SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 1, 135–146 (1971)

# The Portability of the BCPL Compiler

M. RICHARDS
*Computer Laboratory, Corn Exchange Street,*
*Cambridge CB2 3QG, England*

## SUMMARY

**Methods of achieving program portability are discussed, with particular reference to the mobility of compilers. The method of transferring the BCPL compiler is then described including the specification of OCODE which is the language used as an interface between the machine independent and machine dependent parts of the compiler.**

KEY WORDS Portability Programming language Macro processor BCPL OCODE

## INTRODUCTION

One of the most widely used methods of achieving portability of a program is to write it in a universally implemented programming language such as FORTRAN for scientific applications or COBOL for commercial ones. Unfortunately there are many applications where no currently available universal language is suitable and an alternative approach is then necessary. In such situations it is common to write the program in the form of a sequence of macro calls which can readily be transformed into assembly language for any particular machine using a macro processor and suitable macro definitions. There is evidence that this method works well using such macro processors as ML/1[1, 2] or STAGE 2.[3, 4]

In practice, the work involved in effecting a transfer by this method can be divided into the following steps:

(1) Write the macro definitions to convert the macro code into assembly language for the new machine.
(2) Write the machine dependent interface between the macro generated code and the new operating system. This consists mainly of routines for initialization and basic input/output.
(3) Macro generate, assemble and load the program.
(4) Debug the macro definitions and remove any unforeseen machine dependencies of the original macro code.
(5) Polish, modify and extend the program so that it fits cleanly into the new operating environment.

Each stage in the process is usually fairly well defined and straightforward. However, occasionally unexpected difficulties arise. It often turns out that the time spent writing the macro definitions themselves is quite small compared with the total time used to complete the transfer, since, for instance, the interface with the operating system may require some drastic redesign and so take longer than expected to write, or the third step of the transfer may become rather cumbersome if the quantity of macro code is very large.

*Received 23 October 1970*

In order to produce reasonably good code it is valuable to have a fairly sophisticated macro processor which is capable of performing simple computations at macro generation time. This will, however, mean that the macro processor itself is a sizeable program which may need to be transferred before the main transfer can take place. This initial transfer can be avoided if one performs all the macro generation on the donor machine; this mode of transfer is sometimes called 'pushing' and it has the advantage that the transfer can be done in one stage using the relatively sophisticated and convenient macro system which already exists on the donor machine. However, it suffers from having a rather slow debugging loop since errors detected on the recipient machine are often caused by bugs in the macro definitions that were used on the donor machine. Also, without the macro generator being implemented on the recipient machine, the maintenance of the transferred program will be most inconvenient. Thus, there are a number of reasons why it would be useful to transfer the macro generator first.

If a macro generator is written in a macro language which it can process, then as soon as it works on the recipient machine it will be capable of macro generating itself on that machine and thus be independent of the donor. ML/1 is written in this way and has been transferred successfully a number of times by pushing from the donor machine to the recipient.

If one has no access to the donor machine then one must resort to an alternative method of transfer where all the work is done on the recipient machine; this method is sometimes called 'pulling' and usually involves bootstrapping in a number of stages from a very simple start. A good example of this approach is the bootstrapping of the macro processor—STAGE 2. Waite[3] has written STAGE 2 in an extremely simple macro language called FLUB which can be processed by a trivially simple macro generator called SIMCMP which is itself implemented in FORTRAN.

Even though portability can be achieved very effectively using a macro generator, this technique is not always ideal. Its main drawback is that any language which can be compiled by a macro generator is necessarily of rather a low level. For instance, even with a really powerful macro generator it is not usually feasible to generate code for general arithmetic expressions automatically since this tends to involve a great many rather complicated definitions to achieve even a mediocre translation. Furthermore, by making the definitions more difficult to write, one is beginning to lose the portability that the use of a macro generator was originally meant to provide. No one has ever suggested seriously that a language such as ALGOL 60 should be compiled using only a macro generator. Since there are applications requiring portability where a general high level language would be far more suitable than any macro language, one must resort to using either a universal language or a language with a portable compiler.

A compiler, unfortunately, is not a particularly portable program since its code generation end will need extensive modification for each new machine. A natural strategy for transferring such a program is to break it into two parts: a first phase that is machine independent and a code generator that is not. The first half can be transferred using the standard macro generation technique, but the second presents a greater problem. One possible solution is to use a macro language as the interface between the two parts of the compiler since then we can use a macro processor to perform the code generation; and indeed, if we choose to write the first phase of the compiler in the same macro language as is used in the interface then we will have reduced the problem of transferring a compiler to simply that of transferring a machine independent macro program. It may, of course, be that the interface macro language cannot be used to write the first phase of the compiler; but if it can, then

it is probable that the first phase could be implemented satisfactorily in the language that it compiles. If we have a compiler organized in this way then the steps needed to transfer it to a new machine are as follows:

(a) Bootstrap the macro generator.
(b) Write the macro definitions for the interface language.
(c) Macro generate the first phase of the compiler (which must be supplied as a program written in the interface language).
(d) Interface the assembly language produced with the new operating system.

