Last updated: 2022-07-15 13:09:00-04:00

Contents

1 Building Abstractions with Procedures

Definition (computational process)
Abstract beings that inhabit computers.

Definition (data)
Computational processes manipulate other abstract things called ***data*** as they evolve.

Definition (program)
A pattern of rules by which the evolution of a computational process is directed.

Definition (programming language)
That in which programs are carefully composed from symbolic expressions that prescribe the tasks we want our computational processes to perform.

Definition (bug, glitch)
Small errors.

Definition (debug)
Remove bugs.

Programming in Lisp

See the appendix.

Appendix

A McCarthy 1960

Recursive Functions of Symbolic Expressions and Their Computation by Machine

A.1 Introduction

LISP:

- "**LIS**t **P**rocessor"
- Developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T.
- Designed to facilitate experiments with a proposed system called the *Advice Tracker*:
  - A programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Tracker system could make deductions.
  - Originally proposed in November 1958.
- Came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions, independent of any electronic computer.

In this article:

1. Describe a formalism for defining functions recursively.
2. Describe S-expressions and S-functions, give examples, and describe the universal S-function `apply` which plays the theoretical role of a universal Turing machine and the practical role of an interpreter.
3. Describe the representation of S-expressions in the memory of the IBM 704 by list structures ... and the representation of S-functions by program.
4. Mention the main features of the LISP programming system for the IBM 704.
5. Another way of describing computations with symbolic expressions.
6. Give a recursive function interpretation of flow charts.

A.2 Functions and Function Definitions

A.2.1 Partial Functions

Definition (partial function)
A function that is defined only on part of its domain.

A.2.2 Propositional Expressions and Predicates

Definition (propositional expression)
A ***propositional expression*** is an expression whose possible values are T (for truth) and F (for falsity).

Definition (predicate)
A function whose range consists of the truth values T and F.

A.2.3 Conditional Expressions

Definition (conditional expression)
A device for expressing the dependence of quantities on propositional quantities, denoted:

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

where the p's are propositional expressions and the e's are expression of any kind, read "If p_1 then e_1, otherwise if p_2 then e_2, ... otherwise if p_n then e_n" or "p_1 yields e_1, ..., p_n yields e_n."

How to determine the value of an arbitrary conditional statement $(p_1 \to e_1, \cdots, p_n \to e_n)$:

- Let $i = 1$.
- If the value of p_i is undefined, then the value of the conditional expression is undefined.
- If the value of p_i is T, then the value of the conditional expression is the value of the expression e_i.
- If the value of p_i is F, then increment i and evaluate p_i.

Example.

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

  Starting from the left with the statement $1 < 2 \to 4$, the propositional expression $1 < 2$ is true, therefore the value of the conditional expression is 4.

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

  Starting from the left with the statement $2 < 1 \to 4$, the propositional expression $2 < 1$ is false. Moving on to the next statement $2 > 1 \to 3$, the propositional expression $2 > 1$ is true, therefore the value of the conditional expression is 3.

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

  The propositional expression $2 < 1$ is false, so move on. The propositional expression T is true, therefore the value of the conditional expression is 3.

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

  The propositional expression $2 < 1$ is false, so move on. The propositional expression T is true, therefore the value of the conditional expression is 3.

- $(2 < 1 \to 3,\ T \to \frac{0}{0})$ is undefined.

  The propositional expression $2 < 1$ is false, so move on. The propositional expression T is true, therefore the value of the conditional expression is the value of the expression $\frac{0}{0}$ which is undefined.

- $(2 < 1 \to 3,\ 4 < 1 \to 4)$ is undefined.

  The propositional expression $2 < 1$ is false, so move on. The propositional expression $4 < 1$ is false, so move on. There's nothing else to move on to, so the value of the conditional expression is undefined.

