Lisp

Functions, recursion and lists

Interacting with Scheme

```
(define pi 3.14159)      ; bind pi to 3.14159

(lambda (x) (* x x))     ; anonymous function

(define sq (lambda (x) (* x x)))
(define (sq x) (* x x))  ;  (define sq (lambda (x) (* x x)))

(sq 100)                 ;  100 * 100

(if P E1 E2)             ;  if P then E1 else E2
(cond (P1 E1) (P2 E2) (else E3)) ; (if P1 E1 (if P2 E2 E3))

(let ((x1 E1) (x2 E2)) E3)  ;  declare local variables x1 and x2

(let* ((x1 E2) (x2 E2)) E3)  ; E2 can use x1 as a local variable
```

The Lisp Programming Language

- Stems from interest in symbolic computation
  - Led by John McCarthy in late 1950s
  - Designed for math logic in artificial intelligence
- Functional programming paradigm
  - A program is a expression
    - Expresses flow of data; map input values to output values
    - No side effects or modification to variables
    - No concept of control-flow or statements
  - Functions are first-class objects
    - A function can be used everywhere a regular value is used
    - Functions can take other functions as parameters and return other functions as results (higher-order functions)
- Adding side-effect operations
  - Different occurrences of expressions have different values
- Strength and weakness
  - ✓ Simplicity and flexibility
  - ✓ Build prototype systems incrementally
  - X Not many tools or libraries; low in efficiency (mostly interpreted)

Concepts in Lisp

- Supported value types
  - Atomic values: numbers (e.g. 3, 7.7), symbols (e.g. ‘abc), booleans
  - Compound data structures: lists (car, cons, cdr), functions (lambda)
- Supported operations
  - Function definition and function call
    - (define fname (lambda (parameters) body))
    - (fname arguments)
  - Predefined functions: cons, cond, if, car, cdr, eq?, ……
  - Nested blocks (local variables): let
- Variable declarations : introduces new variables
  - May bind value to identifier, specify type, etc.
  - Global vs. local variables: (define x ‘a) vs. (let ((x a)) (…))

Lists in Lisp/Scheme

- In Lisp/Scheme, a list may contain arbitrary types of values
  - ‘(a b c) ‘(+ 2 (* 3 5))   ‘(lambda (a b) (cons a b))
  - A dynamically typed list can be used to implement most pointer-based data structures, including lists and trees.
  - Can it be used to implement arbitrary graphs? (can we build cycles in lists?)
- Lisp/Scheme lists can be used to naturally implement AST --- a tree data structure used as an internal representation of programs in compilers/interpreters

```
       lambda                       +
      /       \                    /  \
   list      cons                 2     *
   /  \      /  \                      / \
  a    b    a    b                    3   5
```

Lisp Innovations in language design

- Functional programming paradigm
  - A program is composed of expressions
  - Functions are first-class objects
    - Support higher-order functions
  - Abstract view of memory (the Lisp abstract machine)
  - Program as data (dynamic interpretation of program)

Expressions vs Statements

- Expression (x+5)/2
  - Syntactic entity that has a value
  - Need not change accessible memory
    - If it does, has a *side effect*
- Statement load 4094 r1
  - Imperative command
  - Alters the contents of previously-accessible memory
- Example: inserting to an existing list
  - Via pure (side-effect-free) expressions in Lisp/Scheme

```
(define insert (lambda (x y) (cons x y)))
(insert 4 (insert 3 `())
```

  - How do we implement list insertion in C?

Expressions vs. Statements

- Compare to imperative programming in C

```
void insert( int x, Cell* y)  {
     Cell* z = (Cell*)malloc(sizeof(Cell));
     z->val = y->val; z->next = y->next;
     y->val = x; y->next = z;
    }
int main () { Cell* y = (Cell*)malloc(sizeof(Cell));
                    y->val=-1; y->next=0;
                    insert(3, y); insert(4, y); }
```

- Evaluation order
  - Among pure expressions: flow of data
    - Can evaluate each expression as soon as values are ready
  - Among statements: ordering of side effects (modifications)
    - Statement order cannot be changed unless proven otherwise
- Tradeoff: creating new values vs. modifying existing ones?
  - Copying vs. sharing of complex data structures
  - Modification efficiency vs. parallelization of computation

