Appendix to Topic IV

Block-structured procedural languages BCPL and C

References:

- **Chapters 1 to 3** of *BCPL, the language and its compiler* by M. Richards and C. Whitby-Strevens. CUP, 1979.

BCPL

```
LET BCPL BE
  $( LET CPL = "Combined Programming Language"
     WRITEF("Basic %S", CPL) $)
```

- Designed by Martin Richards in 1967 at MIT.
- Originally developed as a *compiler-writing* tool, has also proved useful as a *systems-programming* tool.
- BCPL adopted much of the syntactic richness of CPL; however, in order to achieve the *efficiency* necessary for systems-programming, its scale and complexity is far less than that of CPL.
- BCPL has only *one data type*: the bit-pattern.

BCPL[a]

Philosophy

- **Abstract machine.**

  The most important feature is the *store*: a set of numbered *storage cells* arranged so that the numbers labelling adjacent cells differ by one.

  All storage cells are of the same size and each of them holds a *value* (= bit-pattern). A value is the only kind of object that can be manipulated directly in BCPL, and every variable and expression in that language will always evaluate to one of these.

[a]Notes from Chapter 1 of BCPL, the language and its compiler by M. Richards and C. Whitby-Strevens. CUP, 1979.

Many basic operations on values are provided. One of these, of fundamental importance, is *indirection*. This operation takes one operand with is interpreted as an integer and yields the contents of the storage cell labelled by that integer.

- **Data types.**

  The design of BCPL distinguishes between two classes of data types.

  1. *Conceptual types.* The kind of abstract object the programmer had in mind.
  2. *Internal types.* Basic types for modelling conceptual types.

  Much of the flavour of BCPL is the result of the conscious design decision to provide only one internal type. The most important effects on language design are:

1. There is no need for type declarations in the language.

   This helps to make programs concise and also simplifies problems such as the handling of actual/formal parameter correspondence and separate compilation.

2. It gives the language nearly the same power as one with dynamically varying types (as in LISP), and yet retains the efficiency of a language (like FORTRAN) with manifest types. In languages (such as Algol) where the elements of arrays must all have the same type, one needs some other linguistic device in order to handle dynamically varying data structures.

3. Since there is only one internal type in the language there can be no automatic type checking, and it is possible to

write nonsensical programs which the compiler will translate without complaint.

- **Variables.**

  The purpose of a *declaration* in BCPL is: to introduce a name and specify its scope; to specify its extent; to specify its initial value.

  In BCPL, variables may be divide into two classes:

  1. *Static variables.* The extent of a static variable is the entire execution time of the program. The storage cell is allocated prior to execution and continues to exist until execution is complete.

  2. *Dynamic variables.* A dynamic variable is one whose extent starts when its declaration is executed and continues until

execution leaves the scope of the variable. Dynamic variables are usually necessary when using routines recursively.

- **Recursion.**

  Procedures may be used recursively, and in order to allow for this and yet maintain very high execution efficiency, there is the restriction that the free variables of a procedure must be static.

- **Modularity.**

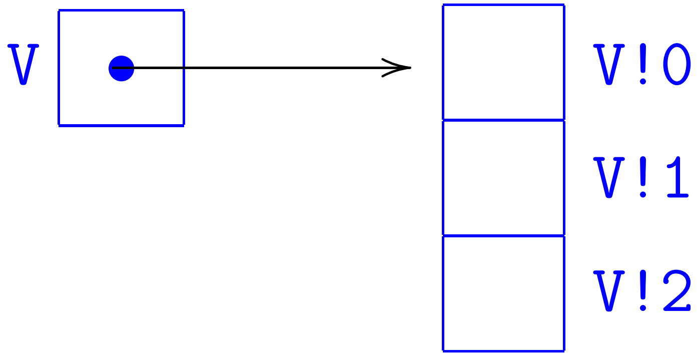
  BCPL uses a form of static storage, called *global vector*, which allows separately compiled modules to reference and call each other and to share data. This facility is not unlike the FORTRAN `COMMON` storage area.

