Language Design is LEGO Design and Library Design

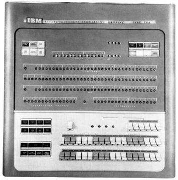
Stephen A. Edwards

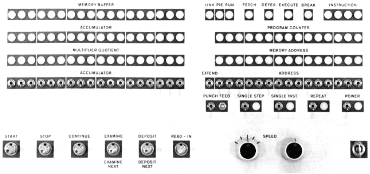
Columbia University

Forum on Specification & Design Languages
Southampton, United Kingdom, September 3, 2019

User-defined functions and pointers in imperative languages

Language design choices are often heavily influenced by processor architectures. Understand the processor to understand the language





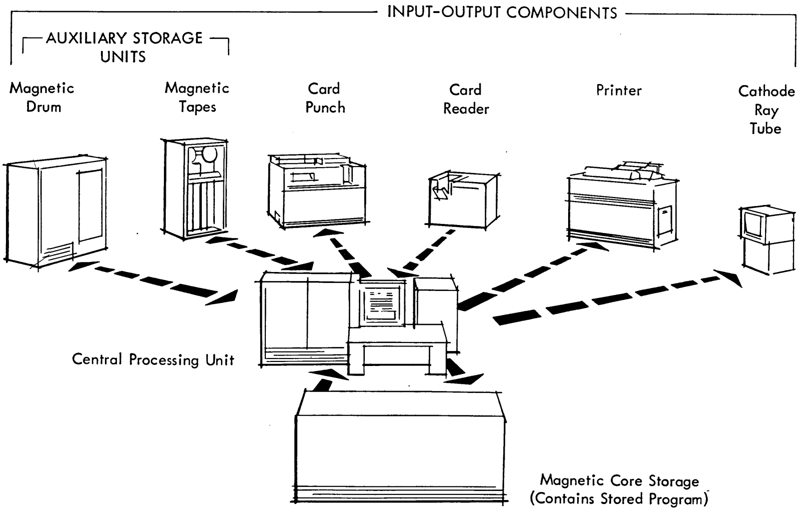
Best to understand how to compile a feature before adding it to the language

Report on the Algorithmic Language ALGOL 60

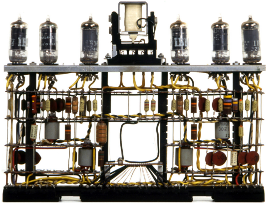
Peter Naur (*Editor*)

J. W. Backus	C. Katz	H. Rutishauser	J. H. Wegstein
F. L. Bauer	J. McCarthy	K. Samelson	A. van Wijngaarden
J. Green	A. J. Perlis	B. Vauquois	M. Woodger

1954: The IBM 704 Electronic Data-Processing Machine



[IBM 704 Manual of Operation, 1955]



36-bit Integer & Floating-point ALU

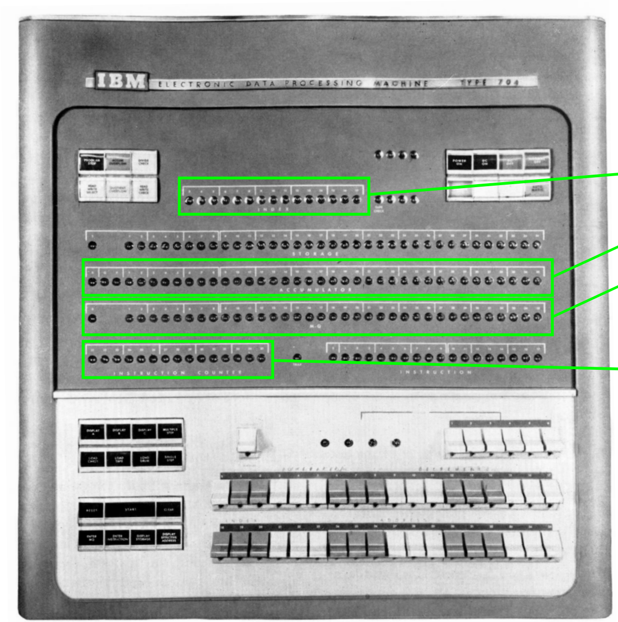
36-bit instructions

Core: 4–32K words

Incubated FORTRAN and LISP

"Mass Produced": IBM sold 125 @ $2M ea.

