The main features of CPL

By D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon and C. Strachey

The paper provides an informal account of CPL, a new programming language currently being implemented for the Titan at Cambridge and the Atlas at London University. CPL is based on, and contains the concepts of, ALGOL 60. In addition there are extended data descriptions, command and expression structures, provision for manipulating non-numerical objects, and comprehensive input-output facilities. However, CPL is not just another proposal for the extension of ALGOL 60, but has been designed from first principles and has a logically coherent structure.

1. Introduction

This paper provides an informal description of the main features of CPL, a programming language developed jointly by members of the University Mathematical Laboratory, Cambridge, and the University of London Computer Unit. (CPL is mnemonic for Combined Programming Language.) The object in developing CPL was to produce a language which could be used for all types of problem, numerical and non-numerical, and would allow programmers to exploit all the facilities of a large and powerful computer without having to "escape" into machine code.

CPL is to a large extent based on ALGOL 60 (a knowledge of which is assumed), and we readily acknowledge our debt to the authors of the ALGOL Report (Naur, 1963). The publication of this report marked a turning point in the development of programming languages, since it concentrated attention on, and to a large extent solved, the problems of unambiguously defining a computational process or algorithm. Unfortunately, in the present state of development of computers, a precise definition of an algorithm is not equivalent to a computer program for the performing of that process. An example of this difference is in the precision of arithmetic operations; an algorithm assumes that all operations are carried out to a sufficient precision, whereas the actual computer program must specify for each operation whether it is to be done single length, double length, etc. For this reason ALGOL 60 is not entirely satisfactory as a programming language, and in CPL we have endeavoured to produce a language which allows the programmer to retain contact, where it is necessary, with the realities of an actual computer; we have also extended the forms of data description to allow manipulation of a variety of non-numerical objects. The result is a language which is machine-independent, but is oriented towards actual computers. There are some points in the language at which account must be taken of the actual object machine on which programs will run, and here we have had in mind the machines on which CPL will first be implemented, the Atlas at London, and the Titan at Cambridge. In those features which affect the language (e.g. word length and arithmetic facilities) these two machines are identical, and

we shall therefore refer subsequently to the Atlas implementations, without distinguishing the two. During the development of CPL various general principles have evolved, and as these have been recognized the language has been refined to conform to them, with the result that it forms (we hope) a logically coherent and unified structure, with a minimum of *ad hoc* rules. One important principle that has guided us is the essential unimportance of syntax. By this we mean that although the syntax of the language is of great importance to the user and to those who construct compilers, it is of little importance whilst the content of the language is being determined. To put it less formally, one should decide what one wants to say before deciding how to say it. Following this principle, the present paper is devoted to a semantic account of the main features of CPL; it does not pretend to be complete, nor should the examples be regarded as precise definitions of the syntax. A definitive description of the language, with formal syntax, is in preparation and will be published later.

2. Structure of a CPL program

A CPL program is made up of *definitions* and *commands*. The definitions may be regarded as instructions to associate certain names with data items, data structures, functions, or process descriptions. The commands may be regarded as instructions to perform some evaluation and/or some rearrangement of the information held in the computer. Programs are divided into subsections called *blocks*; in the simplest case a block consists of a series of definitions which are activated simultaneously, followed by a series of commands which are executed in sequence (unless a command specifies a change of sequence). Blocks may be nested within other blocks. This type of block will be familiar from ALGOL; the greater power and flexibility of the CPL block is described in Section 20.

Definitions and commands are made up of *expressions*. Much of the power of CPL comes from the number of different kinds of expression, and the ways in which they can be combined; in this the language exemplifies the trend towards more complicated expressions embodied in a very few basic command forms.

134

3. Items and types

The CPL programming language is concerned with the handling of basic items of information of various types. The items most often encountered are numbers: these may be of the types **real, integer,** and **complex.** An item of the type **real** is a real number held to the precision and within the range permitted by any particular implementation of the language, an item of type **integer** is an integer within the permitted range, and a **complex** item is an ordered pair of **real** items. These numerical items may be held to **double** the standard precision.

Type **integer** is a machine-dependent facility. In the Atlas implementations the type **integer** will be held in the same internal representation as **real,** and arithmetic expressions involving integers will be evaluated as though they were **real,** an appropriate rounding function being invoked on assignment to an **integer.** An **index** variable is an integer within the range that can be handled in the B-registers (index registers) of the object machine: it is introduced to speed up some indexing operations. Double-precision working and complex arithmetic may not be implemented in the first compilers.

The non-numerical types include **Boolean, logical, long logical, string** and **label.** A **Boolean** item is a truth-value, a **logical** item is a binary pattern of the size permitted by the implementation (24 bits on Atlas), and a **long logical** item is a similar binary pattern to be used where relevant to the implementation (i.e. 48 bits on Atlas). A **string** item is a sequence of symbols from the CPL alphabet, and a **label** item is a command label as defined below in Section 13.

There is also a type **general** which designates an item whose type is not fixed and may, therefore, vary at run time. Further types may be introduced in later versions of CPL.