Pointers and arrays

- A *pointer* in BCPL is the address of a word of store.
- The unary operator `@` is used to produce the *address* of a variable.
- The unary operator `!` is used to access the store cell pointed to by an address.
- The *array* declaration

```
LET V = VEC 2
```

establishes: (i) an array of three consecutive locations, and (ii) a separate variable `V` which is initialised to the address of the first location of the array:



$(\texttt{V!E} \stackrel{\text{def}}{=} \texttt{!(V+E)})$

Here `V` behaves like any other local variable, the main difference being that it is initialised by the compiler as a pointer. Hence its value can be copied into another variable (which as a result will also point to the same array), or passed as a parameter to a procedure.

Parameter passing

The BCPL procedure call uses the *call-by-value* technique for parameter passing.

As simple variables are passed by value, a copy is made of the actual parameters for the called procedure to use. Assigning to the formal parameters will not change the values of the original variables specified as actual parameters. This is similar to the Algol call-by-value mechanism, and in contrast to the FORTRAN parameter-passing mechanism.

The *effect* of the parameter-passing mechanism in BCPL is that simple variables are passed by value, and vectors by reference.

Procedures

```
LET COUNT(ARRAY, SIZE) = VALOF
$( LET NUMBER = 0
   FOR I = 0 TO SIZE DO ARRAY!I := 0
   $( LET C = READN()
      IF C < 0 RESULTIS NUMBER
      IF C > SIZE THEN C := SIZE
      ARRAY!C := ARRAY!C + 1
      NUMBER := NUMBER + 1
   $) REPEAT
$)
```

Note that there is no mention that `ARRAY` is an array. It is the programmer's responsibility to make sure that if a parameter is treated as an array inside a procedure, then an array is provided in the procedure call.

Procedures as values

- BCPL has been carefully designed so that it is possible to represent a procedure by a simple BCPL value, called the *procedure value*. The procedure value is placed in a variable bearing the name of the procedure.
- Procedure values can be assigned to ordinary variables.

```
...
    LET CH = GETBYTE(FORMAT, P)
    SWITCHON CH INTO
    $( ...
       CASE ‘S’: F:= WRITES; GOTO L
       CASE ‘C’: F:= WRCH; GOTO L
       ...
    $)
...
L: F(ARG, N)
```

Thus, a procedure may be passed as a parameter to another procedure, or returned as the result of a function call.

Example: I/O streams

```
LET NEXT(S) = (S!0)(S)
LET OUT(S,X) BE (S!1)(S,X)
```

The relevant information concerning a particular stream `S` is stored in an array to which `S` points. The first few items in this array are procedure values. The array takes the following form

```
S ⟶ | NEXT.SOURCE |
    | OUT.SINK    |
    |      ⋮      |
```

The procedure value held in the zeroth element of `S` represents the function which implements `NEXT`, *etc.*

C

- Designed and implemented from 1969 to 1973, as part of the Unix operating system project at Bell Labs.
- C was designed by Dennis Ritchie, as an evolution of Ritchie and Ken Thompson's language B, which was in turn based on BCPL.

  B was a pared-down version of BCPL, designed to run on the small computer used by the Unix project. The main difference between B and C is that B was untyped whereas C has types and type-checking rules.

- An important feature of C has been the tolerance of C compilers to type errors. This is partly because C evolved from typeless languages. As C evolved further and was later standardised by an ANSI committee in the mid-1980s, backward compatibility with the then-existing C code also prevented strong typing restriction. One of the most commonly cited advantages of C++ over C is the fact that C++ provides better type checking.

Summary

- The C programming language is similar to Algol 60, Algol 68, and Pascal in some respects: command-oriented syntax, blocks, local declarations, and recursive functions. However, C also shares some features with its untyped precursor BCPL, such as pointer arithmetic. C is also more restricted than most Algol-based languages in that functions cannot be declared inside nested blocks: All functions are declared outside the main program. This simplifies storage management.