1954: IBM 704 Processor Architecture



1954: Calling a Subroutine on the IBM 704

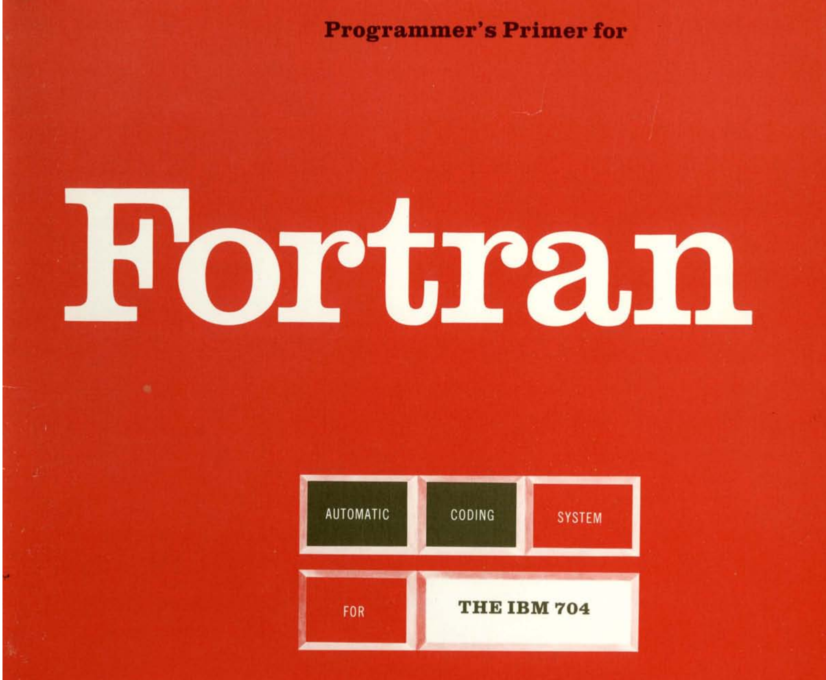
IDENTIFICATION	LOCATION	OPERATION CODE	ADDRESS SYMBOLIC	ADDRESS ABSOLUTE	TAG	DECREMENT SYMBOLIC	DECREMENT ABSOLUTE	COMMENTS
	C. SEQ	SXD	CBOX		C			Save contents of C
		TSX	SINX		C			Transfer to SIN X
								Storage for X
		LXD	CBOX		C			Restore contents of C
	- - - -	- - - -	- - - -	- - - -				
	CBOX							Erasable storage in main program

IDENTIFICATION	LOCATION	OPERATION CODE	ADDRESS SYMBOLIC	ADDRESS ABSOLUTE	TAG	DECREMENT SYMBOLIC	DECREMENT ABSOLUTE	COMMENTS
	SINX	CLA		1	C			Place X in AC
		STO	SINX	+ i				Store X
		SXD			C			Save index C
		- - -	- - - -	- - - -				Computation for sin x
	BOXC							Storage for C in subroutine
		- - -	- - - -	- - - -				
		LXD			C			Restore C
		TRA		2	C			Exit to main program

