
How BCPL evolved from CPL

MARTIN RICHARDS

*University of Cambridge Computer Laboratory, William Gates Building, JJ Thomson Ave,
Cambridge CB3 0FD, UK
Email: mr@cl.cam.ac.uk*

This paper describes how the programming language BCPL developed from the work on the design and implementation of CPL. It shows that BCPL is essentially just CPL with all the difficult bits removed and with just a few extensions. The CPL project lasted from early 1962 to December 1966 and the first outline of BCPL appeared in 1966. Its first compiler was implemented at MIT in early 1967 and, surprisingly, the language is still used commercially and by individuals all over the world.

Keywords: CPL; BCPL; B; C; PAL

Received 00 December 2011; revised 00 Month 2011

1. INTRODUCTION

CPL (Combined Programming Language) was developed jointly by members of the University Mathematical Laboratory (now the Computer Laboratory), Cambridge and the University of London Computer Unit. The first ideas for CPL appeared in a paper by Strachey and Wilkes[1] in 1961. A paper by Barron et al.[2] two years later gave a summary of its main features. Some of the finer details of CPL can be found in the CPL Working Papers[3]. Both London and Cambridge Universities independently constructed compilers running on different versions of the Ferranti Atlas computer. The Cambridge machine was simplified to reduce its cost and was not usable until early in 1964. Initially it only had a very rudimentary operating system with a simple assembler, no linkage editor and no filing system. Much of the early development of the Cambridge CPL compiler was done by research students using Edsac 2, a beautifully designed but slow machine with limited memory. Research students typically stayed for three years with much of their final years devoted to their own research and producing theses. During this time many of the senior staff were busy developing the multi-user system for the new machine. Progress on the Cambridge compiler was thus slow and did not result in a satisfactory implementation.

It was clear that the compiler needed to be implemented in a way that allowed easy transfer from Edsac 2 to Atlas. The approach taken was to write it in a subset of CPL and hand code it into a sequence of macro calls that could generate assembly language for either Edsac 2 or Atlas. Strachey designed a elegant macrogenerator called GPM[4] for the purpose.

2. GPM

GPM was a text to text converter that basically copied text from input to output except for certain special characters which changed the internal state of the macrogenerator. The open square bracket ([) caused caused output to be diverted to internal memory where macro arguments were stored. Commas (,) were used to separate macro arguments, and close square bracket (]) caused a macro to be called with the current set of arguments. The name of the macro was in argument zero and was looked up in the chain of defined macros. Processing continued from the start of its body. A substitution item of the form #*n* would be replaced by a copy of the *n*th argument. Sometimes, especially when defining macros, it was necessary to disable normal processing of special characters, and this was done using quotation marks (< and >) which could be nested. Finally, there was a comment character ‘ which caused text to be skipped up to the first non white space character on a later line of input. The special characters used in different versions of GPM varied to suit the the characters available and the needs of the application.

There were several built in macros the most important being **def** that allowed macros to be defined. A typical example is as follows:

```
[def,hi,<Hello #1>]
[hi,Sally]
[hi,James]
```

This would add the macro **hi** to the chain of defined macros and call it twice generating the following result:

```
Hello Sally
Hello James
```

GPM included some numeric capability and was found to be remarkably powerful. For instance, one

could define a macro **prime** that would generate the n^{th} prime, so that **[prime,100]** would expand to 541. A version of GPM called **bgpm** is a command in the BCPL distribution[5].

Some of the macros used in the Cambridge CPL compiler are exemplified by considering the implementation of the following CPL definition of the factorial function.

```
let rec fact(n) = n = 0 -> 1, n * fact(n-1)
```

The corresponding macro translation could be as follows.

