CS560
Programming Languages

Dr. Amie Souter

Slides from John Mitchell; supplements to *Concepts in Programming Languages*

Announcements

- Homework 1 posted due in two weeks
- Term project posted part 1 due in three weeks

Class Tonight

- Brief look at Computability and how related to PL
- LISP
  - functions, recursion, and lists
  - historically important language
  - illustrating a number of general points in PL

Foundations: Partial,Total Functions

- Value of an expression may be undefined
  - Undefined operation, e.g., division by zero
    - 3/0 has no value
    - implementation may halt with error condition
  - Nontermination
    - f(x) = if x=0 then 1 else f(x-2)
    - this is a *partial* function: not defined on all arguments
    - cannot be detected at compile-time; this is halting problem
  - These two cases are
    - "Mathematically" equivalent
    - Operationally different

How is this related to PL?

- Programs define partial functions for two reasons
  - partial operations (like division)
  - nontermination
    - f(x) = if x=0 then 1 else f(x-2)

Computability

- Definition

  Function f is computable if some program P computes it:

  For any input x, the computation P(x) halts with output f(x)

- Terminology

  Partial recursive functions

  = partial functions (int to int) that are computable

Halting function

- Decide whether program halts on input
  - *Given program P and input x to P,*

$$Halt(P,x) = \begin{cases} \text{yes} & \text{if } P(x) \text{ halts} \\ \text{no} & \text{otherwise} \end{cases}$$

Clarifications

Assume program P requires one string input x
Write P(x) for output of P when run in input x
Program P is string input to *Halt*

Fact: There is no program for *Halt*

Main points about computability

- Some functions are computable, some are not
  - Halting problem
- Programming language implementation
  - *Can* report error if program result is undefined due to division by zero, other undefined basic operation
  - *Cannot* report error if program will not terminate

Lisp, 1960

- Look at Historical Lisp
  - Perspective
    - Some old ideas seem old
    - Some old ideas seem new
  - Example of elegant, minimalist language
  - Not C: a chance to think differently
  - Many general themes in language design
- Supplementary reading
  - McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Communications of the ACM,* Vol 3, No 4, 1960.

John McCarthy

- Pioneer in AI
  - Formalize common-sense reasoning
- Also
  - Proposed timesharing
  - Mathematical theory
  - ….
- Lisp

  stems from interest in symbolic computation (math, logic)

Lisp summary

- Many different dialects
  - Lisp 1.5, Maclisp, …, Scheme, ...
  - CommonLisp has many additional features
  - This course: a fragment of Lisp 1.5, approximately

    But ignore static/dynamic scope until later in course
- Simple syntax

  (+ 1 2 3)

  (+ (* 2 3) (* 4 5))

  (f x y)

  Easy to parse. (Looking ahead: programs as data)

Atoms and Pairs

- Atoms include numbers, indivisible "strings"

  <atom> ::= <smbl> | <number>

  <smbl> ::= <char> | <smbl><char> | <smbl><digit>

  <num> ::= <digit> | <num><digit>
- Dotted pairs
  - Write (A . B) for pair
  - Symbolic expressions, called *S-expressions*:

    <sexp> ::= <atom> | (<sexp> . <sexp>)

Basic Functions

- Functions on atoms and pairs:
  cons car cdr eq atom
- Declarations and control:
  cond lambda define eval quote
- Example
  (lambda (x) (cond ((atom x) x) (T (cons 'A x))))
  function f(x) = if atom(x) then x else cons("A",x)
- Functions with side-effects
  rplaca rplacd set setq

Evaluation of Expressions

- Read-eval-print loop
- Function call (function arg_1 ... arg_n)
  - evaluate each of the arguments
  - pass list of argument values to function
- Special forms do not eval all arguments
  - Example (cond (p_1 e_1) ... (p_n e_n))
    - proceed from left to right
    - find the first p_i with value true, eval this e_i
  - Example (quote A) does not evaluate A

Examples

(+ 4 5)
  expression with value 9
(+ (+ 1 2) (+ 4 5))
  evaluate 1+2, then 4+5, then 3+9 to get value
(cons (quote A) (quote B))
  pair of atoms A and B
(quote (+ 1 2))
  evaluates to list (+ 1 2)
'(+ 1 2)
  same as (quote (+ 1 2))

Another Example

```
(define what (lambda (x y)
  (cond (( equal y nil) nil)
     ((equal x (car y)) x)
     (true (what x (cdr x (cdr y)))
)))

(what 'jack '(jill hill over jack fell
  down))
```

