NPRG086: Functions, trees, processes

Functions and functional programming

Tomáš Petříček, 204 (2nd floor)
petricek@d3s.mff.cuni.cz
@tomasp.net
https://tomasp.net

Charles University

From Lisp

Functional language?





Origins

Mark Priestley on Lisp

LISP was not based on the lambda calculus, despite using the word "LAMBDA" to denote functions. ... McCarthy was aware of (Church 1941) but had not studied it.

Charles University

Real origins of Lisp

Geometry theory machine

- Theorems in plane geometry
- Own research on AI topics

FLPL (1958)

- FORTRAN-compiled List-Processing Language
- Using 36-bit words for linked lists
- No recursion and other limitations

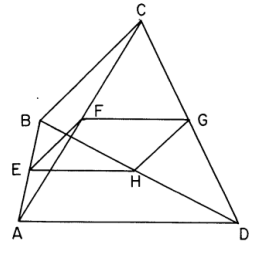
```
   PREMISES
**************
QUAD-LATERAL ABCD
POINT E MIDPOINT SEGMENT AB
POINT F MIDPOINT SEGMENT AC
POINT G MIDPOINT SEGMENT CD
POINT H MIDPOINT SEGMENT BD

   TO PROVE
**************
PARALELOGRAM EFGH

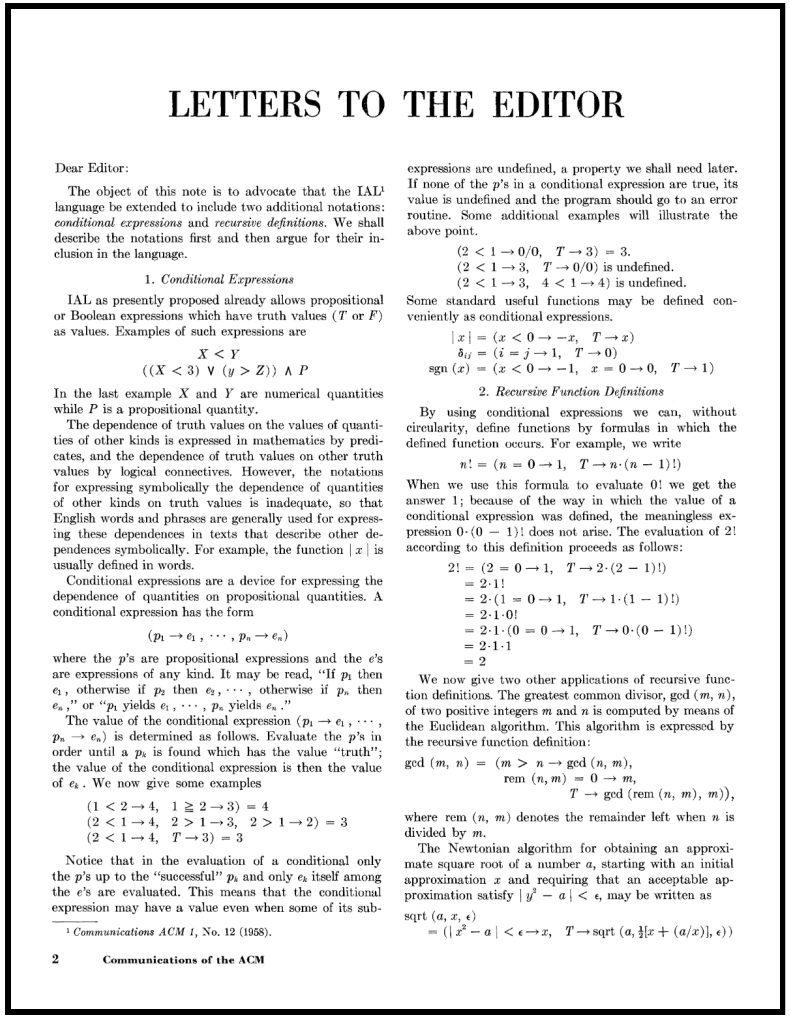
   SYNTACTIC SYMMETRIES
**************************
BA, AB, DC, CD, EE, HF, GG, FH,
CA, DB, AC, BD, GE, FF, EG, HH,
DA, CB, BC, AD, GE, HF, EG, FH,

   PROOF
***********
SEGMENT DG EQUALS SEGMENT GC
      DEFINITION OF MIDPOINT
SEGMENT CF EQUALS SEGMENT FA
      DEFINITION OF MIDPOINT
TRIANGLE DCA
      ASSUMPTION BASED ON DIAGRAM
PRECEDES DGC
      DEFINITION OF MIDPOINT
PRECEDES CFA
      DEFINITION OF MIDPOINT
SEGMENT GF PARALLEL SEGMENT AD
      SEGMENT JOINING MIDPOINTS OF SIDES OF TRIANGLE IS PARALLEL TO BASE
SEGMENT HE PARALLEL SEGMENT AD
      SYNTACTIC CONJUGATE
SEGMENT GF PARALLEL SEGMENT EH
      SEGMENTS PARALLEL TO THE SAME SEGMENT ARE PARALLEL
SEGMENT HG PARALLEL SEGMENT FE
      SYNTACTIC CONJUGATE
QUAD-LATERAL HGFE
      ASSUMPTION BASED ON DIAGRAM
PARALELOGRAM EFGH
      QUADRILATERAL WITH OPPOSITE SIDES PARALLEL IS A PARALLELOGRAM


   TOTAL ELAPSED TIME   1.03 MINUTES
```



