BCPL: A tool for compiler writing and system programming

by MARTIN RICHARDS*

University Mathematical Laboratory
Cambridge, England

INTRODUCTION

The language BCPL[1] (Basic CPL) was originally developed as a compiler writing tool and as its name suggests it is closely related to CPL[2,3] (Combined Programming Language) which was jointly developed at Cambridge and London Universities. BCPL adopted much of the syntactic richness of CPL and strived for the same high standard of linguistic elegance; however, in order to achieve the efficiency necessary for system programming its scale and complexity is far less than that of CPL. The most significant simplification is that BCPL has only one data type—the binary bit pattern—and this feature alone gives BCPL a characteristic flavour which is very different of that of CPL and most other current programming languages.

BCPL has proved itself to be a very useful compiler writing tool and it also has many qualities which make it highly suitable for other system programming applications.

We will first outline the general structure of BCPL and later discuss how well it is suited to applications in the fields of compiler writing and system programming.

The language

BCPL has a simple underlying semantic structure which is built around an idealised object machine. This method of design was chosen in order to make BCPL easy to define accurately and to facilitate machine independence which is one of the fundamental aims of the language.

The most important feature of the object machine is its store and this is represented diagrammatically in Figure 1. It consists of a set of numbered boxes (or storage cells) arranged so that the numbers labelling adjacent cells differ by one. As will be seen later, this property is important.

Each storage cell holds a binary bit pattern called an Rvalue (or Right hand value). All storage cells are of the same size and the length of Rvalues is a constant of the implementation which is usually between 24 and 36 bits. An Rvalue is the only kind of object which can be manipulated directly in BCPL and the value of every variable and expression in the language will always be an Rvalue.

Rvalues are used by the programmer to model abstract objects of many different kinds such as truth values, strings and functions, and there are a large number of basic operations on Rvalues which have been provided in order to help the programmer model the transformation of his abstract objects. In particular, there are the usual arithmetic operations which operate on Rvalues in such a way that they closely model integers. One can either think of these operations as ones which interpret their operands as integers, perform the integer arithmetic and convert the result back into the Rvalue form, alternatively one may think of them as operations which work directly on bit patterns and just happen to be useful for representing integers. This latter approach is closer to the BCPL

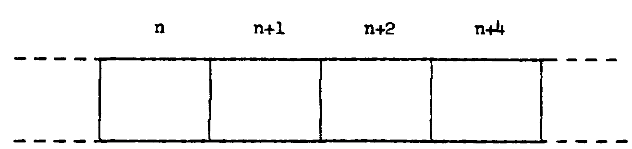

Figure 1—The machine's store

* The work was started while the author was employed by Massachusetts Institute of Technology. It was supported, in part, by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

philosophy. Although the BCPL programmer has direct access to the bits of an Rvalue, the details of the binary representation used to represent integers are not defined and he would lose machine independence if he performed non numerical operations on Rvalues he knows to represent integers.

An operation of fundamental importance in the object machine is that of Indirection. This operation has one operand which is interpreted as an integer and it locates the storage cell which is labelled by this integer. This operation is assumed to be efficient and, as will be seen later, the programmer may invoke it from within BCPL using the *rv* operator.

Variables and manifest constants

A variable in BCPL is defined to be a name which has been associated with a storage cell. It has a value which is the Rvalue contained in the cell and it is called a variable since this Rvalue may be changed by an assignment command during execution. Almost all forms of definition in BCPL introduce variables. The only exception is the *manifest* declaration which is used to introduce manifest constants.

A manifest constant is the direct association of a name with an Rvalue; this association takes place at compile time and remains constant throughout execution. There are many situations where manifest constants can be used to improve readability with no loss of runtime efficiency.

Lvalues and modes of evaluation

As previously stated each storage cell is labelled by an integer; this integer is called the Lvalue (or Left hand value) of the cell. Since a variable is associated with a storage cell, it must also be associated with an Lvalue and one can usefully represent a variable diagrammatically as in Figure 2.

Within the machine an Lvalue is represented by a binary bit pattern of the same size as an Rvalue and so an Rvalue can represent an Lvalue directly. The process of finding the Lvalue or Rvalue of a variable is called Lmode or Rmode evaluation respectively. The idea of mode of evaluation is useful since it applies to expressions in general and can be used to clarify the semantics of the assignment command and other features in the language.

