(FP Dag 2026)

A Geneaology of Functional Programming Languages

Prepared by Philip KALUĐERČIĆ
https://cs.ru.nl/~pkal/pub/gfp/

Presented on 2026-01-09, last typeset on January 8, 2026

The Conventional Summary…

1. λ-Calclus (1936)
2. LISP (1958) — Implemented λ-Calculus
3. ML (1978) — Add Static Types
4. Haskell (1989) — Add Monads

The Conventional Summary…

1. λ-Calclus (1936)
2. LISP (1958) — Implemented λ-Calculus
3. ML (1978) — Add Static Types
4. Haskell (1989) — Add Monads
5. Java 8/C++ 11/Rust/... — *FP becomes mainstream/sells out?*

History is messy…

History is messy. . .
and life imitates art!

2. Conversion and λ-definability. We select a particular list of symbols, consisting of the symbols { , }, (,), λ, [,], and an enumerably infinite set of symbols *a*, *b*, *c*, · · · to be called *variables*. And we define the word *formula* to mean any finite sequence of symbols out of this list. The terms *well-formed formula, free variable,* and *bound variable* are then defined by induction as follows. A variable $\boldsymbol{x}$ standing alone is a well-formed formula and the occurrence of $\boldsymbol{x}$ in it is an occurrence of $\boldsymbol{x}$ as a free variable in it; if the formulas $\boldsymbol{F}$ and $\boldsymbol{X}$ are well-formed, $\{\boldsymbol{F}\}(\boldsymbol{X})$ is well-formed, and an occurrence of $\boldsymbol{x}$ as a free (bound) variable in $\boldsymbol{F}$ or $\boldsymbol{X}$ is an occurrence of $\boldsymbol{x}$ as a free (bound) variable in $\{\boldsymbol{F}\}(\boldsymbol{X})$; if the formula $\boldsymbol{M}$ is well-formed and contains an occurrence of $\boldsymbol{x}$ as a free variable in $\boldsymbol{M}$, then $\lambda\boldsymbol{x}[\boldsymbol{M}]$ is well-formed, any occurrence of $\boldsymbol{x}$ in $\lambda\boldsymbol{x}[\boldsymbol{M}]$ is an occurrence of $\boldsymbol{x}$ as a bound variable in $\lambda\boldsymbol{x}[\boldsymbol{M}]$, and an occurrence of a variable $\boldsymbol{y}$, other than $\boldsymbol{x}$, as a free (bound) variable in $\boldsymbol{M}$ is an occurrence of $\boldsymbol{y}$ as a free (bound) variable in $\lambda\boldsymbol{x}[\boldsymbol{M}]$.

Figure 1. From Church' "An Unsolvable Problem of Elementary Number Theory" (1936)

$$I_{\alpha\alpha} \rightarrow \lambda x_\alpha x_\alpha.$$

$$K_{\alpha\beta\alpha} \rightarrow \lambda x_\alpha \lambda y_\beta x_\alpha.$$

$$0_{\alpha'} \rightarrow \lambda f_{\alpha\alpha} \lambda x_\alpha x_\alpha,$$

$$1_{\alpha'} \rightarrow \lambda f_{\alpha\alpha} \lambda x_\alpha (f_{\alpha\alpha} x_\alpha),$$

$$2_{\alpha'} \rightarrow \lambda f_{\alpha\alpha} \lambda x_\alpha (f_{\alpha\alpha}(f_{\alpha\alpha} x_\alpha)),$$

$$3_{\alpha'} \rightarrow \lambda f_{\alpha\alpha} \lambda x_\alpha (f_{\alpha\alpha}(f_{\alpha\alpha}(f_{\alpha\alpha} x_\alpha))), \text{ etc.}$$

$$S_{\alpha'\alpha'} \rightarrow \lambda n_{\alpha'} \lambda f_{\alpha\alpha} \lambda x_\alpha (f_{\alpha\alpha}(n_{\alpha'} f_{\alpha\alpha} x_\alpha)).$$

Figure 2. From Chrch’ “A Formulation of the Simple Theory of Types” (1940)

$$Ix = x$$

$$Kxy = x$$

$$Bxyz = x(yz)$$

$$Cxyz = xzy$$

$$Sxyz = xz(yz).$$

Figure 3. From Curry's "Grundlagen der Kombinatorischen Logik" (1930)

These are all formal systems, not programming languages!

These are all formal systems, not programming languages!