4. Names

The items of information which a CPL program handles are identified by *names,* which all obey the same rules. There are two kinds of name, large and small, and either kind may be used for any purpose.

A *small name* is a single lower-case letter, possibly followed by one or more primes. x, y', y'' are small names.

A *large name* is an upper-case letter, possibly followed by a string of letters and digits, possibly followed by one or more primes. A, $Ab3C'$, ABC are large names. A large name which is not ended by one or more primes must be followed by

(i) any printed character other than a letter or digit

or (ii) a space or newline

or (iii) an underlined letter or digit.

As in ALGOL, underlined words* are basic symbols

* For typographical reasons bold type is used in printed documents and is synonymous with underlined words in handwritten or typewritten documents.

of the language, and underlined letters and digits only occur in this context (except in strings): no distinction is made between upper and lower-case letters in underlined words.

5. Constants

In addition to variables known by names, constants may appear in the language. *Numerical constants* may be written in any of the usual forms, using the decimal notation. 59, $+8 \cdot 76$, $-0 \cdot 35_{10} - 15$ are numerical constants.

Logical constants are bit patterns expressed in binary or octal form as indicated by a preceding **2** or **8.** The pattern is assumed to be right justified; however, it may be shown to be justified from the left or right by the presence of a bar on the left or right of the constant. Examples of logical constants are:

$$8\ 777 \qquad 8\,|\,55 \qquad 2\ 111000\,|$$

Boolean constants are the basic symbols **true** and **false.** A *string constant* is represented by a sequence of symbols enclosed in string quotes: ' and '. The symbols are those that can be contained in a single print position on the paper including spaces, with a special notation for specifying *Newline, Tab,* etc.

6. Expressions

There are two possible modes of evaluation of an expression in CPL, known as the *left-hand* (LH) and *right-hand* (RH) modes. All expressions can be evaluated in RH mode, but only certain kinds of expression are meaningful in LH mode. When evaluated in RH mode an expression is regarded as being a rule for the computation of a value (the RH value). When evaluated in LH mode an expression effectively gives an address (the LH value): the significance of this is discussed further in Section 8.

7. Numerical expressions

A *numerical expression* yields a rule for the computation of a numerical value. It is constructed from constants and variables which may be of mixed numerical types, joined by the customary arithmetic operators. Multiplication may be expressed implicitly and parentheses are used in the normal way. Note that a large name must be terminated by at least one space if followed by another name or constant, and the two are not separated by an operator.

The following precedence rules apply. The operators $\times$, $/$, $\div$ and $\uparrow$ are of equal precedence and associate from right to left. Hence the expression

$$ab/c \uparrow de$$

is equivalent to

$$(a(b/(c \uparrow (de))))$$

The infixed operators $+$ and $-$ are the less binding and associate from left to right.

When + and — are prefixed (i.e. monadic) they are treated as if replaced by (+1) and (−1) respectively (hence they have the precedence of a multiplication and associate to the right). Prefixed + and — are all those which do not occur immediately after a name, constant or closing bracket.

Thus the expression

$$a/bc - d + ef/ -g \uparrow -h$$

is equivalent to

$$(a/(bc) - d) + (e(f/(-(g \uparrow (-h)))))$$

or

$$\frac{a}{bc} - d + \frac{ef}{-g^{-h}}$$

This system of precedence rules and association rules has been adopted because it seems to lead to close correspondence, in most cases, between the meaning of an expression as interpreted by the CPL rules and its interpretation by a mathematician when read on the paper. This is most easily seen in the case of terms such as a/bc.

The following are examples of numerical expressions:

$$b \uparrow 2 - 4ac$$

$$Second \uparrow 2 \cdot 0 - 4 \; First \; Third$$

$$Par1 \times Par2 - Par1 \times Par3$$

Variables of different numerical types may be mixed freely in numerical expressions, and the apparent type of any constant introduced does not affect the type of arithmetic carried out. The arithmetic in which an expression is evaluated is a function of the types and precisions of the terms in the expression. The meaning of each operator is dependent on the operands which it relates. The precision is the highest precision of the operands, and the type is the highest type as given by the following hierarchy:

> **complex**
>
> **real**
>
> **integer**
>
> **index**

Suitable transfer functions are inserted automatically as required by these rules.

Occasionally one may wish to circumvent these rules, for example when multiplying two single-precision numbers to obtain a double-length result. In these circumstances the special operators **plus, minus** and **mult** may be used; their effect is to produce a result whose precision is greater than the precision of the operands.

8. Left-hand expressions and assignment commands

An expression which is evaluable in LH mode produces as part of the result of evaluation an address (or a reference to some member of a data structure). The simplest possible expression evaluable in LH mode is, therefore, a variable name.

An *assignment command* is an instruction to the computing system to assign, to the "address" specified by the LH value of the left-hand side, the value obtained from evaluating the RH value of the right-hand side. A simple example is

$$x := yx$$

Any expression which can be evaluated in LH mode is meaningful on the left-hand side of an assignment command.