Example. Applications:

- $|x| = (x < 0 \to -x,\ T \to x)$
- $\delta_{ij} = (i = j \to 1, T \to 0)$
- $sgn(x) = (x < 0 \to -1,\ x = 0 \to 0,\ T \to 1)$

A.2.4 Recursive Function Definitions

Example. Recursive applications:

- $n! = (n = 0 \to 1,\ T \to n \cdot (n-1)!)$
- $\gcd(m, n) = (m > n \to \gcd(n, m),\ rem(n, m) = 0 \to m,\ T \to \gcd(rem(n, m), m))$
- $\text{sqrt}(a, x, \epsilon) = (|x^2 - a| < \epsilon \to x,\ T \to \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))$

Example. Propositional connectives defined by conditional expressions:

- $p \wedge q = (p \to q,\ T \to F)$
- $p \vee q = (p \to T,\ T \to q)$
- $\neg p = (p \to F,\ T \to T)$
- $p \supset q = (p \to q,\ T \to T)$

A.2.5 Functions and Forms

Definition (form)
The expression $y^2 + x$ is an example of a form.

Definition (function)
A function substitutes arguments into a form to evaluate a result. It's necessarily to know which arguments map to which variables of the form. Something like $y^2 + x(3, 4)$ is not a function because it's not clear how $(3, 4)$ should map to x and y. If we make explicit the positions of the variables in the ordered tuple of arguments, then the mapping is clear. So, $\lambda((x, y),\ y^2 + x)$ is a function, where e.g. $\lambda((x, y),\ y^2 + x)(3, 4)$ evaluates to $4^2 + 3 = 19$.

A function of n arguments is denoted

$$\lambda((x_1, \ldots, x_n),\ \mathcal{E})$$

where $\mathcal{E}$ is a form of n variables $x_1, \ldots, x_n$.

Definition (list of variables)
Given a function $\lambda((x_1, \ldots, x_n),\ \mathcal{E})$, the ordered tuple $(x_1, \ldots, x_n)$ is called the ***list of variables***.

Definition (dummy variable, bound variable)
Given a function $\lambda((x_1, \ldots, x_n),\ \mathcal{E})$, the variables in the list of variables are called ***dummy variables*** or ***bound variables*** .

The symbol used for any bound variable may be changed without changing the value of the expression, provided that we make the same change for each occurrence of the variable and do not make two variables the same that previously were different.

Example. The functions $\lambda((x, y),\ y^2 + x)$, $\lambda((u, v),\ v^2 + u)$, and $\lambda((y, x),\ x^2 + y)$ are equivalent.

Definition (free variable)
Given a function $\lambda((x_1, \ldots, x_n),\ \mathcal{E})$, a ***free variable*** is a variable which occurs in the expression $\mathcal{E}$ but does not occur in the list of variables. Such an expression may be regarded as defining a function with parameters.

Definition (collision of bound variables)
A difficulty which arises in combining functions described by λ-expressions, or by any other notation involving variables, because different bound variables may be represented by the same symbol.

A.2.6 Expressions for Recursive Functions

Note. The λ-notation is inadequate for naming functions defined recursively. For example,

$$\text{sqrt}(a, x, \epsilon) = (\left|x^2 - a\right| < \epsilon \to x,\ T \to \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))$$

converts into λ-notation as

$$\text{sqrt} = \lambda((a, x, \epsilon),\ (\left|x^2 - a\right| < \epsilon \to x,\ T \to \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon)))$$

However, the right-hand side cannot serve as an expression for the function because there would be nothing to indicate that the reference to `sqrt` within the expression stood for the expression as a whole.

Definition (label($a, \mathcal{E}$))
label$(a, \mathcal{E})$ denotes the expression $\mathcal{E}$ provided that occurrences of a within $\mathcal{E}$ are to be interpreted as referring to the expression $\mathcal{E}$ as a whole.

Example. The `sqrt` function can be written as

$$\text{label}(\text{sqrt},\ \lambda((a, x, \epsilon),\ (\left|x^2 - a\right| < \epsilon \to x,\ T \to \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))))$$

A.3 Recursive Functions of Symbolic Expressions

A.3.1 A Class of Symbolic Expressions

Objectives:

1. Define a class of symbolic expressions in terms of ordered pairs and lists.
2. Define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples.
3. Show how these functions themselves can be expressed as symbolic expressions.
4. Define a universal function `apply` that allows us to compute from the expression for a given function its value for given arguments.
5. Define some functions with functions as arguments and give some useful examples.

Definition (S-expression)
1. Atomic symbols are S-expressions.
2. If e_1 and e_2 are S-expressions, then so is $(e_1 \cdot e_2)$.

where *atomic symbols* refers to the symbols `.` `(` and `)` as well as digits, strings of letters, and any other symbols that would be used. S stands for 'symbolic'.