[Prog,Fact]	The start of a function named Fact
[Link]	Store recursive function linkage information onto the stack
[LRX,1]	Load the first argument of the current function onto the stack
[LoadC,0]	Load the constant zero onto the stack
[Eq]	Replace the top two values on the stack by TRUE if they were equal and FALSE otherwise
[JumpF,2]	Inspect and remove the top value of the stack and jump to label 2 if it was FALSE
[LoadC,1]	Load the constant 1 onto the stack
[End,1,1]	Return from the current function returning one result in place of its single argument
[Label,2]	Set label 2 to this position in the code
[LRX,1]	Load the first argument of the current function onto the stack
[Fn,Fact]	Call the function named Fact
[LRX,1]	Load the first argument of the current function onto the stack
[Mult]	Replace the top two values on the stack by the product of their values
[End,1,1]	Return from the current function returning one result in place of its single argument

To call a function, its arguments are loaded onto the stack in reverse order followed by the subroutine jump, typically **[fn,Fact]**. On entry to the function, linkage information is pushed onto the stack by the **link** macro. After evaluating the body of the function, a return is made using the **end** macro that specified how many arguments to remove and how many results to copy in their place from the top of the stack. More details of this macro language and how it influenced the design of BCPL are given later. But first there is a brief introduction to CPL itself.

3. CPL

The designers of CPL were strongly influenced by ALGOL 60 but wished to extend it to make it more efficient and usable for a wider variety of applications. As the language developed general design principles evolved and helped the language to be logically coherent

and have a minimum of *ad hoc* rules. One of these principles was the separation of the lexical and syntactic details from the semantics. But the use of Strachey's left and right hand values, and the application of λ -calculus to help resolve subtleties concerning the scope of identifiers were probably more significant.

3.1. Left and Right hand values

In Fortran, function arguments are passed by *reference*, that is they are a collection of pointers to the memory locations where the values can be found. In Algol, the programmer can specify that some arguments are to be passed by *value*, the remaining ones being passed by *name*, a somewhat subtle mechanism specified in terms of run time copying of the actual parameters of a call modifying the function body. Strachey coined the terms Lvalue and Rvalue to describe the kind of value passed by reference and value parameters, respectively. These terms were derived from the kind of values needed when evaluating expressions on the left and right hand sides of assignment commands. Lmode evaluation could be performed on simple variables, subscripted expressions and, perhaps more surprisingly, on functions calls.

If an idea is good it should be used wherever it was applicable, and in CPL this was done to extreme. Not only could function parameters be passed by value or reference, but ordinary variables could be defined by value or reference, as in

```
let x, y = 1, 2
let a, b  $\simeq$  x, y
let i = a
```

Here, **x** and **y** are defined by value having two new memory locations initialised to 1 and 2, respectively, while **a** and **b** are declared by reference giving them the same Lvalues as **x** and **y**. In the declaration of **i** the variable **a** has to be evaluated in Rmode and so one level of dereferencing is performed. The data types of declared variables were normally deduced from the types of their initialising values. To cope with recursive definitions the type deduction mechanism was iterative. Initially all variables were given type **unknown** and successively changed by repeated iterations over the program until convergence was reached. In difficult situations some variables may be given the type **general** which can hold a value of any type combined with a type code. Where necessary the programmer could specify types explicitly. Variables could be given the type **general** while **unknown** was only used internally within the compiler.

Another use of L and Rvalues was to distinguish between fixed and free functions. In CPL as in Algol, a function could be defined inside other functions and thus variables used inside a function may refer to variables declared outside it. Such variables are called free variables and require a mechanisms such as Dijkstra displays or static chains down the run

time stack. In CPL, the free variables of a function were collected to form a free variable list. If the function was defined using an equal sign (=) it was called a fixed function and its free variable list contained Rvalues, but, if it was defined using \equiv , it was a free function with a free variable list of Lvalues. A CPL function was represented at run time by a pointer to the first compiled instruction of its body combined with a pointer to its free variable list.

Although it was not necessary and caused immense problems, CPL originally included the ALGOL facility of calling function parameters by name. This was called *call by substitution*. The compiler had to recognise when this mode was in use and pass a parameterless function to evaluate the argument expression. Since local variables were analogous to formal parameters, it was natural to have a third form of declaration. This used the defining operator \equiv as in:

```
let w  $\equiv$  xx + 2xy + yy
```

Every time **w** is evaluated it recomputes **xx + 2xy + yy** using the current values of **x** and **y**. In CPL, **xx** means **x** multiplied by **x**. Multi-character identifiers had to start with capital letters.