Charles University



LETTERS TO THE EDITOR

Dear Editor:

The object of this note is to advocate that the IAL[1] language be extended to include two additional notations: *conditional expressions* and *recursive definitions*. We shall describe the notations first and then argue for their inclusion in the language.

1. *Conditional Expressions*

IAL as presently proposed already allows propositional or Boolean expressions which have truth values (T or F) as values. Examples of such expressions are

$$X < Y$$
$$((X < 3) \vee (y > Z)) \wedge P$$

In the last example X and Y are numerical quantities while P is a propositional quantity.

The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function $|\,x\,|$ is usually defined in words.

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

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

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

The value of the conditional expression $(p_1 \to e_1\,,\ \cdots,\ p_n \to e_n)$ is determined as follows. Evaluate the p's in order until a p_k is found which has the value "truth"; the value of the conditional expression is then the value of e_k. We now give some examples

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

Notice that in the evaluation of a conditional only the p's up to the "successful" p_k and only e_k itself among the e's are evaluated. This means that the conditional expression may have a value even when some of its subexpressions are undefined, a property we shall need later. If none of the p's in a conditional expression are true, its value is undefined and the program should go to an error routine. Some additional examples will illustrate the above point.

$$(2 < 1 \to 0/0,\quad T \to 3) = 3.$$
$$(2 < 1 \to 3,\quad T \to 0/0) \text{ is undefined.}$$
$$(2 < 1 \to 3,\quad 4 < 1 \to 4) \text{ is undefined.}$$

Some standard useful functions may be defined conveniently as conditional expressions.

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

2. *Recursive Function Definitions*

By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

$$n! = (n = 0 \to 1,\quad T \to n \cdot (n-1)!)$$

When we use this formula to evaluate 0! we get the answer 1; because of the way in which the value of a conditional expression was defined, the meaningless expression $0 \cdot (0-1)!$ does not arise. The evaluation of 2! according to this definition proceeds as follows:

$$\begin{aligned} 2! &= (2 = 0 \to 1,\quad T \to 2 \cdot (2-1)!) \\ &= 2 \cdot 1! \\ &= 2 \cdot (1 = 0 \to 1,\quad T \to 1 \cdot (1-1)!) \\ &= 2 \cdot 1 \cdot 0! \\ &= 2 \cdot 1 \cdot (0 = 0 \to 1,\quad T \to 0 \cdot (0-1)!) \\ &= 2 \cdot 1 \cdot 1 \\ &= 2 \end{aligned}$$

We now give two other applications of recursive function definitions. The greatest common divisor, $\gcd(m, n)$, of two positive integers m and n is computed by means of the Euclidean algorithm. This algorithm is expressed by the recursive function definition:

$$\begin{aligned} \gcd(m, n) = (&m > n \to \gcd(n, m), \\ &\operatorname{rem}(n, m) = 0 \to m, \\ &T \to \gcd(\operatorname{rem}(n, m), m)), \end{aligned}$$

where $\operatorname{rem}(n, m)$ denotes the remainder left when n is divided by m.