Example.

- AB [Note: AB is an atomic symbol. If it were a composition of atomic symbols A and B, then it would be written as $(A \cdot B)$.]
- $(A \cdot B)$
- $((AB \cdot C) \cdot D)$

Remark. An arbitrary list

$$(m_1, m_2, \ldots, m_n)$$

can be represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))$$

where NIL is an atomic symbol used to terminate lists.

Notation.

- (m) stands for $(m \cdot NIL)$
- $(m_1, \cdots, m_n)$ stands for $(m_1 \cdot (\cdots (m_n \cdot NIL) \cdots))$
- $(m_1, \cdots, m_n \cdot x)$ stands for $(m_1 \cdot (\cdots (m_n \cdot x) \cdots))$
- $((AB, C), D)$ stands for $((AB \cdot (C \cdot NIL)) \cdot (D \cdot NIL))$
- $((A, B), C, D \cdot E)$ stands for $((A \cdot (B \cdot NIL)) \cdot (C \cdot (D \cdot E)))$

A.3.2 Functions of S-expressions and the Expressions That Represent Them

Definition (function of S-expressions, M-expression)
Functions, written in conventional functional notation, taking S-expressions as argument. The expression for a function of S-expressions is called an ***M-expression***; M for 'meta'.

Example.

- $car[x]$
- $car[cons[(A \cdot B); x]]$
- Note the use of square brackets instead of parentheses and the semicolon instead of the comma.

A.3.3 The Elementary S-functions and Predicates

Definition (atom[X])
atom[X] has the value of T or F according to whether X is an atomic symbol. For example:

- atom[X] = T
- atom[(X · A)] = F

Definition (eq[x;y])
Given atomic symbols x,y, eq[x;y] = T iff x and y are equivalent atomic symbols. For example:

- eq[X;X] = T
- eq[X;A] = F
- eq[X;(X · A)] is undefined since (X · A) is not an atomic symbol.

Definition (car[X])
car$[(e_1 \cdot e_2)] = e_1$. car[X] is undefined if X is an atomic symbol. For example:

- car[(X · A)] = X
- car[((X · A) · Y) = (X · A)]

Definition (cdr[X])
cdr[$(e_1 \cdot e_2)$] = e_2. cdr[X] is undefined if X is an atomic symbol. For example:

- cdr[(X · A)] = A
- cdr[((X · A) · Y)] = Y

Definition (cons[x;y])
cons[x;y] = (x · y). For example:

- cons[X;A] = (X · A)
- cons[(X · A); Y] = ((X · A) · Y)

Remark.

- car[cons[x;y]] = x
- cdr[cons[x;y]] = y
- cons[car[x];cdr[x]] = x, provided that x is not atomic
- cons[car[(X · A)]; cdr[(X · A)]] = cons[X;A] = (X · A)

A.3.4 Recursive S-functions

Definition (ff[X])
ff[X] is equivalent to the first atomic symbol of the S-expression X. For example:

- ff[((A ·B) · C)] = A
- ff[x] = [atom[x] → x; T → ff[car[X]]]

Definition (subst[x;y;z])
subst[x;y;z] is equivalent to the S-expression resulting from substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. For example:

- subst[x;y;z] = [atom[z] → [eq[z;y] → x; T → z]; T → cons[subst[x;y;car[z]]; subst[x;y;cdr[z]]]]
- subst[(X · A); B; ((A · B) · C)] = ((A · (X · A)) · C)

Definition (equal[x;y])
equal[x;y] = T iff x and y are equivalent S-expressions. For example:

- equal[x;y] = [atom[x] ∧ atom[y] ∧ eq[x;y]] ∨ [¬atom[x] ∧ ¬atom[y] ∧ equal[car[x]; car[y]] ∧ equal[cdr[x]; cdr[y]]]

Remark.