Lisp: Adding Side Effects

- Pure Lisp
  - Expressions do not modify observable machine states
- Impure Lisp
  - Allow modifications to memory. May increase efficiency of programs (eg. modify an element in a list)
    - (set! x y)   Replace the value of x with y
    - (rplacea ’(A B) y) or (set-car! ’(A B) y) Replace A with y
    - (rplaced ’(A B) y) or (set-cdr! ’(A B) y) Replace B with y
  - Sequence operator
    - (progn (set! x y) x) or (begin (set! x y) x)
      - Set the value of x to be y; then returns the value of x
- Compare Lisp with C
  - Lisp: no return statement, but needs operator for sequencing
  - C: no sequencing operator, but needs a return statement

Exercises
Programming in Lisp(Scheme)

- Programming steps
  - What are the input parameters? What values could each parameter take?
  - Enumerate each combination of input parameters, give a return value for each case
- Exercise problems
  - Define a function Find which takes two parameters, x and y. It returns x if x appears in y, and returns an empty list (‘()) otherwise.
  - Define a function substitute which takes three parameters, x, y, and z. It returns a new list which replaces all occurrences x in y with z.

Solutions
Programming in Lisp(Scheme)

- Define a function Find which takes two parameters, x and y. It returns x if x appears in y, and returns an empty list otherwise.

```
(define Find (lambda (x y)
  (cond ((cons? y)
           (if (eq? (Find x (car y)) x) x (Find x (cdr y))))
        ((eq? x y) x)
         (else `())))))
```

- Define a function substitute which takes three parameters, x, y, and z. It returns a new list which replaces all occurrences of x in y with z.

```
(define substitute (lambda (x y z)
  (cond ((cons? y) (cons (substitute x (car y) z)
                         (substitute x (cdr y) z)))
        ((eq? x y) z)
        (else y))))
```

Functional Programming

- Functions are first-class objects
  - Functions treated as primitive values (What about C/C++)?
  - Can build anonymous and higher-order functions
- Higher order functions are functions that either
  - Take other functions as arguments or return a function as result
  - First-order function: parameters/result are not functions
  - Second-order function: take first-order functions as parameters or return them as result
  - Third-order functions: take as parameters or return second-order functions
- Example: function composition

```
 (lambda (f  g x)  (f  (g   x)))
vs.
 (lambda (f  g)   (lambda (x)  (f  (g   x)))))
```

Pass Functions as Parameters

- Apply a function to each element in a list

```
(define  maplist  (f  x)
   (cond  ((null? x)   nil)
           (else  (cons (f (car x))
                        (maplist f (cdr x))))))
```

vs.

```
Cell* maplist(int (*f)(...),  Cell* x)
{   if (x == NULL) return NULL;
    else {
        Cell* res = (Cell*) malloc (sizeof(Cell));
        res->val=f(x->val);
        res->next=maplist(f,x->next);
        return res;
    }
}
```

- Goal: apply different functions to complex data
  - Enforce a uniform interface for all the functions

Return functions as results

- Function composition

```
(define compose
    (lambda (f  g)   (lambda (x)  (f  (g   x))))))
vs. int compose(int (*f)(...), int (*g)(...), int x)
        { return f(g(x)); }
```

- In Scheme
  - The function compose takes only two parameters
  - The result of compose is another function
- in C
  - The function compose takes three parameters
  - The result of compose is a concrete value
  - Does not allow functions being returned as results, why?
- Goal: allow calling context (parameter values, global variables) be saved and used in the future

Programming With Higher-order Functions

- Apply a function to each element in a list

```
(define  maplist  (lambda (f  x)
   (cond  ((null? x)   nil)
          (else  (cons (f (car x))  (maplist f (cdr x)))))))
```

- Increment each number in a list by 1

```
(define increment1 (lambda (x)
        (maplist (lambda (e) (if (number? e) (+ e 1) e)) x)))
```

- Reduce a list into a single value

```
(define reduce (lambda (f0 f1 f2 x)
   (cond ((null? x)  f0)
         (else (f2 (f1 (car x)) (reduce f0 f1 f2 (cdr x)))))))
```

- Compute the sum of all numbers in a list

```
(define sum (lambda (x)
        (reduce 0 (lambda (e) (if (number? e) e 0))
                  (lambda (res1 res2) (+ res1 res2)) x)))
```

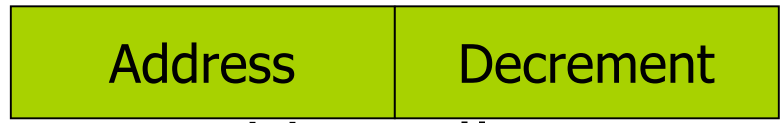
- Exercise:
  - A mapTree function that treat lists as trees
  - A mapTreePostOrder function that traverses a tree in post order