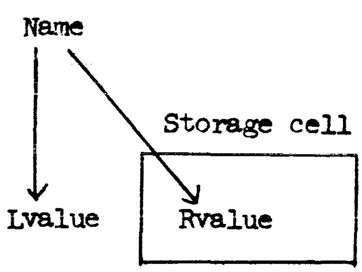

Figure 2—The form of a variable

Simple assignment

The syntactic form of a simple assignment command is:

$$E1 \quad := \quad E2$$

where E1 and E2 are expressions. Loosely, the meaning of the assignment is to evaluate E2 and store its value in the storage cell referred to by E1. It is clear that the expressions E1 and E2 are evaluated in different ways and hence there is the classification into the two modes of evaluation. The left hand expression E1 is evaluated in Lmode to yield the Lvalue of some storage cell and the right hand side E2 is evaluated in Rmode to yield an Rvalue; the contents of the storage cell is then replaced by the Rvalue. This process is shown diagrammatically in Figure 3. The only expressions which may meaningfully appear on the left hand side of an assignment are those which are associated with storage cells, and they are called Ltype expressions.

The terms Lvalue and Rvalue derive from consideration of the assignment command and were first used by Strachey in the CPL reference manual.

The lv operator

As previously stated an Lvalue is represented by a binary bit pattern which is the same size as an Rvalue. The *lv* expression provides the facility of accessing the

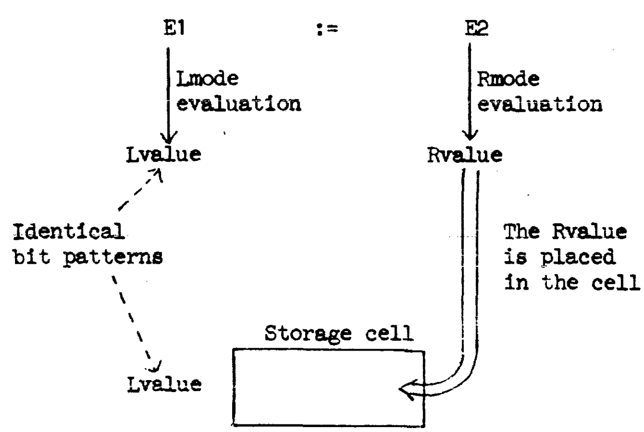

Figure 3—The process of assignment

Lvalue of a storage cell and, as will be seen, this ability is very useful.

The syntactic form of an *lv* expression is:

lv E

where E is an Ltype expression. The evaluation process is shown in Figure 4. The operand is evaluated in Lmode to yield an Lvalue and the result is a bit pattern identical to this Lvalue. The *lv* operator is exceptional in that it is the only expression operator to invoke Lmode evaluation, and indeed in all other contexts, except the left hand side of the assignment, expressions are evaluated in Rmode.

The rv operator

The *rv* operator is important in BCPL since it provides the underlying mechanism for manipulating vectors and data structures; its operation is one of taking the contents (or Rvalue) of a storage cell whose address (or Lvalue) is given.

The syntactic form of an *rv* expression is as follows:

rv E

and its process of evaluation is shown diagrammatically in Figure 5. The operand is evaluated in Rmode and then the storage cell whose Lvalue is the identical bit pattern is found. If the *rv* expression is being evaluated in Rmode, then the contents of the cell is the result; however, it is also meaningful to evaluate it in Lmode, in which case the Lvalue of the cell is the result. An *rv* expression is thus an Ltype expression and so may

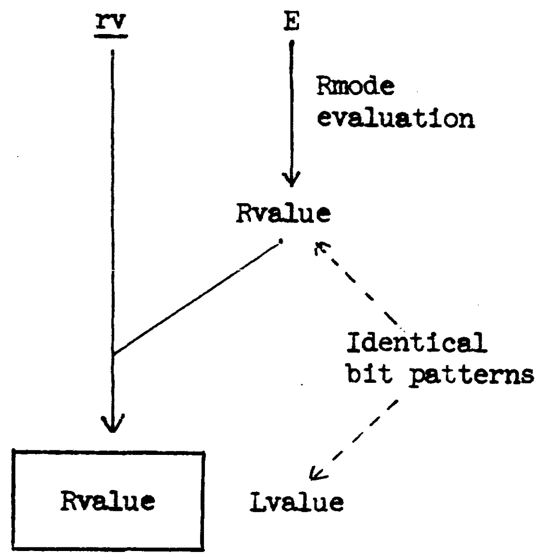

Figure 5—The evaluation of an RV expression

appear on the left hand side of an assignment command, as in:

rv p := t

and one can deduce that this command will update the storage cell pointed to by p with the Rvalue of t.