- car$[(m_1, m_2, \ldots, m_n)]$ = car$[(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))]$ = m_1
- cdr$[(m_1, m_2, \ldots, m_n)]$ = cdr$[(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots)))]$ = $(m_2 \cdot (\cdots (m_n \cdot NIL) \cdots))$ = $(m_2, \ldots, m_n)$
- cdr$[(m)]$ = cdr$[(m, NIL)]$ = NIL
- cons$[m_1; (m_2, \ldots, m_n)]$ = $(m_1, m_2, \ldots, m_n)$
- cons$[m; NIL]$ = (m, NIL) = (m)

Definition (null[x])
null[x] = atom[x] ∧ eq[x;NIL]

Definition (cadr[x], caddr[x])

- cadr[x] = car[cdr[x]] - car (a) and cdr (d) makes c(ad)r
- caddr[x] = car[cdr[cdr[x]]] - car (a) and two cdr's (dd) makes c(add)r
- etc...

Notation. $[e_1; e_2; \ldots; e_n]$ = cons$[e_1;$ cons$[e_2;$ ...; cons$[e_n; NIL]...]]$

This function gives the list $(e_1, \cdots, e_n)$ as a function of its elements.

Definition (append[x;y])
append[x;y] = [null[x] → y; T → cons[car[x]; append[cdr[x]; y]]]

This appends y to the end of the list x. For example:

- append[(A,B); (C,D,E)] = [null[(A,B)] → (C,D,E); T → cons[car[(A,B)]; append[cdr[(A,B)]; (C,D,E)]]]
- cons[car[(A,B)]; append[cdr[(A,B)]; (C,D,E)]]
- cons[A; append[B; (C,D,E)]]
- append[B; (C,D,E)] = [null[B] → (C,D,E); T → cons[car[B]; append[cdr[B]; (C,D,E)]]]
- cons[car[(B,NIL)]; append[cdr[(B,NIL)]; (C,D,E)]]

- cons[B; append[NIL; (C,D,E)]]
- append[NIL; (C,D,E)] = (C,D,E)
- append[B; (C,D,E)] = cons[B; append[NIL; (C,D,E)]] = cons[B; (C,D,E)] = (B,C,D,E)
- append[(A,B); (C,D,E)] = cons[A; append[B; (C,D,E)]] = cons[A; (B,C,D,E)] = (A,B,C,D,E)

Definition (among[x;y])
among[x;y] = ¬null[y] ∧ (equal[x; car[y]] ∨ among[x; cdr[y]])

This is true iff x occurs in the list y.

Definition (pair[x;y])
pair[x;y] = [(null[x] ∧ null[y]) → NIL; ¬atom[x] ∧ ¬atom[y] → cons[list[car[x]; car[y]]; pair[cdr[x]; cdr[y]]]]

This function gives the list of pairs of corresponding elements of the lists x and y. For example: pair[(A,B,C); (X,(Y,Z),U)] = ((A,X),(B,(Y,Z)),(C,U)).

Definition (assoc[x;y])
assoc[x;y] = [eq[caar[y]; x] → cadar[y]; T → assoc[x; cdr[y]]]

If y is a list of the form $((u_1, v_1), \cdots, (u_n, v_n))$ and x is one of the u's, then assoc[x;y] is the corresponding v. For example: assoc[X; ((W,(A,B)),(X,(C,D)),(Y,(E,F)))] = (C,D).

Definition (sub2[x;y], sublis[x;y])

- sub2[x;z] = [null[x] → z; eq[caar[x]; z] → cadar[x]; T → sub2[cdr[x]; z]]
- sublis[x;y] = [atom[y] → sub2[x;y]; T → cons[sublis[x; car[y]]; sublis[x; cdr[y]]]]

x is assumed to have the form of a list of pairs $((u_1, v_1), \cdots, (u_n, v_n))$ where the u's are atomic and y may be any S-expression.

z is assumed to be atomic. sublis[x;y] doesn't use sub2 until y is atomic. Until y is atomic, sublis is recursively applied to the left and right parts of y. Effectively, sublis[x;k] is applied to every atomic symbol k of y such that if k is equivalent to u_i in x, then replace k in y with v_i. Then a new S-expression is constructed with the same structure as y but with the appropriate substitutions having been made. sublis[x;y] returns this new S-expression.

