if (> a 0) (+ a 1)(- a 1))
103
```

- *define* is a *special form* that doesn’t follow the regular evaluation rules
  - Scheme only has a few of these
- Define doesn’t evaluate its first argument
- *if* is another special form
  - What do you think is special about if?

Quoting

- Is (F A B) a call to F, or is it just data?
- All *literal data* must be quoted (atoms, too)
- (QUOTE (F A B)) is the list (F A B)
  - QUOTE is not a function, but a **special form**
  - Arguments to a special form aren't evaluated or are evaluated in some special manner
- '(F A B) is another way to quote data
  - There is just one single quote at the beginning
  - It quotes *one* S-expression

Symbols

- Symbols are atomic names

  > ’foo

  foo

  > (symbol? ‘foo)

  #t

- Symbols are used as names of variables and procedures
  - (define foo 100)
  - (define (fact x) (if (= x 1) 1 (* x (fact (- x 1)))))

Basic Functions

- car returns the head of a list

  (car ‘(1 2 3)) => 1

  (first ‘(1 2 3)) => 1 ;; for people who don’t like car

- cdr returns the tail of a list

  (cdr ‘(1 2 3)) => (2 3)

  (rest ‘(1 2 3)) => (2 3) ;; for people who don’t like cdr

- cons constructs a new list beginning with its first arg and continuing with its second

  (cons 1 ‘(2 3)) => (1 2 3)

More Basic Functions

- eq? compares two atoms for equality

  (eq ‘foo ‘foo) => #t

  (eq ‘foo ‘bar) => #f

  Note: eq? is just a pointer test, like Java’s ‘=‘
- equal? tests two list structures

  (equal? ‘(a b c) ‘(a b c)) =#t

  (equal? ‘(a b) ‘((a b))) => #f

  Note: equal? compares two complex objects, like a Java object’s equal method

Other useful Functions

- (null? S) tests if S is the empty list
  - (null? ‘(1 2 3) => #f
  - (null? ‘()) => #t
- (list? S) tests if S is a list
  - (list? ‘(1 2 3)) =>#t
  - (list? ‘3) => #f

More useful Functions

- list makes a list of its arguments
  - (list 'A '(B C) 'D) => (A (B C) D)
  - (list (cdr '(A B)) 'C) => ((B) C)
- Note that the parenthesized prefix notation makes it easy to define functions that take a varying number or arguments.
  - (list ‘A) => (A)
  - (list) => ( )
- Lisp dialects use this flexibility a lot

More useful Functions

- append concatenates two lists
  - (append ‘(1 2) ‘(3 4)) => (1 2 3 4)
  - (append '(A B) '((X) Y)) => (A B (X) Y)
  - (append ‘( ) ‘(1 2 3)) => (1 2 3)
- append takes any number of arguments
  - (append ‘(1) ‘(2 3) ‘(4 5 6)) => (1 2 3 4 5 6)
  - (append ‘(1 2)) => (1 2)
  - (append) => null
  - (append null null null) => null

If then else

- In addition to cond, Lisp and Scheme have an if special form that does much the same thing
- (if <test> <then> <else>)
  - (if (< 4 6) ‘foo ‘bar) => foo
  - (if (< 4 2) ‘foo ‘bar) => bar
- In Lisp, the then clause is optional and defaults to null, but in Scheme it’s required

Cond

- cond (short for conditional) is a special form that implements the *if ... then ... elseif ... then ... elseif ... then ...* control structure

```
(COND
      (condition1   result1 )
      (condition2   result2 )
      . . .
      (#t   resultN ) )
```

Cond Example

```
(cond
  ((not (number? x)) 0)
  ((< x 0)  0)
  ((< x 10)  x)
  (#t  10)
)
```

```
(if (not (number? x))
    0
    (if (<x 0)
        0
        (if (< x 10)
            x
            10)))
```

Defining Functions

(DEFINE (*function_name parameter_list)*
function_body)

Examples:

```
;; Square a number
(define (square n) (* n n))


