Lisp

Functions, recursion and lists

The Lisp Programming Language

- Stems from interest in symbolic computation
  - math, logic, artificial intelligence
- Functional programming paradigm
  - A program/function is a expression
    - Expresses flow of data in evaluations ==> map input values to output values
    - No concept of control-flow or statements (the ordering of expressions does not matter)
  - Functions are first-class objects
    - A function can be used everywhere a regular value (e.g., an integer) 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 systems incrementally as the results of experiments
  - X Not many tools or libraries; Slow in speed/efficiency (interpreted)

Concepts in Lisp

- Supported value types
  - Atomic values: numbers, symbols, booleans
  - Compound data structures: lists (car, cons, cdr), functions (lambda)
- Supported operations
  - Function definition (abstraction)
    - (lambda (parameter-list) body)
  - Predefined functions: cons, cond, if, car, cdr, eq?
- Language Syntax

  exp ::= value | variable | (expList)

  expList ::= exp expList | exp

  or (shorthand)

  exp ::= value | variable | (exp exp …)
- Innovations in the design of Lisp
  - Functional programming paradigm
    - A program is an expression based on function abstractions and applications
    - First class functions and higher-order functions
  - Abstract view of memory (the Lisp abstract machine)
  - Program as data (dynamic interpretation of program)

Expressions, Statements and Declarations

- 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
- Declaration integer x
  - Introduces new identifiers (variables)
    - May bind value to identifier, specify type, etc.
  - Scoping rules for variables: which variable does each name refers to?

Interacting with Scheme

```
(define pi 3.14159)       ; bind pi to 3.14159
(lambda (x) (* x x))      ; anonymous function
(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)  ; ((lambda (x1 x2) E3) E1 E2)
                    ; ((lambda (x1) (lambda (x2) E3)) E1) E2)
(let* ((x1 E2) (x2 E2)) E3)
                    ; (lambda (x1) ((lambda (x2) E3) E2) E1)
```

- Programming environment
  - **DrScheme**
    - available on Windows/Linux machines
    - Documents available from class web site

Function Definitions and Variables

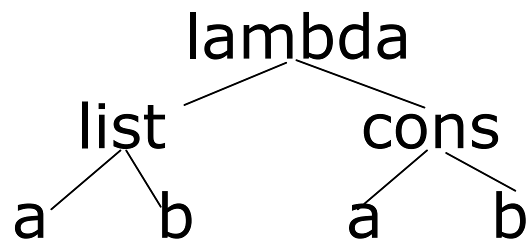
- Recursive function definition

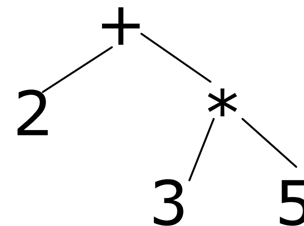
```
(define find (lambda (x y)
        (cond ((eq? y nil) nil)
              (eq? x (car y)) x)
              (else (find x (cdr y)))
```

  - Recursively traverse the tail of y --- what about (car y)?
    - Is y a flat list or a nested list (a tree)?
- Static scoping of variables
  - Runtime context can differ from the definition context
    - **(define y 100)**
    - **(define addx (lambda (x) (+ x y)))))**
    - What is the results of **((lambda (y) (addx y)) 3)**
    - **(+ x 3)** or **(+ x 100)**
  - Static scoping: determine binding of variables by statically looking at surrounding scopes (without running the program)

Lists in Scheme

- In Scheme(Lisp), a list may contain arbitrary kinds 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
    - lists, trees, graphs, etc.
- Scheme lists can be used to naturally implement ASTs (abstract syntax trees)





- What about graphs (can we build cycles in lists)?

Lisp: Adding Side Effects

- Pure Lisp
  - Expressions do not modify observable machine states
    - No side effects
- Impure Lisp
  - Allow modifications to memory
    - May increase efficiency of programs
      - Eg. Modify a single 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 (the address field of '(A B)) with y
    - (rplaced '(A B) y) or (set-cdr! '(A B) y)
      - replace B (the decrement field of '(A 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

Evaluation Using Expressions vs. Statements

- No side-effect expressions (no variable modifications)
  - In Scheme

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

  - What is the result?
- 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: explicit in data flow
    - Can evaluate each expression as soon as values are ready
  - Among statements: implicit in side effects (modifications, error)
    - Control flow among statements cannot be changed unless expressions are independent and no error cases could occur

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.
  - Define a function Max which takes a single parameter x, and returns the maximal number that appears in x.