Data structures

The considerable power and usefulness of the *rv* operator can be seen by considering Figure 6. This

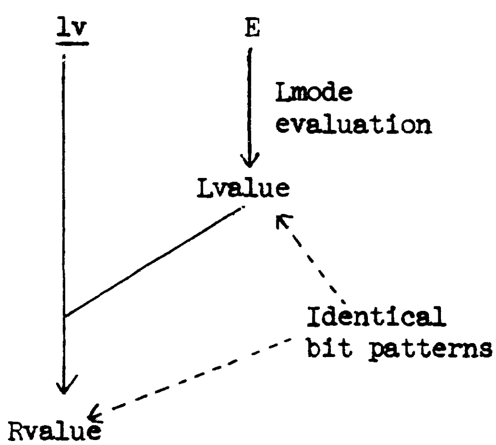

Figure 4—The evaluation of an LV expression

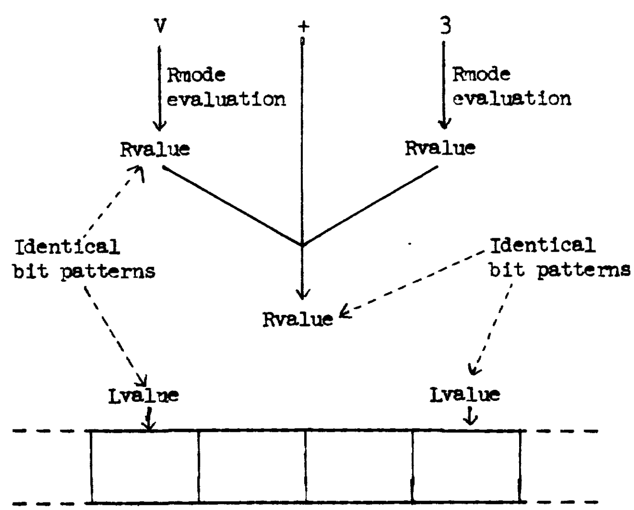

Figure 6—An intrepretation of V + 3

diagram shows a possible interpretation of the expression V + 3. Some adjacent storage cells are shown and the left most one has an Lvalue which is the same bit pattern as the Rvalue of V. One will recall that an Lvalue is really an integer and that Lvalues of adjacent cells differ by one, and thus the Rvalue of V + 3 is the same bit pattern as the Lvalue of the rightmost box shown in the diagram. If the operator *rv* is applied to V + 3, then the contents of that cell will be accessed. Thus the expression:

$$rv\ (V + i)$$

acts very like a vector application, since, as i varies from zero to three, the expression refers to the different elements of the set of four cells pointed to by V. V can be thought of as the vector and i as the integer subscript.

Since this facility is so useful, the following syntactic sugaring is provided:

E1 ↓ E2 is equivalent to *rv* (E1 + E2)

and a simple example of its use is the following command:

V ↓ (i + 1) : = V ↓ i + 2

One can see how the *rv* operation can be used in data structures by considering the following:

$$\begin{aligned} V \downarrow 3 &\equiv rv\ (V + 3) \text{ by definition} \\ &\equiv rv\ (3 + V) \text{ since + is commutative} \\ &\equiv 3 \downarrow V \end{aligned}$$

Thus V ↓ 3 and 3 ↓ V ares emantically equivalent; however, it is useful to attach different interpretations to them. We have already seen an interpretation of V ↓ 3 so let us consider the other expression. If we rewrite 3 ↓ V as Xpart ↓ V where Xpart has value 3, we can now conveniently think of this expression as a selector (Xpart) applied to a structure (V). This interpretation is shown in Figure 7.

By letting the elements of structures themselves be structures it is possible to construct compound data structures of arbitrary complexity. Figure 8 shows a structure composed of integers and pointers.

Data types

The unusual way in which BCPL treats data types is fundamental to its design and thus some discussion