McCarthy's 1960 Paper

- Interesting paper with
  - Good language ideas, succinct presentation
  - Some feel for historical context
  - Insight into language design process
- Important concepts
  - Interest in symbolic computation influenced design
  - Use of simple machine model
  - Attention to theoretical considerations
    Recursive function theory, Lambda calculus
  - Various good ideas: Programs as data, garbage collection
  - From McCarthy Himself

Motivation for Lisp

- Advice Taker
  - Process sentences as input, perform logical reasoning
- Symbolic integration, differentiation
  - expression for function --> expression for integral
    (integral '(lambda (x) (times 3 (square x))))
- Motivating application part of good lang design
  - Keep focus on most important goals
  - Eliminate appealing but inessential ideas

Lisp	symbolic computation, logic, experimental prog.
C	Unix operating system
Simula	simulation
PL/1	"kitchen sink", not successful in long run

Execution Model (Abstract Machine)

- Language semantics must be defined
  - Too concrete
    - Programs not portable, tied to specific architecture
    - Prohibit optimization (e.g., C eval order *undefined* in expn)
  - Too abstract
    - Cannot easily estimate running time, space
- Lisp: IBM 704, but only certain ideas …
  - Address, decrement registers -> cells with two parts
  - Garbage collection provides abstract view of memory

Abstract Machine

- Concept of abstract machine:
  - Idealized computer, executes programs directly
  - Capture programmer's mental image of execution
  - Not too concrete, not too abstract
- Examples
  - Fortran
    - Flat register machine; memory arranged as linear array
    - No stacks, no recursion.
  - Algol family
    - Stack machine, contour model of scope, heap storage
  - Smalltalk
    - Objects, communicating by messages.

Theoretical Considerations

- " … scheme for representing the partial recursive functions of a certain class of symbolic expressions."
- Lisp uses
  - Concept of computable (partial recursive) functions
    - Want to express *all* computable functions
  - Function expressions
    - known from lambda calculus (developed A. Church)
    - lambda calculus equivalent to Turing Machines, but provide useful syntax and computation rules

Innovations in the Design of Lisp

- Expression-oriented
  - function expressions
  - conditional expressions
  - recursive functions
- Abstract view of memory
  - Cells instead of array of numbered locations
  - Garbage collection
- Programs as data
- Higher-order functions

Parts of Speech

- Statement — load 4094 r1
  - Imperative command
  - Alters the contents of previously-accessible memory
- Expression — (x+5)/2
  - Syntactic entity that is evaluated
  - Has a value, need not change accessible memory
  - If it does, has a *side effect*
- Declaration — integer x
  - Introduces new identifier
  - May bind value to identifier, specify type, etc.

Function Expressions

- Example:

  (lambda (*parameters*) (*function_body*))
- Syntax comes from lambda calculus:

  $\lambda f. \lambda x. f\ (f\ x)$

  (lambda (f) (lambda (x) (f (f x))))

Function expression defines a function but does not give a name to it.

```
( (lambda (f) (lambda (x) (f  (f  x))))
   (lambda (y) (+ 2 y)))
)
```

Conditional Expressions in Lisp

- Generalized if-then-else
  - (cond $(p_1\ e_1)$ $(p_2\ e_2)$ … $(p_n\ e_n)$)
    - Evaluate conditions p_1 … p_n left to right
    - If p_i is first condition true, then evaluate e_i
    - Value of e_i is value of expression

    Undefined if no p_i true, or

    p_1 … p_i false and p_{i+1} undefined, or

    relevant p_i true and e_i undefined

  Conditional statements in assembler

  Conditional expressions apparently new in Lisp

Examples

(cond ((<2 1) 2) ((<1 2) 1))

(cond ((<2 1) 2) ((<3 2) 3))

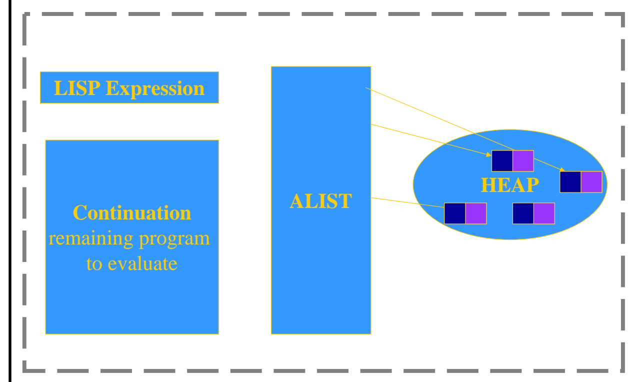
(cond (diverge 1) (true 0))

(cond (true 0) (diverge 1))