`TSX   SINX, C`   *Branch to SINX, remember PC in index register C*

`TRA   2, C`   *Return to 2 words past address in index register C*

1954: FORTRAN

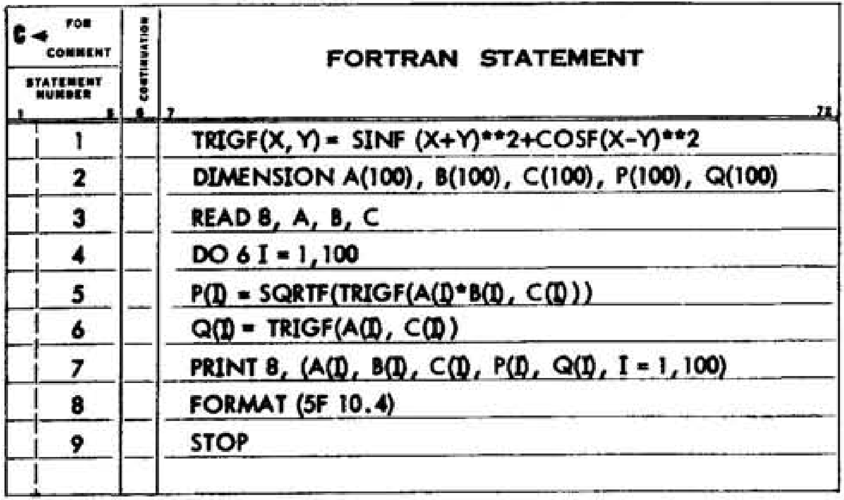


1954: FORTRAN

Since FORTRAN should virtually eliminate coding and debugging, it should be possible to solve problems for less than half the cost that would be required without such a system. Furthermore, since it will be possible to devote nearly all usable machine time to problem solution instead of only half

— J. W. Backus, H. Herrick, and I. Ziller.
Specifications for the the IBM Mathematical FORmula TRANSlating System.
IBM, November 10, 1954.

1957: FORTRAN I on the IBM 705



C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)
1		TRIGF(X,Y) = SINF (X+Y)**2+COSF(X-Y)**2
2		DIMENSION A(100), B(100), C(100), P(100), Q(100)
3		READ 8, A, B, C
4		DO 6 I = 1,100
5		P(I) = SQRTF(TRIGF(A(I)*B(I), C(I)))
6		Q(I) = TRIGF(A(I), C(I))
7		PRINT 8, (A(I), B(I), C(I), P(I), Q(I), I = 1,100)
8		FORMAT (5F 10.4)
9		STOP

[Programmer's Primer for FORTRAN Automatic Coding System for the IBM 704, 1957]

1, 2, 3D arrays

Arithmetic expressions

Integer and floating-point

Loops and conditionals

User-defined functions: expressions only

1957: FORTRAN I User-Defined Functions

C FOR COMMENT / STATEMENT NUMBER	CONTINUATION	FORTRAN STATEMENT
1		FIRSTF(X) = X**2 + A**2
2		SECONDF(R, S) = SQRTF(FIRSTF(R/(R+S)))
		.
		.
		.
15		Q(I) = FIRSTF(Y*B(I))
		.
		.
		.
27		P = SECONDF(1.7*DELTA, ALPHA)*PI

Notice that it is permissible to use a previously defined function in the definition of subsequent functions. Notice also that the variable A is involved in the definition of FIRSTF but is not an argument. A may be used in the same way as any other variable in the problem, and its current value is used each time FIRSTF is evaluated.

Free variables are globals

No recursion; backward references only

No arrays

"Activation Records" allocated statically

1957: EQUIVALENCE Statement for Sharing Storage

GENERAL FORM	EXAMPLES
"EQUIVALENCE (a, b, c, . . .), (d, e, f, . . .), . ." where a, b, c, d, e, f, . . . are variables optionally followed by a single unsigned fixed point constant in parentheses.	EQUIVALENCE (A, B(1), C(5)), (D(17), E(3))

The EQUIVALENCE statement enables the programmer, if he wishes, to control the allocation of data storage in the object program. In particular, it permits him to economise on data storage requirements by causing storage locations to be shared by two or more quantities, when the logic of his program permits. It also permits him, if he wishes, to call the same quantity by several different names, and then ensure that those names are treated as equivalent.

Memory scarce

No stack, functions, or automatic variables

EQUIVALENCE for sharing memory of non-overlapping uses of variables/arrays

A sort of manual "register" allocation

[Programmer's Reference Manual for the FORTRAN Automatic Coding System for the IBM 704 EDPM, 1956]

1958:

FORTRAN II: User-defined Subprograms

Six new statements:

CALL	Call a subroutine
RETURN	Return from function or subroutine
END	End-of-file & compiler directives
SUBROUTINE	Define a subroutine name & arguments
FUNCTION	Define a function name & arguments
COMMON	Like EQUIVALENCE, but between subprograms Also for creating global variables