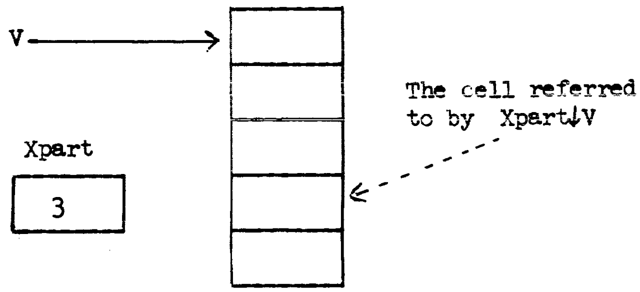

Figure 7—An interpretation of X part ↓ V

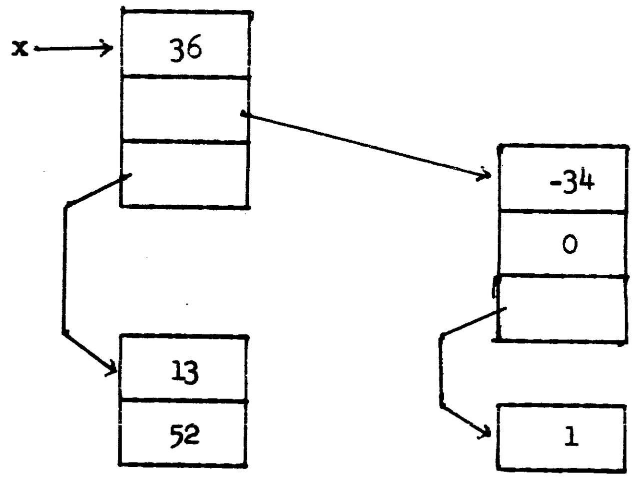

Figure 8—A structure of integers and pointers

of types is in order here. It is useful to introduce two classes:

a. Conceptual types
b. Internal types

The Conceptual type of an expression is the kind of abstract object the programmer had in mind when he wrote the expression. It might be, for instance, a time in milliseconds, a weight in grams, a function to transform feet per second to miles per hour, or it might be a data structure representing a parse tree. It is, of course, impossible to enumerate all the possible conceptual types and it is equally impossible to provide for all of them individually within a programming language. The usual practice when designing a language is to select from the conceptual types a few basic ones and provide a suitable internal representation together with enough basic operations. The term Internal type refers to any one of these basic types and the intention is that all the conceptual types can be modelled effectively using the internal types. A few of the internal types provided in a typical language, such as CPL, are listed below:

real

integer

label

integer function

(*real*, *Boolean*) *vector*

Much of the flavour of BCPL is the result of the conscious design decision to provide only one internal type, namely: the binary bit pattern (or Rvalue). In order to allow the programmer to model any conceptual type many useful primitive operations have been provided. For instance, the ordinary arithmetic operators +, −, * and / have been defined for Rvalues in such a way as to model the integer operations directly. The six standard relational operators have been defined and a complete set of bit manipulating operations provided. In addition, there are some stranger bit pattern operations which provide ways of representing functions, labels and, as we have already seen, vectors and structures. All these operations are uniformly efficient and can usually be translated into one or two machine instructions.

The most important effects of designing a language in this way can be summarized as follows:

1. There is no need for type declarations in the language, since the type of every variable is already known. This helps to make programs concise and also simplifies such linguistic problems as the handling of actual parameters and separate compilation.
2. It gives the language nearly the same power as one with dynamically varying types (e.g., PAL[4]) and yet retains the efficiency of a language (like FORTRAN[5]) with manifest types; for, although the internal type of an expression is always known by the compiler, its conceptual type can never be. For instance it may depend on the values of variables within the expression, as in the vector application V ↓ i, since the elements of a vector are not necessarily of the same conceptual type. One should note that in languages (such as ALGOL[6] and CPL) where the elements of vectors must all have the same type, one needs some other linguistic device in order to handle dynamically varying data structures.
3. Since there is only one internal type there can be no automatic type checking and it is possible to write nonsensical programs which the compiler will translate without complaint. This slight disadvantage is easily outweighed by the simplicity, power and efficiency that this treatment of types makes possible.

Syntactic features of BCPL

One of the design criteria of BCPL was that it should be a useful system programming tool and it was felt that high readability was of extreme importance. The readability of a program largely depends on the skill and style of the programmer; however, his task is greatly simplified if he is using a language with a rich set of expressive but concise constructions and if all the little syntactic details have been well thought out.

The syntax of BCPL is based on the syntax of CPL and, although the underlying semantics of the two languages are very different, they look superficially alike.

One of the most important requirements necessary before one can obtain a reasonable degree of readability is an adequate character set which contains both capital and small letters. A comparison has been made between two hardware versions of the same large BCPL program, one using nearly the full ASCII character set and the other using the same set without any small letters. Although there is no accurate measure of readability, it was agreed by all who made the comparison that the difference between the two versions was very significant. The lengthening of identifiers to avoid clash of names, and the fact that system words and identifiers were no longer distinct both increased the difficulty of reading the program. There are satisfactory implementations of BCPL using a restricted character set, but such a set should only be used where absolutely necessary.

BCPL follows CPL in the selection of commands. There are the three basic commands: assignments, routine commands and jumps, and there is the large variety of syntactic constructions to control the flow of control within an algorithm; some example forms are given below:

test E *then* C *or* C

if E *do* C

unless E *do* C

until E *do* C

while E *do* C

C *repeatuntil* E

C *repeatwhile* E

C *repeat*

for Name = E *to* E *do* C

where E denotes any expression and C any command. A very useful pair of additional commands are

a. *break* which causes a jump out of the smallest enclosing loop command, and
b. *return* which causes a return from the current routine.