9. Boolean and logical expressions and assignment commands

Expressions may be built up from Boolean variables, parentheses, constants and the following operators (in descending order of binding power).

$\sim$	(not)
$\wedge$	(and)
$\vee$	(or)
$\equiv$	$\not\equiv$
imp	(implies)

The infixed operators associate from left to right.

A *Boolean expression* yields a rule for obtaining a Boolean value. A Boolean assignment command is an instruction to the computing system to assign to the (Boolean) variable specified by the left-hand side expression the (Boolean) value specified by the right-hand side expression.

An *arithmetic relation* is a pair of arithmetic expressions separated by one of the relational operators $< \leqslant = \neq \geqslant >$. It is in fact a Boolean expression having the value **true** or **false** and may occur as a term in Boolean expressions.

An extended arithmetic relation is introduced; for example, a relation such as

$$a \leqslant b < c$$

is interpreted as $(a \leqslant b) \wedge (b < c)$

Logical expressions and assignment commands have the same appearance as their Boolean counterparts and in fact have identical syntax, with the exception that arithmetic relations cannot be introduced. Their meaning, however, is concerned with manipulation of the bit-patterns which are the logical values. For example, if a, b, c are **logical** variables, the command

$$a := b \wedge c$$

means: assign to the variable a the bit-pattern which has 1-bits wherever the corresponding positions of b and c both have 1-bits, and which has 0-bits in all other positions.

10. Conditional expressions

Any type of expression described above may be conditional. The form adopted for such expressions is

$$B \to E_1, E_2$$

where B denotes a Boolean expression, and E_1 and E_2 are expressions which may themselves be conditional. If B is **true**, the value of the expression is the value of E_1; if B is **false** then it is the value of E_2. Conditional expressions of a more elaborate form are possible:

$$B_1 \to E_1, B_2 \to E_2, \ldots B_n \to E_n, E_{n+1}$$

In this case the Boolean components are evaluated from left to right until a **true** one is found, and the following expression is taken as the value of the whole expression. The value of E_{n+1} is taken if none of the Boolean components is **true**.

Normally the types of the expression components will be the same, but this is not mandatory. A conditional expression may be written as part of a larger expression by enclosing it in round brackets.

11. Compound commands, blocks and simple definitions

A sequence of commands may be grouped, by enclosing them within the *section brackets* § and §, to form a *compound command*, the whole then being equivalent to, and a syntactically valid replacement for, a single command.

A *block* consists of one or more definitions followed by one or more commands, the whole being enclosed within section brackets. Its principal purpose is to introduce a new level of nomenclature as in ALGOL. Blocks may be nested in the ALGOL manner.

The most elementary form of definition which exists in the language is the definition of simple variables. This takes the form of type followed by a list of names. Thus, the definitions

real a, b, c; **index** x; **logical** *Mask*

at the start of a block will define **real** variables a, b and c, **index** variable x and a **logical** variable *Mask*, the scope of these being the body of the block.

A pair of matching section brackets may be identified by adding a sequence of letters, digits and decimal points, e.g.

§1.2.1 . . . §1.2.1

If compound commands are nested, then a closing section bracket of this form automatically terminates any compound commands opened between it and its matching opening section bracket. Section brackets are also used in other contexts (see Section 20 below), and the above method of identification is useful for distinguishing one set from another.

12. Initialized definitions

The concept of an *initialized definition* is introduced in CPL. One of the three possible forms which this may take is 'initialization by value'; by this means, when a variable is defined, it may be assigned an initial value obtained by evaluating an expression (names occurring in this expression are global to the definition).

For example

> **real** $z = 15A - 37 \cdot 2$
>
> **index** $x = 128$; **logical** *Mask* = 8 77707

Note that in these definitions A is a global variable which must have had a value assigned to it prior to this definition.

In ALGOL 60 a block may only be entered at its head. In CPL a block may be labelled, and then a transfer may be made to any labelled command in the block (the way in which this is expressed is given in the next Section); the effect is that the definitions and declarations (see Section 20) at the heads of any blocks entered are activated before the transfer into the body of an inner block becomes effective.

13. Labels, designational expressions, and transfer commands

Any command or block may be labelled by a name. A *designational expression* is a rule for finding such a label; a simple expression may consist of a label or of a variable of type **label**, whose value is a label. Such an expression may occur on the right-hand side of an assignment command whose left-hand side specifies a **label** variable, or in a transfer command.

The main uses for **label** variables are in the setting of links and switches. For example, a switch could be set up by the initialized definition

label *Switch* $= x < 0 \to L1, L2$

Then, the transfer command

go to *Switch*

will cause a transfer of control to one of the commands labelled $L1$ or $L2$ dependent on the value of x when the switch was defined.

The value of a **label** variable is a label, together with the name of the block in which the label is defined. Normally, a label is only used in this scope, but if it is desired to refer to a label outside its scope, as for example when jumping into the middle of a block, the label of the block must be prefixed to the label, thus: B **at** L. When control is transferred into the middle of a block from outside, the definitions and declarations at the head of the block are activated before any commands in the block are executed.