The Lisp Abstract machine

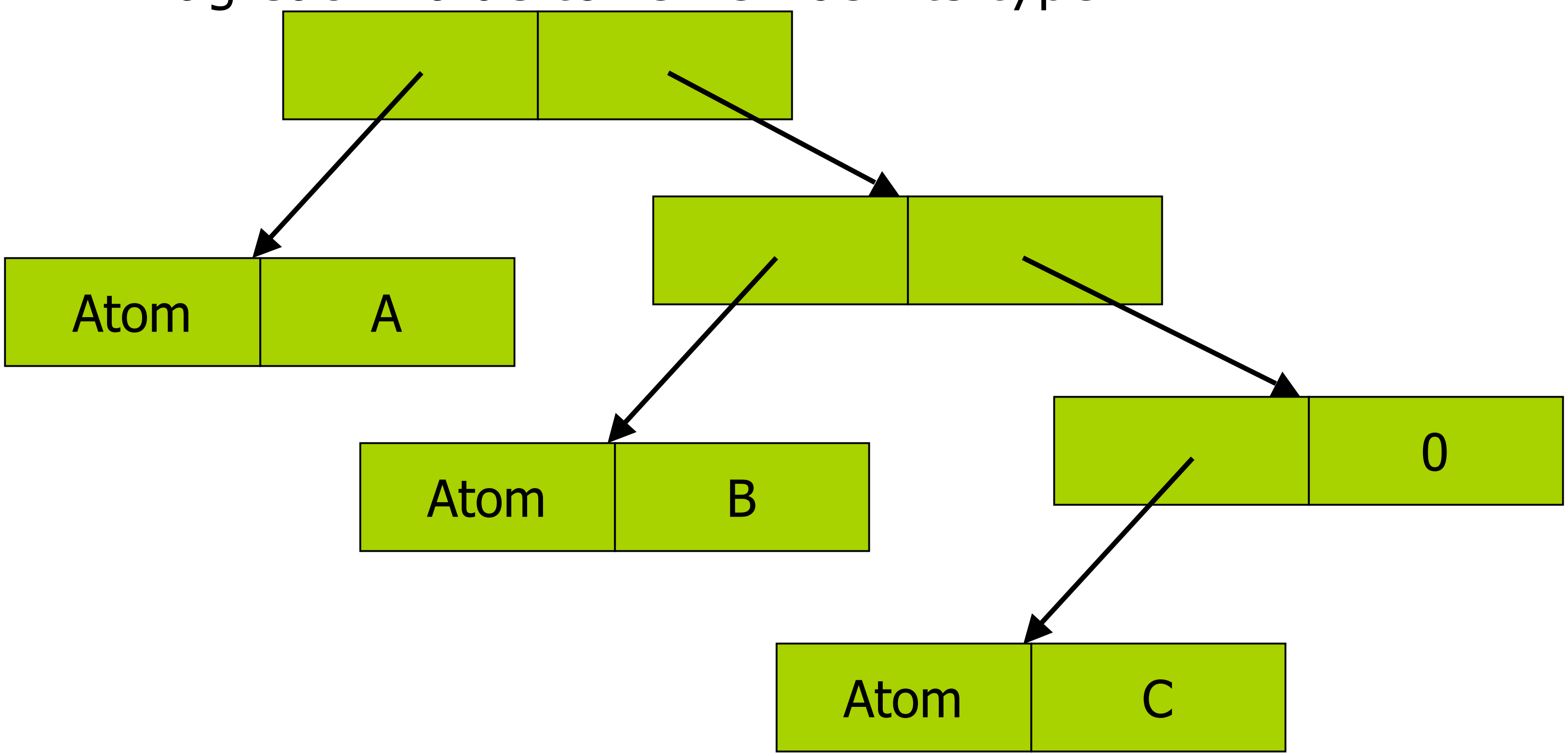
- Abstract machine
  - The runtime system (software simulated machine) based on which a language is interpreted
  - In short, the internal model of the interpreter that implements the language
- Lisp Abstract machine
  - A Lisp expression: the current expression to evaluate
  - A continuation: the rest of the computation
  - A-list : variable->value mapping
  - A set of cons cells (dynamic memory)
    - pointed to by pointers in A-list
    - Each cons cell is a pair
      - (car cdr) => linked data structures (lists)
      - (atm a) => a single atom
- Garbage collection
  - Automatic collection of non-accessible cons cells

