

# Christopher Strachey and the Development of CPL

*by*

Martin Richards

`mr@cl.cam.ac.uk`

`www.cl.cam.ac.uk/users/mr10`

Computer Laboratory

University of Cambridge

Thu Jun 29 17:28:32 BST 2017

## Abstract

Chrisopher Strachey was the most significant contributor to the design and implementation of the programming language CPL. At the time there was little understanding of the complexities of computer language design and how type systems could cope with lists and the kinds of structures needed to represent, for instance, parse trees. The CPL project cannot be regarded as being successful since it did not result in a usable CPL compiler. The reasons being that the language became too large and complicated, there were insufficient people to implement the compiler and, in the middle of the three year project, all work had to be transferred from Edsac 2 to Titan, a newly designed version of the Ferranti Atlas computer which as yet had no operating system. Even so, we can be proud of the work that went into CPL and its influence on the design of many later languages.

**Keywords:** Edsac, Titan, CPL, BCPL

## 1 Introduction

Before discussing the development of CPL it is necessary to have some understanding of what computer facilities were available at Cambridge at the time.

The first electronic computer in Cambridge was constructed under the Direction of Maurice Wilkes and successfully ran its first program in May 1949. It was a serial machine with up to 512 35-bit words of memory stored in mercury delay lines. Each word could contain two 17-bit instructions. Wilkes made two important decisions that contributed to the machine's success. Firstly, he chose to use a fairly slow clock rate of 500kHz arguing that reliability and ease of design were more important than speed since the machine was already going to be able to do calculations thousands of times faster than was possible by hand. Secondly, he made the machine available to anyone in the University who could make use of it. This was enthusiastically taken up by many departments including the radio astronomers, the theoretical chemists, and mathematicians, directly contributing to at least two Nobel Prizes.

One advantage of the serial design was that it allowed high precision arithmetic to be implemented using rather few logic gates. Although the instruction set was quite primitive with no

subroutine jump instructions and no index registers, it did include an instruction to add the product of two 35-bit numbers to a 71-bit accumulator. Indeed, it was a good machine for serious numerical calculation. For more information look at the poster[12] which is on the wall of the Cambridge Computer Laboratory's first floor Coffee Room describing how the incredibly cunning EDSAC initial orders worked, written by David Wheeler. In a mere 31 instructions, it could load a program from paper tape written in assembly language and execute it.

In 1958 the EDSAC was replaced by Edsac 2. This was a bit sliced computer controlled by a micro program consisting of a matrix 32x32 ferrite cores. Each core corresponded to a different micro instruction. When a core was fired, windings on it would give the  $(x, y)$  address of the next micro instruction to obey and other windings would send signals to control the machine. Some micro instructions used signals from the machine, such as the sign of the accumulator, to modify the address of the next core to fire. The micro program consisted of up to 1024 instructions many of which were conditional. David Wheeler wrote the micro program and as with the Initial Orders of the EDSAC, it allowed paper tapes of assembly language to be loaded and executed. But unlike the EDSAC, the instruction set was much more powerful and convenient to use and included some rather wonderful debugging aids. For instance, by pressing a switch on the control panel, the machine would output a paper tape giving a compact representation of the recent flow of execution of the program, typically in the form of recently executed nested loops. Another intriguing feature was that the fifth bit of the effective address of every instruction was fed to a loudspeaker normally producing a rich sound that programmers soon learnt to interpret. For users not using assembly language the preferred language was Edsac Autocode, designed and implemented by David Hartley.

Initially, Edsac 2 had a memory of 2048 20-bit words. These were used to represent machine instructions, but some instructions used pairs of these words to represent 40-bit integers or floating point numbers. The machine also had a read only memory filled with many useful subroutines, typically entered by the instruction with function code 59. The main memory was extended in 1962 by an additional 16K words purchased from Ampex. Access to this memory was less convenient since it required the use of a new base register.

Around 1961, it was clear that the University needed a more powerful computer, and it was decided to install a version of the Ferranti Atlas, but, to save money, a cut down version was designed. Following the success of the previous two computers, it is perhaps not surprising that modifications to the machine were designed and built in Cambridge. Some users would probably have preferred a more conventional machine such as the IBM 7094 allowing greater compatibility with other universities, particularly in America.