Once the transfer is complete we have a fairly flexible compiler on the new machine which is partly written in its own high level language and partly in the form of macro definitions. We have, thus, achieved our initial objective of constructing a portable compiler; however, a compiler so produced will be very slow being limited by the inefficiency of the macro generator and assembler and one is naturally tempted to explore ways of improving it.

An obvious improvement is to combine the macro generator and assembler into one program and thereby save the inefficient outputting of assembly language text for later re-input to the assembler. A further improvement can be made by building the macro definitions into the code generator and thus allowing for far more efficient decoding of the macros and the elimination of much of the cost caused by the interpretive nature of most macro generators. A scheme very similar to this is the compiled macro assembler suggested by Maurer,[5] but without such a powerful assembler one has to resort to writing the code generator by hand, but by doing so one reaps a number of additional advantages. One is that the implementer may find it easier to generate optimized code since he can optimize by algorithm rather than by a complicated set of macro definitions. Another advantage is that the original need for the interface to be a macro language is no longer valid. Since it is output by program and read by program there is no need for it to be readable by a programmer and indeed it need not even be represented in character form. The design criteria for such an interface language are rather different than for a macro code which is to be written by hand, and one can take advantage of this to produce an intermediate object code which is reasonably compact and easy for the first phase of the compiler to generate as well as being easy for the code generator to translate. By using an intermediate object code in place of a macro code we may lose the portability that was our prime objective, but this can be overcome either by designing the intermediate object code so that it can easily be turned into machine code by a simple non-optimizing code generator, or by supplying with the compiler a code converter that will translate the intermediate code into a simple macro code.

BCPL[6, 7] is a language whose compiler is constructed in exactly this way in order to make it portable, and the rest of this paper is devoted to a description and discussion of OCODE which is the intermediate object code used. Since OCODE is very simple there is space here to give it in sufficient detail for the reader to implement a code generator for it on his machine; this may then be used to convert the OCODE form of the BCPL compiler into machine code and by adding suitable interface routines an implementation of BCPL may be obtained.

BCPL is a general purpose programming language which was designed as a tool for compiler writing and other system programming applications. It is somewhat similar to ALGOL 60 since it is block structured and has recursive functions, but unlike ALGOL it has only one size of data item and it allows the user to perform machine independent address arithmetic. There is no need for the reader to be familiar with BCPL since any relevant features will be explained where necessary.

## THE INTERMEDIATE OBJECT CODE—OCODE

One feature of BCPL which allows its intermediate object code to be particularly simple is that all its variables and expressions have values which are of the same size. This allows both for a very simple addressing structure as well as a type-free stack mechanism. OCODE is a macro-like low level language containing fifty-six different statements which are all very simple. Since it is primarily designed as an interface between two programs, its detailed internal representation is not specified since this is often a binary form which is highly machine dependent. However, when defining OCODE and when using it during a transfer to a new machine it is best to use a standard character representation since this simplifies its description and eases the debugging of the code generator by making test data easier to prepare.

Each statement starts with a key word (represented by a sequence of letters) specifying one of the fifty-six possible statements. The key word is possibly followed by arguments which can either be labels (appearing as L followed by digits) or positive or negative integers. The number of arguments depends on the key word and occasionally on the first argument. Key words and arguments are terminated by spaces or newlines.

BCPL functions and routines are recursive and so all arguments, anonymous results and most variables are allocated space on a runtime stack and are addressed relative to an activation pointer called P which points to the base of the region of the stack belonging to the current call. This region of the stack is called the 'current stack frame' and its size varies during execution as the number of local variables and anonymous results changes, however, the design of BCPL is such that the size of the stack frame is always known at compile time for every point in the compiled code. This allows the first phase of the compiler to allocate cells for local variables declared within a routine and address them relative to P.

To simplify the description of OCODE we will introduce the variable S which gives the position relative to P of the next free stack location, but one should remember that both the first phase of the compiler and the code generator can deduce the value of S at every stage and so S need not exist at runtime. Many OCODE operations are concerned with loading, storing or modifying items on or near the top of the stack, and we will use P[S − 1] and P[S − 2] to denote the top and second locations of the current stack frame. This notation is shown pictorially by the following diagram.