The Newtonian algorithm for obtaining an approximate square root of a number a, starting with an initial approximation x and requiring that an acceptable approximation satisfy $|\,y^2 - a\,| < \epsilon$, may be written as

$$\begin{aligned} &\operatorname{sqrt}(a, x, \epsilon) \\ &\quad = (|\,x^2 - a\,| < \epsilon \to x,\quad T \to \operatorname{sqrt}(a, \tfrac{1}{2}[x + (a/x)], \epsilon)) \end{aligned}$$

[1] *Communications ACM 1*, No. 12 (1958).

Dear Editor:

Two proposed additions to IAL (later Algol)

1. Conditional expression
2. Recursive functions

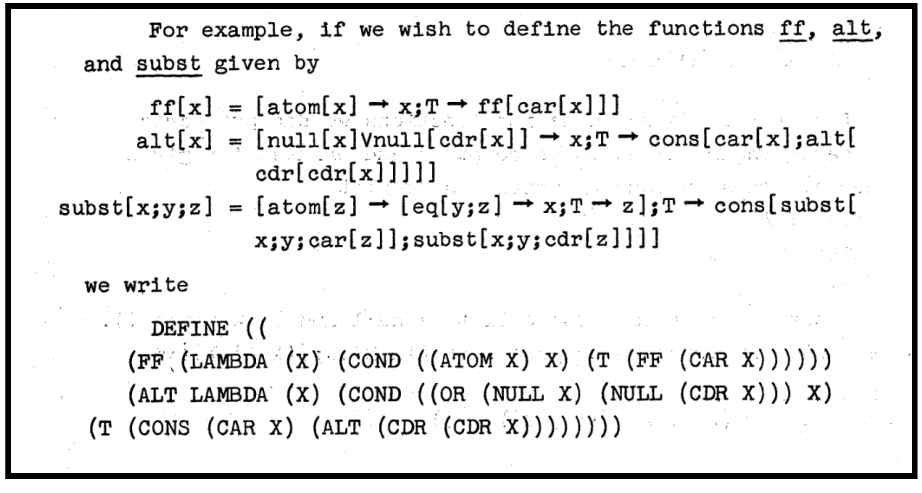
The first was added, the second not forbidden

Charles University

LISP I - Manual (1960)

Two notations

- "Normal" M-expressions
- List-based S-expressions
- **`LAMBDA`** in the latter



For example, if we wish to define the functions ff, alt, and subst given by

```
ff[x] = [atom[x] → x;T → ff[car[x]]]
alt[x] = [null[x]Vnull[cdr[x]] → x;T → cons[car[x];alt[
         cdr[cdr[x]]]]]
subst[x;y;z] = [atom[z] → [eq[y;z] → x;T → z];T → cons[subst[
         x;y;car[z]];subst[x;y;cdr[z]]]]
```

we write

```
DEFINE ((
(FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))))
(ALT LAMBDA (X) (COND ((OR (NULL X) (NULL (CDR X))) X)
(T (CONS (CAR X) (ALT (CDR (CDR X)))))))))
```

Language features

- Lists based on FLPL
- First-class recursion support
- Imperative programming with **`goto`**
- Higher-order functions like **`maplist`**