3.2. λ -Calculus

Peter Landin wrote a paper[6] in two parts showing how λ -calculus could be used to clarify some of the more subtle semantic details of Algol 60. He was a member of the CPL design team and his use of λ -calculus affected the design of the language. The expression $\lambda x.x + 1$ is a good mathematical representation of the function **f** defined by: **let f(x) = x+1**, and $(\lambda x.x + 1)3$ is a good representation of the meaning of **valof { let x = 3; resultis x+1 }**. This suggests that it is sensible to allow local variables to be declared by value, reference and substitution since these modes are available to function parameters. An effect of this correspondence is that **valof { let f(n) = n=0 -> 1, n*f(n-1); resultis f(5) }** is represented by $(\lambda f.f(5))(\lambda n.n = 0 \rightarrow 1, n \times f(n - 1))$, indicating that **f** is not a recursive function but requires a previous definition of **f** used by the inner call **f(n-1)**. This explains the necessity of **rec** as in **let rec f(n) = n=0 -> 1, n*f(n-1)** when defining recursive functions. The λ -calculus equivalent of this recursive definition involves the combinator **Y** and is outside the scope of this paper.

Another example concerns CPL's **in** declarations which provided an alternative to the rather unsatisfactory Algol **own** variable mechanism. The declaration **let x = 0 in f() = valof { x:=x+1; resultis x }** defines **f** to be $(\lambda x.(\lambda().(x := x + 1; x)))(0)$. This effectively means that the **x** is initialised to zero and has a scope limited to the body of **f**. So, **f** increments **x** on every call, but programs calling **f** have no direct access to **x**.

Another effect of λ -calculus on CPL was the inclusion

of call by substitution (or by name in Algol) since this is closely resembles how λ -expressions are evaluated. It unfortunately was expensive to implement and caused semantic problems in languages that had imperative constructs such as assignments and goto commands. By mid 1966 call by substitution and the corresponding form of variable definition were removed.

4. THE EVOLUTION OF BCPL

As we have seen the CPL compiler was written in an informal subset of itself and hand coded into a sequence of GPM macro calls being essentially the assembly language of an abstract machine. The values on the stack were all the same size and were used to represent quantities of any conceptual type such as a character, a truth value, an integer, a pointer or even a function, just as the bit pattern held in a CPU general register can represent a value of any type. Such values could be thought of as being of type **general** but without the associated type information.

The call **[LRX,1]** would load the first argument of a function whatever its conceptual type. We did not need a separate version of **LRX** for each conceptual type, just as the machine instruction to move a 32-bit quantity into a general register does not care what conceptual type the value has. The Lvalues of CPL were implemented by pointers. For instance, the call **[LLX,1]** would load a pointer to the first argument of the current function. This pointer could be thought of as the address of where the first argument was stored, but could also be thought of as an integer. Since all variables were of the same size, it was natural to address them by consecutive integers just as on Edsac 2 or the IBM 7094. Adding one to a pointer thus creates a pointer to the next location of memory. The call **[RV]** replaced the top stack item by the contents of the memory it pointed to, and so, supposing **v** and **i** were the first two arguments of a function, the value of **v[i]** could be computed by the calls: **[LRX,1]** **[LRX,2]** **[PLUS]** **[RV]**.

In a period of about a year between 1965 and 1966 this GPM based abstract machine was fully developed and used to implement a significant proportion of CPL producing compilers that ran on both Edsac 2 and the Atlas. After moving to MIT in December 1966, I filled in the syntactic details of the informal subset of CPL we had used, treating the abstract machine as though it were the operational semantics of the language. Where possible the original CPL syntax was used. The following sections explore the features of CPL that were omitted and why, and what else was added. But first there is the thorny question of character sets to consider.

