Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I

by John McCarthy

Ik-Soon Kim
Winter School 2005
Feb 18, 2005

Overview

By John Mitchell

- Interesting paper with
  - Good language ideas, succinct presentation
  - 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:
    Program as data, garbage collection

Motivation for Lisp

- Advice Taker
  - process declarative and imperative sentences
  - make logical reasoning
- Lisp was designed to facilitate experiments with Advice Taker
- Motivating application part of good language design
  - Lisp symbolic computation, logic, experimental
  - C Unix O/S
  - Simula simulation
  - Java web applet

Mathematical concepts in Lisp

- Lisp implements the following mathematical concepts:
  - Partial functions
  - Propositional expressions and predicates
  - Conditional expressions
  - Lambda functions and recursive functions

Partial functions

- A function defined on a subset of its domain
- Common in real computation since
  - partial operations
    ex) division

  - nontermination
    ex) f(x) = if x=1 then 1 else x*f(x-1)

Propositional expressions and predicates

- Propositional expressions
  - have T or F as possible values
  - have logical connectives: □ ("and"), □ ("or") and ¬ ("not")
  - ex)

    x < y

    (x < y) □ (b = c)

- A predicate
  - is a function whose range consists of T or F
  - ex) prime(x)

Conditional expressions

$(p_1 \to e_1, \ldots, p_n \to e_n)$

- Generalized if-then-else
- If p_1 then e_1 otherwise if p_2 then e_2 ,..., otherwise if p_n then e_n .
  - ex)

$$(1 < 2 \to 4, 1 > 2 \to 3) = 4$$

$$(2 < 1 \to 4, T \to 3) = 3$$

$$(2 < 1 \to \frac{0}{0}, T \to 3) = 3$$

$$(2 < 1 \to 3, T \to \frac{0}{0}) \quad \text{undefined}$$

$$(2 < 1 \to 3, 4 < 1 \to 4) \quad \text{undefined}$$

Lambda functions and recursive functions

- Express anonymous functions
  - form $x^2 + y$
  - function f $f(x,y) = x^2 + y$
  - anonymous function $\lambda((x,y)\ x^2 + y)$
- Inadequate for naming functions defined recursively
  - label $(fact, \lambda((x,y)\ (n = 0 \to 1, T \to n \cdot fact(n-1))))$

Theoretical consideration

- Lisp is “based on 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

Recursive functions of symbolic expressions

- Presents the Lisp syntax and semantics

  - S-expressions
  - S-functions
  - Translation of S-functions into S-expressions
  - Universal function *eval* (meta-circular interpter for Lisp)

S-expressions

- Atoms are distinguishable symbols
- Atomic symbols are S-expressions
- If e_1 and e_2 are S-expressions, so is $(e_1 . e_2)$
- ex) A
  (A . B)
  ((A . B) C)

- Lists can be represented by S-expressions
  - (e) (e . NIL)
  - $(e_1\ e_2\ \dots\ e_m)$ $(e_1 . (e_2 . (\dots (e_m . NIL) \dots)))$
  - $(e_1\ e_2\ \dots\ e_m . x)$ $(e_1 . (e_2 . (\dots (e_m . x) \dots)))$

S-functions

- S-functions are written in M-expressions
  - $fname[arg_1;arg_2; \dots ;arg_n]$

- Elementary S-functions
  - atom[x] check whether x is an atomic symbol
  - eq[x;y] check whether x and y are the same symbol
  - car[x] $car[(e_1 . e_2)] = e_1$
  - cdr[x] $cdr[(e_1 . e_2)] = e_2$
  - cons[x;y] cons[x;y] = (x . y)

Recursive and Higher-order S-functions

- Recursive S-functions
  - append[x;y] = (null[x]→ y ,T→ cons[car[x];append[cdr[x];y]])
  - null[x] = atom[x] □ eq[x;NIL]

- Higher-order functions
  - takes a function as an argument or
  - returns a function as a result

compose[f;g] = [λ [x] f[g[x]]]

maplist[x;f]=(null[x]→ NIL,T→ cons[f[car[x]];maplist[cdr[x];f])