A caricature of PL history:
1950s: What are programming languages?
1960s: What else can programming languages be?
1970s: What do programming languages need to be?
1980s: What do programming languages need?
1990s: What do programmers need?
2000s: How do programming languages scale?
...

```
apply[fn;x;a] =
      [atom[fn] → [eq[fn;CAR] → caar[x];
                   eq[fn;CDR] → cdar[x];
                   eq[fn;CONS] → cons[car[x];cadr[x]];
                   eq[fn;ATOM] → atom[car[x]];
                   eq[fn;EQ] → eq[car[x];cadr[x]];
                   T → apply[eval[fn;a];x;a]];
      eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
      eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                  caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
      atom[car[e]]→
               [eq[car[e],QUOTE] → cadr[e];
               eq[car[e];COND] → evcon[cdr[e];a];
               T → apply[car[e];evlis[cdr[e];a];a]];
      T → apply[car[e];evlis[cdr[e];a];a]]
```

Figure 4. The “Maxwell-Equations of Software” from the LISP 1.5 Programmers Manual (1962)

Lisp

- Not first that manipulates expressions (**BACAIC**)
- New: (Proper) Conditional Expressions,

  "Language $\cong$ Encoding"

- Real-world code was imperative and **FORTRAN** then **ALGOL**-like
- Not until **Scheme** (1975) was lexical scoping and TCO implemented

```
(prog (x y z)   ;x, y, z are prog variables   - temporaries.
   (setq y (car w) z (cdr w))       ;w is  a free  variable.
loop
   (cond ((null y) (return x))
         ((null z) (go err)))
rejoin
   (setq x (cons (cons (car y) (car z))
                 x))
   (setq y (cdr y)
         z (cdr z))
   (go loop)
err
   (break are-you-sure? t)
   (setq z  y)
   (go rejoin))
```

Figure 5. An example of the `prog` macro from the MacLisp manual (1974)

$\underline{1}$ $\quad a_0 11 : q_0 11,$

$\underline{2}$ $\qquad f_{50} = + \surd \text{ abs } c_1 \cdot \cdot \cdot 5\ c_1\ c_1\ c_1,$

$\underline{3}$ $\quad d_{12} b_1 = r_0,$

$\underline{4}$ $\quad d_{22} b_2 = \leq b_3\ 400,\ b_3,\ 999,$

$\underline{5}$ $\qquad b_3 = f_{50}\ a_0\ r_0,$

$\underline{6}$ $\qquad r_0 = -10\ q_0,$

$\underline{7}$ $\quad \forall\ 0\ q_0\ 10\ b_0 = f_0\ b_1\ b_2,$

Figure 6. From Knuth's "The Early Development of Programming Languages" (1975); ADES was implemented in 1956

```
TPK:  begin integer i; real y; real array a[0:10];
        real procedure f(t); real t; value t;
           f := sqrt(abs(t)) + 5 × t ↑ 3;
        for i := 0 step 1 until 10 do read(a[i]);
        for i := 10 step -1 until 0 do
        begin y := f(a[i]);
           if y > 400 then write(i,"TOO LARGE")
                      else write(i,y);
        end
      end.
```

Figure 7. From Knuth's "The Early Development of Programming Languages" (1975)

$$f(b+2c) + f(2b-c)$$
$$\textbf{where } f(x) = x(x+a)$$

$$f(b+2c) + f(2b-c)$$
$$\textbf{where } f(x) = x(x+a)$$
$$\textbf{and } b = u/(u+1)$$
$$\textbf{and } c = v/(v+1)$$

$$g(f \textbf{ where } f(x) = ax^2 + bx + c,$$
$$u/(u+1),$$
$$v/(v+1))$$
$$\textbf{where } g(f, p, q) = f(p+2q, 2p-q)$$

Figure 8. ISWIM: From Landin's "The Next 700 Programming Languages" (1966)

If You See What I Mean

- A family of idealized languages
- Syntactically a variation of ALGOL with more "mathematical notation"
- Never implemented, but inspired many subsequent languages (**POP-2**, **GEDANKEN**, **PAL**)
- Introduces `where` and `let` notation

*An important distinction is the one between indicating what behavior, **step-by-step**, you want the machine to perform, and merely **indicating what outcome you want**. [...]*

*An important distinction is the one between indicating what behavior, **step-by-step**, you want the machine to perform, and merely **indicating what outcome you want**. [...]
The word "**denotative**" seems more appropriate than nonproeedural, declarative or functional. The antithesis of denotative is "**imperative**." — Landin*

```
module ordered_trees
  pubtype otree
  pubconst empty, insert, flatten

  data otree == empty ++ tip(num)
                      ++ node(otree#num#otree)

  dec insert : num#otree -> otree
  dec flatten : otree -> list num

  --- insert(n,empty)  <= tip(n)
  --- insert(n,tip(m))
               <= n<m then node(tip(n),m,empty)
                      else node(empty,m,tip(n))
  --- insert(n,node(t1,m,t2))
               <= n<m then node(insert(n,t1),m,t2)
                      else node(t1,m,insert(n,t2))

  --- flatten(empty)  <= nil
  --- flatten(tip(n)) <= [n]
  --- flatten(node(t1,n,t2))
                 <= flatten(t1) <> (n::flatten(t2))

end
```