Charles University

```
      The S-function apply is defined by

      apply[f;args] = eval[cons[f;appq[args]];NIL]

where

      appq[m] = [null[m] → NIL;T → cons[list[QUOTE;car[m]];appq[
                cdr[m]]]]

and

      eval[e;p] = [
        atom[e] → [assoc[e;p];
        atom[car[e]] → [
          eq[car[e];QUOTE] → cadr[e];
          eq[car[e];ATOM] → atom[eval[cadr[e];p]];
          eq[car[e];EQ] → eq[eval[cadr[e];p];eval[caddr[e];p]];
          eq[car[e];COND] → evcon[cdr[e];p];
          eq[car[e];CAR] → car[eval[cadr[e];p]];
          eq[car[e];CDR] → cdr[eval[cadr[e];p]];
          eq[car[e];CONS] → cons[eval[cadr[e];p];eval[caddr[e];p]];
             T → eval[cons[assoc[car[e];p];evlis[cdr[e];p]];p]];
          eq[caar[e];LABEL] → eval[cons[caddar[e];cdr[e]];cdr[e]];
             cons[list[cadar[e];car[e]];p]];
          eq[caar[e];LAMBDA] → eval[caddar[e];append[pair[cadar[e];
             evlis[cdr[e];p];p]]]]

and

      evcon[c;p] = [eval[caar[c];p] → eval[cadar[c];p];T → evcon[
                   cdr[c];p]]

and

      evlis[m;p] = [null[m] → NIL;T → cons[list[eval[car[m];p];
                   evlis[cdr[m];p]]]]
```

Theoretical links

Theory of recursive functions

Refernece to partial recursive functions

Universal function

`eval` function - first theoretical idea, later implemented

(No reference to ADES)

Charles University

Lisp

How functions and variables worked

- Reduction in lambda-calculus $(\lambda x.e_1)e_2 \rightarrow_\beta e_1[x/e_2]$
- Not practical for actual implementation!
- Variables stored in association list (a-list)
- Most recent variable from a-list used
- Dynamic scoping - not as in lambda calculus!

Charles University

The Function of FUNCTION in LISP: Why the FUNARG problem Should be Called the Environment Problem

- `a` assigned `2` inside `g`
- function `f` is then returned
- `a` assigned `0` before calling it

```
define f(x) =
   begin;
      if a = 0 then return x
      else return -x;
      end;
define g(x) =
   begin local a;
      a = 2;
   return f
      end;
begin local a,b,h;
   a = 0;
   h = g(2);
   b = h(3);
   end;
```

Not obvious!

- Weizenbaum (1968), The FUNARG Problem Explained
- Moses (1970), The Function of FUNCTION in LISP Why the FUNARG problem...
- Sandewall (1971), A proposed solution to the FUNARG problem
- Steele & Sussman (1975), The Lambda papers

Charles University

Faulty function (LISP I)

$$testr[x, p, f, u] \leftarrow \textbf{if } p[x] \textbf{ then } f[x] \textbf{ else if } atom[x] \textbf{ then } u[\,] \textbf{ else } testr[cdr[x], p, f, \lambda{:}testr[car[x], p, f, u]].$$

I must confess that I regarded this difficulty as just a bug and expressed confidence that Steve Russell would soon fix it. He did fix it, but by inventing the so-called FUNARG device that took the lexical environment along with the functional argument.

(McCarthy, 1978)

Charles University

Aside: Semantics of Algol

The mechanical evaluation of expressions (1964)

> *We represent the value of a λ-expression by a bundle of information called a "closure," comprising the λ-expression and the environment relative to which it was evaluated.*

Language semantics motivation

- Landin's model of Algol using lambda calculus
- Recall Mervyn Pragnell's reading group!
- "similar to FUNARG" and in contrast to "literal substitution"

Charles University

Language semantics

How to model `goto` and returns?

- van Wijngaarden (1964): "Provide each procedure declaration with an extra formal parameter... and insert at the end [goto] leading to that formal parameter."
- Many names for the concept: Landin (1964) SECD "dump", Mazurkiewicz (1969) "tail function", Morris (1970) "label values", Abdali (1973) "program remainder"
- Wadsworth (1970) "Once I'd coined the word [continuation] it all clicked and the rest followed."

