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

- 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) \( f(x) = \text{if } x=1 \text{ then } 1 \text{ else } x*f(x-1) \)

**Propositional expressions and predicates**

- Propositional expressions
  - have T or F as possible values
  - have logical connectives: ("and"), ("or") and ("not")
  - ex) \( \begin{align*} x < y \\
    (x < y) \land (b = c) \end{align*} \)

- A predicate
  - is a function whose range consists of T or F
  - ex) \( \text{prime}(x) \)
Conditional expressions

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

\[
\begin{align*}
(1 < 2 \rightarrow 4, & 2 \rightarrow 3) = 4 \\
(2 < 1 \rightarrow 4, T & \rightarrow 3) = 3 \\
(2 < 1 \rightarrow 0, O & \rightarrow 3) = 3 \\
(2 < 1 \rightarrow 3, T & \rightarrow 0) & \rightarrow \text{undefined} \\
(2 < 1 \rightarrow 3, 4 & \rightarrow 4) & \rightarrow \text{undefined}
\end{align*}
\]

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
  - \text{label} \( \text{fact} \((x,y) \ (n = 0 \rightarrow 1, T & \rightarrow n \text{fact}(n \rightarrow 1)))\)
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

 Mathematical concepts in Lisp

Recursive functions of symbolic expressions

- Presents the Lisp syntax and semantics
  - S-expressions
  - S-functions
  - Translation of S-functions into S-expressions
  - Universal function eva/ (meta-circular interpreter for Lisp)
**S-expressions**

- Atoms are distinguishable symbols
- Atomic symbols are S-expressions
- If $e_1$ and $e_2$ are S-expressions, so is $(e_1 \cdot e_2)$
- e.g. $A$
  
  $$(A \cdot B)$$

  $$((A \cdot B) \cdot C)$$

- Lists can be represented by S-expressions
  - $$(e)$$
  - $$(e_1 e_2 \ldots e_m \cdot x)$$
  - $$(e_1 e_2 \ldots e_m \cdot x)$$

**S-functions**

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

- Elementary S-functions
  - $atom[x]$ check whether $x$ is an atomic symbol
  - $eq[x;y]$ check whether $x$ and $y$ are the same symbol
  - $car[x]$ $car[(e_1 \cdot e_2)] = e_1$
  - $cdr[x]$ $cdr[(e_1 \cdot e_2)] = e_2$
  - $cons[x;y]$ $cons[x;y] = (x \cdot y)$
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
    
    \[
    \text{compose}[f; g] = [\lambda x f[g(x)]]
    \]
    
    \[
    \text{maplist}[x; f] = \{\text{null}[x] → \text{NIL}, \text{T} → \text{cons}[f[\text{car}[x]]; \text{maplist}[\text{cdr}[x]; f]]\}
    \]

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; \ldots; e_n]\) is translated into \(f^* e_1^* \ldots e_n^*\)
    - \(\{p_1 → e_1^* \ldots p_n^* e_n^*\}\) is \(\text{COND} (p_1^* e_1^* \ldots (p_n^* e_n^*))\)
    - \(\{\lambda (x_1^* ; x_n^* ; M)\}\) is \(\text{LAMBDA} (x_1^* \ldots x_n^* ) M^*\)
    - \(\{\text{label} \{f; M\}\}\) is \(\text{LABEL} f^* M^*\)

- Regard program as data
Translation example

Label:\texttt{append; \lambda \[(x; y); (\text{null}(x) \rightarrow y, T \rightarrow \text{cons}(\text{car}(x); \text{append}(\text{cdr}(x); y)))]}

\textbf{Recursive functions of symbolic expressions}

\begin{verbatim}
(LABEL APPEND
 (LAMBDA (X Y)
  (COND
   ((NULL X) Y)
   (T
    (CONS (CAR X) (APPEND (CDR X) Y))))))
\end{verbatim}

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 function
    \texttt{(INTEGRAL (QUOTE (LAMBDA (X) (* 3 SQUARE X)))))}

- Staged computations
  - Manipulate code at runtime
    - Macro processing
    - Runtime code generation
Universal function \texttt{eval}

\[(\text{eval } \text{exp } \text{env})\]

- Compute \textit{exp} under the \textit{env} environment
  - \textit{exp} an S-expression translated from an S-function
  - \textit{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 \texttt{eval}/

Recursive functions of symbolic expressions

Lisp programming system

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

The Lisp Programming System
### Representation of S-expressions

- Memory cells
  - **Address**
  - **Decrement**

- Atoms and lists represented by cells
  - ![Diagram of atom A and atom B with addresses](image)

- Prohibit circular lists for printing problem (allowed later)

---

### Shared lists

- ![Diagram of shared lists](image)

- 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)`

---

The Lisp Programming System
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 function 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 collection

- Public push-down list (stack) for recursive calls
- Programs as data
- Higher-order functions

The Lisp Programming System

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 \texttt{eval}
  - Garbage collection

The Lisp Programming System
References

- John Mitchell’s CS242 lecture note