One of the most noticeable ways in which this large selection of constructions improves readability is by the considerable reduction in the need for labels and goto commands. For instance, the BCPL compiler itself consists of 88 pages of BCPL program and contains only 29 labels which is about one label per three pages of program. It is interesting to see how experienced FORTRAN programmers fill their first few BCPL programs with labels and how their programming style improves as they gain facility.

The BCPL syntax for declarations and definitions is far simpler than the corresponding syntax in CPL; this is mainly due to the elimination of types from the language, and the lower emphasis placed on declarations in BCPL.

The purpose of a declaration in BCPL is threefold:

a. To introduce a name and specify its scope.
b. To specify its extent.
c. To specify its initial value.

The *scope* of a name is the textual region of program in which it may be used to refer to the same data item; this region is usually a block or the body of a function or routine, and it depends on the way in which the name was declared. The *extent* of a variable is the time through which it exists and is associated with a storage cell. Throughout the extent of a variable, its Lvalue remains constant and its Rvalue is only changed by assignment. Most forms of declaration initialize the variables they define, as in:

let f (t) = 2*t + 3

let *x* = 36 + f(4)

In this example, the variable f is initialized to a value which represents the function defined, and x is initialized to 47.

In BCPL, variables may be divided into two classes:

1. *Static variables*
   The extent of a static variable is the entire execution time of the program; the storage cell is allocated prior to execution and continues to exist until execution is complete.
2. *Dynamic variables*
   A dynamic variable is one whose extent starts when its declaration is executed and continues until execution leaves the scope of the variable. Dynamic variables are particularly useful when using recursive functions and routines. The kind of variable declared depends on the form of declaration used; out of the nine methods of declaring names in BCPL, five declare static variables, three produce dynamic variables and the remaining one declares manifest constants.

Function and routine calls

In BCPL as in CPL, there is a rigorous distinction between expressions and commands which shows itself in the syntax of the language; it also causes the semantic separation of functions from routines. In many respects functions and routines are rather similar; however, a function application is a kind of expression and yields a result, whereas a routine call is a kind of command and does not.

The syntactic form of both function applications and routine calls is as follows:

E1(E2, E3 . . . ., En)

where E1 to En all denote expressions. The expressions E2 to En are called actual parameters. The evaluation process is as follows:

1. All the expressions E1 to En are evaluated in Rmode to yield Rvalues.
2. A set of n-1 adjacent new storage cells are found and the values of E2 and En are stored in them.
3. The function or routine corresponding to the value of E1 is found and the formal parameters are associated with the cells containing the arguments. This association is performed from left to right and there is no need for the number of actual parameters to equal the number of formals.
4. The body of the function or routine is then executed in the new environment.
5. When the body has been completely evaluated, execution returns to the call. For a routine call there is no result and execution is now complete; however, for a function application there is a result which is the Rvalue of the function body.

All functions and routines in BCPL are automatically recursive and so, for instance, one can call a function while an activation of that function is already in existence. In order to allow for recursion and yet maintain very high execution efficiency, the restriction has been imposed that all free variables of both functions and routines must be static. Randell and Russell[7] give a good description of the kind of mechanism normally required for recursive calls in ALGOL; however, with this restriction, a recursive call in BCPL can be very efficient.

The mobility of the BCPL Compiler

A program is machine independent if it can be transferred from one machine to another without change. Complete machine independence is rarely achieved; however, it is a goal well worth striving for. For large systems, mobility is often a more useful measure than machine independence. Mobility is a measure of how easy it is to transfer a system from one machine to another; it differs from machine independence because the program often requires some redesign and reprogramming. For example, when moving a compiler from one machine to another it is necessary to rewrite the code generator and usually part of the lexical analyzer. Writing a compiler in a machine independent language is an important factor in obtaining mobility but it does not ensure it; it is at least as important to design the overall structure of the compiler with mobility in mind. The BCPL compiler has been designed with this aim and has been transferred successfully to seven other machines without much difficulty.

BCPL is a simple language to compile and it has a straightforward compiler written in BCPL. The compiler is easy to follow and it produces fairly good object code at an acceptably fast speed. Its general structure is shown in Figure 9. The rectangular boxes represent the different logical parts of the compiler and the round boxes the various intermediate forms the BCPL program takes while it is being compiled. These intermediate forms will be briefly sketched by considering the transformations of the program shown in Figure 10.