Charles University

Continuations

Shifting motivations



Programming language semantics

Wijngaarden, Landin, Morris, Wadsworth



Proving program properties

Mazurkiewicz, Strachey



Managing environments efficiently

Fischer, Steele and Sussman



Scheme: The Lambda papers

- Scheme: An Interpreter for Extended LC (1975)
  Lexically scoped Lisp with closures, explaining actors
- Lambda: The Ultimate Imperative (1976)
  GOTO, assignment and loops modelled with CPS
- Lambda: The Ultimate Declarative (1976)
  Lambda as naming; does not need to return!
- Debunking the "Expensive Procedure Call" Myth (1977)
  Using tail calls to compile calls effectively

Charles University

Lessons from the Lisp history

Functions in Lisp

- Funarg problem fixed via workarounds
- "Monster" function to be "fixed"
- 15 years from Lisp to Scheme!
- Not a practical engineering problem?



Influences across levels

- AI motivations, weak lambda calculus link
- Continuations move from semantics to implementation
- Theoretical influence leads to (successful) Scheme

Towards Haskell

Functional programming!



Peter Landin

Actual lambda calculus influence

From Roger Penrose via Mervy Pragnell's reading group

Lambda calculus for semantics and design

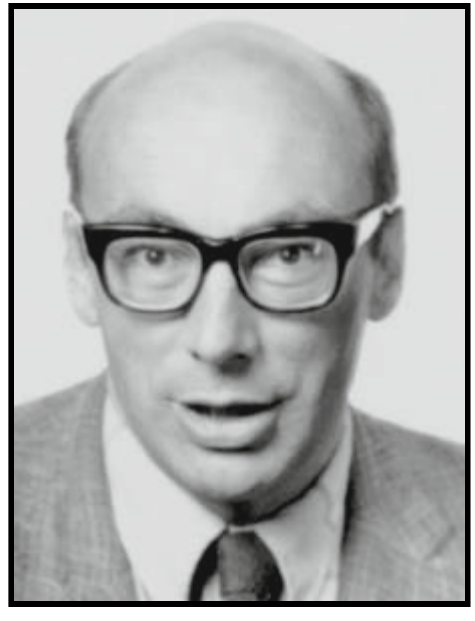
- The mechanical evaluation of expressions (1964)
- A correspondence between ALGOL 60 and LC (1965)
- A Generalization of Jumps and Labels (1965)
- The next 700 programming languages (1966)



Peter Landin

Papers on semantics

- SECD (stack, env, control, dump)
- Language constructs translate to LC
- Closure = function + environment



ISWIM and 700 PLs

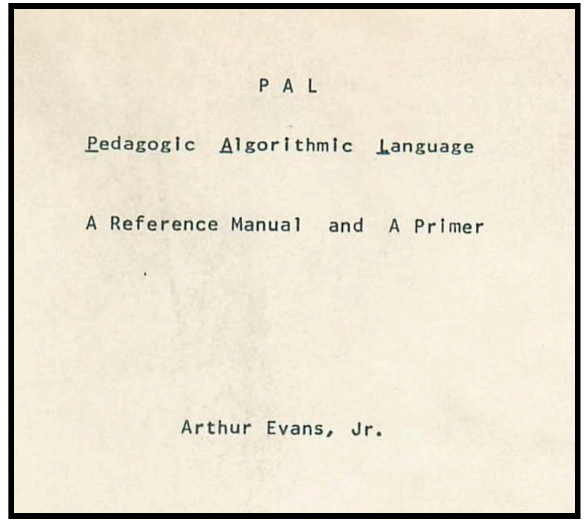
- Design range of LC-based languages
- Prefers denotative over functional or declarative
- Function definitions in a modern functional sense

Charles University

Early functional-style languages

PAL (MIT, 1967)

- ISWIM implementation in BCPL
- Small applicative subset (lambdas, functions, tuples, conditional, let)
- Imperative (assignment, goto)


P A L

Pedagogic Algorithmic Language

A Reference Manual and A Primer

Arthur Evans, Jr.

