The London CPL1 compiler

By G. F. Coulouris,* T. J. Goodey,* Roberta W. Hill,* R. W. Keeling† and D. Levin

The CPL language has been described elsewhere, but as yet no implementations of it have been described. The authors have been concerned in the implementation of a subset of the full language, the implementation proceeding concurrently with the design of the language. Since the majority of the designers were not members of the implementation team, the choice of a subset was partly dictated by the state of the design process at the time. Some other motivations for the choice are given in Coulouris (1967). The implementation was intended to clarify some of the problems of implementing CPL, and to gain feedback information from user experience to assist in the design process. The authors believe that CPL is a powerful language for the demonstration of computing concepts, and that the implementation has therefore provided a useful teaching tool.

(First received September 1967)

The compiler which is the subject of this paper is implemented on the London Atlas for a language CPL1, very similar to CPL (Barron *et al.*, 1963; CPL Working Papers, 1966), but less powerful. (It has no types **general**, **complex** or **double length**. The list processing facilities mentioned in Barron *et al.* (1963) and Barron and Strachey (1966) are not part of CPL as defined by the reference manual (2), and are not in CPL1.) Despite this CPL1 is more powerful than ALGOL 60, and some details of its implementation may therefore be of interest.

The CPL1 compiler has been written using the Compiler Compiler of Brooker and Morris. This takes a 'statement' at a time, analyses it, and produces object code. In its application to CPL1 it has been necessary to define the complete program as a single source statement; this is forced by the '**where**' clause which allows trailing definitions and by the labels which are not defined at the head of a block. In a situation such as:

```
§1 let B = 0·01
§  let x = B
... Program ...
          B: §1
```

it is not known, until the whole program is read in, whether x is of type **real** or **label** (in fact it is **label**).

Declarations

At the head of a block, local variables can be defined to have a certain type, and may also be initialised to the value or to the address of some expression. Thus '**let** x, y **be real**' corresponds to ALGOL '**real** x, y'; '**let** $x, y = 0\cdot 0, 30\cdot 6$' is both an implicit type declaration and an initialisation program.

'**let** $A \simeq B$' initialises A to contain the address of cell B and all references to A are indirect references to B. The two ways of evaluating an expression, to produce a value or an address, are called *right-hand* and *left-hand evaluation* respectively, the nomenclature deriving from the fact that these are the modes of evaluation required on each side of an assignment statement.

In CPL1 there are nine data types to which dynamic assignments may be made. They are listed in **Table 1**. It is a principle in CPL that the type of any expression can be recognised by inspection. Thus when the compiler meets '**let** $b =$ '*cat*' ' it sets up b as a *string* variable because it recognises '*cat*' as a string. Constants have easily recognisable types, and variables have their type specified in their definition. The type of an expression is determined by the main operator and its operands—i.e. by the operator and operands which result from the first stage of syntactic analysis of the expression. (If the operands are themselves expressions, this process is applied recursively.) Thus $c > a + b$ is **boolean**, $a + b$ is **real** if either a or b is **real**, otherwise it is **integer**, $A \vee b$ is **logical** unless both A and b are **boolean**, when it is **boolean**. The conditional expression $b \rightarrow A, B$ has a type which is the 'worse' of the two types of A and B, in the sense of having the lower type number in Table 1, so '$b \rightarrow 1, 3\cdot 24$' is **real**. The type of an expression can also be forced by using the explicit transfer functions *Real*, *Integer*, etc.

The compile-time routine that generates object program to evaluate an expression exits with a global

Table 1

TYPE NUMBER	TYPE
1	**real**
2	**integer**
3	**logical**
4	**boolean**
5	**string**
6	**routine**
7	**function**
8	**array**
9	**label**

* Present address, *Centre for Computing and Automation, Imperial College, London.*
† Present address, *Department of Computer Science, University of Edinburgh.*
This work was carried out at the *Institute of Computer Science*, 44 *Gordon Square, London W.C.1.*

variable set to the type number of the result it has found. If the expression is the RHS of a definition then the defined variable and its type are entered in the compile-time dictionary.

