Lisp
John Mitchell

Reading: Chapter 3          Homework 1: due Oct 3

Lisp, 1960

• Look at Historical Lisp
  – Perspective
    • Some old ideas seem old
    • Some old ideas seem new
  – Example of elegant, minimalist language
  – Not C, C++, Java: a chance to think differently
  – Illustrate general themes in language design
• Supplementary reading (optional)
  – McCarthy, Recursive functions of symbolic expressions and their computation by machine, CACM, Vol 3, No 4, 1960. [see CS242 web site]

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

Note on syntax
• Book uses some kind of pidgin Lisp
• Handout provides executable alternatives, so examples run in Scheme
• In Scheme, a pair writes as (A . B), but (A . B) is not an expression
Evaluation of Expressions

- Read-eval-print loop
- Function call \((\text{function } \text{arg}_1 \ldots \text{arg}_n)\)
  - evaluate each of the arguments
  - pass list of argument values to function
- Special forms do not eval all arguments
  - Example \((\text{cond } e_1 \ldots e_n)\)
    - proceed from left to right
    - find the first \(p_i\) with value true, eval this \(e_i\)
  - Example \((\text{quote } A)\) 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
\[ (\text{cons} (\text{quote } A) (\text{quote } B)) \]
pair of atoms \(A\) and \(B\)
\[ (\text{quote} (+ 1 2)) \]
evaluates to list \((+ 1 2)\)
\[ '(+ 1 2) \]
same as \((\text{quote} (+ 1 2))\)

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

Motivation for Lisp

- Advice Taker
  - Process sentences as input, perform logical reasoning
- Symbolic integration, differentiation
  - expression for function \(\rightarrow\) expression for integral
  - \((\text{integral} \left\{ \text{lambda} (x) \left( \text{times} 3 \left( \text{square} x \right) \right) \right\})\)
- Motivating application part of good language 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 expr)
  - Too abstract
    - Cannot easily estimate running time, space
- Lisp: IBM 704, but only certain ideas ...
  - Address, decrement registers \(\rightarrow\) 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

- McCarthy’s description
  - “... scheme for representing the partial recursive functions of a certain class of symbolic expressions”
- Lisp uses
  - Concept of computable (partial recursive) 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
  - Imperative command
  - Alters the contents of previously-accessible memory
- Expression
  - Syntactic entity that is evaluated
  - Has a value, need not change accessible memory
  - Defines a function but does not give it a name
- Declaration
  - Introduces new identifier
  - May bind value to identifier, specify type, etc.

Function Expressions

- Form
  - (lambda (parameters) (function_body))
- Syntax comes from lambda calculus:
  - λx. f(x)
  - λf. λx. f(f(x))
- (define inc (lambda (x) (+ 1 x)))
- (define twice (lambda (f) (func (f x)))))

Example

```
(define twice
  (lambda (f) (lambda (x) (f (f x))))))
)

(define inc (lambda (x) (+ 1 x)))

(twice inc) 2
⇒ 4
```

Conditional Expressions in Lisp

- Generalized if-then-else
  - (cond (p1 e1) (p2 e2) ... (pn en))
    - Evaluate conditions p1 ... pn left to right
    - If pi is first condition true, then evaluate ei
    - Value of ei is value of expression
    - No value for the expression if no pi true, or pi false and pim has no value, or relevant pi true and ei has no value

Conditional statements in assembler
Conditional expressions apparently new in Lisp
Examples

- (cond ((< 2 1) 2) ((< 1 2) 1)) has value 1
- (cond ((< 2 1) 2) ((< 3 2) 3)) has no value
- (cond (diverge 1) (true 0)) no value, if expression diverge loops forever
- (cond (true 0) (diverge 1)) has value 0

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))
  - (+ e₁ e₂)

Details: (define loop (lambda (x) (loop x)))
  diverge ≈ (loop ())

Lisp Memory Model

- Cons cells
- Atoms and lists represented by cells

Sharing

- Both structures could be printed as ((A.B) (A.B))
- Which is result of evaluating
  (cons (cons 'A 'B) (cons 'A 'B))?
  Note: Scheme actually prints using combination of list and dotted pairs

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 ( e₁ ) ( e₂ )))
  Cells created in evaluation of $e₂$ may be garbage, unless shared by $e₁$ 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-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 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
- Eval function used to evaluate contents of list
- Example: substitute x for y in z and evaluate
  (define substitute (lambda (x y z)
    (cond ((null z) z)
      ((atom z) (cond ((eq z y) x) (true z)))
      (true (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 '3 'x '(+ x 1))

Recursive Functions

- Want expression for function f such that
  \( (f x) = \text{cond (eq x 0) 0} \true + x (f (- x 1))]\)
- Try
  \( (\text{lambda } (x) (\text{cond (eq x 0) 0} \true + x (f (- x 1)))))\)
  but in function body is not defined.
- McCarthy’s 1960 solution was operator “label”
  \( (\text{lambda } x (\text{cond (eq x 0) 0} \true + x (f (- x 1)))))\)

Recursive vs. non-recursive declarations

- Recursive definition
  - (define f
    (lambda (x) (cond ((eq x 0) 0) \true (+ x (f (- x 1))))))
  - Legal Scheme: treats inner f as function being defined
- Non-recursive definition
  - (define x (+ x 2))
  - Error if x not previously defined
    While evaluating arguments to + in expression (+ x 2):
    Unbound variable: x
    ABORT: (misc-error)

Higher-Order Functions

- Function that either
  - takes a function as an argument
  - returns a function as a result
- Example: function composition
  \( (\text{define compose (lambda } (f g) (\text{lambda } (x) (f (g x)))))\)
- Example: maplist
  \( (\text{define maplist (lambda } (f)\ (\text{cond ((null x) ())))
    \true (cons (f (car x)) (maplist f (cdr x)))\)))\)
**Example**

```scheme
(define inc (lambda (x) (+ x 1)))
(maplist inc '(1 2 3))
⇒ (2 3 4)
```

Scheme preamble:
```
(define true #t)
(define false #f)
(define eq eq?)
(define null null?)
```

**Efficiency and Side-Effects**

- Pure Lisp: no side effects
- Additional operations added for “efficiency”
  - `(rplaca x y)` replace car of cell x with y
  - `(rplacd x y)` replace cdr of cell x with y
- What does “efficiency” mean here?
  - Is `(rplaca x y)` faster than `(cons y (cdr x))`?
  - Is faster always better?

**Example**

```scheme
(define p '(A B))
(rplaca p 'C)
(rplacd p 'D)
p ⇒ (C . D)
```

**Language speeds**

- Scheme preamble
  ```scheme
  (define rplaca set-car!)
  (define rplacd set-cdr!)
  ```

**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.