THE DESIGN AND IMPLEMENTATION OF CPL-LIKE PROGRAMMING LANGUAGES

by

M. Richards

St. John's College

This reconstruction is currently under development

Revision date: Mon Apr 22 16:22:42 BST 2019

A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge,
November, 1966.

Notes about this document

My thesis was typed using a conventional typewriter producing three copies with the aid of carbon paper. The top copy in now held in the University Library, the second copy was given to Christopher Strachey and I still have the third copy. Since its print quality is poor I have decided to reconstruct the thesis using LaTeX. I have attempted to make it as close as I can to the original but have used different fonts. The main text is in Times Roman with chapter headings and sections in bold. Program code is written in a typewriter font where possible to simulate the look of code written using a flexowriter. Reserved words such as $\underline{\texttt{if}}$ and $\underline{\texttt{while}}$ are underlined, and some symbols such as $\underline{<}$ and $\neq$ involve overprinting which was poassible on the flexowriter using the backspace key. The flexowriter had a paragraph sign used in the representation of CPL section brackets. In this document they are represented by § and $\not\S$. On the first implementation of BCPL at MIT they were represented by `$(` and `$)` and when the ASCII character set became available they were replaced by `{` and `}`.

I have not used the LaTeX facilities for chapter and section headings, and have not used the LaTeX mechanisms for table of contents, bibliography or the appendices.

This reconstruction is in the early stages of development. More content will appear when I have time to type it in, but this is likely to be rather slow.

Indication of progress with the reconstruction

In the table of contents gives the page numbers for sections that have been reconstructed up to the next section heading. For all other sections the page number appears as `ddd`.

Content

Introduction

Summary

CPL (Combined Programming Language) is a programming language which has been jointly developed by London and Cambridge Universities; since its inception in 1962 it has become more extreme in its underlying philosophy and structure and has thereby become a language of academic interest. Much of the thought that has gone into CPL and its compiler is applicable to CPL-like langages in general.

This dissertation is a presentation of CPL, describing and discussing some of the language concepts and implementation problems with particular reference to the existing CPL compiler at Cambridge (throughout this dissertation this is called the Cambridge CPL compiler). Many of the new ideas in CPL cause some difficulty in the design of its compiler since existing techniques were not powerful enough or not suitable for other reasons.

Chapter 1 describes and discusses CPL and Chapter 2 considers some of the implementation problems. Chapter 3 is concerned with the detailed design of the CPL compiler and discusses various algorithms for transforming CPL text in an internal semantic structure called the Applicative tree.

In Chapter 4 the translation process from this semantic structure to machine code is discussed and an intermediate object code called CPLOCODE is suggested.

Chapter 5 is concerned with the actual implementation of the compiler; it describes a macro code system that was specially developed for it and this is compared with an altternative method of coding using a very basic high level language, called BCPL, which evolved from the macro system.

Acknowledgements

I am indebted to all the members of the CPL development group and in particular to Mr. C. Strachey and Dr. D. Park for many long and interesting discussions on all aspects of CPL and its compiler. I would also like to acknowledge the invaluable assistance of Mr. L.J.D MaIntyre for his work on helping to implement the Cambridge CPL compiler.

I owe a great deal to Dr. Barron and Dr. Park for introducing me to language design and compiler writing, and for continual supervision of my research.

I gratefully acknowledge the financial assistance of the Science Research Council for my first year and the ICT Research Studentship for my second and Last years of my research.

Originality

Both the design and implementation of CPL have be undertaken by groups of people and so it is often difficult to decide which ideas in this dissertation are original. I have been one of the authors of CPL for two and a half years years and have had some influence on its design especially those concerning those features, such as the switch command, which have evolved from experience of writing large systems in CPL. The notes on segmentation in 1.2.7 are derived from a suggestion made by Mr. C. Strachey.

The Cambridge implementation of CPL was mainly developed and written by Dr. D. Park and its brief description in Chapter 2 is included since it is very relevant to the design of the compiler.