14. Conditional commands

There are three forms of conditional command:

> **if** ⟨Boolean expression⟩ **then do** ⟨command⟩   *
>
> **unless** ⟨Boolean expression⟩ **then do** ⟨command⟩
>
> **test** ⟨Boolean expression⟩ **then do** ⟨command⟩
>
> **or do**   ⟨command⟩

* It is assumed that the reader is familiar with this notation for expressing syntactic forms, which is used in the ALGOL Report.

The **if** command is similar to the ALGOL form, while the **unless** command is equivalent to **if not** . . . The **test** command gives the facilities of the ALGOL **if** . . . **then** . . . **else,** but it avoids ambiguity when the **then do** is followed by a conditional command, since the **or do** followed by a command is mandatory. Thus there is no restriction on the form of the commands following the symbols **then do** and **or do**: either may be conditional.

15. Cycles

The purpose of a *cycle command* is to organize the repetition of a command (usually compound) called the *body,* a certain number of times. The repetition may be for an indefinite number of times, controlled by a Boolean expression:

> **while** ⟨Boolean expression⟩ **do** ⟨command⟩
>
> **until** ⟨Boolean expression⟩ **do** ⟨command⟩

These commands will cause repetition of the body for as many times as the Boolean expression remains **true** or **false** respectively. If on entry the value is otherwise, *no activation of the controlled command occurs.* The forms

> ⟨command⟩ **repeat while** ⟨Boolean expression⟩
>
> ⟨command⟩ **repeat until** ⟨Boolean expression⟩

cause at least one activation of the controlled command, which is the shortest command that can be found scanning backwards from the **repeat.**

The third type of cycle command is the **for** command, in which a precise specification is given of the number of repetitions and the values of a controlled variable. The controlled variable values may be given as an explicit list:

> **for** $v = 1, 3, 6, 10$ **do** ⟨command⟩

or by any list expression (see Section 23) which defines a group of values. The specification of the controlled variable and the list of values are similar to an initialized definition, so that in the above example, v is a variable whose scope is the body of the **for** command. A particularly useful list expression is

> **step** E_1, E_2, E_3

where E_1, E_2, E_3 are numerical expressions. Thus

> **for** $v = $ **step** E_1, E_2, E_3

is similar to the ALGOL

> **for** $v := E_1$ **step** E_2 **until** E_3

The **for** list may be made up by concatenation, e.g.

> **for** $v = $ **step** $0, 1, 10$ **then step** $40, 2, 60$

During execution of the **for** body, the controlled variable may be altered by assignments to it. Regardless of such changes, on entry to the next repetition the controlled variable will be set to the next value in the correct repetition sequence.

Values in the **for** list are given by expressions of a type corresponding to the controlled variable. Any such expressions are notionally evaluated on entry to the **for** command.

The body of a **for** command may be regarded as resembling the body of a block, with the repetition definition playing the same role as the initialized definitions at the head of a block. Transfers into the body are therefore allowed in the same way as transfers into a block; all expressions in the repetition description are evaluated, and the controlled variable is set to the first value of the repetition list before the transfer becomes effective.

16. Other forms of initialization

We have already introduced one form of initialization for variable definitions. This is characterized by the '=' sign and denotes initialization by value. There are two other forms: 'by reference' (denoted by ≃) and 'by substitution' (denoted by ≡).

When a variable is initialized by reference its LH value is set to be the LH value of the right-hand expression. For example, if A is a one-dimensional array (see Section 22), the definition

$$\textbf{real } x \simeq A[i]$$

makes x equivalent to the variable $A[i']$ where i' is the value of i at the moment of definition. Thus, the value of x is the value of $A[i']$ and an assignment to x is an assignment to $A[i']$. Clearly, the right-hand side expression must yield an LH value.

Initialization by substitution denotes that a variable is equivalent to the right-hand side expression, which must be evaluated afresh each time the variable is used. Thus

$$\textbf{real } x \equiv a + b$$

essentially means that $(a + b)$ must be substituted for each occurrence of x. Also

$$\textbf{real } y \equiv A[i]$$

makes the value of y the same as the value of $A[i]$, and thus, as i changes during the program, then so does the LH value of y.

17. Function definitions and calls

Functions are used in CPL to specify, in complicated ways, the computation of a value. Examples of a function definition are

> **function** $F[x, y] = ax \uparrow 2 + 2bxy + cy \uparrow 2$
>
> **index function** G [**index** i, j] $= i(i - 1) + j(j - 1)$

The left-hand side of the definition gives the name of the function and a list of formal parameters, and the right-hand side is an expression. The type of the result of the function may also be written on the left-hand side, otherwise this takes the type of the right-hand side expression. The formal parameters may have type

138

specifiers: in their absence, a parameter has the type of the previous one. If the first parameter in the list is unspecified then it is understood to be **real**. Parameters of type **routine** in general are not allowed, since functions cannot have side effects (routines are discussed in Section 19 below).