IBM

Reference Manual

FORTRAN II

for the IBM 704 Data Processing System

C ← FOR COMMENT / STATEMENT NUMBER (1–5)	CONTINUATION (6)	FORTRAN STATEMENT (7–72)

```
      FUNCTION SUM (A, NA, B, NB)
      DIMENSION A(500), B(500)
      SUM=A(1)
      DO 5 J=2, NA
    5 SUM=SUM + A(J)
      DO 10 I=1, NB
   10 SUM=SUM + B(I)
      RETURN
```

```
      DIMENSION X(500), Y(500), V(500), W(500)
      READ 2, NX, NY, NV, NW, X, Y, V, W
      AVERG=(SUM(X,NX,Y,NY) + SUM(V,NV,W,NW))/FLOATF (NX + NY + NV + NW)
      PRINT 10, AVERG
    2 FORMAT (4I8/ (1P5E14.5))
   10 FORMAT (35H AVERAGE OF X, Y, V, AND W LISTS IS 1PE14.5)
      STOP
```

FORTRAN

1957 optimizing compiler far ahead of its time:
register allocation, common subexpression elimination, strength reduction

Static-only storage allocation philosophy ultimately a dead end

No implicit stack or notion of an activation record

Recursion wasn’t standarized until FORTRAN 90

EQUIVALENCE and COMMON were ripe for abuse

1960: ALGOL

Report on the Algorithmic Language ALGOL 60

PETER NAUR (*Editor*)

J. W. BACKUS, C. KATZ, H. RUTISHAUSER, J. H. WEGSTEIN, F. L. BAUER, J. MCCARTHY, K. SAMELSON, A. VAN WIJNGAARDEN, J. GREEN, A. J. PERLIS, B. VAUQUOIS, M. WOODGER

```
procedure Transpose(a)Order:(n)   ;  value n   ;
array a   ;  integer n   ;
begin real w   ;  integer i , k   ;
for i := 1 step 1 until n do
     for k := 1+i step 1 until n do
     begin w := a[i,k]   ;
           a[i,k] := a[k,i]   ;
           a[k,i] := w
     end
end Transpose
```

[CACM 3(5):299–314, May 1960]

Block-structured; simple memory reuse

Nested procedure/function definitions

Call-by-name (substitution) semantics subtle, difficult to implement

Recursion introduced stealthily by Dijkstra et al.
[Naur, SIGPLAN Notices, 13(8), 1978]

1960: Dijkstra Advocates Stacking Activation Records

Numerische Mathematik 2, 312—318 (1960)

Recursive Programming*

By

E. W. DIJKSTRA

The Aim

If every subroutine has its own private fixed working spaces, this has two consequences. In the first place the storage allocations for all the subroutines together will, in general, occupy much more memory space than they ever need *simultaneously*, and the available memory space is therefore used rather uneconomically. Furthermore—and this is a more serious objection—it is then impossible to call in a subroutine while one or more previous activations of the same subroutine have not yet come to an end, without losing the possibility of finishing them off properly later on.

Static links for accessing non-local variables · Displays for efficiency

1961: Side-Effects Complicate Call-By-Name

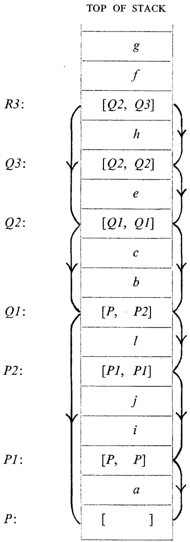
```
begin real B, D; array A[1:10, 2:20];
  procedure P(a, b, c, d); real a, b, c, d;
    a := b := c * d + a + c;

  real procedure C(dd); real dd; begin
     dd := dd + 5;
     C := dd - 3
  end C;

  D := 5; B := 4; A[10, 7] := -20;
  P( A[D, B+3], B, C(D) - 4, D)
end
```

Passing $C(D) - 4$ for c means every reference to c adds 5 to D as a side-effect.
This changes the meaning of a, $A[D, B+3]$ [Jensen and Naur, P. BIT 1(1):38–47, 1961]

Parameters passed as addresses of *thunks*: short address-generating subroutines [Ingerman, CACM, 4(1):55–58, 1961]

ALGOL

```
begin real a;
      procedure Q1;
      begin real b,c;
            . . .
      Q2:  begin real e;
                 procedure R3;
                 begin real f,g;
                       . . .
                 L :   g:=0
                 end R3;
           Q3:   begin real h;
                       . . .
                  M:   R3;
                       . . .
                 end;
                 . . .
           end
      end Q1;
P1:   begin real i,j;
            . . .
      P2:  begin real l;
           N:    Q1;
                 . . .
           end
      end
end
```