The overall scheme for a CPL compiler was suggested by Mr.C. Strachey and partly implemented for a subset language on EDSAC II by Dr J.B. Hext. The detailed development of this into the full CPL compiler and the suggested alternatives are my own work except where explicit acknowledgement is given.

I declare that no part of this dissertation has been submitted for any university degree.

St. John's College
Cambridge.

November, 1966. M. Richards.

CHAPTER 1

1.1 Outline of CPL

Since this dissertation is largely concerned with CPL and its implementation it is necessary to start with a brief description of the language.

1.1.1 CPL character set

Although the hardware representation is not an important part of the logical structure of CPL, it is in the philosophy of CPL to be readable and, where possible, self explanatory; the use of a large character set is thus natural. The examples of CPL programs that occur in this dissertation will use a character set that includes capital and small letters, the digits 0 to 9 and the following symbols

```
+  -  ×  /  =  <  >
(  )  [  ]  ∧  ∨  ∼
'  ;  :  ,  .  ⋆  →  §  |
```

overprinting with vertical or diagonal bars and the use of underlining further extends the charater set.

1.1.2 Basic symbols

The basic symbols of CPL are constructed from sequences of one or more of these characters. This set of symbols includes many underlined words and a number of composite characters, such as:

`:=` and `<=>`

1.1.3 Identifiers, string constants and numbers

An identifier is either a small letter followed optionally by one or more primes or a capital letter optionally followed by letters, digits and dots and possibly terminated by primes.

```
e.g.  a    x     x'
      Ab1  L.x3  p''
```

A string constant is a sequence of characters from the string constant character set, its first and last characters are primes. The string constant character set set includes all the CPL charaters and the following six composite characters:

`*' *| *s *t *n **`

which are used to represent respectively the six string characters

`' ||` space tab newline `*`

A number is a sequence of digits and possibly one decimal point; no spaces are allowed in numbers.

1.1.4 Canonical symbols

Canonical CPL is a representation independent form of CPL consisting of a sequence of basic symbols, identifiers, string constants and numbers; its purpose is to assist the description of the language by defining the boundary between representation dependent and independent features. It can be used as an intermediate stage in the translation from one hardware representation of CPL to another.

1.1.5 Data items and types

A CPL program is concerned with relations and operations on data items; these may represent either numerical or non-numerical objects, and their types describe the sort of objects represented and often the precision of the representations.

The numerical types include:

`index`	a positive or negative small whole number
`integer`	an integer using the same machine representation as `real`.
`real`	a number in the range $\pm 10^{100}$ with a precision of about 10 decimal digits.
`double`	a number with twice the precision of `real`.
`complex`	an ordered pair of `real`numbers with arithmetical properties similar to those of A+iB,
`double` `complex`	a complex number of about twice the precision of `complex`.

Among the non-numerical types are:

`boolean`	a truth value which may only be `true`or `false`.
`logical`	a standard length string of binary digits.
`long` `logical`	a twice standard length string of binary digits.
`string`	a string of CPL characters.
`label`	a representation of a program point.

1.1.6 Expressions

Data items may be combined together with operators to for expressions; these data items may occyr explicitly as a written constant or they may be referred t by their identifiers, as in:

$$\begin{aligned} & 3xx + 4xy - 2yy \\ \text{or}\quad & p\wedge\underline{8}77 \vee q\wedge\underline{8}777777 \end{aligned}$$

Syntactically the most basic form of expression is called the individual; this may be a either a written constant, or identifier, an array or function application or a block expression; these will be described later.

The expression operators which combine these individuals have binding powers and association rules which make the interpretation of CPL expressions as close to the natural mathematical meaning as possible. For the more complicated expressions round brackets may be needed for grouping subexpressions; there is no limit to the depth of nesting or the degree of complexity allowed.

Every expression can be evaluated to yield a data item and the type of this data item is termed the type of the expression and can be statically determined from the written form of the expression and the types of its component data items.