A *function call* consists of the function name followed by a list of actual parameters enclosed in square brackets. If these do not correspond in type to the corresponding formal parameters in the definition of the function, suitable transfer functions are inserted if this is possible. All actual parameters may be expressions; all actual parameters are evaluated at the moment of call.

A function call is a call for the computation of a value, the rules of computation being defined by the expression on the right-hand side of the definition. As such it may be used as a primary in expressions of suitable type, thus:

$$3p \uparrow 2 + f[p, q] - 3f[q, (3p \uparrow 2 \cdot 4 - q)]$$

The treatment of non-formal parameters occurring in the definition is controlled by the sign which separates the left- and right-hand sides of the definition. If this is = as in the above examples, the values of non-formals are taken at the time of declaration; if the sign is ≡, they are evaluated at the time of call. In effect, the connecting sign specifies the initialization method to be applied to the non-formal parameters, and a function definition may be regarded as another kind of initialization procedure.

In addition to single values, the results of functions may be lists, arrays (see later) and functions themselves, but not routines.

18. "Result of" expressions

These provide a means of forming an expression from a compound command or block. The body must contain an assignment to the special variable **result**, and the value assigned is the value of the whole expression. Other local variables may be defined, and the body may not contain operations, such as assignments to non-local variables, which would cause side effects.

The form of a **result of** expression is shown in the example below.

$x := \textbf{result of } \S \textbf{ real } p; p := yy - z; \textbf{result} := \text{Sin}[p] - pp \S$

A frequent use of **result of** expressions is in the right-hand sides of function definitions. This notation was first suggested by P. J. Landin; it is described in Strachey and Wilkes (1961).

19. Routine definitions and calls

A *routine* is a way of defining for frequent use a complicated piece of program which may produce many results. An example of a routine definition is

> **routine** *Work* [**real** *a*, *b*, *c*, **index** *d*, **label** *e*]
>
> > **value** *a*, *e*; **ref** *c*; **subst** *b*, *d*
> >
> > ⟨command⟩

The formal parameter list and specifications take the same form as in function definitions, except that parameters of type **routine** may be included. In addition to the types of the formal parameters, it is also necessary to specify the way in which each parameter is to be called. Three modes of parameter call are possible: call by **value** (which is equivalent to the ALGOL call by value), call by **substitution** (equivalent to ALGOL call by name), and call by **reference**. In the latter case, the LH value of the actual parameter is handed over: this corresponds to the "call by simple name" suggested by Strachey and Wilkes (1961). Note the correspondence to the three kinds of initialization.

As the definition takes the form of a command (which may be compound), a *routine call* is a command consisting of the routine name and an actual parameter list:

$$Work \; [p, q, r, s, t]$$

20. Scopes, definitions, and declarations

In the ALGOL block, the declarations at the head of the block are regarded as being performed simultaneously on entry to the block, and the scope of the declared items is the body of the block. A more sophisticated system has been introduced in CPL to cope with initialized definitions and with complicated recursive definitions.

A series of definitions may be grouped into a *declaration* by enclosing them within section brackets thus:

> **dec** § **real** *x*, *y*; **index** *p* = 4; **logical** *L* = **8** | 74 §

The definitions thus grouped are regarded as being activated simultaneously. A block head may contain several such declaration paragraphs, in which case the paragraphs are activated sequentially. Isolated definitions which are not grouped are treated as if contained within brackets; that is, as a declaration.

The *scope* of the items defined in a declaration is the body of the block as in ALGOL, but in addition it includes all subsequent declarations in the block head. If a definition or compound definition is preceded by the symbol **rec**, this scope is further extended to include any right-hand sides and routine bodies of the definition.

21. The "where" clause

The **where** clause provides a means of adding local definitions to commands and expressions, for example

$$p := ax \uparrow 2 + bx + c/x \textbf{ where } x = 2a \uparrow 2 + b$$

A **where** clause following a command qualifies the largest immediately preceding command, and its body is obeyed before the command is obeyed. Otherwise a **where** clause qualifies the largest preceding expression, and is obeyed before this expression is evaluated.

By its use actual parameters for a function or routine call may be declared local to a call; e.g.

$$Quad \; [a, b, F] \textbf{ where } F[x] = G[x, y]$$

139

Where clauses may themselves be modified by other **where** clauses, and more than one definition may be made using the construction

$$\textbf{where} \ \S \ldots ; \ldots \S$$

The **where** clause essentially gives the facilities of the Lambda calculus (Church, 1941) and has been derived from the Auxiliary Equations feature of the GENIE language (Iliffe, 1961).

22. Arrays

An *array* in CPL is basically similar to an array in ALGOL: a multidimensional array of items of the same type. Individual members are addressed by the array name followed by a subscript list enclosed in square brackets, and can be used throughout the language in the same way as simple variables.