David Wheeler had a major hand in the modified hardware design. Since the machine was going to allow simultaneous access for many users, it required hardware memory protection, but rather than the full blown paging mechanism of the full Atlas, David designed a much simpler system using a base address and limit register. For efficiency reasons, the base address was ORed into the effective address rather than being added. This meant that with a main memory of 32K words, small programs of say 4K could be placed in 8 positions in memory but programs larger than 16K only had one choice. To make time sharing responsive, small programs were strongly encouraged. This cut down machine was called Titan but was not ready to use until 1964 and even then it needed an operating system, an assembler and at least one high level programming language. With the limited resources available, an assembler was constructed on time but the operating system was clearly going to be late, and the decision to design and implement CPL which was substantially more complicated than ALGOL 60 was extraordinarily optimistic. At a late stage, Peter Swinnerton-Dyer was called in and miraculously implemented, in about six weeks,

a simple operating system including an assembler, and a version of Edsac Autocode. Edsac 2 was switched off on 1 November 1965.

## 2 Development of CPL

The initial plan was to design a new language related to ALGOL 60, taking advantage of its many good features such as its block structure, but removing or modifying features that caused its implementation to be difficult or inefficient. A paper by Strachey and Wilkes[21] proposed some possible improvements. One idea was to allow programmers to state when functions were non recursive and another was a scheme that attempted to stop functions having side effects to allow compilers more freedom to optimise the order of evaluation within expressions. They also felt that a simpler form of ALGOL 60's call-by-name would be useful. They called this call-by-simple-name being equivalent to call-by-reference we use today. These ideas found their way into the design of CPL. The work on CPL ran from about 1962 to 1966. The first outline of CPL was in the paper "The main features of CPL"[2] published in 1963 and a later document "CPL Papers"[3] was written in 1966.

I was involved with CPL when I was a research student for three years starting in October 1963. By then CPL changed from being Cambridge Programming Language to Combined Programming Language since the project was now a collaboration with the University of London, Institute of Computer Science. The committee involved in the design of CPL was headed by Strachey and included his assistant Peter Landin, David Hartley, David Park, David Barron, from Cambridge and, from London, John Buxton, Eric Nixon and George Coulouris who implemented a satisfactory subset of CPL on the London Atlas computer. As a junior research student involved with the Cambridge implementation, I was lucky enough to attend most of these meetings. They usually took place either in Cambridge or in Strachey's house in Bedford Gardens, London. The discussions were lively and animated trying to reach compromises between the conflicting desires of the participants. Some wanted a firm underlying mathematical structure to the design, others only wanted features that were useful, and those involved with the implementation favoured a design that could be compiled efficiently. In general, the kind of discussions were like those fictitious language design discussions appearing in chapter 8 of "System Programming with Modula-3" edited by Greg Nelson[10]. During breaks in the discussions, Christopher and Peter would sometimes entertain us with piano duets.

Clearly the designers of ALGOL 60 were familiar with Church's  $\lambda$ -calculus since the ALGOL 60 Report describes procedure calling and call-by-name parameter passing in term of textual replacement of procedure calls by suitably modified copies of the procedure bodies, in more or less the same way that  $\lambda$ -calculus used  $\alpha$ -conversion and  $\beta$ -reduction. This was clarified in Peter Landin's paper "Correspondence between ALGOL 60 and Church's Lambda-notation"[7]. With such a close relationship between ALGOL 60 and  $\lambda$ -calculus, it was natural to allow procedures to be defined within other procedures.

CPL was a natural extension of ALGOL 60 with a wider variety of numerical values, including integers, reals and complex numbers both in single and double precision as well as bit pattern values called logicals. It also contained a wide variety of useful and easily implemented conditional constructs using keywords such as **if**, **unless**, **while**, **until** and **repeat**. Strachey was always full of great ideas for other extensions, not all of which ended up in CPL. He was keen on the idea of L and R values and L and R mode evaluation. These corresponded to the different ways in which expressions on the left and right hand side of assignment statements were processed. He also

liked the idea of referential transparency which essentially means an expression can be replaced by any other expression that evaluates to the same result. This led to the belief that conditional expressions and even function calls should be allowed on the left hand side of assignments and in call-by-reference parameters. Strachey thought it important to make a distinction between functions that produced results and routines whose sole purpose was to have side effects. He liked the idea that some functions could be defined in such a way that they could not have side effects. He called them fixed functions and had a way of implementing them. Rather than use the common choice of dynamic and static chains through the runtime stack, he proposed the construction of a list of free variable values at the time a function was defined. For fixed functions, the values were R values and assignments to them would not cause side effects. Free functions on the other hand would use L values, allowing them to have side effects. His liking for L values extended to the idea of creating a general form consisting of a load-update pair (LUP), allowing a user to, for instance, construct the L value of a field of bits within a logical variable.