COWSEL/POP-1 (1966)

- Partial function application (misnomer)
- Dynamic scoping (for memory/engineering reasons)
- Frozen formals `(% ... %)` akin to **FUNARG**

POP-2 (Edinburgh, 1970)

- Algol-inspired with "functions first-class"
- Burstall addressing Strachey's complaint about Algol

Charles University

Theoretical work

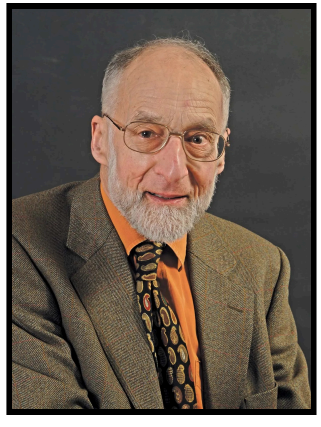
Typed lambda calculus

- Scott (1970s) - Type-theoretical alternative to ISWIM
- Typed LC model with denotational semantics
- Functions are partial (bottom element)
- Made unnecessary by model of untyped LC
- Morris, Reynolds (1970s) - Theory of type structure

Charles University

LCF/ML (Milner, late 1970s)

LCF theorem prover



- Aims to prove compiler correctness
- Scott's Logic for Computable Function
- Interactive Stanford LCF theorem prover (made possible by time-sharing?)
- Edinburgh LCF adds meta-language

Meta-language

- Write code to construct theorems in LCF
- Types used to ensure correctness
- Becomes a programming language (HOPE, VAX ML, SML)

Formal verification

From PL/CV to NuPrl

- ✓ PL/1 language with support for correctness proofs
- → Difficulties getting "functions right" (first-class)
- 🎓 ALGOL 68 does not; λ-calculus "not respectable"
- X_1 Martin-Löf's theory to clarify "intuitionistic mathematics"
- ☑ Adopted by Constable for Nuprl (PL/CV follow-up)

Charles University

Functional programming

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

Backus, Turing Award lecutre (1978)

Just as structured programming eschews many control statements to obtain programs with simpler structure, better properties, and uniform methods for understanding their behavior, so functional programming eschews the lambda expression, substitution, and multiple function types.

Functional programming terminology, but talks about algebra of programs and formal reasoning



Backus lecture

Unexpected effects

 Agreement on von Neumann bottleneck

 Adoption of the term functional programming

 Most kept lambda functions and variables

 Presents past work from LC perspective

Charles University

Caveats

Dijkstra's criticism of the talk

> *In short, the article is a progress report on a valid research effort but suffers badly from aggressive overselling of its significance, long before convincing results have been reached. This is the more regrettable as it has been published by way of Turing Award Lecture.*

Well-founded critique

- Algebra of programs exists, but more academic
- Later work on FP tries to develop optimizations...



Functional programming

SASL (Turner, 1978), KRC (Turner, 1982)

Lazy, purely functional programming languages

Miranda (Turner, 1985)

Purely functional, non-strict language with Hindley–Milner polymorphism in one system

Haskell (1990)

Program is a function, mostly follows Miranda, Type classes as a main additional novelty



Conclusions

Functional programming



Many ideas struggle

Possibly before firm link across formal, implementation and intuitive levels?

- First-class functions (ALGOL, PL/CV, LISP)
- Links to recursion & lambda calculus (ADES, LISP)
- Variable binding (von Neumann) and scoping (LISP)
- Continuations (from semantics to implementation)

Charles University

Interactions across levels

Between theory and practice

- Translation to a small formal model
- ISWIM as theory, Scheme as implementation
- Not obvious well-documented link

From theory to intuition

- Backus' FP system and von Neuman bottleneck
- Main effect on thinking about functional programming!



Monsters



Pseudo-functions with effects (print) in LISP 1.5

Eliminated in pure functional languages



Laziness in LC, but not mathematical functions

Simplifies calling conventions of Algol



Funarg problem seen as something to be fixed

Fixes eventually replaced by different approach

Charles University


Charles
University