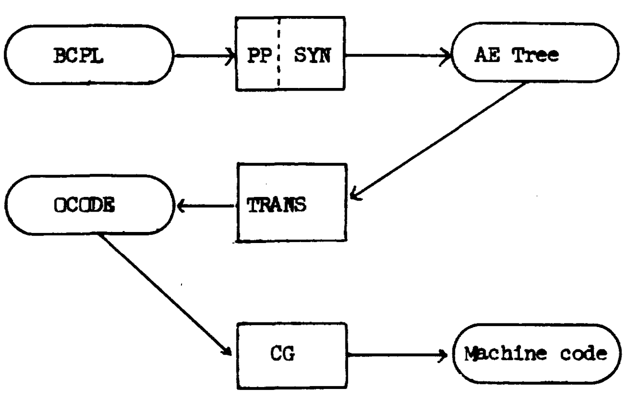

Figure 9—The structure of the BCPL compiler

The input form of the program is first transformed into an internal tree structure called the Applicative Expression Tree (AE Tree); this is done by the syntax analyzer which is composed of a set of machine independent parsing functions (SYN) and a lexical analyzer routine (PP). The AE tree structure for our example program is shown in Figure 11.

The AE tree is then translated by Trans into an intermediate object code called OCODE. OCODE was specially designed for BCPL and it is a simple language whose statements cause basic transformations on an imaginary stack machine; it was designed to be as machine independent as is practical to keep the changes to Trans to a minimum when the compiler is moved to a new machine.

The Code generator translates OCODE statements into the machine code of the object machine. The implementor is free to choose between relocatable binary and assembly language, and it is usually found that the ability to generate both is very valuable.

```
$(  let x = 0
    if x ne 2 do   x := x + 2
    finish   $)
```

Figure 10—An example BCPL program

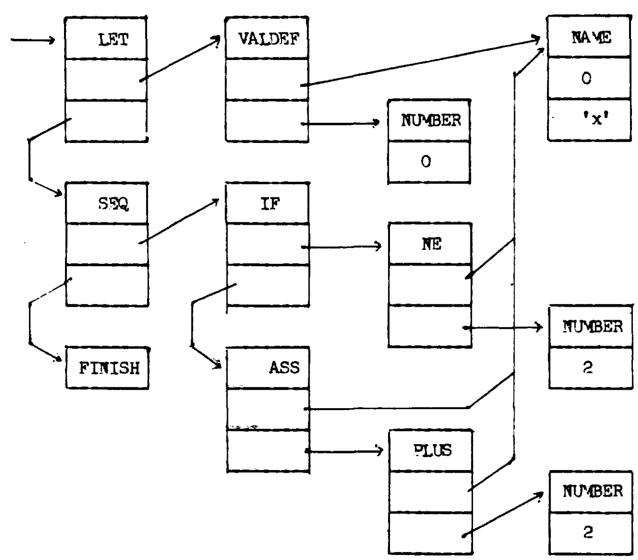

Figure 11—The AE tree form of Figure 10

In order to transfer BCPL to a new machine, one must choose a suitable strategy and this usually depends on the locality of the machines, their basic compatibility and the facilities available on the recipient machine. The basic process is to write both a code generator for the new machine and a suitable machine code interface with the new operating system; one then modifies and corrects the few machine dependencies in the syntax analyser and Trans, and finally compiles the new compiler using the new code generator. The process is complicated by the fact that the work cannot be carried out entirely on one machine. In practice, the more work that can be done on the donor machine the better; however, one often has no direct access to that machine and a different strategy must be applied. In this situation it is usually better to implement a temporary code generator for the recipient machine in some standard language such as FORTRAN or SNOBOL[8] and then use it to compile the OCODE files of the syntax analyzer and translation phase of the compiler. One can then construct a temporary BCPL compiler on the new machine and use it to compile itself. The compiler can then be polished to fit well in its new operating environment.

The cost of transferring BCPL depends largely on the computing facilities available, and one can expect it to be between one and five man months.