Example. sublis[((X, (A, B)), (Y, (B, C))), (A, X · Y)] = (A, (A, B), B, C)

Expand the first argument ((X, (A, B)), (Y, (B, C))) into dot notation:

- (A, B) = (A · (B · NIL))
- (B, C) = (B · (C · NIL))

- (X, (A, B)) = (X · ((A, B) · NIL)) = (X · ((A · (B · NIL)) · NIL))
- (Y, (B, C)) = (Y · ((B, C) · NIL)) = (Y · ((B · (C · NIL)) · NIL))
- x = ((X, (A, B)), (Y, (B, C))) = ((X · ((A · (B · NIL)) · NIL)) · ((Y · ((B · (C · NIL)) · NIL)) · NIL))

The second argument (A, X · Y) expands into dot notation as:

- y = (A, X · Y) = (A · (X · Y))

The function sub2 isn't called until y is atomic. Until then, sublis is called on the left and right parts of y. This recursion unfurls the tree of y until sublis reaches each of the leaves of the tree of y. The leaves are atomic symbols, so they are passed on to sub2. sub2 then compares the leaves with the u_i's of x and returns v_i if there's a match or otherwise returns the leaf back. A new S-expression is constructed with the structure of y and the appropriate substitutions. sublis[x;y] returns this S-expression.

From y = (A · (X · Y)):

- sub2[x;A] = A
- sub2[x;X] = (A, B)
- sub2[x;Y] = (B, C)

Therefore:

- sublis[x;y] = (A · ((A, B) · (B, C))) = (A · ((A, B) · (B · (C · NIL)))) = (A, (A,B), B, C)

A.3.5 Representation of S-Functions by S-Expressions

Rules for translating M-expressions into S-expressions:

1. If $\mathcal{E}$ is an S-expression, then $\mathcal{E}^*$ is (QUOTE, $\mathcal{E}$).
2. Variables and function names that were represented by strings of lowercase letters are translated to the corresponding strings of the corresponding uppercase letters. E.g.: car* is CAR and subst* is SUBST.
3. A form $f[e_1; \ldots; e_n]$ is translated to $(f^*, e_1^*, \ldots, e_n^*)$. E.g.: cons[car[x];cdr[x]]* is (CONS, (CAR, X), (CDR, X)).
4. $[p_1 \to e_1; \ldots; p_n \to e_n]^*$ is (COND, (p_1^*, e_1^*), $\cdots$, $(p_n^* \cdot e_n^*)$).
5. $\lambda[[x_1; \ldots; x_n]; \mathcal{E}]^*$ is (LAMBDA, $(x_1^*, \cdots, x_n^*)$, $\mathcal{E}^*$).
6. $label[a; \mathcal{E}]^*$ is (LABEL, a^*, $\mathcal{E}^*$).

Example. The M-expression:

label[subst; λ[[x; y; z]; [atom[z] → [eq[y; z] → x; T → z]; T → cons[subst[x; y; car[z]]; subst[x; y; cdr[z]]]]]]

has the S-expression:

(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND, ((ATOM, Z), (COND, ((EQ, Y, Z), X), ((QUOTE, T), Z))), ((QUOTE, T), (CONS, (SUBST, X, Y, (CAR, Z)), (SUBST, X, Y, (CDR, Z))))))))

Note. What this function does: Unfurl the tree of z. For every leaf, if the leaf is equivalent to y then replace the leaf with x, otherwise leave the leaf alone. Return the resulting S-expression.

A.3.6 The Universal S-Function `apply`

Definition (apply[f; $args$])
apply[f; $args$] = eval[cons[f; appq[$args$]]; NIL]

Given S-expression f representing S-function f', and given $args$ is a list of arguments of the form $(arg_1, \ldots, arg_n)$ where $arg_1, \ldots, arg_n$ are arbitrary S-expressions: apply[f; $args$] is defined to be equivalent to $f'[arg_1, \ldots, arg_n]$.

Example. The S-function:

$$\lambda[[x; y]; cons[car[x]; y]][(A,B); (C,D)]$$

is equivalent to:

$$apply[(LAMBDA, (X,Y), (CONS, (CAR, X), Y)); ((A,B), (C,D))] = (A,C,D)$$

Definition (appq[m])
appq[m] = [null[m] $\rightarrow$ NIL; T $\rightarrow$ cons[list[QUOTE; car[m]]; appq[cdr[m]]]]

Definition (eval[e; a])

Index