4.1. Character sets

It was not until July 1967 that a specification[7] of a character set was published that closely resembles the

ASCII character set we have today. Prior to that, computers typically had their own character sets and these were often quite limited. For instance, on the first machine on which BCPL ran, namely an IBM 7094 running CTSS, the standard code used 6-bit characters packed in 36-bit words. The characters available were essentially those used in Fortran and did not include square or curly brackets ([] { }), semicolon (;), double quote (") or underscore (_), and commonly used terminals such as the Model 35 Teletype only permitted letters in upper case. However, at Cambridge, CPL programs used a much richer character set since they were typically prepared using a Flexowriter which was an electric typewriter that was combined with a 7-track paper tape reader and punch. The available characters included backspace and so overprinting was possible and used to represent symbols such as \neq . System words such as **while** were underlined to distinguish them from ordinary identifiers. BCPL on the 7094 thus had to represent lexical tokens quite differently.

In CPL an identifier was either a lower case letter or an uppercase letter followed by letters, digits and dots. Both sorts of names could be optionally followed by primes ('). Identifiers in BCPL followed the ALGOL convention but used dots rather than spaces to separate words within identifiers. Ideally underline (.) would have been chosen had it been available. Some identifiers such as **while** and **for** were reserved and so could not be used as variable names.

Since square brackets were not available, open square bracket was represented by `* (` and the matching close square bracket was represented by a close parenthesis without the star. The star in `* (` was chosen because it was available, multiplication was a rarely used, and it seemed appropriate for subscription since it was used as the indirection operator in the FAP assembly language used on CTSS. Note that star is still used today as the indirection operator in C. Later, when the exclamation mark became available, `* (` was replaced by `! (` and exclamation mark became both a dyadic and monadic indirection operator. Exclamation mark was chosen because it was an approximation to a down arrow which was a natural choice for subscription in expressions like x_i . Later a byte subscription operator (%) was added eliminating the need for the library functions `getbyte(s,i)` and `putbyte(s,i,x)`.

Before double quotes (") became available, single quotes were used for character strings, except a string of length one was a character constant represented by the appropriate character code. Escape sequences in strings such as `*n` and `*t` used stars as in CPL.

Integer constants were sequences of decimal digits, or represented in binary, octal or hex by preceding the appropriate sequence of digits by `#b`, `#o` or `#x`. More recently, BCPL has allowed underlines in numbers as in `100_000_000` as a more readable way of representing a hundred million than `100000000`, or `#b00101_0_000111101`, the first instruction (T123S) of

the program[8] to run on the EDSAC computer in May 1949.

CPL used a section symbol (§) to be equivalent to Algol's **BEGIN** and a section symbol overprinted with a slash to represent **END**. BCPL adopted `$(` and `)$` for these tokens and as with CPL such section brackets could be tagged, allowing a close section bracket to close multiple sections. Unfortunately this convention led to rather obscure programming errors and so when curly brackets (`{` and `}`) became available, they were only used as untagged section brackets and the use of `$(` and `)$` was discouraged.

The remaining lexical tokens were straightforward using reserved words whenever suitable characters were not available.

5. THE BCPL ABSTRACT MACHINE

The BCPL abstract machine was almost the same abstract machine of macro calls used in the implementation of CPL, but since its statements were to be generated by the front end of the compiler and read by a codegenerator there was no need for it to consist of textual macro calls. The code for BCPL's abstract machine was called OCODE and is logically just a sequence of integers, although, for presentation, it is often written in a slightly more mnemonic form. See the `procode` command in the standard BCPL Cintcode distribution[5]. The use of OCODE rather than GPM macros reduced compile time and allowed better optimisation of the target code.

Block structured languages like Algol and CPL use a stack to hold function arguments and local variables that have transient lifetimes. In the abstract machine of macro calls used in the CPL compiler, there was a pointer, called the P pointer, that pointed to the return link information of the current function, and arguments and local variables were accessed relative to this pointer. In OCODE, arguments and local variables are indistinguishable both being stored on the same side of the P pointer above the return link which normally required three words. So the first argument or local variable is at position 3 relative to P and can be loaded and updated by the OCODE statements `LP 3` and `SP 3`.