The use of BCPL for compiler writing

There are many reasons why BCPL is suitable for compiler writing and probably one of the most important of these is the ease of programming in the language. This together with its inherently high readability combine to make BCPL a very flexible language. The richness and variety of useful commands are valuable and the built in recursion is almost essential. In order to see how well these features may be used together we will consider a short excerpt from the BCPL compiler. Figure 12 shows the overall structure of the main part of translation phase. The directive *get* 'HEAD2' causes the compiler to insert a file of BCPL text and compile it with what follows. This insertion facility is very useful when co-ordinating many separate parts of a large program. This example shows how the *switchon* command may be used with manifest constants to good effect. It is executed by evaluating the controlling expression H1 ↓ x and then jumping to the case corresponding to the value found. The expression appearing in the case labels are manifest constants and denote the possible AE tree operators that Trans must deal with (the constants LET, TEST and REPEAT are declared in the inserted file HEAD2. If the value of H1 ↓ x does not correspond to any case then execution continues at the default label. Since all the case constants are known by the compiler it is possible to implement the switch very efficiently, even constructing a hash table if this method of switching is appropriate. This combination of manifest constants and *switchon* commands is very effective and it has been used frequently in the BCPL compiler.

```
get 'HEAD2'
let Trans(x) be
    $(1 switchon H1↓x into
        $( default: return
           case LET:     - - -
                         - - -
           case TEST:    - - -
                         - - -
           case REPEAT:  - - -
                         - - -      $)1
```

Figure 12—The structure of Trans

Recursion is also very useful in many compiling situations particularly in parts concerned with the creation and analysis of tree structures. Figure 13 is a detailed excerpt, again taken from Trans, and it provides an example of a typical use of recursion. The section of program shown is used to translate the AE tree form of a *test* command into OCODE. The variable x is the formal parameter of Trans and it points to a

```
case TEST: $( let L, M = Nextparam(), Nextparam()
              Jumpcond(H2↓x, false, L)
              Trans(H3↓x)
              Compjump(M)
              Complab(L)
              Trans(H4↓x)
              Complab(M)
              return     $)
```

Figure 13—A detail from the body of Trans

TEST node. The form of this node is shown in Figure 14; the pointers to E, C1 and C2 represent the branches to nodes for the Boolean expression and two alternative commands of the *test* command. These components can be accessed by the expressions H2 ↓ x, H3 ↓ x, and H4 ↓ x, respectively. To execute a test command, first the Boolean expression is evaluated and then, if the result is true, the first alternative is executed, alternatively the second. The general form of the object code is as follows:

1. Code to evaluate E.
2. Code to jump to label L if the result is false.
3. Code corresponding to the translation of C1.
4. An unconditional jump around the code for C2.
5. A directive to set label L.
6. Code for C2.
7. A directive to set the label used in step 4.

As can be seen the program to generate this code is very straightforward. First, two local variables L and M are declared for the two labels. The call for Jumpcond then compiles the code for steps 1 and 2. Its first argument is the Boolean expression of the *test* command and the other arguments specify the kind of conditional jump required and the label to jump to. The next statement is a call for Trans which will compile the first alternative C1. This is an example of the recursive use of Trans. The calls for Compjump and Complab generate code corresponding to steps 4 and 5, and then there is a second recursive call for Trans to translate C2. Finally, a directive to set label M is compiled and then, since the *test* command has now been completely translated, a return is made to the current call of Trans.

One should note how convenient it is not to have to declare the types of the variables such as x, L and M, and one should also note how well the use of manifest constants, switchon commands, recursion and simple data structures combine to produce a very effective and readable means of writing this part of the compiler. Although there is considerable variance in the style of programming used in the different parts of the compiler,

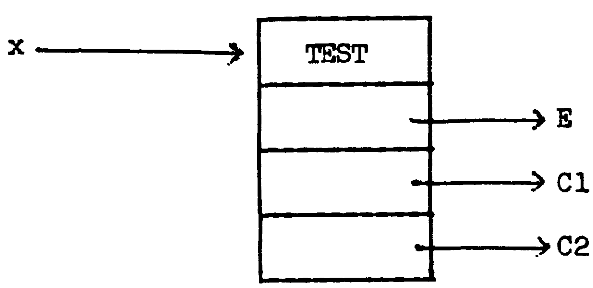

Figure 14—The AE tree form of a test command

the facilities and syntactic qualities of BCPL have made it possible to achieve this high standard of simplicity and readability throughout.

The way in which BCPL treats data types allows the programmer great freedom to organize his symbol tables, property lists, tree structures and stacks in the most suitable fashion for his particular application. Admittedly BCPL only provides the basic operations and the compiler writer must write his own system, but this is easy to do and he does not suffer the disadvantage of having to use a system in which inappropriate design decisions have already been made. The philosophy of BCPL is not one of a tyrant who thinks he knows best and lays down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering all his services to the best of his ability without complaint even when confronted with apparent nonsense. The programmer is always assumed to know what he is doing and he is not hemmed in by petty restrictions. Machine code programmers tend to like the way in which BCPL combines the advantages of a high level language with the power to do address arithmetic and to be able to manipulate binary bit patterns without invoking a great weight of expensive machinery.