Recursive

Stack of activation records

Static and dynamic links for accessing non-local variables

Procedures can be passed as arguments, but not returned

Procedures can only return simple types (real, integer, or Boolean), a syntactic restriction

[Randell and Russell, ALGOL-60 Implementation, 1964]

1962: CPL

The main features of CPL

By D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon and C. Strachey

```
function Euler [function Fct, real Eps; integer Tim]= result of
     §1 dec §1.1 real Mn, Ds, Sum
                 integer i, t
                 index n=0
                 m = Array [real, (0, 15)] §1.1
        i, t, m[0] := 0, 0, Fct[0]
        Sum := m[0]/2
        §1.2 i := i + 1
             Mn := Fct[i]
             for k = step 0, 1, n do
                 m[k], Mn := Mn, (Mn + m[k])/2
             test Mod[Mn] < Mod[m[n]] ∧ n < 15
                 then do Ds, n, m[n+1] := Mn/2, n+1, Mn
                 or do   Ds := Mn
             Sum := Sum + Ds
             t := (Mod[Ds] < Eps) → t + 1, 0 §1.2
        repeat while t < Tim
        result := Sum §1.
```

[The Computer Journal, 6(2), 1963]

- Cambridge and London
- Very ambitious
- Based on ALGOL 60
- Richer types, type checking, and type inference
- Nested function definitions
- Call-by-name plus call-by-value and call-by-reference
- Fixed (side-effect-free) and free procedures

1962: CPL was too complicated

While attempting to write the CPL compiler in a subset of CPL,

> *"We found we did not need to define functions within other functions. This allowed us to represent functions by just their entry points without any additional environment information. This also meant that function calls did not need to implement either Dijkstra displays or Strachey's free variable lists. It also allowed the compiler to be broken into several sections each compiled separately. We only needed called-by-value arguments, since pointers could be used for call-by-reference arguments, and call-by-name could be implemented by passing functions. It is worth noting that the CPL program given in Strachey's GPM paper only used call-by-value and never defined a function within another."*

—Martin Richards, *Christopher Strachey and the Developent of CPL*, 2016

1967: BCPL

BCPL: A tool for compiler writing and system programming

by MARTIN RICHARDS*

University Mathematical Laboratory
Cambridge, England

Exactly one data type: a machine word (24–36 bits)

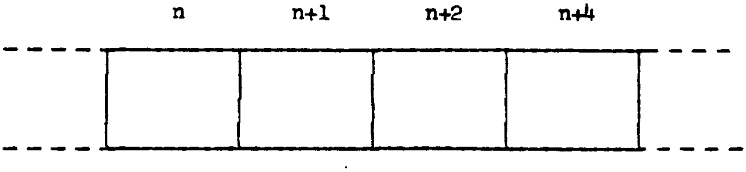


Figure 1—The machine's store

[Spring Joint Computer Conference, 1969]

1967: BCPL Nested Functions, But Free Variables Had To Be Static

BCPL: A tool for compiler writing and system programming

by MARTIN RICHARDS*

University Mathematical Laboratory
Cambridge, England

All functions and routines in BCPL are automatically recursive and so, for instance, one can call a function while an activation of that function is already in existence. In order to allow for recursion and yet maintain very high execution efficiency, the restriction has been imposed that all free variables of both functions and routines must be static. Randell and Russell[7] give a good description of the kind of mechanism normally required for recursive calls in ALGOL; however, with this restriction, a recursive call in BCPL can be very efficient.

```
let Node (x) = valof
      $(  let P = Freelist
          Freelist := P + 3
          P!0, P!1, P!2 := x, 0, 0
          resultis P $)
and Put (x, t) be
      $(  if t!0 = x return
          t := t!0 < x -> t + 1, t + 2
          test rv t = 0
          then rv t := Node (x)
          or   Put (x, rv t)    $)
```