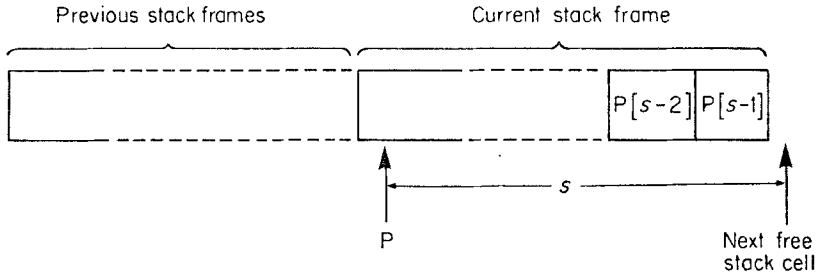


*Figure 1. The runtime stack*

There are only three areas where BCPL variables can be allocated and there are correspondingly three sets of OCODE statements which access and update them.

## LOCAL VARIABLE ACCESS

Local variables including function arguments are allocated cells on the stack and are addressed relative to P. The operators LP, LLP and SP may be used to load the value of a

local variable, load the left hand value (or address) of a local variable or store a value in a local variable. More formally, this may be described as follows:

LP *n* means S := S+1; P[S−1] := P[*n*]

LLP *n* means S := S+1; P[S−1] := @P[*n*]

where @ stands for 'address of'.

SP *n* means P[*n*] := P[S−1]; S := S−1

## GLOBAL VARIABLE ACCESS

The global vector in BCPL is, in many respects, similar to FORTRAN common and is used for linking and global communication between separately compiled segments of program. Variables may be allocated specific locations within the global vector and the OCODE statements to access them are as follows:

LG *n* means S := S+1; P[S−1] := G[*n*]

where G[*n*] denotes the *n*th global variable.

LLG *n* means S := S+1; P[S−1] := @G[*n*]

SG *n* means G[*n*] := P[*n*−1]; S := S−1

It is possible to initialize global variables at load time to contain either integers or pointers to the entry points of functions and routines and the OCODE statements for these are as follows:

INITGN *g k* means initialize G[*g*] to value *k*.

INITGL *g* L*n* means initialize G[*g*] to the entry point labelled by the assembly parameter L*n*.

## STATIC VARIABLE ACCESS

Static variables in BCPL are similar to *own* variables in ALGOL or local variables in FORTRAN and are allocated storage cells at compile time which can be intermixed with the compiled code. Such cells are allocated using the ITEMN and ITEML statements and they are addressed using assembly parameters. The statement DATALAB L*n* will set the assembly parameter L*n* to the address of the next static cell to be allocated. ITEMN *k* will allocate a static cell and give it the initial value *k* which must be a positive or negative integer and ITEML L*n* will allocate a cell and give it an initial value which points to the entry point labelled L*n*.

Static variables may be accessed using LL, LLL and SL defined as follows:

LL L*n* means S := S+1; P[S−1] := Cell[L*n*]

where Cell[L*n*] denotes the static cell labelled by assembly parameter L*n*.

LLL L*n* means S := S+1; P[S−1] := @ Cell[L*n*]

SL L*n* means Cell[L*n*] := P[S−1]; S := S−1

## LOADING CONSTANTS

Numerical and Boolean constants may be loaded using LN, TRUE and FALSE defined as follows:

LN $k$ means $S := S+1;\ P[S-1] := k$

where $k$ is a positive or negative integer.

TRUE means $S := S+1;\ P[S-1] := not\ 0$

FALSE means $S := S+1;\ P[S-1] := 0$

where 0 and *not* 0 is used to denote bit patterns entirely composed of zeros and ones respectively.

The LSTR statement will load a value which represents a string. Its first argument gives the number of characters in the string and the remaining arguments are the integer values of the characters given in some standard code. The design of BCPL is such that the first phase of the compiler must know the integer values of string characters and so during the initial stage of bootstrapping one must work in the character code of the donor machine. However, as soon as the compiler can compile itself, the first phase can be changed to use a different code and this usually involves a change in only one function. Strings are packed in BCPL and the packing is necessarily machine dependent since it depends strongly on the word and byte sizes of the object machine. The usual internal representation of a string value is as a pointer to the first of a set of words holding the length and packed characters of the string. The zeroth byte is usually justified to the start of a word and holds the length of the string with successive bytes holding the characters and padded with zeros (or possibly spaces) at the end to fill the last word. In order to handle strings in as machine independent way as possible packing, unpacking and writing of strings is done using library routines which are defined in the machine dependent interface with the operating system.