As an example of how BCPL evolved, consider the following fragment of CPL:

```
let x = 10
let a  $\simeq$  x
let i = a
```

If this were translated into OCODE, it would be as follows assuming that x was allocated position 3 relative to P.

```

LN 10  Load the integer 10 as the initial value of
        variable x
LLP 3   Load the Lvalue of x as the initial value of a
LP 4    Load a, an Lvalue
RV      Load the contents of what a is pointing at to
        give the initial value of variable i

```

The BCPL code to generate this sequence could be:

```

LET x = 10 // Initialise x to be 10
LET a = @x // Declare a to be a pointer to
           // variable x
LET i = !a // Declare i to be the contents
           // of memory pointed to by a

```

There is considerable similarity between the two languages, partly because the programmer rarely had to specify CPL types explicitly. The at sign (@) is the modern replacement of LV to compute the Lvalue of a variable or subscripted expression. It was chosen since it looks like an *a* inside an *O* which could be read as *address of*.

Values of all CPL types were first class citizens in the sense that they could be passed as arguments to functions or assigned to suitably declared variables. Since all BCPL values and variables were of the same size there was no problem in letting BCPL do the same. The only constraint was that every kind of value had to be representable by a single word. For functions, this meant that there was no room for a free variable list pointer in addition to the entry address. This forced all free variables to be in locations known at compile time or be manifest constants, and so called dynamic (argument and local) free variables were disallowed. The same constraint applies to C functions but is enforced by the more satisfactory scheme of disallowing functions to be defined inside other functions. Although disallowing dynamic free variables in BCPL annoyed Algol and Pascal programmers, it greatly eased the problem of linking libraries of separately compiled code.

Labels and **goto** commands in CPL were problematic. The scope of a label was the current routine or **valof** block in which it occurred, and a **goto** statement could cause execution to jump out of the scope of some local variables and even out of one or more functions. It was even possible to jump into a block initialising local variables on the way. These features clearly got into CPL before Dijkstra's letter[9] in the CACM pointed out the harmful nature of **goto** statements. BCPL still allows labels to be passed as parameters and assigned to variables, but, since labels are just addresses in the program, **GOTO** in BCPL cannot jump out of functions even though it can jump out of the scope of local variables. Non local jumps in BCPL use the library functions **level** and **longjump**.

A similar restriction applied to the **RESULTIS** command which required the enclosing **VALOF** construct to be within the same function or routine. In CPL,

result is and even **break** could cause execution to leave one or more functions.

BCPL followed CPL's lead in syntactically separating expressions from commands. The primary purpose of an expression is to produce a result, while commands are written for the effect (particularly assignments) they have on the environment. BCPL provided all the CPL expression operators except exponentiation and string concatenation, but limited the arithmetic operators to apply only to integers. Although CPL used square brackets for both function arguments and array subscripts, BCPL had to make these two forms syntactically distinct, using parentheses for calls and square brackets for subscription. BCPL included CPL's simultaneous assignments but allowed the order of evaluation to be implementation dependent. So an assignment like: **p,v!p:=p+1,p** should therefore be written as two assignments to clarify its meaning. The rich collection of conditional and looping commands were all included in BCPL although some (particularly the **FOR** command) were simplified. So, **IF**, **TEST**, **WHILE**, **UNTIL**, **REPEAT**, **REPEATWHILE**, and **REPEATUNTIL** are all present. BCPL followed CPL's lead in having rules that almost eliminated the need for semicolons to separate statements.

6. STRUCTURES

Although various schemes for providing structures (or records) in CPL were discussed in design meetings, no version was deemed adequate for inclusion in the language before I left the project. However, structures of some kind were needed to represent data such as parse trees and hash tables within the compiler and so the CPL subset used to write the compiler did allow the creation of structures that were essentially dynamically created vectors using subscription to access the fields. A parse tree node representing the product of two expressions could be created by the call: **mk3(Mult,t,x,y)** and if the result was placed in **t**, its fields could be accessed by expressions **t[0]**, **t[1]** or **t[2]**. Since the abstract machine was typeless, there was no problem in allowing **x** to represent any kind of expression such as a name, a monadic or dyadic node. BCPL used exactly this mechanism for its structures. Normally, names declared with compile time constant values were used as a more readable way to identify the fields. Strachey coined the word manifest for such names and that is why this word was used in BCPL's **MANIFEST** declarations. In many ways these are like **enum** declarations in C and have recently been extended to allow the compiler to allocate distinct values automatically to the names declared.