However, to permit the handling of non-rectangular arrays, the method of defining and forming arrays is new. The essential feature is in the way storage for arrays is created, which is in the form of a function rather than as part of a declaration. Array variables are defined in a similar way to simple variables, the definition giving the dimensionality and type of element, but not the bounds; for example

real 2 **array** *A*

defines a matrix of **real** elements. Storage for this matrix may be created by a built-in function in the following manner

Array [**real**, (1, 10), (1, 12)]

Thus, the result of this function, which is an *array expression*, may be assigned to an array variable, either by an assignment command or by an initialized definition. In the latter case, the notation can be shortened so that an array definition might be

A = *Array* [**real**, (1, 10), (1, 12)]

The symbols **vector** and **matrix** are synonyms for 1 **array** and 2 **array** respectively. At present the only forms of array expression are array variables, or functions whose results are arrays; later, it is hoped to introduce more complicated forms, e.g. direct matrix operations.

23. Lists

A *list* is a data structure consisting of an ordered group of members, similar in many respects to those of the LISP system (McCarthy, 1960). The group is dynamically variable in length and consists of items each of which is a single LH value. A **list** variable has a single LH value, and hence any element of a list may itself be a list. A **list** variable may be defined (and initialized) as follows

list *L* = *a, b, c*

This defines a list *L* with three elements; these elements

are the LH values of *copies* of the RH values of *a, b* and *c*. The right-hand side of this definition is an *explicit list*, a form which occurs at many places in the CPL language. An explicit list is written as a sequence of expressions (including list expressions) separated by commas, and brackets may be used to convert an explicit list into a list expression. For example

$$a, b, c, (d, e, f), g, h$$

is an explicit list of six members, of which one (the fourth) is a list of three members. An explicit list may always be written in place of a list expression unless this occurs as part of a larger list expression.

The symbol **then** is an infixed list-forming operator whose arguments are lists. It is more binding than a comma (which is also an infixed list-forming operator), and it concatenates the arguments. List structures may be assigned to list variables by assignment commands.

In order to specify the members of a list individually, a transfer function is provided which converts a list expression into an explicit list:

Members [⟨list expression⟩]

24. Simultaneous assignment commands

The general form of an assignment command can now be given. Normally this is an expression yielding an LH value, followed by :=, followed by an expression yielding an RH value. However, if an explicit list is written on the left-hand side then the right-hand side is either an explicit list, or a list expression; in the latter case the transfer function *Members* is automatically invoked. In this form the two explicit lists must contain the same number of members, and the command denotes a simultaneous assignment of each right-hand member to the corresponding left-hand member. Thus, if *L* is a **list** variable and *a, b, c* are **real** variables,

$$L := a, b, c$$

is an assignment to the **list** variable, while

$$a, b, c := L$$
$$a, b := b, a$$

are simultaneous assignments.

When an assignment is made to an item of type **general**, the type of the right-hand expression is assigned, as well as the value. A special form of assignment command permits either an explicit list or an array variable on the left-hand side, and a single-valued expression on the right. In this case, the command must start with the symbol **all** to show that the value of the right-hand side expression is to be assigned to each member of the explicit list or to each element of the array. For example

real *a, b, c*; **real vector** *A*
all *a, b, c* := 0
A := *Array* [**real**, (0, *n* − 1)]
all *A* := 0

25. Input and output

Input and output depend to some extent on the object machine and, in the case of the Atlas implementations, on the operating system. The following is only an outline of the facilities provided.

There are in CPL two complementary input-output schemes; one based on routines for reading and writing data items in a conventional manner, the other based on the reading and writing of files. The former scheme provides the facilities normally found in a "scientific" language, whereas the latter scheme resembles the input-output mechanism of a "commercial" language such as COBOL. The file input-out scheme is described in Section 27; the remainder of the present Section is devoted to the simpler scheme, which is adequate for any problem which does not involve a large amount of input or output.

25.1 *Input and output streams*

A program is regarded as processing data arriving in one or more *input streams*, and producing results in one or more *output streams*. In principle there can be any number of streams, though in practice there will be a limitation imposed by the object machine. The streams are identified by numbers, and the current stream is selected by the built-in routines *Input* and *Output*; thus the command

$$Input\ [3]$$

selects stream 3 for subsequent input operations, and this selection is maintained until another stream-selecting command is obeyed. Two functions, *Source* and *Destination*, provide a means for the program to discover which input or output stream is currently selected. (In the Atlas implementations the operating system sets up the correspondence between stream numbers and actual input or output devices.)

25.2 *Routines for input*

The basic routine for input of numbers is called *Read*. It has as its formal parameter an explicit list; thus

$$Read\ [a, b, c, A[i]]$$

will read the next four items from the currently selected stream and assign them to the variables named. Appropriate transfer functions are invoked if the command calls for the input of an **index** or **integer** variable. Logical constants may be read, either by preceding them on the input medium by the basic symbols 2 or 8, or by using variants of the *Read* routine called *Read* 2 and *Read* 8. Obviously the corresponding items in the parameter list must be **logical** or **long logical** variables. By using an explicit list as the formal parameter we obtain a routine with an apparently variable number of actual parameters, thus avoiding the difficulties of the KDF9 ALGOL Input-Output scheme (Duncan, 1963).