Implementing Lisp --- The Memory Model

- Cons cells

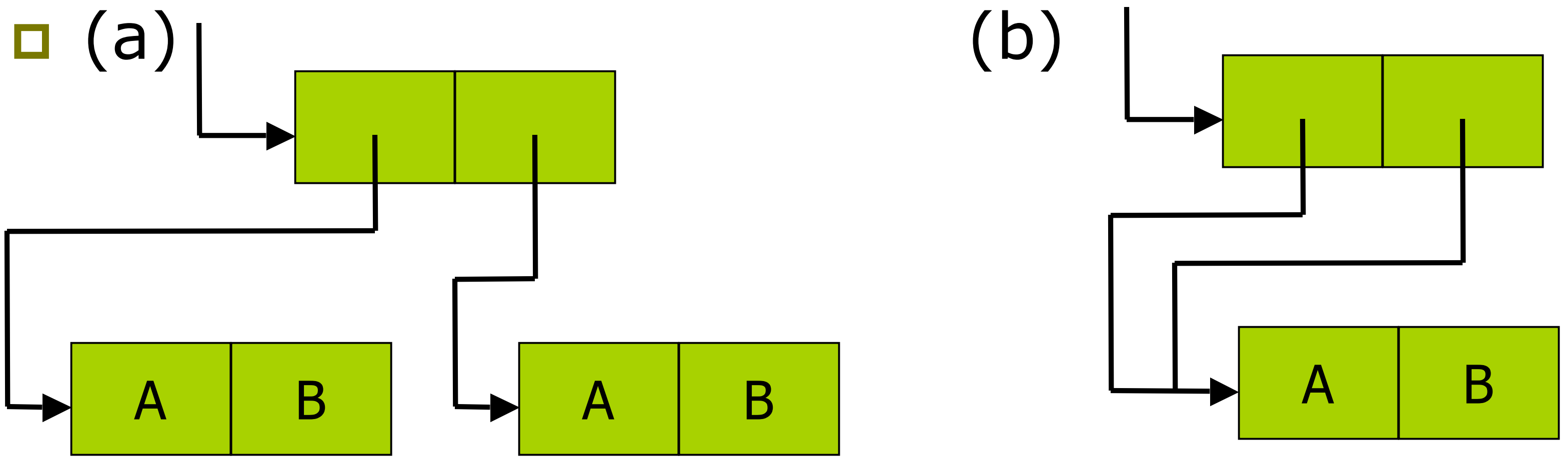


- Atoms and lists represented by cells
  - Tag each value to remember its type



Sharing

- (a) (b)



- Both structures could be printed as (A.B).(A.B)
- Which are the results of evaluating
  - (cons (cons ‘A ‘B) (cons ‘A ‘B)) ?
  - ((lambda (x) (cons x x)) (cons ‘A ‘B))
- Equality of compound structures
  - What is the result of (eq? ‘a ‘a) ?
  - What is the result of (eq? ‘(a b) ‘(a b)) ?

Garbage Collection

- Memory management at runtime
  - Maintains a list of available memory cells
  - Receive and satisfies allocation requests
  - When available space is below threshold
    - Invoke garbage collector
- Garbage collection
  - Detecting memory cells no longer used
    - Reclaim memory cells
  - Garbage: memory locations that are no longer accessible
    - Example (car (cons (e_1) (e_2)))
    - Cells created in evaluation of e_2 may be garbage, unless shared by e_1 or other parts of program
- Need to keep track of how many active pointers are pointing to each store

Meta-programming
Programs As Data

- Meta programming languages
  - Computer programs can write or manipulate other programs (or themselves) as their data
  - If can modify themselves --- reflective programming
- Lisp program can be represented using Lisp atoms and lists
  - Can be built/modified at runtime and then evaluated
- An eval function used to evaluate contents of list
  - in Scheme, need to choose a more advanced language level

```
(define atom? (lambda (x) (or (symbol? x) (number? x) (boolean? x))))
(define substitute (lambda (x y z)
    (cond ((null? z) z)
          ((atom? z) (if (eq? z x) y  z))
          (else  (cons (substitute x y (car z)) (substitute x y (cdr z)))))))
(define substitute-and-eval (lambda (x y z) (eval (substitute x y z))))
(substitute-and-eval  ’x  ’3  '(+ x 1))
```