Compile-time dictionaries

It is meaningful to redefine locally some variables in terms of their previous global values. The definition '**let** $i, j = j + 1, i + 1$' has this effect. It is necessary to compile the expressions on the right-hand side before the names being defined are put on the dictionary. i and j are said to be defined in parallel.

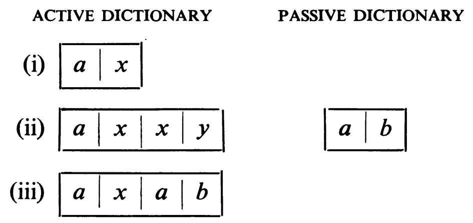

Fig. 1

It is convenient to have two dictionaries (called *active* and *passive*) where the passive dictionary holds all those definitions currently being defined in parallel. At the end of the definition the entries are transferred from passive to active dictionaries. Consider the declarations.

```
'let a, x = 0
      let a = result of § let x, y = 1
                   result = x + a §
      and b = 0'
```

Fig. 1 shows the states of the dictionaries

(i) after compiling the first definition
(ii) while compiling '**result** $= x + a$'
(iii) after compiling the second declaration.

If a definition is intended to be recursive it must be preceded by the directive '**rec**':

'**let rec** a = **result of** § . . . a . . . §'

which forces the identifiers being defined onto the active dictionary before the right-hand sides are evaluated. In this case a preliminary scan of the right-hand sides is necessary to discover their type.

'Where' clauses

When a definition is only needed over a single CPL expression or command it is not necessary to create an additional block for it. A command, expression or definition can be qualified by a **where** clause, which is a trailing definition. Variables defined in a **where** clause remain on the active dictionary throughout the scope of the **where** clause. To compile

'**let** $A = x + y$ **where** $x = 3$'

x is defined equal to 3 and put on the active dictionary, A is defined on the passive dictionary, finally x is removed from the active dictionary and A placed there. The local storage occupied by x at run-time is not recovered but is treated as 'own' to the definition.

Functions are thought of as expressions with parameters and can be defined as such, e.g.

let *Radius* $[x, y] \equiv$ *Sqrt* $[x \uparrow 2 + y \uparrow 2]$'

Variables which are not parameters of the expression are called *free* variables. Assignments to these variables alter the functions behaviour. In the above case the only free variable is *Sqrt*. In CPL a further sort of function can be defined with the property that assignments to its free variables do not alter its behaviour, but in CPL1 these functions are not available. A function is represented as an entry address to a piece of object program which will left-hand-evaluate the expression, thus allowing the function to be used on the left-hand or right-hand side of assignments. In CPL the commands for evaluating the left-hand and right-hand sides can be defined separately, in CPL1 this is not possible.

Type transfers

The programmer is not required to say what type his expressions are, so before any expression is compiled the compiler can make no assumptions about its type. The routine for compiling expressions discovers their type and exits with a global variable set to the type of the compiled expression. A compile-time transfer function is called to generate code transferring the value of the expression to a value of the type required. If the two types are the same, or if the transfer is **integer** to **real**, no other action is needed. Otherwise if the transfer is possible a soft report or note is printed, and if the transfer is impossible a hard report is printed. After a hard report the program will not normally be executed.

For instance in the boolean expression '$x = n$' where x, n are **real**, **integer**, n is transferred from **integer** to **real** for the comparison. Transfers are invoked in assignments, during evaluation of actual parameters of routines and functions, and for matching subexpressions to their operators. Thus compile-time messages from the transfer routine are frequent and sometimes useful.