1.1.6.1 Function applications

A function application is a short hand notation for a complicated expression. The expression is associated with the function name by a function definition, see (1.1.8.3), and is evaluated at each application of the function. The syntactic form of this is

$$E_1\,\texttt{[}E_2\texttt{]}$$

where E_1 is an identifier, a function or array application or an expression in round brackets and E_2 is a list (which may be empty) of expressions; this list is called the actual parameter list and is used to convey information from the application to the body of the function in exactly the way used in routine calls, see (1.1.8.2).

A function can have more than one result and it may also be evaluated in Left Hand Mode, see (1.1.7.1).

1.1.6.2 Arrays

A data item whose type is `array` represents a collection of elements which are all of the same type; the individual elements are referred to by their coordinates which are numbers of type `index`, and the number of indices required is called the dimension of the array. The syntactic form of an array reference is the same as for a function application; the following is an example:

`A[i, j+1]`

An array of dimension more than one can be thought of and used as vector of arrays of dimension one less. Thus if A is a <u>`real`</u> 2 <u>`array`</u> then `A[i]` is a reference to a <u>`real`</u> 1 <u>`array`</u> and the expression `A[i][j+1]` is exactly equivalent to `A[i,j+1]`.

The dimension and component types are static and cannot be changed by assignment. An array is constructed using the function `Newarray`; this constructs a rectangular array of any number of dimensions and the argument list contains the component type and the bounds of each dimension, thus `Newarray[`<u>`index`</u>`,(1,10),(0,100)]` is an expression whose result is a rectangular array such that `A[i,j]` is of type <u>`index`</u> and references elements whose coordinates lie in the ranges `1<=i<=10` and `0<=j<=100`.

`Newarray` only creates the space and structure of the array, it does not initialise the elements.

1.1.6.3 Expression operators

The expression operators are broadly separated into three groups in decreasing order of binding power – arithmetic, relational and logical.

1.1.6.4 Arithmetic operators

The arithmetic operators are

`+`   `-` $\times$ `/` $\uparrow$

Juxtaposition between individuals implies multiplication and is treated exactly as though the multiplication sign was not absent.

Multiplication ($\times$), division (`/`) and exponetiation ($\uparrow$) are all of equal precedence and associate to the right while addition (`+`) and subtraction (`-`) are less binding and associate to the left. The monadic operators `+` and `-` are recognised by not having individuals on their left and they have the same binding power as their dyadic forms, thus `-a`$\uparrow$`b` means `-(a`$\uparrow$`b)`.

1.1.6.5 Relational operators

The relational operatores are

`=` $\neq$ `<` $\leq$ `>` $\geq$ `<<`   `>>`

They are all equally binding and may occur in an extended relational expression as in

$$a < b \leq C+2$$

The type of such an expression is always boolean and it value is `trueif` and only if all the component relations are true, that is, in the above example, if a < b and b ≤ C+2.

The other relational operators have meanings close to the natural mathematical sense.

1.1.6.6 Logical operators

The following are the logical operators listed in decreasing order of precedence.

$$\tilde{} \quad \wedge \quad \vee \quad \not\equiv \quad \equiv$$

These operators yield logical or boolean results depending on the types of their operands. Where the result is a bit string the dyadic operators produce a value whose n^{th} bit depends only on the corresponding bits of the two operands and is determined from the following table.

Operands	Operator			
	$\wedge$	$\vee$	$\equiv$	$\not\equiv$
both 0	0	0	1	0
both 1	1	1	1	0
otherwise	0	1	0	1

where the type of the result is boolean its value may be determined from the same table by replacing 0 by `false` and 1 by `true`.

The operator ˜ is monadic and its result is the logical negative of its operand.

1.1.6.7 Concatenation of strings

Strings may be concatenated using the operator `<=>`; its binding power is less than the arithmetic operators though more than the relations. The result of concatenation is a string consisting of the first argument joined on to the left of the second argument.