Strictness

- An operator or expression form is *strict* if it can have a value only if all operands or subexpressions have a value.
- Lisp cond is not strict, but addition is strict
  - (cond (true 1) (diverge 0))
  - (+ e_1 e_2)

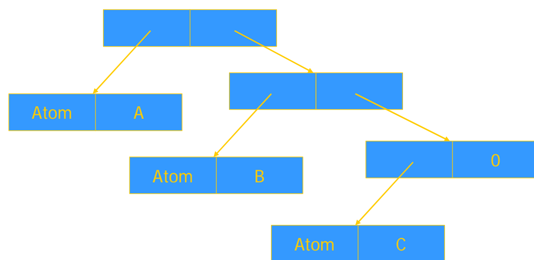
LISP Abstract Machine



Lisp Memory Model

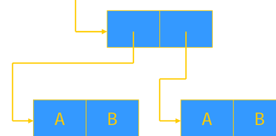
- Cons cells

- Atoms and lists represented by cells

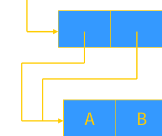


Sharing

(a)



(b)



- Both structures could be printed as
- Which is result of evaluating

(cons (cons 'A 'B) (cons 'A 'B)) ?

Garbage Collection

- Garbage:

At a given point in the execution of a program *P*, a memory location *m* is *garbage* if no continued execution of *P* from this point can access location *m*.

- Garbage Collection:
  - Detect garbage during program execution
  - GC invoked when more memory is needed
  - Decision made by run-time system, not program

This is can be very convenient. Example: in building text-formatting program, ~40% of programmer time on memory management.

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.

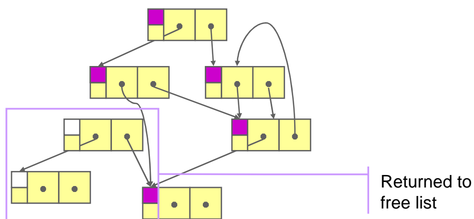
Mark-Sweep

- The first tracing garbage collection algorithm
- Garbage cells are allowed to build up until heap space is exhausted (i.e. a user program requests a memory allocation, but there is insufficient free space on the heap to satisfy the request.)
- At this point, the mark-sweep algorithm is invoked, and garbage cells are returned to the free list.

Mark-Sweep

- Performed in two phases:
  - **Mark phase:** identifies all live cells by setting a mark bit. Live cells are cells reachable from a root.
  - **Sweep phase:** returns garbage cells to the free list.

Mark-Sweep Example



Mark-Sweep: Advantages and Disadvantages

- Advantages:
  - Cyclic data structures can be recovered.
  - Tends to be faster than reference counting.

Mark-Sweep: Advantages and Disadvantages

- Disadvantages:
  - Computation must be halted while garbage collection is being performed
  - Garbage collection becomes more frequent as heap occupancy of a program increases.
  - May fragment memory.
  - Has negative implications for locality of reference. Old objects get surrounded by new ones.

1/13/2005

Mark-and-Sweep Algorithm

- Assume tag bits associated with data
- Need list of heap locations named by program
- Algorithm:
  - Set all tag bits to 0.
  - Start from each location used directly in the program. Follow all links, changing tag bit to 1
  - Place all cells with tag = 0 on free list

Why Garbage Collection in Lisp?

- McCarthy's paper says that this is "more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists."
- Does this reasoning apply equally well to C?
- Is garbage collection "more appropriate" for Lisp than C? Why?

Programs As Data

- Programs and data have same representation
  - McCarthy again
- Eval function used to evaluate contents of list
- Example: substitute x for y in z and evaluate

```
(define substitute (lambda (x y z)
   (cond  ((atom z) (cond ((eq z y) x ) (T z)))
             (T  (cons (substitute x y (car z))
                          (substitute x y (cdr z))))))
(define substitute-and-eval
    (lambda (x y z) (eval (substitute x y z))))
```

Recursive Functions

- Want expression for function f such that
  (f x) = (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))
- Try
  (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1))))))
  but f in function body is not defined.
- McCarthy's 1960 solution was operator "label"
  (label f
  (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1)))))))

Higher-Order Functions

- Function that either
  - takes a function as an argument
  - returns a function as a result
- Example: function composition

```
(define compose
    (lambda (f g) (lambda (x) (f (g x)))))
```

- Example: maplist

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

Summary: Contributions of Lisp

- Successful language
  - symbolic computation, experimental programming
- Specific language ideas
  - Expression-oriented: functions and recursion
  - Lists as basic data structures
  - Programs as data, with universal function `eval`
  - Stack implementation of recursion via "public pushdown list"
  - Idea of garbage collection.
- Closing Remarks

Any Questions