By convention, the first field of each parse tree node contained an integer (the operator) identifying the kind of node and it was be able to select code to execute depending on which operator was found. This naturally lead to the inclusion of the **SWITCHON**

command in BCPL which was retro-fitted into CPL in 1966. `SWITCHON` is a frequently used command, it occurs 28 times in the BCPL to Cintcode compiler and is executed 182591 times when the compiler compiles itself.

Recently, the specification of simultaneous definitions such as `LET days,msecs,flag=0,0,0` was tightened to force the variables to be in consecutive locations in memory. This allowed a pointer to these three variables to be passed to a function as in `dat_stamp(@days)` which would set `days` to the number of days since 1 January 1970 and `msecs` to the number of milli-seconds since midnight.

7. DECLARATIONS

BCPL includes CPL's simple variable declarations by value. The reference form being easily provided with the aid of the address of operator (`LV`, and later `@`). Simultaneous declarations and declarations connected by `AND` were allowed.

BCPL permitted both function and routine declarations just as in CPL but insisted that they contained no dynamic free variables. They were automatically recursive without the need for `REC`, and their arguments were always called by value. The effect of call by reference can be achieved by passing pointers, and call by substitution by passing parameterless functions.

One dimensional arrays, often called vectors, could be allocated by calls of `getvec` just as `malloc` is used in C. For local vectors, declarations such as: `LET v = VEC 10` could be used. This example would declare a variable `v` initialised with a pointer to a vector on the stack with subscripts ranging from 0 to 10. This vector would cease to exist when control leaves the scope of `v`.

A **where** declaration in CPL qualified the largest declaration, command or expression to its left. This rule caused syntactic problems and was often misunderstood by users and so was not included in BCPL. The construct using `in` was also not included.

8. SEPARATE COMPILATION

Although Fortran had allowed modules of a program to be compiled separately and brought together for execution, most block structured language like Algol, CPL, Algol W and Pascal back then typically provided no such mechanism, making precompiled libraries inconvenient or even impossible. For BCPL, separate compilation was deemed essential, so some mechanism had to be found. At the time object module formats and linkage editors were in their infancy and so a scheme was devised specifically for BCPL. It used an area of memory called the global vector similar to `BLANK COMMON` of Fortran to allow separately compiled modules to share variables. The `GLOBAL` declaration allowed the programmer to name elements of the global

vector, and there was a rule that global variables would be initialised to the entry points of functions if their names matched the names of declared global variables. This initialisation took place when modules were loaded. Although this mechanism is not beautiful and much hated by some, it is simple and has served BCPL well for the last 45 years.

Global variables are statically allocated and so could be accessed within functions. Although not directly connected with separate compilation, `STATIC` declarations were added to allow variables to be shared between functions defined within the same module.

9. INFLUENCES

As has been shown BCPL has essentially much of the same syntax as CPL but with many semantic restrictions to allow a simpler implementation. Machine independent pointer arithmetic and indirection were added to allow Lvalues, vectors and structures to be handled. Function arguments were called by value and laid out in consecutive locations making it was easy to define functions with variable numbers of arguments. This allowed functions like `writeln` (similar to C's `printf`) to be implemented without special privilege.

BCPL and hence CPL influenced several other languages. For instance, Thompson designed a cut down variant of BCPL he called B[10] which was later extended by Ritchie[11] resulting in C, made popular by its use in the implementation of Unix. Another development was the design of PAL[12] which was a language designed for use at MIT in a course to teach programming linguistics. It was a version of sugared λ -calculus strongly influenced by Landin's work on ISWIM[13] and implemented in BCPL using his SECD machine[14]. It had dynamic types (like CPL's type `general`), a garbage collector and syntax related to BCPL. It had updatable variables, and labels based on Landin's program points which allowed the intriguing possibility of jumping back into a function that had previously been left. A version of PAL is available in the standard BCPL distribution[5].