Use of the Compiler Compiler

We give here a simplified version of the routine for compiling CPL1 assignments written in the Compiler Compiler language to illustrate the method used. The names in square brackets refer to the instances of phrases that the syntax recogniser has recognised in the source program. The '/1' and '/2' postfixes distinguish separate instances of the same class of phrase. Characters not in square brackets refer to instances of themselves. Compiler Compiler routines are allowed to be recursive.

```
ROUTINE [COMMAND] ≡ [EXPRESSION/1] := [EXPRESSION/2]

LHEV [EXPRESSION/1]   | plants code to evaluate (i.e.
                      | fetch) the left-hand
                      | value (address) of
                      | [EXPRESSION/1] to
                      | index register 1.

A1 = B2               | a compiler-time assignment.
                      | A1 is a local integer vari-
                      | able of the compiling
                      | routine, and
                      | B2 is the global variable
                      | always
                      | containing the current type
                      | (set by LHEV).

RHEV [EXPRESSION/2]   | plants code to evaluate
                      | [EXPRESSION/2] in an
                      | accumulator and (of course)
                      | reloads B2.

TRANSFER B2 TO A1     | the implicit transfer func-
                      | tion, to transfer
                      | the contents of the accumu-
                      | lator from the type
                      | in B2 to the type in A1.

DUMP ACC IN 1         | plants code to store the
                      | accumulator
                      | in the location pointed to
                      | by index register 1.

END
```

Expression compilation

The type of an expression is determined by its principal operators, e.g. an expression involving '$\wedge$' or '$\vee$' is of type **logical** and its subexpressions will be transferred to this type. Expressions are processed recursively from the bottom up; by the time coding to evaluate the expression is being planted, code to evaluate all the subexpressions has already been planted and all of their types are known. The routine RHEV[EXPRESSION], for compiling expressions, will involve calls to subroutines, one for each subexpression. The nature of expression compilation can be seen from a simple example.
The subexpression [FEXPN] is an expression involving at worst the operator '$\vee$' and is defined syntactically:

[FEXPN] = [EEXPN] $\vee$ [FEXPN] *or* [EEXPN]

The Compiler Compiler will use this definition for syntactic analysis, and will need two corresponding parts to the routine, one for each alternative. The subexpression is compiled by routine FRHEV:

```
ROUTINE FRHEV [FEXPN]
→ 1   UNLESS [FEXPN]
      ≡ [EEXPN]              | is the operator absent
      ERHEV [EEXPN]          | if so the type and
                             | value of [FEXPN]
                             | are that of [EEXPN].
      END
(1)   LET [FEXPN]            | a command control-
      ≡ [EEXPN] ∨ [FEXPN]    | ling exploration
                             | of the syntax tree.
      FRHEV [FEXPN]
      TRANSFER B2 TO 3       | transfers type to
                             | logical.
      A1 = B3                | B3 is the next available
                             | workspace address.
      DUMP ACC               | dump the accumulator
                             | moving B3 on by one
                             | word.
      ERHEV[EEXPN]
      TRANSFER B2 TO 3
      PLANT 1646 ,0,0, A1    | plants Atlas extracode
                             | which OR s the
                             | accumulator with
                             | word at address A1.
      UPDUMP 2               | move B3 pointer back
                             | two half words.
      END                    | (i.e. one whole word.)
```

The routine LHEV [EXPRESSION] compiles code for the evaluation of the left-hand value of an expression, which could be a conditional expression such as $b \rightarrow A[i], C[i]$.
If it finds an expression not possessing a left-hand value (e.g. '$x = y$') it calls RHEV to right-hand evaluate it and then creates a new left-hand value in which to store the result. The creation is performed by a routine DUMP ACC. The compile-time global variable B3 points to the next available word of run-time storage. Routine DUMP ACC moves up the pointer B3, looks at B2 (the type of latest expression compiled) to decide which accumulator to dump and plants code to dump the accumulator. The effect is to create a left-hand value for the expression most recently evaluated in right-hand mode. There is an inverse routine UPDUMP [N] which, by moving back the B3 pointer, deletes the last N left-hand values created. This simulation of a stack at compile time allows all addressing to be absolute, the technique is described in Coulouris (1967). Under recursion the local variables are preserved and restored on a run-time stack.

Data representation

Variables can be initialised by **value** to contain the right-hand value of an expression or by **reference** to contain the left-hand value of an expression. A variable can also be initialised by **substitution** which sets it up as a parameterless function.