In addition to call-by-value and call-by-reference, CPL had ALGOL 60 style call-by-name using the keyword **subst**, but this was soon dropped since passing a function as an argument was a simpler way to provide the same facility.

One notable feature of CPL was its ability to deduce the data types of all expressions and variables with little explicit help from the programmer. Partly because we had insufficient understanding of type systems, we needed a type **general** which carried the actual type of values around at runtime. The much more recent language ML[9] has a superb polymorphic type system with the ability to statically deduce the types of all expressions and variables. The design of ML was a tour de force and was cited as one of the reasons why Robin Milner received the Turing Award in 1991. Although ML can deduce the type of every expression in an ML program, this is only just true because the process can take exponential time. For instance, the type of **f** in the following short program is many millions of characters long.

```
fun a x y = y x x;
fun b x   = a(a x);
fun c x   = b(b x);
fun d x   = c(c x);
fun e x   = d(d x);
fun f x   = e(e x);
```

### 3 GPM and the CPL Compiler

In order to write the CPL compiler in a form that could be easily transferred from Edsac2 to Titan, it was decided to implement the compiler in essentially a subset of CPL and hand translate it into a sequence of macro calls using the GPM macrogenerator that Strachey specifically designed for the purpose. GPM was described in detail in a paper in the Computer Journal[18]. Since GPM had such a significant influence on how the compiler was constructed it is necessary to learn a little about how GPM worked.

Unfortunately Strachey's paper contains a subtle bug as a result of his attempt to implement GPM using a single stack. I therefore present a re-implementation of GPM called BGPM that uses two stacks. BGPM is implemented in BCPL (not in CPL) using only ASCII characters rather than those available on the flexowriters used at Cambridge. Its behaviour is almost identical to GPM. In BGPM a typical macro call is: `[aaa,bbb,ccc]`. The character '[' marks the start of a macro call,

' , ' separates the macro arguments and ']' indicates that a macro call is complete and should be expanded. BGPM's left hand stack holds incomplete macro calls while their arguments are being formed. On encountering ']' the latest macro call is now complete and is moved to the right hand stack. This stack contains complete macro calls and also the environment chain of defined macros. The arguments of a macro are numbered from zero upwards, and when a complete call is moved to the right hand stack its zeroth argument is looked up in the environment causing input to be temporarily changed to the start of the body of the matching macro definition. An error occurs if the macro is undefined.

While executing the body of a macro, a substitution item of the form **#*n*** is automatically replaced by a copy of the *n*<sup>th</sup> argument. Sometimes, especially when defining macros, it was necessary to disable normal processing of special characters, and this was done by enclosing the text in curly brackets ({ and }) which can 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.

A new macro definition can be inserted at the front of the environment using the predefined macro **def** as in the following demonstration.

```
[def,hi,{Hello #1}]'
[hi,Sally]
[hi,James]
```

This would add the macro **hi** with body **Hello #1** to the chain of defined macros and call it twice, generating the following result:

```
Hello Sally
Hello James
```

Another built in macro, **set**, allows the body of a macro to be updated, essentially making it behave like a variable, and with the aid of the **eval** macro, it is possible to perform integer arithmetic, as in:

```
[def,counter,0000]'
[def,inc,{[set,#1,[eval,[#1]+1]]}]'
[inc,counter]counter = [counter]
[inc,counter]counter = [counter]
```

This will generate the following.

```
counter = 1
counter = 2
```

A notable feature of BGPM is that, if macros are defined within an argument list of a macro call, they are removed when the macro finishes execution. Using this feature, it is easy to define conditional macros such as **ifeq**. BGPM is thus simple but remarkably powerful, allowing one to define, for instance, the macro **prime** to generate the *n*<sup>th</sup> prime, so that **[prime,100]** would expand to **541**.