Recursive functions

Function pointers

Static and dynamic (stacked) storage

Nested functions supported, but
free variables must be static

1964: The DEC PDP-7

18-bit word-based

4K to 32K word magnetic core memory

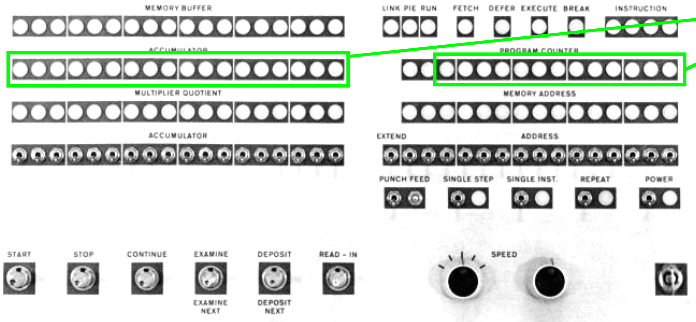
Only $72,000 in 1964

Transistor-based

500 kg, 2000 W

DEC sold 120 of them

1964: DEC PDP-7 Processor Architecture



18-bit Accumulator

13-bit Program Counter

1969: B

> *Thompson was faced with a hardware environment cramped and spartan even for the time: the DEC PDP-7 on which he started in 1968 was a machine with 8K 18-bit words of memory and no software useful to him.*
>
> *Thompson decided that Unix … needed a system programming language. After a rapidly scuttled attempt at Fortran, he created instead a language of his own, which he called B. B can be thought of as C without types; more accurately, it is BCPL squeezed into 8K bytes of memory and filtered through Thompson's brain.*

—Dennis Ritchie, *The Development of the C Language*, SIGPLAN Notices, 28(3) 1993

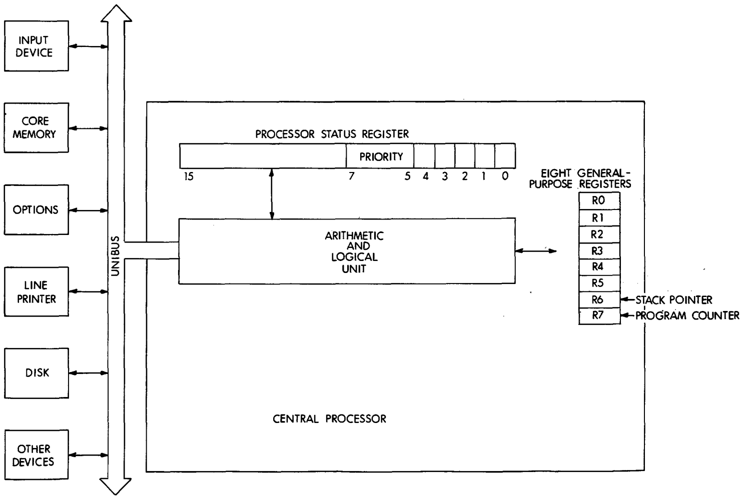
1969: B

```
/* The following program will calculate the constant e-2 to about
   4000 decimal digits, and print it 50 characters to the line in
   groups of 5 characters.  The method is simple output conver-
   sion of the expansion
     1/2! + 1/3! + ... = .111...
   where the bases of the digits are 2, 3, 4, ... */

   main() {
      extrn putchar, n, v;
      auto i, c, col, a;

      i = col = 0;
      while(i<n)
         v[i++] = 1;

      while(col<2*n) {
         a = n+1;
         c = i = 0;
         while(i<n) {
            c =+ v[i]*10;
            v[i++] = c%a;
            c =/ a--;
         }
         putchar(c+'0');
         if(!(++col%5))
            putchar(col%50?' ':'*n');
      }
      putchar('*n*n');
   }

   v[2000];
   n 2000;
```

[Thompson, Users' Reference to B, Bell Labs MM-72-1271-1, 1972]

1970: The DEC PDP-11

1970: DEC PDP-11 Architecture



16-bit architecture

Byte and word operations

8 16-bit general-purpose registers

Floating-point arithmetic

16-bit virtual addresses

Stack support

Many addressing modes, including register+index

1971: C

> *The machines on which we first used BCPL and then B were word-addressed … . The advent of the PDP-11 exposed several inadequacies of B's semantic model. First, its character-handling mechanisms … were clumsy … even silly, on a byte-oriented machine. Second, although the original PDP-11 did not provide for floating-point arithmetic, the manufacturer promised that it would soon be available. …*
> *Finally, the B and BCPL model implied overhead in dealing with pointers: the language rules, by defining a pointer as an index in an array of words, forced pointers to be represented as word indices. Each pointer reference generated a run-time scale conversion from the pointer to the byte address expected by the hardware.*