1.1.6.8 Conditional expressions

The syntactic form of a conditional expression is

$$E_1 \texttt{ -> } E_2\texttt{,}\ E_3$$

where E_2 and E_3 are general expressions and E_1 is not itself a conditional expression.

To evaluate a conditional expression, first E_1, which must be boolean, is evaluated to yield a result `true` or `false`, then the result is the value of E_2 if E_1 was `true` otherwise the value of E_3.

1.1.7 Commands

A data item has a value and this may be changed or updated by means of the assignment command. Before describing this command it is necessary to introduce the concept of mode of evaluation.

1.1.7.1 Left hand mode

The result of evaluating in Left Hand Mode (Lmode) is a representation of a reference to a data item and is called an Lvalue.

1.1.7.2 Right hand mode

Evaluation in Right Hand Mode (Rmode) produces the value of the data item and this is termed the Rvalue. In an assignment command it the Rvalue of the data item that is changed; there is no way in which an Lvalue of the data item can be changed once it has been created.

1.1.7.3 The assignment command

The basic assignment operation acts on two operands, one an Lvalue and the other an Rvalue; its effect is to replace the Rvalue of the data item whose Lvalue is the first operand by the Rvalue which is the second operand.

The CPL assignment command is a generalisation of this basic assignment; its syntactic form is

$$E_1 \texttt{ := } E_2$$

where E_1 and E_2 are both expressions or lists of expressions separated by commas as in

```
i, Symb[i] := i+1, C
```

The left hand side is evaluated in Lmode to produce a list of one or more Lvalues and the right hand side is evaluated in Rmode to produce a list of the same length of Rvalues; appropriate type transfer functions are then applied to the Rvalues where necessary to make their types correspond to those of the Lvalues, and finally the basic assignment operation is performed to the corresponding pairs of L and Rvalues taken in turn.

1.1.7.4 Command sequence

Command are normally obeyed in sequence one after the other until either the end of the command sequence is reached or a transfer command is encountered.

Syntactically the commands in a command sequence are separated by semicolons or newlines; the newline only acts as a command separator where it makes sense, it is otherwise ignored.

1.1.7.5 Labels

A command may be labelled with an identifier, as in

```
L: R[x, y]
```

In this example, `L` is said to be a label set by colon and its Rvale is a representation of this point in the program. It may be referred to by its identified from any place in the routine body or result block in which the labelled command occurs provided that the definition of the same identifier does not shiel the label. The sole purpose of a label is as the goal of a transfer command.

1.1.7.6 Transfer command

The syntactic form of a transfer command is

$$\underline{\texttt{goto}}\ E$$

where E is an expression of type $\underline{\texttt{label}}$.

The effect of this command is to break the normal sequence of execution and resume it at the command which is labelled with the label whose value is equal to the Rvalue of the expression E.

A label may have several activations if it occurs in a funtion or routine that is used recursively. At a routine call a new set of data items is created whose Rvalues represent each of the labels in the routine body and these are different from the Rvalues of all other labels, in particular they are different from the Rvalues of labels of an outer activation of the same routine. A transfer command therefore must first exit from as many functions and routines as necessary until the right activation is reached and then resume execution at the appropriate label. If this involves entering any blocks the declarations in the block heads are performed so that when execution is finally resumed as the bound variables have been declared. If while performing a declaration another transfer command is encountered this supercedes the current transfer.

1.1.7.7 Routine command

The routine command is a shorthand notation to call for the execution of the complicated command which occurs in the definition of the routine being called.

The syntactic form of a routine call is

$$E_1\ \texttt{[}\ E_2\ \texttt{]} \qquad \text{or} \qquad \text{an identifier}$$

where E_1 is either an expression in round brackets, a function or array application, or just a single identifier, and E_2 is either empty or a list of one or more expressions; E_2 is termed the actual parameter list. The following are examples of routine calls

```
                Rexp[n]
B -> f, g) [ a, b, 1, L ]
```

Information may be handed from the call to the body of the routine via the actual parameter list, and the expressions in this list are evaluated at the time of the call in the types and modes declared in the routine's definition, this is described in detail in (1.1.8.2).