Value variables of type **real, integer, logical** or **boolean** have a single Atlas word assigned to them at compile-time which will contain their run-time value. **Routine, function** and **label** variables also have a single Atlas word containing their entry address in the compiled program. Strings and arrays have a pointer which at run-time points to their currently assigned storage area, or is zero if they are currently empty. Strings are stored as a vector of half words. The first half word is the length of the string, subsequent half words are consecutive characters, simple or overprinted. Arrays are stored as vectors of words. Most of them contain the right-hand value of one element. However, the first few words of the vector contain the total number of elements, the dimension of the array and upper and lower bounds for each dimension. Every variable declared by value has reserved for it at compile-time an Atlas word, the contents of which is changed on assignment to the variable. The address of the reserved word represents the left-hand value of the variable.

A **reference** variable is represented by a word containing a pointer to the variable or array element it is referencing. Instructions using reference variables are implemented by indirect addressing. In most languages left-hand evaluation is done implicity as part of a fetch or store instruction. However, in CPL1 left-hand evaluation can involve evaluating functions and conditional expressions so it is more convenient to calculate the left-hand value separately and to leave the result in an index register.

A **substitution** variable is represented as a pointer to a piece of object program which calculates the left-hand value of an expression. Substitution variables can be evaluated for use on either side of an assignment. If the right-hand value is required the variable is at first left-hand evaluated then the right-hand value is fetched to the accumulator.

Routine and function parameters

A routine call must have actual parameters matching its declared formal parameters. When a routine or function is defined, a list of the types of its parameters is stored in the dictionary. In compiling the call, the types of the actual parameters are matched to those of the formal parameters, failure to match generating an error message from the compiler's transfer routine.

Inside the body of the routine or function, the formal declaration of parameters together with the substitution of actual parameters acts like an initialised definition. Just as variables may be initialised by '$=$', '$\simeq$' or '$\equiv$', parameters can be called by value, reference or substitution. Thus after

'**let routine** *R*6 [**string ref** *S*, **real value** *t*]

. . . Body . . .'

the routine call '*R*6 [*Sam*, 2.7]' will make the body of the routine act as if the local declaration

'**let** $S \simeq Sam$ **and** $t = 2 \cdot 7$'

had been made.

Copying

For variables with changing storage requirements (functions, routines, arrays and strings) the question of whether to copy on assignment arises. The bodies of routines and functions cannot be changed or deleted so no copying is necessary for them when they are assigned, only the entry address is handed across. Arrays in CPL1 are always copied on assignment as this seems to cause least confusion to programmers (although in CPL no copying is done unless the function '*Copy*' is invoked). In CPL1 strings are not copied and the system never collects the disused storage space. The programmer must organise his own garbage collection.

Standard routines and functions

A full set of standard routines and functions is included with the compiler; they do not necessarily correspond with those in the CPL reference manual. The '*Input*' and '*Output*' routines accept any number of arguments of any type. '*Output*' will output its arguments in a standard format determined by their type. Other output routines allow the format to be specified, and further flexibility can be achieved by using the string generating functions. Full specification of the standard routines and functions are given in Coulouris and Goodey (1966).

Predeclared labels

During execution of a program, faults such as accumulator overflow and array bound violation can occur. When this happens the program jumps to a predefined label variable where monitoring action is taken. By assigning labels in his program to these predefined label variables the programmer can arrange to take his own action when the fault occurs. All label jumps are indirect jumps through a given store location, and a label assignment simply writes a new address into this location.

Run-time postmortem

When compiling the program the compiler prints a program 'map' which assigns to each block of the program a block number. If the program fails in execution then a postmortem is printed. This contains a one-line identification of the fault and a list of blocks that the program is dynamically inside. With each block is printed a list of the local identifiers together with their current right-hand values printed in their standard output formats.

There is also a 'block trace' mechanism which prints out each block or routine entry and exit. This can be turned on and off dynamically.

Acknowledgements

The authors would like to extend their thanks to the Director of the University of London Institute of Computer Science, Prof. R. A. Buckingham, for making the work possible, and to the staff of the Atlas Computing Service for making their interaction with the machine somewhat less painful.

Thanks are also due to Dr. Derek Morris for his unfailing help with the Compiler Compiler.

References

(1) Barron, D. W., Buxton, J. N., Hartley, D. F., Nixon, E., and Strachey, C. (1963). The main features of CPL, *Computer Journal*, Vol. 6, pp. 134–143.