;; absolute difference between two numbers.
(define (diff x y)  (if (> x y)  (- x y) (- y x)))
```

Example: member

- member is a built-in function, but here's how we'd define it

```
(define (member x lst)
   ;; x is a top-level member of a list if it is the first
   ;; element or if it is a member of the rest of the list
  (cond ((null lst) #f)
        ((equal x (car lst)) list)
        (#t (member x (cdr lst)))))
```

Example: member

- We can also define member using if:

```
(define (member x lst)
      (if (null> lst) null
          (if (equal x (car lst)) lst
              (member x (cdr lst)))))
```

- We could also define it using boolean ops not, and & or, all built-in

```
(define (member x lst)
   (and (not (null lst))
        (or (equal x (car lst))
            (member x (cdr lst)))))
```

Example: define append

- (append ‘(1 2 3) ‘(a b)) => (1 2 3 a b)
- Here are two versions, using if and cond:

```
(define (append l1 l2)
    (if (null l1)
        l2
        (cons (car l1) (append (cdr l1) l2)))))

(define (append l1 l2)
    (cond ((null l1) l2)
            (#t (cons (car l1) (append (cdr l1) l2)))))
```

Example: SETS

- Implement sets and set operations: union, intersection, difference
- Represent a set as a list and implement the operations to enforce uniqueness of membership
- Here is set-add

```
(define (set-add thing set)
  ;; returns a set formed by adding THING to set SET

  (if (member thing set) set
      (cons thing set)))
```

Example: SETS

- Union is only slightly more complicated

```
(define (set-union S1 S2)
  ;; returns the union of sets S1 and S2
  (if (null? S1)
      S2
      (add-set (car S1)
               (set-union (cdr S1) S2))))
```

Example: SETS

Intersection is also simple

```
(define (set-intersection S1 S2)
   ;; returns the intersection of sets S1 and S2
   (cond ((null s1) nil)
            ((member (car s1) s2)
             (set-intersection (cdr s1) s2))
            (#t (cons (car s1)
                      (set-intersection (cdr s1) s2)))))
```

Reverse

- Reverse is another common operation on Lists
- It reverses the "top-level" elements of a list
  - That is, it constructs a new list equal to its argument with the top level elements in reverse order.
- (reverse '(a b (c d) e)) => (e (c d) b a)

```
(define (reverse L)
  (if (null? L)
     null
     (append (reverse (cdr L)) (list (car L)))))
```

Reverse is Naïve

- The previous version is often called naïve reverse because it's so inefficient
- What's wrong with it? two problems
  - The kind of recursion it does grows the stack when it does not need to
  - It ends up making lots of needless copies of parts of the list

Tail Recursive Reverse

- The way to fix the first problem is to employ tail recursion
- The way to fix the second problem is to avoid append.
- So, here is a better reverse:

```
(define (reverse2 L) (reverse-sub L null))

(define (reverse-sub L answer)
   (if (null? L)
       answer
       (reverse-sub (cdr L) (cons (car L) answer))))
```

Still more useful functions

- (LENGTH L) returns the length of list L
  - The "length" is the number of *top-level* elements in the list
- (RANDOM N) , where N is an integer, returns a random integer >= 0 and < N
- EQUAL tests if two S-expressions are equal
  - If you know both arguments are atoms, use EQ instead

Programs in files

- Use any text editor to create your program
- Save your program on a file with the extension .ss
- (Load “foo.ss”) loads foo.ss
- (load “foo.bar”) loads foo.bar
- Each s-exprssion in the file is read and evaluated.

Comments

- In Lisp, a comment begins with a semicolon (;) and continues to the end of the line
- Conventions for ;;; and ;; and ;
- Function document strings:

```
(defun square (x)
  "(square x) returns x*x"
  (* x x))
```

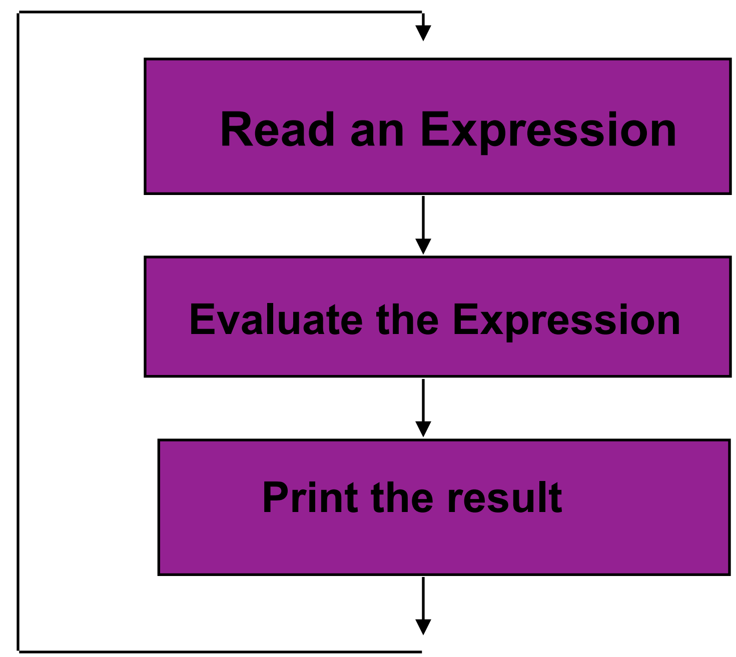
Read – eval - print

Lisp's interpreter essentially does:

(loop (print (eval (read)))

i.e.,

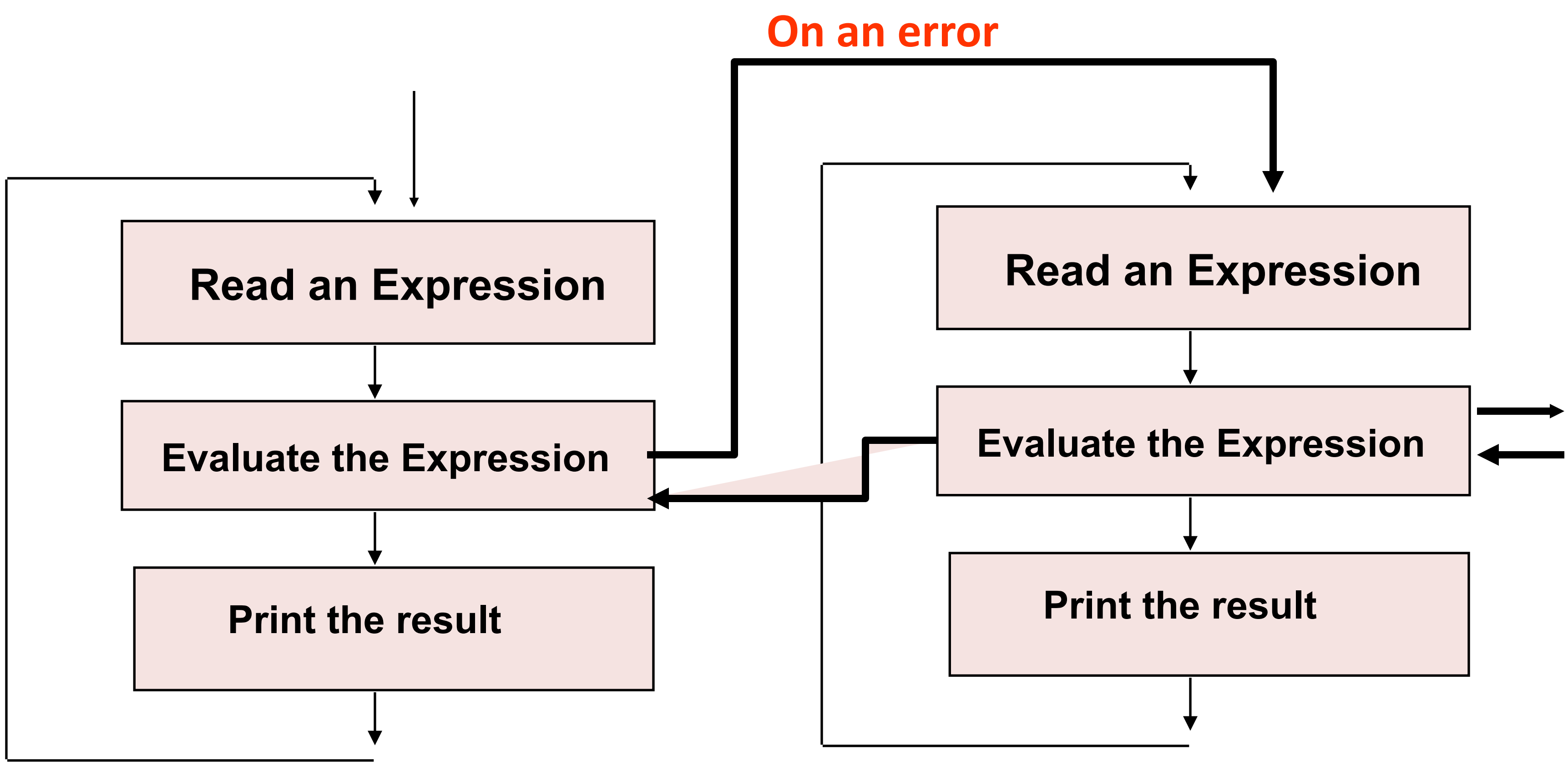
1. Read an expression
2. Evaluate it
3. Print the resulting value
4. Repeat from 1.



Understanding the rules for evaluating an expression is key to understanding lisp.

Reading and printing, while a bit complicated, are conceptually simple.

When an error happens



Eval(S)

- If S is an atom, then call evalatom(A)

- If S is a list, then call evallist(S)

EvalAtom(S)

- Numbers eval to themselves
- T evals to T
- NIL evals to NIL
- Atomic symbol are treated as variables, so look up the current value of symbol

EvalList(S)

- Assume S is (S1 S2 …Sn)
  - If S1 is an atom representing a special form (e.g., quote, defun) handle it as a special case
  - If S1 is an atom naming a regular function
    - Evaluate the arguments S2 S3 .. Sn
    - Apply the function named by S1 to the resulting values
  - If S1 is a list … more on this later …

Variables

- Atoms, in the right context, as assumed to be variables.
- The traditional way to assign a value to an atom is with the SET function (a special form)
- More on this later