1.1.7.8 Conditional commands and loops

These commands are to assist in specifying and controlling the order of evaluation of basic commands; a wide variety of such commands is available and they are listed below giving their syntactic forms.

`if` E `do` C
`unless` E `do` C
`while` E `do` C
`until` E `do` C
C `repeat` `while` E
C `repeat` `until` E
C `repeat`
`test` E `then` `do` C_1 `or` C_2

In all these commands E is a boolean expression and C, C_1 and C_2 all stand for commands. The value of E determines the flow of execution; for instance, the effect of the test-command is that of executing C_1 or C_2 depending on whether the value of E is `true` or `false` respectively. The meanings of the other commands in this group are self explanatory.

1.1.7.9 For-commands

A for-command is a loop command in which a control variable is successively set to the values of a sequence specified by the for-list. The syntactic form of the for-command is

`for` N = *Flist* `do` C

where N is an identifier called the control variable, C is the controlled command and *Flist* is the for-list which is a syntactic represntation of a list of values; basically it is either a single list of expressions or one of the following notations for an arithmetic progression:

E_1 <u>`to`</u> E_2 equivalent to E_1, $E_1 + 1, \ldots, E_2$
<u>`step`</u> E_1 , E_2 , E_3 equivalent to E_1, $E_1 + E_2, \ldots,$ E_3
or E_1 , E_2 , . . . , E_n

The general for-list consists of a string of these baisc forms separated by commas.

The sequence of values specified by the for-list is evaluated once and only once on first encountering the for-command, then the body C is repeatedly executed with the control variable successively declared and set equal to each value in turn until either the sequence is exhausted or execution is transferred out of the for-command by a jump.

The control variable is local to the body of the for-command, however if successive assignments to an external data item is required the following syntactic form could be used.

<u>`for`</u> <u>`ext`</u> E = *Flist* <u>`do`</u> C

where E is one expression.