When planning and writing a compiler in a commercial environment one must make a compromise between the quality of the product and its cost. The quality of a compiler is affected by many factors such as its size, its compile speed, the efficiency of the object code produced, the usefulness of the error diagnostics, the accuracy and quality of its documentation, its maintainability and in some cases its flexibility and mobility. Only the first two of these are directly improved by writing the compiler in a more efficient language, while the others tend to suffer because the compiler is harder to write. Although efficiency is important in a compiler writing language, this consideration should not totally dominate its design. The author believes that the compromise in the design of BCPL between efficiency and linguistic effectiveness is near optimal for compilers of medium to large scale languages especially if flexibility is required.

REFERENCES

1 M RICHARDS
The BCPL reference manual
Memorandum-69/1 The University Mathematical Laboratory Cambridge England January 1969

2 J BUXTON J C GRAY D PARK
CPL elementary programming manual
Edition II The University Mathematical Laboratory Cambridge England 1966

3 C STRACHEY (editor)
CPL working papers
Cambridge University Mathematical Laboratory and London Institute of Computer Science 1965

4 A EVANS JR
A language for teaching programming linguistics
Proc 23rd National ACM Conference 1968

5 IBM Reference Manual
709/7094 FORTRAN Programming System
Form C28-6054-2

6 P NAUR (editor)
Revised report on the algorithmic language ALGOL 60
The Computer Journal Vol 5 January 1963 349

7 B RANDELL L J RUSSELL
ALGOL 60 implementation
Academic Press 1964

8 D J FARBER et al
SNOBOL, a string-manipulation language
J ACM Vol 11 No 1 January 1964

APPENDIX

The syntax given below is Bachus Naur Form with the following extensions:

1. For improved readability, the syntactic categories for expressions, commands and definitions (namely E, C and D) are not surrounded by meta linguistic brackets.
2. The symbols { and } are used to indicate repetition, for example:

   E { , E }$_0^\infty$ means

   E | E, E | E, E, E | . . . etc

This syntax is ambiguous and is simply intended to list all the syntactic contructions available.

The canonical syntax of BCPL

E ::= ⟨name⟩ | ⟨stringconst⟩ | ⟨charconst⟩ |
⟨number⟩ | *true* | *false* | (E) | *valof* ⟨block⟩ |
lv E | *rv* E | E(⟨E list⟩) | E() | E⟨diadic op⟩ E |
⟨monadic op⟩ E | E → E, E |
table ⟨constant⟩ {, ⟨constant⟩ }$_0^\infty$

⟨diadic op⟩ ::= ↓ | * | / | *rem* | + | − |
= | ≠ | *ls* | *gr* | *le* | *ge* |
lshift | *rshift* | ∧ | ∨ | ≡ | ≢

⟨monadic op⟩ ::= + | − | *not*

⟨E list⟩ ::= E {, E}$_0^\infty$

⟨constant⟩ ::= E

C ::= ⟨E list⟩ := ⟨E list⟩ | E(⟨E list⟩) | E() |
goto E | ⟨name⟩ : C | *if* E *do* C | *unless* E *do* C |
while E *do* C | *until* E *do* C | C *repeat* |
C *repeatuntil* E | C *repeatwhile* E |
test E *then* C *or* C | *break* | *return* | *finish* |
resultis E | *for* ⟨name⟩ = E *to* E *do* C |
switchon E *onto* ⟨block⟩ | *case* ⟨constant⟩ : C |
default : C | ⟨block⟩ | ⟨empty⟩

D ::= ⟨name⟩ (⟨FPL⟩) = E | ⟨name⟩ (⟨FPL⟩) *be* C |
⟨name list⟩ = ⟨E list⟩
⟨name⟩ = *vec* ⟨constant⟩

⟨FPL⟩ ::= ⟨name list⟩ | ⟨empty⟩

⟨name list⟩ ::= ⟨name⟩ {, ⟨name⟩ }$_0^\infty$

⟨block⟩ ::= $(⟨block body⟩ $)

⟨block body⟩ ::= C{; C}$_0^\infty$ |
⟨declaration⟩ }$_0^\infty${; C}$_0^\infty$

⟨declaration⟩ ::= *let* D {*and* D}$_0^\infty$ | *static* ⟨decl body⟩ |
manifest ⟨decl body⟩ |
global ⟨decl body⟩

⟨decl body⟩ ::= $(⟨C def⟩ {; ⟨C def⟩}$_0^\infty$$)

⟨C def⟩ ::= ⟨name⟩ : ⟨constant⟩ |
⟨name⟩ = ⟨constant⟩

⟨program⟩ ::= ⟨block body⟩