Figure 9. From “Hope: An Experimental Applicative Language” (1980)

Pattern Matching

- **NPL** (1977) and **HOPE** (1988) implement pattern matching as control flow!
- (They implemented “set/list comprehension”)
- Not the first: **Refal** (1968, Turchin) preceded
- Pattern-matching on input: **SNOBOL** (1962), **AWK** (1977)

```
Fact { 0 = 1;
   s.N = <* s.N <Fact <- s.N 1>>>; }

Fact { s.n = <Loop s.n 1>; };
Loop {
   0 s.f = s.f;
   s.n s.f = <Loop <- s.n 1> <* s.n s.f>>; }
```

Figure 10. An example from the “Refal” Wikipedia page

Recursive definitions can be quite complicated, as in the following example, which recognizes a simple class of arithmetic expressions.

```
       &ANCHOR  =  1
       VARIABLE  =  ANY('XYZ')
       ADDOP  =  ANY('+-')
       MULOP  =  ANY('*/')
       FACTOR  =  VARIABLE  |  '('  *EXP  ')'
       TERM  =  FACTOR  |  *TERM  MULOP  FACTOR
       EXP  =  ADDOP  TERM  |  TERM  |  *EXP  ADDOP  TERM
LOOP   STRING  =  TRIM(INPUT)                               :F(END)
       STRING  EXP RPOS(0)                                  :F(NOGOOD)
       OUTPUT  =  STRING  '  IS AN EXPRESSION.'             :(LOOP)
NOGOOD OUTPUT  =  STRING  '  IS NOT AN EXPRESSION.'         :(LOOP)
END
```

Output for typical data is

```
X+Y*(Z+X)  IS AN EXPRESSION.
X+Y+Z  IS AN EXPRESSION.
XY  IS NOT AN EXPRESSION.
```

Figure 11. From Griswold's "The SNOBOL 4 Programming Language" (1968/71)

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!
- **LCF ML** used as language for tactics

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!
- **LCF ML** used as language for tactics
- New: *Strong, Polymorphic Type Inference*

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!
- **LCF ML** used as language for tactics
- New: *Strong, Polymorphic Type Inference*
- With inspiration from **HOPE**, it matured into **Standard ML** (1983-1989, 1997)

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!
- **LCF ML** used as language for tactics
- New: *Strong, Polymorphic Type Inference*
- With inspiration from **HOPE**, it matured into **Standard ML** (1983-1989, 1997)
- **CAML** (1985), **OCaml** (996) developed in INRIA

1978 ML; 1985: CAML; 1996: OCaml

- ML := ISWIM with type inference (also exceptions)!
- **LCF ML** used as language for tactics
- New: *Strong, Polymorphic Type Inference*
- With inspiration from **HOPE**, it matured into **Standard ML** (1983-1989, 1997)
- **CAML** (1985), **OCaml** (996) developed in INRIA
- Inspired **F#** (2005), **Rust** (2006), **Rocq** (1989)

```
absrectype * tree = * + * tree # * tree
 with leaf n = abstree(inl n)
  and node (t1, t2) = abstree(inr(t1, t2))
  and isleaf t = isl(reptree t)
  and leafval t = outl(reptree t) ? failwith 'leafval'
  and leftchild t = fst(outr(reptree t) ? failwith 'leftchild'
  and rightchild t = snd(outr(reptree t) ? failwith 'leftchild'
```

(Example from Leroy's "25 years of OCaml")

[redacted]

(1) As we said in the introduction, the addition of extra (primitive) type operators such as $\times$ (Cartesian product), $+$ (disjoint sum) and *list* (list forming), causes no difficulty. Together with $\rightarrow$, these are the primitive type operators in the language ML. For $\times$ one has the standard polymorphic functions

pair: $\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$ (one could add the syntax (e, e') for pair $(e)(e')$),
fst: $\alpha \times \beta \rightarrow \alpha$,
snd: $\alpha \times \beta \rightarrow \beta$.

For $+$, one has

inl: $\alpha \rightarrow \alpha + \beta$,	inr: $\beta \rightarrow \alpha + \beta$	(left and right injections),
outl: $\alpha + \beta \rightarrow \alpha$,	outr: $\alpha + \beta \rightarrow \beta$	(left and right projections),
isl: $\alpha + \beta \rightarrow$ *bool*,	isr: $\alpha + \beta \rightarrow$ *bool*	(left and right discriminators)

Figure 12. From Millner's "A Theory of Type Polymorphism in Programming" (1978)

LCF ML as "typed Lisp" (1978)

```
letrec sumtree t =
  if isleaf t then
    leafval t
  else
    sumtree (leftchild t)
    + sumtree (rightchild t)
```

Core ML toward SML (1983)