Although not directly related to BCPL, ML[15], like PAL, is a sugared λ -calculus related to Landin's work that was first implemented in 1974. It is a functional language extended to include mutable objects, but, unlike PAL, it has a type system that allows the types of all values to be inferred statically given little or no explicit type information in the program, as was true of CPL. Unlike CPL, its type system allows structures such as trees to be represented and manipulated conveniently. The design is a *tour de force* for which Milner won the British Computer Society Award for Technical Excellence in 1987.

10. CURRENT STATUS OF BCPL

Although CPL did not survive for long, BCPL was used extensively at places like MIT, Xerox PARC and

Cambridge, and is still in use both commercially and by individuals. An implementation based on a compact interpretive code (Cintcode) is freely available[5] together with its manual[16]. Coroutines[17] were a significant addition to the BCPL library in 1977 and were used extensively in the Tripos[18] Portable Operating System, a version of which is currently available as Cintpos[19].

- [19] Richards, M. (2011) *The Cintpos Distribution*. www.cl.cam.ac.uk/users/mr/Cintpos/cintpos.{tgz,zip}.

REFERENCES

- [1] Strachey, C. and Wilkes, M. (1961) Some Proposals for Improving the Efficiency of ALGOL 60. *Comm.A.C.M.*, **4**, 448.
- [2] Barron, D., Buxton, J., Hartley, D., Nixon, E., and Strachey, C. (1963) The main features of CPL. *The Computer Journal*, **6**, 134–143.
- [3] Editor: C. Strachey (1966) CPL Working Papers, V53-57. Technical report. Computer Laboratory, Cambridge University.
- [4] Strachey, C. (1965) A General Purpose Macrogenerator. *The Computer Journal*, **8**, 225–241.
- [5] Richards, M. (2011) *The BCPL Cintcode Distribution*. www.cl.cam.ac.uk/users/mr/BCPL/bcpl.{tgz,zip}.
- [6] Landin, P. (1965) Correspondence between ALGOL 60 and Church's Lambda-notation, Parts 1 and 2. *Comm. ACM*, **8**.
- [7] USA Standards Code for Information Interchange (1967) *USAS X3.4-1967, revision of X3.4-1965*. United States of America Standards Institute, USA.
- [8] Richards, M. (2009) *EDSAC Initial Orders and Squares Program*. www.cl.cam.ac.uk/users/mr/edsacposter.html.
- [9] Dijkstra, E. (1968) Letter to the Editor: Go To Statements Considered Harmful. *Comm. ACM*, **11**, 147–148.
- [10] Johnson, S. and Kernighan, B. (1997) *The Programming Language B*. <http://cm.bell-labs.com/who/dmr/bintro.html>.
- [11] Ritchie, D. (1997) *The Development of C*. <http://cm.bell-labs.com/who/dmr/chist.html>.
- [12] Art Evans, Jr (1968) PAL - A language designed for teaching programming linguistics. *Proceedings of 1968 23 ACM Conference, pp 395-403 August 1968*. Thompson Book Company, Washington, DC.
- [13] Landin, P. (1966) The next 700 programming languages. *Comm. ACM*, **9**, 157–166.
- [14] Landin, P. (1964) A mechanical evaluator of expressions. *Computer Journal*, **6**, 308–320.
- [15] Milner, R., Tofte, M., and Harper, R. (1990) *The Definition of Standard ML*. MIT Press, Cambridge, MA.
- [16] Richards, M. (2011) *The BCPL Programming Manual*. www.cl.cam.ac.uk/users/mr/bcplman.pdf.
- [17] Moody, K. and Richards, M. (1980) *A Coroutine Mechanism for BCPL. Software-Practice and Experience*, **10**, 765–771.
- [18] Richards, M., Aylward, A., Bond, P., Evans, R., and Knight, B. (1979) The Tripos Portable Operating System for Minicomputers. *Software-Practice and Experience*, **9**, 513–527.