1.1.7.10 Break, return and finish commands

More text to follow

1.1.7.11 result-is command and result blocks

More text to follow

1.1.7.12 Blocks

More text to follow

1.1.8 Declarations and definitions

More text to follow

1.1.8.1 Simple definitions

More text to follow

1.1.8.2 Routine definitions and calls

More text to follow

1.1.8.3 Function definitions

More text to follow

1.1.8.4 Fixed functions

More text to follow

1.1.8.5 Free functions

More text to follow

1.1.8.6 Free routines

More text to follow

1.1.8.7 Definitions by reference

More text to follow

1.1.8.8 Definition structure

More text to follow

1.2 Discussion of CPL

More text to follow

1.2.1 Aims of general purpose languages

More text to follow

1.2.2 Some useful terms

More text to follow

1.2.2.1 Constant and variable

More text to follow

1.2.2.2 Static and dynamic

More text to follow

1.2.2.3 Extent and existance

More text to follow

1.2.2.4 Version and activation

More text to follow

1.2.3 Recursion

More text to follow

1.2.4 Re-entrant

More text to follow

1.2.5 The removal of call by substitution

More text to follow

1.2.6 Some additions to CPL

More text to follow

1.2.6.1 Switch-commands

More text to follow

1.2.6.2 Lookup-expressions

More text to follow

1.2.7 Segmentation

More text to follow

1.2.7.1 Input and output of fixed functions

More text to follow

1.2.7.2 Static segmentation

More text to follow

Chapter 2

2 The implementation of CPL

More text to follow

2.1 Arrays

More text to follow

2.2 Declared data items

More text to follow

2.3 Other features requiring dynamic storage

More text to follow

2.4 The runtime space allocation system

More text to follow

2.5 The implementation of jumps

More text to follow

2.5.1 The representation of a label

More text to follow

2.5.2 The execution of a jump

More text to follow

2.6 Debugging aids

More text to follow

Chapter 3

3 Compiler design

More text to follow

3.1 Aims of the compiler

More text to follow

3.1.1 Internal representation

More text to follow

3.1.2 Segmentation of the compiler

More text to follow

3.1.3 Method of coding the compiler

More text to follow

3.2 Transformation to the AE form

More text to follow

3.2.1 Line imager and Preprocessor

More text to follow

3.2.2 Syntax analysis

More text to follow

3.2.3 The choice of syntactic description

More text to follow

3.2.4 Condensation

More text to follow

3.2.5 Output of Diagram

More text to follow

3.2.6 Applicative tree structure

More text to follow

3.2.6.1 Representation of the tree

More text to follow

3.2.7 Syntax description by decomposition

More text to follow

3.2.8 An alternative transformation to AE form

More text to follow

3.3 Operations on the AE tree

More text to follow

3.3.1 Routine to canonicalize the variables (RCV)

More text to follow

3.3.1.1 The RCV process

More text to follow

3.3.1.2 TBVL

More text to follow

3.3.1.3 TB

More text to follow

3.3.2 Typematching(TM)

More text to follow

3.3.2.1 The typematching process

More text to follow

3.3.2.2 Recursive definitions

More text to follow

3.3.2.3 Result blocks

More text to follow

3.3.2.4 General remarks

More text to follow

3.3.2.5 Auxiliary functions and routines

More text to follow

3.3.2.6 Function definitions

More text to follow

3.3.2.7 Conditional expressions

More text to follow

Chapter 4

4 Object code

More text to follow

4.1 Object code streams

More text to follow

4.1.1 The PE stream

More text to follow

4.1.2 The program stream

More text to follow

4.2 Intermediate opbject code

More text to follow

4.2.1 Zero address intermediate code

More text to follow

4.3 Flatten

More text to follow

4.3.1 Output of intermediate code

More text to follow

4.3.2 The flatten process

More text to follow

4.3.2.1 The translation of commands

More text to follow

4.3.2.2 The translation of expressions

More text to follow

4.3.2.3 The translation of jumps

More text to follow

4.3.2.4 The translation of definitions

More text to follow

4.4 The code generator

More text to follow

4.5 Optimisation

More text to follow

4.6 A suggestion for an intermediate code

More text to follow

4.7 An optimising code generator

More text to follow

4.8 An internal assembly language

More text to follow

4.8.1 Description of IAL

More text to follow

4.8.1.1 Parameters

More text to follow

4.8.1.2 Simple loading operations

More text to follow

4.8.1.3 Parameter setting

More text to follow

4.8.1.4 Modifying the latest word loaded

More text to follow

4.8.1.5 Evaluation of expressions

More text to follow

4.8.1.6 Loading hunks

More text to follow

4.8.1.7 Conditional loading

More text to follow

4.8.1.8 Stream changing

More text to follow

4.8.1.9 Start directive

More text to follow

4.8.1.10 Conclusion

More text to follow

Chapter 5

5 Implementation of the compiler

More text to follow

5.1 Description of macro code

More text to follow

5.2 Segmentation

More text to follow

5.3 Compiler debugging aids

More text to follow

5.4 Conclusion

More text to follow

5.5 A bootstrap system

More text to follow

5.6 The design of BCPL

More text to follow

5.6.1 Expressions

More text to follow

5.6.2 Mode of evaluation and storage

More text to follow

5.6.3 Definitions

More text to follow

5.6.4 Commands

More text to follow

5.6.5 Segmentation and scope rules

More text to follow

5.6.6 Static names constants

More text to follow

Appendix 1

The AE tree of CPL after CAE

A list is represented in this notation by items separated by commas and surrounded by round brackets; some list may be marked as compounds or Vexpressions by preceeding the open bracket by c or v, respectively. Names which occur in the constructions without any definition are basic items. Alternative constructions are on different lines or are separated by /.