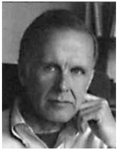
```
type 'a tree =
  | Leaf of 'a
  | Node of 'a tree
          * 'a tree
```

```
let rec sumtree t =
  match t with
  | Leaf n -> n
  | Node (l, r) ->
      sumtree l
      + sumtree r
```

```
        ∇IN[⎕]∇
      ∇ Z←A IN B;J
[1]     J←(A[1]=B)/⍳⍴B
[2]     J←(J≤1+(⍴B)-⍴A)/J
[3]     Z←(B[J∘.+¯1+⍳⍴A]∧.=A)/J
      ∇
        ∇IN1[⎕]∇
      ∇ T←A IN1 B
[1]     T←A IN B
[2]     →2×J<⍴T←(~(⍳⍴T)∊J←1+((⍴A)>|-/[1](2,1+⍴T)⍴T)⍳1)/T
      ∇

        W←'THE'
        T←'THE MEN THEN WENT HOME,'
        W IN T
1  9
        W IN1 T
1  9
        'ABA' IN 'NOWABABABABABABABA'
4  6  8  10  12  14  16
        'ABA' IN1 'NOWABABABABABABABA'
4  8  12  16
```

Figure 13. From the “APL\360 User Manual” (1968)



5.2 A Functional Program for Inner Product

Def Innerproduct

$$\equiv (\text{Insert } +)\circ(\text{ApplyToAll } \times)\circ\text{Transpose}$$

Or, in abbreviated form:

Def IP $\equiv (/+)\circ(\alpha\times)\circ\text{Trans.}$

Figure 14. From Backus' " Can Programming Be Liberated from the von Neumann Style" (1977)

- Backus introduced **FP** in his 1977 Turing Award Paper, inspired by **APL**
- Focus is on combinations of “functional forms”
- Functionals: Composition, conditionals, apply-to-all, insert-right
- Untyped, not based on λ-Calculus, succeeded by **FL**

David Turner

- **SASL** (1976) implemented the functional subset of ISWIM
- Initially **eager**, later turned **lazy**
- Used as Burroughs to implement an Operating System
- Turner later developed **KRC** (1982) and **Miranda** (1985)
- None of these functions hat λ Expressions
- *Miranda was proprietary software*, interest in a free alternative spawned Haskell

```
sieve (from 2)
where
from n = n: from (n+1)
sieve (p : x) = p: sieve (filter x)
                where
                filter (n : x) =
                     n rem p = 0 → filter x;
                     n: filter x
```

Figure 6. The list of all the prime numbers

Figure 15. SASL: From Turner's "A New Implementation Technique for Applicative Languages" (1979)

```
abstype stack *
with empty :: stack *
     isempty :: stack *->bool
     push :: *->stack *->stack *
     pop :: stack *->stack *
     top :: stack *->*

stack * == [*]
empty = []
isempty x = (x=[])
push a x = a:x
pop(a:x) = x
top(a:x) = a
```

Figure 16. From Turner's "Miranda: A non-strict functional language with polymorphic types" (1985)

Haskell (1989)

- Lazy Evaluation grew more popular from late 70's onwards.
- Designed in late 1980s to concentrate disparate efforts (Miranda, Lazy ML, Orwell, Alfl, Id, ...).
- Goal: A pure, lazy, functional language standard
- "Type classes" used to solve SML's eq-type and arithmetic-overloading Problems
- Initially many implementations, over time Haskell became GHC defacto

More Languages worth Mentioning

Curry (1995) Extends pattern-matching with a special form of Prolog-like unification

Lucid (1985) A "dataflow language" where each program generates a stream of values

Clean (1987) Use of "uniqueness types" as opposed to monads

Erlang (1986) A distributed programming language inspired by the Actor model

Idris (2007) Dependent types in a "real-world" language (+ more)

Unison (2017) Effect system in a "real-world" language (+ more)

```haskell
-- Returns the last number of a list.
last :: [Int] -> Int
last (_ ++ [x]) = x

-- Returns some permutation of a list.
perm :: [a] -> [a]
perm []     = []
perm (x:xs) = insert (perm xs)
 where insert ys     = x : ys
       insert (y:ys) = y : insert ys
```

Figure 17. From Curry's homepage

Closing Comments and Questions

- What is the *sine qua non* of functional programming? (functions, lambdas, static types, composition, pattern matching, referential transparency, ...)

Closing Comments and Questions

- What is the *sine qua non* of functional programming? (functions, lambdas, static types, composition, pattern matching, referential transparency, ...)
- Is functional programming really declarative?

Closing Comments and Questions

- What is the *sine qua non* of functional programming? (functions, lambdas, static types, composition, pattern matching, referential transparency, . . .)
- Is functional programming really declarative?
- Will functional programming be reduced to a historical footnote, as the ideas are absorbed into "mainstream" languages?