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 <b>Fact</b>
[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 <b>TRUE</b> if they were equal and <b>FALSE</b> otherwise
[JumpF,2]	Inspect and remove the top value of the stack and jump to label 2 if it was <b>FALSE</b>
[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
[LoadC,1]	Load the constant 1 onto the stack
[Sub]	Replace the top two items on the stack by $n-1$
[Fn,Fact]	Call the function named <b>Fact</b>
[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. It was helpful to use readable function names such as **Fact** in the macro code. Since the assembly language for both Edsac 2 and Titan only had numerical program labels the macros **Prog** and **Fn** would convert function names to label numbers using macros that mapped function names to label numbers.

The CPL compiler written in a subset of CPL was large, and it gave us a reasonable feel of what programming in CPL was like. Our subset of CPL needed a mechanism to allow us to construct hash tables, lists and structures such as the parse tree generated during syntax analysis. This was easily accomplished using a simple memory model in which all memory locations were of the same size and labelled (or addressed) by consecutive integers. Macros were added to return the integer addresses of arguments (**LLX**), locals (**LLW**) and globals (**LLG**), and a macro **RV** was added to access the contents of a memory location given its address.

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.

We found that we could survive with all values being the same size, since they could represent all the kinds of value we needed, including integers, characters, truth values, arrays and functions, and this greatly reduced the number of macros we had to define. It was just as well that functions were represented by a single pointer.

The resulting subset of CPL that was used is outlined in my PhD thesis[11] and it closely resembles BCPL, but we still felt we were programming in CPL.

Within the compiler, it was frequently necessary to write code to generate error messages and output assembly language. It was clear that the way GPM expanded macros by copying their bodies while replacing substitution items by arguments was ideal. This resulted in the definition of `writeln` whose first argument was like the body of a macro but with slightly extended substitution items such as `%c`, `%s` and `%i5` to cope with characters, strings and numbers. There were as many additional arguments as needed and substitution items did not need to specify argument numbers since arguments were used in sequence. I regard `writeln` as a direct result of our experience with Strachey's invention of GPM.

An extension of `writeln` called `printf` is widely used in C, but due to its polymorphism it is difficult to implement it in strictly typed languages and so, sadly, is not available in ML or Java.

## 4 MIT, BCPL and PAL

Strachey visited Jack Wozencraft at MIT when he was designing a new course to introduce first year students to computer programming. The outcome was that both Peter Landin and I moved to MIT and helped to design a form of sugared  $\lambda$ -calculus, based on the language ISWIM (If you See What I Mean) described by Landin in "The next 700 programming languages"[8]. Landin produced a prototype implementation in LISP using his SECD machine as described in "The mechanical evaluation of expressions"[6]. When I arrived at MIT in December 1966, I quickly constructed a BCPL compiler that ran on the timesharing system CTSS and set about using it to implement a variant of Landin's system called PAL[1] that was then used in the new course. This language was a form of sugared  $\lambda$ -calculus but its syntax was somewhat related to CPL. Based on the SECD machine it was dynamically typed with every value carrying its data type at run time. Like LISP, it required a garbage collector. Many years later this course was superseded by one based on a cleaned up version of LISP called Scheme. The lecture notes for this replacement were turned into the book "Structure and Interpretation of Computer Programs"[4]. A similar course was later started in Cambridge, UK, using ML. BCPL still exists and is freely available[13] and has a manual[14].

## 5 OS6 and Triplos

Throughout my time at MIT and later, on return to Cambridge, I maintained close ties with Strachey. He was enthusiastic about the design and portability of BCPL to the extent that he implemented it on the KDF-9 computer at Oxford, and later he installed it on the CTL Modula One at the Programming Research Group at 45 Banbury Road. He also used BCPL to implement an operating system called OS6 for that machine which, as the story goes, ran successfully, for the first time, on a Sunday evening very few days after the machine arrived. The BCPL source code is published in OS6Pub[19] with the "Commentary on the text of OS6Pub"[20]. BCPL running

under OS6 was the only language available to students at the Programming Research Group for some years.

OS6 included many innovative ideas, one being his implementation of I/O streams. He represented a stream by a small vector, called a stream control block (SCB), that held all the values needed for its implementation, such as a pointer to its buffer and integers giving its size and current position. But importantly, the SCB also held three functions. One (**Next**) to read the next value from the stream, one (**Out**) to write a value to the stream and one (**Close**) to close the stream. **Next** was defined as follows:

```
LET Next(s) = (s!NEXT)(s)
```

where **s** points to an SCB and **NEXT** is a manifest constant identifying the position of the read function. This mechanism could be used to represent streams of characters, words or indeed any kind of value. You could even create a stream that generated prime numbers. It is clear that this structure is similar to an instance of a class in a language such as Java. Some streams required larger SCBs but could still use the same definition of **Next**. This is clearly a precursor to the concept of inheritance found in modern object oriented languages.