Name, Variable lists and Types

```
N1   ::=  Name / c(Comma, Name, ..., Name)
V    ::=  v(Name, T, Address, Properties)
FPL  ::=  Nil / V / c((Comma, V, ..., V)
T    ::=  All basic symbol types
          c(routine) / c(function, T)
          c(n array, T) / c(Commaa, T, ..., T)
```

Definitions

```
D    ::=  c(be, N1, T) / c(all be, N1, T)
          c(valdef, N1n E) / c(all eq, N1, E)
          c((refdef, N1, E) / c(all ref eq, N1, E)
          c(fn eq, N, (FPL, E))
          c(fn subst eq, N, (FPL, E))
          c(rtdef, N, (FPL, C, c(Colon, Returnlabel)))
          c(rec, D, ..., D) / c(in, D, D)
          (D, ..., D)
```

Expressions

More text to follow

Commands

More text to follow

Appendix 2

References

The following abbreviations are used:

Comm. ACM	The Communications of the ACM
CJ	The Computer Journal
A.P.I.C	The Automatic Programming Information Centre (Brighton)
Annual Review	The Annual Review in Automatic Programming
ULICS	The University of London Institue of Computer Science
UML	The University Mathematical Laboratory, Cambridge.

[1] Barron, D.W. et al. 'The main features of CPL' CJ. vol 6, p.134. (1966)

[2] Buxton, J.N. SPL, Paper No.6. CEIR Ltd., (1966)

[3] Brooker, R.A. et al. 'The compiler compiler' Annual Review, vol.2, p.229. A.P.I.C. (1963)

[4] Church, A. 'The calculus of lambda-conversions' Princeton University Press. (1941)

[5] Curtiss, A.R. et al. 'A proposed target language for Atlas' CJ. vol.5, p.100 (1962)

[6] Dijkstra, E.W. 'A primer for ALGOL programming' Academinc Press, New York. (1962)

[7] Dijkstra, E.W. 'An attempt to unify the constituent concepts of serial program execution' in 'Symbolic Languages in Data Processing', Gordon and Breach, London (1962)

[8] Feldman, J.A. 'A formal semantics for computer languages and its application to a compiler-compiler' Comm. ACM Vol.9, p.3 (January 1966)

[9] Hext, J.B. 'Programming languages and compiling teckniques' UML, Y-304 (1965)

[10] Iliffe, J.K. 'The use of the Genie system in numerical calculations', Annual Review, Vol.2, p.1. A.P.I.C. (1962)

[11] Irons, E.T.	'The structure and use of a syntax directed compiler' Annual Review, Vol.3, p.207.
[12] Landin, P.J.	'A correspondence between ALGOL 60 and Church's lambda notation' Comm. ACM. Vol.8, in two parts (February and March 1965)
[13] Matthewman, J.H.	'An experimental syntax directed compiler for EDSAC 2' UML, Technical Memorandum, No.64/4 (1964)
[14] McIntyre, L.J.D.	CPL implementation notes IN/2 to IN/8 - UML (1966)
[15] Naur, P. (Editor)	'Report on the algorithmic language ALGOL 60' Comm. ACM. Vol.3, p.299, (May, 1960)
[16] Park, D. et al.	'Titan CPL object code circulars 4 to 8' UML (1965)
[17] Strachey, C.	'A general purpose macrogenerator' CJ. Vol.8, p.225 (October 1965)
[18] Strachey, C. (Editor)	'CPL working papers' a technical report, UML and ULICS (Jult 1866)
[19] Smith, R.V.	'The literal and the variable in programming languages' IBM research paper RC-1444. (1965)
[20] Wilkes, M.V. and Strachey, c.	'Some proposals for improving the efficiency of ALGOL 60' Comm. ACM, Vol.4, p.488. (November, 1961)
[21] Wirth, N.	'A generalisation of ALGOL' Comm. ACM. Vol.6 p.547. September, 1963)
[22] Wirth, N. and Weber, H.	'EULER: A generalization of ALGOL, and its formal definition' Comm. ACM. Vol.9, p.13 (January and February, 1966).