—Dennis Ritchie, *The Development of the C Language*, SIGPLAN Notices, 28(3) 1993

1971: C and PDP-11 Assembly

```
int gcd(m, n)
{
  int r;
  while ((r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

Frame Pointer: r5
Stack Pointer: r6
Program Counter: r7

```
    .globl _gcd
    .text
_gcd:
    jsr r5, rsave           save SP in FP
L2: mov 4(r5), r1           r1 = n
    sxt r0                  sign extend
    div 6(r5), r0           r0, r1 = m / n
    mov r1, -10(r5)         r = r1 (m % n)
    jeq L3                  if r == 0 goto L3
    mov 6(r5), 4(r5)        m = n
    mov -10(r5), 6(r5)      n = r
    jbr L2
L3: mov 6(r5), r0           r0 = n
    jbr L1
L1: jmp rretrn              return r0 (n)
```

1970: Pascal

Acta Informatica 1, 35–63 (1971)

The Programming Language Pascal

N. Wirth*

Received October 30, 1970

Summary. A programming language called Pascal is described which was developed on the basis of Algol 60. Compared to Algol 60, its range of applicability is considerably increased due to a variety of data structuring facilities. In view of its intended usage both as a convenient basis to teach programming and as an efficient tool to write large programs, emphasis was placed on keeping the number of fundamental concepts reasonably small, on a simple and systematic language structure, and on efficient implementability. A one-pass compiler has been constructed for the CDC 6000 computer family; it is expressed entirely in terms of Pascal itself.

```
procedure Bisect (function f: real; const low, high: real;
    var, zero: real; p: Boolean);
    var a, b, m: real;
begin a := low;  b := high;
    if (f(a) ≧ 0) ∨ (f(b) ≦ 0) then p := false else
    begin p := true;
        while abs(a − b) > eps do
        begin m := (a + b)/2;
            if f(m) > 0 then b := m else a := m
        end;
        zero := a
    end
end
```

Based on ALGOL 60

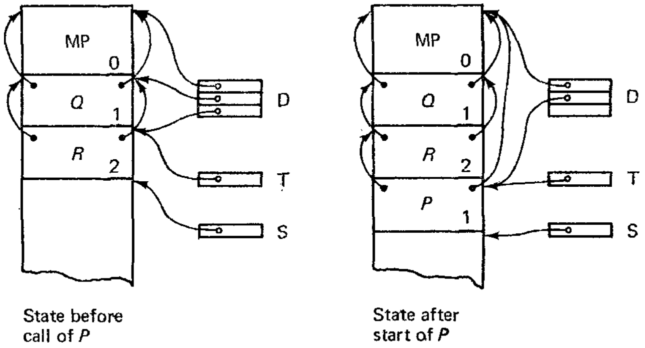
More types: files, records, tagged unions, pointers, sets

Nested procedure/function declarations

Function/procedure arguments, but no variables/types

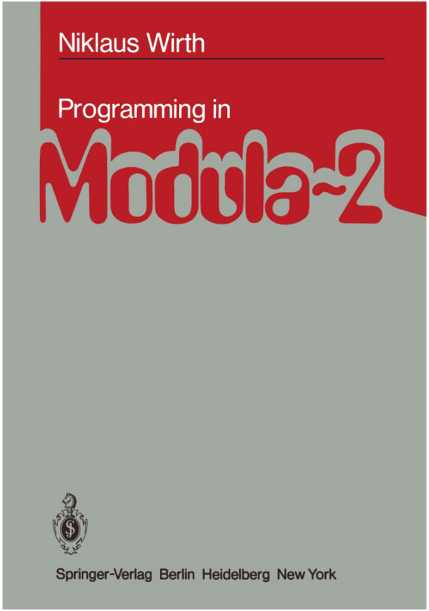
1970: Pascal Nested Procedures, Static Links, and the Display