Some years later I led a team of research students that implemented a portable operating system called Tripos[16]. This was used for some years in Cambridge for research concerned with computer networks and the Cambridge Ring. BCPL running under a version of Tripos is still used commercially controlling the factory floor of many Ford car plants around the world. An interpretive version of Tripos called Cintpos[15] is still freely available.

## 6 Conclusions

I am deeply indebted to Christopher for the help and discussions we had when I was a Research Student and for arranging that I could go to MIT after my PhD. Without him and the CPL project, BCPL would not have been developed and it would not have been seen by Ken Thompson. Ken's language B[5] would not have been designed and his collaboration with Dennis Ritchie would not have resulted in the development of C[17] and its successors such as C++ and Java. Whether this was good is perhaps still debateable.

## References

- [1] A. Evans, Jr. PAL - A language designed for teaching programming linguistics. In *Proceedings of 1968 23 ACM Conference, pp 395-403 August 1968*. Thompson Book Company, Washington, DC, 1968.
- [2] D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. *The Computer Journal*, 6:134-143, July 1963.
- [3] Editor: C. Strachey. CPL Working Papers, V53-57. Technical report, Computer Laboratory, Cambridge University, July 1966.
- [4] G.J. Sussman H. Abelson and J. Sussman. *Structure and Interpretation of Computer programs*. MIT Press, Cambridge, MA, 1985.
- [5] S.C. Johnson and B.W. Kernighan. *The Programming Language B*. <http://cm.bell-labs.com/who/dmr/bintro.html>, 1997.

- [6] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.
- [7] P.J. Landin. Correspondence between ALGOL 60 and Church’s Lambda-notation, Parts 1 and 2. *Comm. ACM*, 8(2 and 3), February and March 1965.
- [8] P.J. Landin. The next 700 programming languages. *Comm. ACM*, 9(3):157–166, 1966.
- [9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [10] Ed. G. Nelson. *System Programming with Modula-3*. Prentice Hall, 1991.
- [11] M. Richards. *The Implementation of CPL-like programming languages*. PhD thesis, Cambridge University, 1966.
- [12] M. Richards. *EDSAC Initial Orders and Squares Program*. [www.cl.cam.ac.uk/users/mr/edsacposter.html](http://www.cl.cam.ac.uk/users/mr/edsacposter.html), 2009.
- [13] M. Richards. *The BCPL Cintcode Distribution*. [www.cl.cam.ac.uk/users/mr/BCPL/bcpl.{tgz,zip}](http://www.cl.cam.ac.uk/users/mr/BCPL/bcpl.{tgz,zip}), 2011.
- [14] M. Richards. *The BCPL Programming Manual*. [www.cl.cam.ac.uk/users/mr/bcplman.pdf](http://www.cl.cam.ac.uk/users/mr/bcplman.pdf), 2011.
- [15] M. Richards. *The Cintpos Distribution*. [www.cl.cam.ac.uk/users/mr/Cintpos/cintpos.{tgz,zip}](http://www.cl.cam.ac.uk/users/mr/Cintpos/cintpos.{tgz,zip}), 2011.
- [16] M. Richards, A.R. Aylward, P. Bond, R.D. Evans, and B.J. Knight. The Tripos Portable Operating System for Minicomputers. *Software-Practice and Experience*, 9:513–527, June 1979.
- [17] D.M. Ritchie. *The Development of C*. <http://cm.bell-labs.com/who/dmr/chist.html>, 1997.
- [18] C. Strachey. A General Purpose Macrogenerator. *The Computer Journal*, 8(3):225–241, October 1965.
- [19] C. Strachey and J. Stoy. *The text of OS6Pub*. Programming Research Group, Oxford University Computer Laboratory, July 1972.
- [20] C. Strachey and J. Stoy. *The text of OS6Pub (Commentary)*. Programming Research Group, Oxford University Computer Laboratory, July 1972.
- [21] C. Strachey and M.V. Wilkes. Some Proposals for Improving the Efficiency of ALGOL 60. *Comm.A.C.M.*, 4:448, 1961.