```
[9]> (set! 'a 100)
100
[10]> a
100
[11]> (set 'a (+ a a))
200
[12]> a
200
[13]> b
*** - EVAL: variable B has no value
1. Break [14]> ^D
[15]> (set 'b a)
200
[16]> b
200
[17]> (set 'a 0)
0
[18]> a
0
[19]> b
200
[20]>
```

Input

- (read) reads and returns one s-expression from the current open input stream.

```
[1]> (read)
foo
FOO
[2]> (read)
(a b
  (1 2))
(A B (1 2))
```

```
[3]> (read)
3.1415
3.1415
[4]> (read)
-3.000
-3.0
```

Output

```
[1]> (print '(foo bar))
(FOO BAR)
(FOO BAR)
[2]> (setq *print-length* 3 )
3
[3]> (print '(1 2 3 4 5 6 7 8))
(1 2 3 ...)
(1 2 3 ...)
[4]> (format  t "The sum of one and one is ~s.~%"
               (+ 1 1))
The sum of one and one is 2.
NIL
```

Let

- (let <vars><s1><s2>…<sn>)
  – <vars> = (<var1>…<varn>)
  – <var1> = <name> or (<name>) or (<name> <value>)
- Creates environment with local variables v1..vn, initializes them in parallel and evaluates the <si>.
- Example:

```
>(let (x (y)(z (+ 1 2))) (print (list x y z)))
(NIL NIL 3)
(NIL NIL 3)
```

Iteration - DO

```
(do ((x 1 (1+ x))
      (y 100 (1- y)))
     ((> x y)(+ x y))
   (princ “Doing “)
   (princ (list x y))
   (terpri))
```