(2) University of London Institute of Computer Science and the Mathematical Laboratory, Cambridge, Technical Report, 'CPL Working Papers', July, 1966.

(3) Coulouris, G. F., and Goodey, T. J. (1966). *The CPL1 System Manual*, University of London Institute of Computer Science.

(4) Coulouris, G. F. (1967). Principles for implementing useful subsets of advanced programming Languages (in *Machine Intelligence* 1, Ed. by N. L. Collins and D. Michie, Edinburgh: Oliver and Boyd).

(5) Brooker, R. A., MacCallum, I. R., Morris, D., and Rohl, J. S. (1963). The Compiler Compiler, *Ann. Rev. Automatic Programming*, Vol. 3, Pergamon Press.

(6) Barron, D. W., and Strachey, C. (1966). Programming (in *Advances in Programming and Non-Numerical Computation*, ed. L. Fox, London: Pergamon Press).

Book Review

Procédures Algol en Analyse Numérique; 324 pages. (Published by Min. de l'Education Nationale, 35 F.)

This book, whose purpose is to provide useful ALGOL procedures for scientific computing, and to encourage the general use of ALGOL, is a combined effort by six French Universities organised by the National Centre for Scientific Research. There are seven chapters, with titles Linear Algebraic Equations (13 *procedures*), Algebraic Eigenvalue Problem (17), Algebraic and Non-linear Systems (11), Differential Systems, Integral and Integro-differential Equations (6), Definite Integrals (10), Approximation (13), Probability and Special Functions (8).

The two linear algebra chapters contain much standard material, such as variants of Gauss and Cholesky for solution of linear equations and matrix inversion, and methods for the eigenvalue problem associated with the names of Jacobi, Givens, Householder, Sturm, Rutishauser, Hessenberg, Wilkinson, Hyman, Laguerre and Newton. In addition there are processes for the termination of iterations based on consideration of a 'neighbouring' problem; least squares methods including that of Golub and Businger; the determination of pseudo inverses; power methods and deflation for eigenvalues; and Jacobi for complex Hermitian matrices. Chapter 3 includes for polynomials the methods of Newton (real and complex) and Laguerre, and those of Lin and Bairstow for finding quadratic factors. For more general functions the method of Muller is programmed, and 'bisection' and Newton iteration are used for single and simultaneous non-linear equations.

Chapters 5 and 6 include programs which probably exist in few other computer installations. The quadrature procedures determine matrices connected with polynomial and trigonometric interpolation, and tensor products of such matrices, and use them for quadrature along a line and over a rectangle, with error estimation based on the interpolating function. Romberg integration is also extended from the line to a rectangle and a parallelepiped. Chapter 7 relates only to initial-value problems, but covers systems of first and second order differential equations, Volterra integral equations and Volterra integro-differential equations of first and second orders. The main techniques are varieties of Runge-Kutta processes recently developed in France.

Chapter 7 also breaks new ground. Six procedures find good or best approximations, using the maximum norm, for continuous functions under a miscellany of conditions and constraints, both discrete and continuous. The Remes algorithm is the basic tool. Four procedures use a least squares norm, and four others produce spline approximations of general and particular orders. In the final miscellaneous chapter we find an additional Runge-Kutta error-minimising procedure, two sections on Mathien functions, one on the inversion of the error function, two on Markoff chains, and two on random sequence generators.

Each section comprises a 'Notice', with information about the program, method, and relevant literature; the '*Procedure*', containing the ALGOL instructions; and an 'Exemple d'utilisation', with problem, program and numerical results obtained. Each chapter also has an introduction, with some numerical analysis, a summary of the procedures and an evaluation thereof, and further references. Finally, each chapter and even each *procedure* have named responsible authors, to permit 'the establishment, of a fruitful dialogue between the authors and the readers (suggestions, criticisms, requests for clarification, etc').

Extreme accuracy, of course, is a necessity for a work of this kind, and the stated checks and methods of producing the printed pages give confidence that this has been achieved. This is undoubtedly an important and practically useful publication, and the team responsible for it deserve our congratulations and thanks for a good idea splendidly carried out.

L. Fox (Oxford)