In the simplest form of the *Read* routine, numbers are assumed to appear in the input medium in a standard format, terminated by end-of-line or multiple space, and there is an error print if any characters are encountered which are out of context for the data type being read. However, if it is desired to read numbers punched in a non-standard format, traps can be specified in the *Read* routine parameter list; these send control to specified labels when particular characters are encountered.

Besides reading numbers it is necessary to be able to read single characters from paper tape, or single columns from a punched card. This is accomplished by the routine *Readsymbol*, which has an explicit list as its formal parameter. The variables making up the actual parameter list may be **index**, **logical** or **long logical** variables. (It is natural to read a symbol as an index if it is to be translated by table look-up.) Symbols read by this routine are removed from the input stream; thus it is analagous to "read tape and advance the reader." When permitted by the facilities available on the object machine, there are two useful variants of *Readsymbol*. These are the functions *Nextsymbol* and *Lastsymbol*. *Nextsymbol* produces as its value the next symbol in the stream, without removing it from the stream (which would be a side effect). *Lastsymbol* has as its value the symbol which has just been read from the stream; it is particularly useful in conjunction with the trap facility in the number reading routine.

25.3 *Routines for output*

The structure of the output system parallels very closely that for input: the routines provided are *Write*, *Write2*, *Write8*, and *Writesymbol*. Like the input routines, these each have an explicit list as a formal parameter. The *Write* routine includes in its parameter list a format descriptor, and it is also possible to put out text by writing a string constant in place of a variable name in the parameter list.

The layout of results on the printed page can be controlled by the routine

$$Layout\ [s, r, c]$$

which arranges the output of stream *s* in blocks of *r* rows and *c* columns, or by use of the routines *Space*, *Tab* and *Newline*.

26. Strings

String variables and constants have been mentioned in previous Sections: their main purpose is to facilitate the handling of textual information, mainly by the input and output routines. However, **string** is a data type and so it is permitted to have string arrays, string functions, string parameters, and assignments to string data items. A string expression can be made up from string variables, constants, and function calls; there is one infixed operator, **then**, which concatenates two strings.

The RH value of a string expression is a sequence of symbols. Built-in functions are provided for more

complex operations on these symbols, including conversion to other data types.

27. Files

A *file* is a data structure which exists in the peripheral environment of the computing system; it may be stored on paper or magnetic tape, printed paper, or any other input-output or long-term storage medium.

The formal properties of a file resemble those of a list, in that its structure may be nested to many levels and it may contain items of many types. However, the structure of a file is fixed at definition time and may not thereafter be changed. The types of data items in a file are more numerous and complex than in the rest of the language, since, for example, provision must be made for holding numerical items with different numbers of decimal characters.

Transfer of information to and from files is performed by reading and writing routines which relate the next group of items on the file to an explicit list of internal data items, inserting the relevant transfer functions. Precise specifications of these routines are currently being formulated.

28. Conclusions

The arguments in favour of an ALGOL-like language are overwhelming, and we have designed CPL in the spirit of ALGOL. A programmer trained in CPL should have no difficulty in learning ALGOL, and vice-versa. However, having decided to develop a new language there seemed little point in keeping to ALGOL in small details unless there was nothing to choose between CPL and ALGOL. In almost all cases where CPL differs from ALGOL we believe that for the average programmer there is some advantage in the CPL approach. Clearly there is a good case for adhering to a familiar concept in a programming language unless its replacement by a new concept brings about a noticeable improvement in consistency, clarity or expediency. We have introduced changes from the "traditional" patterns only when, in our opinion, such an improvement has resulted.

29. Examples

The following are examples of CPL function definitions. *Exp* produces the exponential of a real number by summing a power series, *Fact* 1 is a recursive function

for computing a factorial, while *Fact*2 produces the same result in a more conventional and efficient manner. The function *Euler* comes from the ALGOL Report (Naur, 1963).

function $Exp[x]$ = **result of**
§ **real** $s, t, r = 0, 1, 1$
$s, t, r := s + t, tx/r, r + 1$
repeat until $t < 1_{10} - 8$
result $:= s$ §

---

rec function $Fact1[x]$ = $(x = 0) \to 1, xFact1[x - 1]$

---

function $Fact2[x]$ = **result of**
§ **real** $f = 1$
until $x = 0$ **do**
$f, x := xf, x - 1$
result $:= f$ §

---

function $Euler$ [**function** Fct, **real** Eps; **integer** Tim] = **result of**
§1 **dec** §1.1 **real** Mn, Ds, Sum
   integer i, t
   index $n = 0$
   $m = Array$ [**real**, $(0, 15)$] §1.1
$i, t, m[0] := 0, 0, Fct[0]$
$Sum := m[0]/2$
§1.2 $i := i + 1$
 $Mn := Fct[i]$
 for $k =$ **step** $0, 1, n$ **do**
 $m[k], Mn := Mn, (Mn + m[k])/2$
 test $Mod[Mn] < Mod[m[n]] \wedge n < 15$
  then do $Ds, n, m[n+1] := Mn/2, n+1, Mn$
  or do $Ds := Mn$
 $Sum := Sum + Ds$
 $t := (Mod[Ds] < Eps) \to t + 1, 0$ §1.2
repeat while $t < Tim$
result $:= Sum$ §1.