```
type header = record
                slink, dlink: ↑stack;
                pstatus: address
              end

  procedure P; begin ... end;
  procedure Q;
    procedure R; begin ... P ... end;
  begin ... R ... end;
begin {main program} ... Q ... end;
```



[Wirth, The Design of a PASCAL Compiler, SPE, 1971]

1980: Modula-2



Simplified Pascal for multiprogramming (processes, monitors, signals)

Initially on the PDP-11

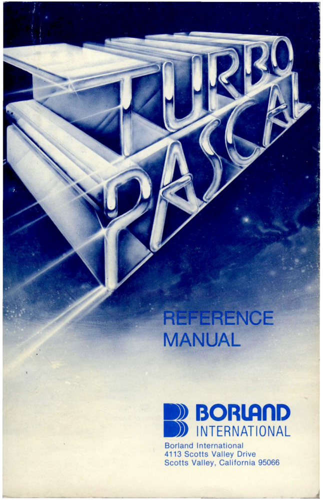
6.8. Procedure types

Variables of a procedure type T may assume as their value a procedure P. The (types of the) formal parameters of P must correspond to those indicated in the formal type list of T. P must not be declared local to another procedure, and neither can it be a standard procedure.

```
$   ProcedureType = PROCEDURE [FormalTypeList].
$   FormalTypeList = "(" [[VAR] FormalType
$        {"," [VAR] FormalType}] ")" [":" qualident].
```

Essentially the rules for C

1983: Turbo Pascal for the IBM PC/Intel 8086



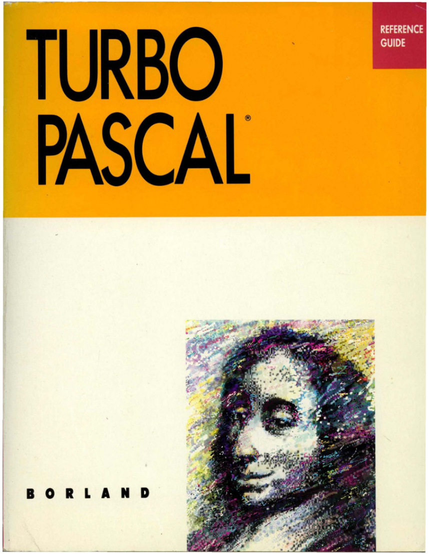
F. TURBO VS. STANDARD PASCAL

The TURBO Pascal language closely follows the Standard Pascal defined by Jensen & Wirth in their **User Manual and Report**, with only minor differencies introduced for the sheer purpose of efficieny. These differencies are described in the following. Notice that the *extensions* offered by TURBO Pascal are not discussed.

F.7 Procedural Parameters

Procedures and functions cannot be passed as parameters.

1989: Turbo Pascal 5.0 Added Procedural Types



Procedural Types

As an extension to Standard Pascal, Turbo Pascal allows procedures and functions to be treated as objects that can be assigned to variables and passed as parameters; *procedural types* make this possible.

```
type
  GotoProc  = procedure(X,Y: integer);
  ProcList  = array[1..10] of GotoProc;
  WindowPtr = ^WindowRec;
  WindowRec = record
                Next: WindowPtr;
                Header: string[31];
                Top,Left,Bottom,Right: integer;
                SetCursor: GotoProc;
              end;
var
  P: ProcList;
  W: WindowPtr;
```

In addition to being of a compatible type, a procedure or function must satisfy the following requirements if it is to be assigned to a procedural variable:

- It must be compiled in the {$*F*+} state.
- It cannot be
  - a standard procedure or function.
  - a nested procedure or function.
  - an **inline** procedure or function.
  - an **interrupt** procedure or function.

Language	Year	Procedures	Recursion	Nested Definitions	Nested References	Function Pointers
FORTRAN I	1957					
FORTRAN II	1958	✓				
ALGOL 60	1960	✓	✓	✓	✓	†
CPL	1962	✓	✓	✓	✓	
BCPL	1967	✓	✓	✓		✓
B	1969	✓	✓			✓
C	1971	✓	✓			✓
Pascal	1970	✓	✓	✓	✓	†
Modula-2	1980	✓	✓	✓	✓	‡
Turbo Pascal	1983	✓	✓	✓	✓	
Turbo Pascal 5.0	1989	✓	✓	✓	✓	‡

† Function arguments only

‡ Pointers to top-level functions only