Translating S-functions into S-expressions

- Translating an M-expression M into M^*
  - If M is an S-expression, M is (QUOTE M)
  - Variable and function names are converted into upper case letters
  - $f[e_1; \dots ; e_n]$ is translated into $(f^* \ e_1^* \dots e_n^*)$
  - $[p_1 \rightarrow e_1, □ , p_n \rightarrow e_n]^*$ is (COND $(p_1^* \ e_1^*)$□ $(p_n^* \ e_n^*)$)
  - $\{\lambda[[x_1;□ ;x_n]; M]\}^*$ is (LAMBDA $(x_1^*$ □ $x_n^*)$ M^*)
  - $\{\text{label } [f;M]\}^*$ is (LABEL f^* M^*)

- Regard program as data

Translation example

label[append; λ[[x;y];(null[x]→ y ,T→ cons[car[x];append[cdr[x];y]])]]

Translation

```
(LABEL APPEND
  (LAMBDA (X Y)
    (COND
      ((NULL X) Y)
      (T          (CONS (CAR X) (APPEND (CDR X) Y))))))
```

Recursive functions of symbolic expressions

Program as data

- Program and data have same representation
- Symbolic computation such as integration and differentiation
  - Lisp handles program (or functions) as input or output
    ex) find integration or differentiation of input funciton
    (INTEGRAL (QUOTE (LAMBDA (X) (* 3 SQUARE X)))))
- Staged computations
  - Manipulate code at runtime
    Macro processing
    Runtime code generation

Universal function *eval*

(eval exp env)

- Compute *exp* under the *env* environment
  - exp   an S-expression translated from an S-function
  - env   a list of pairs of variable and its value
- A Lisp interpreter based on essential S-functions (meta-circular interpreter)
- An operational semantics for Lisp using Lisp
- We will skip the detailed code for *eval*

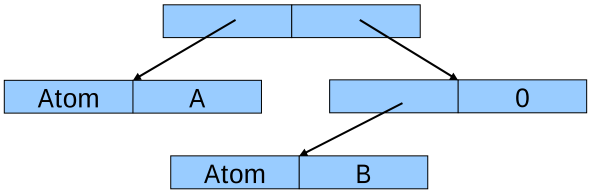
Lisp programming system

- Present implementation issues for Lisp
  - Representation of S-expressions
  - Free Storage List (Garbage Collection)
  - Public push-down list

Representation of S-expressions

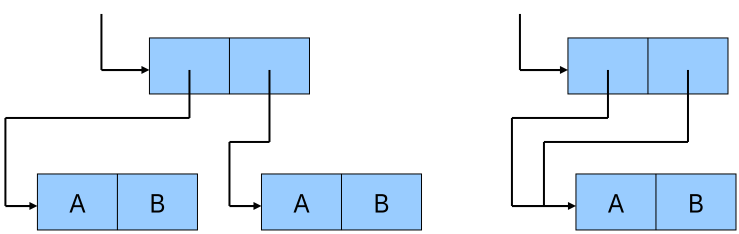
- Memory cells 

- Atoms and lists represented by cells



- Prohibit circular lists for printing problem (allowed later)

Shared lists



- Both structures could be printed as ((A . B) . (A . B))
- Whether lists are shared or not depends on the history of program execution
  - (cons (cons 1 2) (cons 1 2))
  - (cons a a) where a = (cons 1 2)

Free-storage list

- Lisp keeps the free-storage list of free cells automatically
- Assume tag bits associated with data
- Need list of heap locations referred 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-storage list
- "*Mark-and-sweep*" garbage collection algorithm

Public push-down list

- A recursive function uses itself as a subroutine
- When a recursive function begins, it saves registers into public push-down list
- When a recursive functions exits, it restores registers from public push-down list
- It is called a *stack* today

Innovation in the Design of Lisp

- Expression-oriented
  - function oriented
  - conditional expressions
  - recursive functions

- Abstract view of memory
  - Cells instead of array of indexed locations
  - Garbage colletion
- Public push-down list (stack) for recursive calls
- Programs as data
- Higher-order functions

Conclusions

- 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*
  - Garbage collection

References

- McCathy, *Recursive functions of symbolic expressions and their computation by machine,* CACM, Vol 3, No 4, 1960
- John Mitchell's CS242 lecture note