Meta-programming Programs As Data

- Lisp program: exp ::= value | variable | (exp exp exp…)
  - Can be represented using Lisp atoms and lists
  - Can be built 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))
```

Higher-Order Functions

- Function that either
  - takes a function as an argument or returns a function as a result
  - First-order functions: parameters and results are not functions
  - Second-order functions: take as input or return first-order functions
  - Third-order functions: take as input or return second-order functions
- Examples:
  - function composition

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

  - maplist

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

- Functions as first-class objects
  - Functions treated same as primitive values (What about C/C++)?
  - Can be used to build anonymous and higher-order functions

Functions as Parameters

- function composition

```
(lambda (f  g x)  (f  (g   x)))
```

vs.

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

- 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; }
 }
```

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 if a concrete value
  - C does not allow functions being returned as result, why?

The Lisp Abstract machine

- Abstract machine
  - The runtime system (low-level hardware machine or high-level software simulated machine) based on which a language is interpreted
- 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 (the list data structure)
    - 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

Atom | A

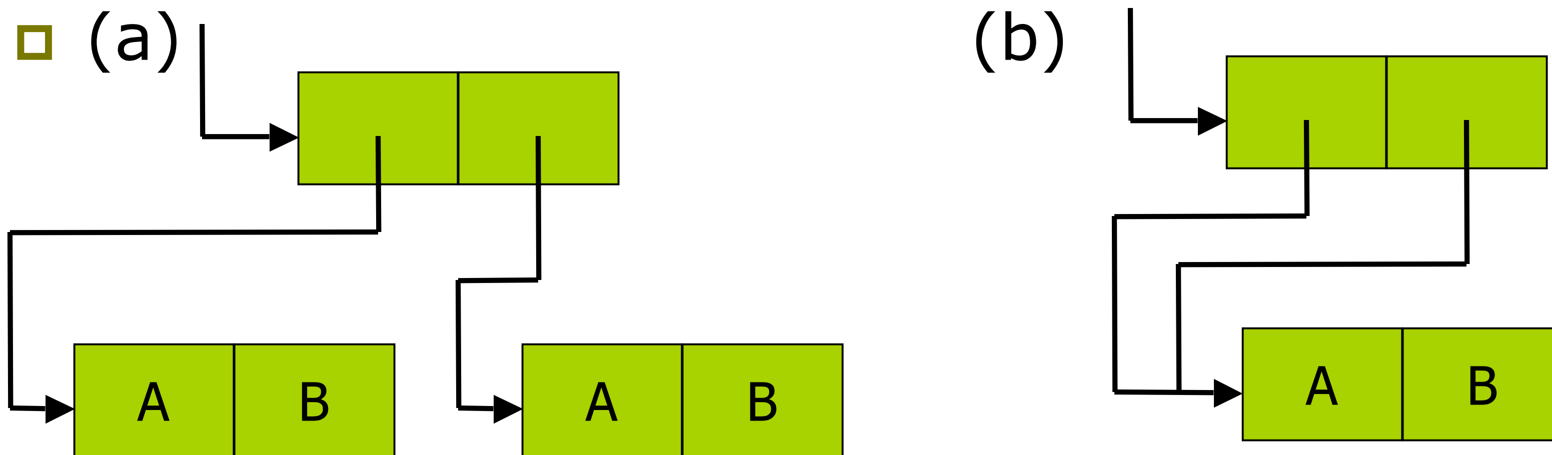
Atom | B

0

Atom | C

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

Detecting Garbage

- Garbage
  - Memory locations in the heap that are no longer accessible
- Examples

  (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

  ((lambda (x) (car (cons (… x…) (... x ...))) '(Big Mess))
  - The car and cdr of this cons cell may point to overlapping structures.