References

CHURCH, A. (1941). *The Calculi of Lambda-Conversion*, Princeton University Press, Princeton, N.J.
DUNCAN, F. G. (1963). "Input and output for ALGOL 60 on KDF 9," *The Computer Journal*, Vol. 5, p. 341.
ILIFFE, J. K. (1961). "The use of the GENIE System in numerical calculation," *Annual Review in Automatic Programming*, Vol. 2, Pergamon Press.
MCCARTHY, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," *Comm. A.C.M.*, Vol. 3, p. 184.
NAUR, P. (Ed.) (1963). "Revised Report on the Algorithmic Language ALGOL 60," *The Computer Journal*, Vol. 5, p. 349.
STRACHEY, C., and WILKES, M. V. (1961). "Some Proposals for Improving the Efficiency of ALGOL 60," *Comm. A.C.M.*, Vol. 4, p. 448.

Book Reviews

A Guide to ALGOL Programming, by DANIEL D. MCCRACKEN, 1962; 106 pages. (New York: *John Wiley and Sons Inc.* London: *John Wiley and Sons Ltd.*, 30s.)

This book is highly recommended for the person who wants to get a rapid grasp of the use of a computer in the solution of problems in science and engineering.

The first chapter is devoted to a general description of the fields in which computers may be used, and the necessary steps to be followed in solving a problem on a computer. The notation of flow charts is introduced, and finally an account of the origin and purpose of ALGOL 60 is given, together with a brief explanation of the necessity for translation of ALGOL programs into machine code. The chapter forms an excellent introduction to the subject of computing; there remains only the minor criticism that the use of flow charts is perhaps unnecessary in view of the elegance of the conditional statements and **for** statements of ALGOL. In fact, the later explanation of the relationship between flow charts and ALGOL conditionals tends to obscure the greater simplicity of the latter.

The description of the ALGOL language given in subsequent chapters is exceptionally clear and thorough; but the book offers far more than an account of the language. As each new facility is introduced, its purpose is explained, and practical advice is given on its proper use. This advice covers points of program efficiency and the validity of the numerical methods used. For example, the chapter on **for** statements mentions the danger of testing for absolute convergence, and suggests the proper method of testing for relative convergence; it also encourages the removal of unnecessary calculations from inner loops. Occasionally a warning is given that certain translators may not accept the notation described, but insufficient notice is given in cases where facilities have been widely rejected by implementors. In particular, the use of integer labels should have been discouraged from the start.

An excellent feature of the book is the large number of worked examples, each of which traces through all the stages in the construction of a well-designed program. These examples discuss and solve questions of problem analysis, design of input and output, and the use of computer storage. The exercises and their answers give further examples of the reasoning needed in the construction of good programs, not merely programs that work. After a study of this book, many scientists and engineers should be competent to use a computer for their own problems, with little or no assistance from an experienced programmer.

C. A. R. HOARE.

Input Language for Automatic Programming Systems, by A. P. YERSHOV, G. I. KOZHUKHIN, U. M. VOLOSHIN, 1963; 70 pages. (London: *Academic Press*, 35s.)

The spectre of ALGOL, which has haunted Europe since 1958, at last shows signs of being laid to rest. The third volume of the A.P.I.C. Studies in Data Processing contains a description of a source language designed in the Soviet Union for mathematical procedure description. Though ALGOL-like in some respects it departs sufficiently from the spirit of the original to be counted as a new venture in automatic programming.

The departure is away from pure sequential procedure description towards mathematically convenient forms. In an excellent introduction to the English edition, R. W. Hockney summarizes the most significant of these, which I will comment on here. First let it be said, however, that what might be called the body of the book, consisting of a sentence-by-sentence revision of the ALGOL 60 report, is only "readable" in the sense of a work of reference to the fine structure of the Input Language, so one could hardly call this a work of value for anyone but a specialist steeped in ALGOL lore.

The first modification is in the use and definition of arrays. Conventional matrix operations are permitted in expressions, and a meaning is given to the less conventional forms which can arise (though I think better use could have been made of these varieties). Multi-dimensional arrays can be declared in the usual way, but in addition the elements of an array can themselves be declared to have an array form. New arrays, moreover, can be formed by adding old ones "in parallel" (increasing the dimensionality) or "in series" (increasing the length in one dimension). Elaborate notations are developed for referring to elements or sub-arrays. All this seems very attractive for mathematical work, but there is one unaccountable drawback, that an array must at all times keep the shape of a hyper-rectangle. My experience in using a scheme of this type is that its greatest use is in representing a complete system of programs and data in an array form, which is certainly much less regular than that demanded by the Input Language. Using codeword techniques (Iliffe and Jodeit, *The Computer Journal* Vol. 5, p. 200) this facility is easily gained, and it has obvious applications in both mathematical and data-processing work.

One other significant departure is towards what might be called initialized declarations, in which a value may be assigned to a variable at the same time as it is declared. A special case, a function-expression declaration, is included