## DIADIC EXPRESSION OPERATORS

All expression operators take their operands from the top of the stack and leave the results in their place. The typical diadic operator is defined as follows;

O$p$ means $P[S-2] := P[S-2]\ F_{op}\ P[S-1];\ S := S-1$

where O$p$ denotes the OCODE key word and $F_{op}$ the corresponding diadic operator.

There are seventeen such operators defined as follows:

Integer arithmetic operators—

MULT, DIV, REM, PLUS and MINUS

REM denotes the remainder after integer division.

Relation operators—

EQ, NE, LS, GR, LE and GE

These correspond to the six usual relation operators and yield Boolean results.

Logical operators—

LSHIFT, RSHIFT, LOGAND, LOGOR, EQV and NEQV

The operators LSHIFT and RSHIFT perform logical shifts on their left operands to the left and right respectively. The number of places to shift is given by the right operand. The other key words provide the bit pattern operations of logical and, logical or, equivalence and non-equivalence.

## MONADIC EXPRESSION OPERATORS

These all take one operand and yield one result and so do not affect S. They are defined as follows:

NEG means P[S−1] := −P[S−1]

NOT means P[S−1] := *not* P[S−1]

RV means P[S−1] := Cont[P[S−1]]

where Cont[$x$] is the primitive function
to access the contents of the cell whose address is x.

In order to achieve machine independent address arithmetic BCPL requires that the representation of addresses and integers is such that adding one to an address will yield the address of the adjacent storage cell.

## COMMAND OPERATORS

Assignments to simple variables may already be translated using SP, SL and SG defined above; however, assignments to locations whose addresses need to be evaluated require an additional assignment operator called STIND defined as follows:

STIND means Cont[P[S−1]] := P[S−2]; S := S−2

Conditional commands require corresponding conditional statements in OCODE, these are JT L*n* and JF L*n* which cause the program to jump conditionally to the point labelled by the assembly parameter L*n* depending on the value of P[S−1]. JT will jump if it represents *true* and JF will jump if it is *false*; whether the jump is made or not S is reduced by one. The assembly parameter is set using the statement LAB L*n* at the appropriate place in the OCODE program.

An unconditional jump may be specified by the statement JUMP L*n* and a jump to a computed label by the GOTO statement defined as follows:

GOTO means S := S−1; *goto* P[S]

The statement FINISH will cause execution of the compiled program to cease.

A BCPL switch is compiled using the SWITCHON statement whose form is:

SWITCHON $k$ L$d$ $K_1$ L$n_1$ ... $K_k$ L$n_k$

The top item of the stack P[S−1] is used to control the switch; if it equals any one of the $k$ case constants $K_1$ ... $K_k$ then a jump is made to the corresponding case label, otherwise a jump is made to the default label L$d$. S is reduced by one during the execution of the statement. Since the number and range of case constants is given, the code generator has ample opportunity to select an appropriate switching strategy.

Occasionally it is necessary for the first phase of the compiler to tell the code generator where the top of the stack is relative to P. This happens, for instance, when vectors are declared or at the ends of blocks. The OCODE statement to pass this information is STACK $k$ and its effect is to set S in the code generator to the value $k$.

A second directive, STORE, is provided so that the first phase of the compiler can indicate the point dividing the declarations at the head of a block from the body that follows. Its effect is to cause the code generator to compile code to standardize the runtime state of the machine so that all stacked items are physically held in their appropriate store locations rather than being held in central processor registers. Without such a directive it would be difficult for an optimizing code generator to know when stacked items could be held safely in machine registers.

BCPL has a construction whereby commands can be embedded in an expression. The textual extent of the commands in the body of a *valof* block and the return from such a block is performed by a *resultis* command. This return causes a value to be passed back and a jump to take place, and the OCODE statements to provide this mechanism are as follows:

RES L*n* means A := P[S − 2]; S := S − 1; *goto* L*n*

RSTACK *k* means S := *k* + 1; P[S − 1] := A

where A is used to denote the location in which the result is passed.

## FUNCTION AND ROUTINE CALLING

The syntactic form of a BCPL function or routine call is E0 (E1, E2, ... E*n*) where E0 ... E*n* are all general expressions; E0 evaluates to give the value of the function or routine to be called and E1 to E*n* give the arguments of the call. The parameters are called by value and, as was stated earlier, the calling mechanism is recursive. It is essential for BCPL that calls be compiled efficiently and with this in view the OCODE statements for calls have been designed with great care.

Since the called function is recursive it needs a new stack frame and this is allocated just beyond the limit of the current stack frame whose size is, of course, known by the compiler. The arguments are passed by initializing the lower cells of the new stack frame with the values of the expressions E1 to E*n* after leaving two cells free for link information. Once the arguments have been dealt with, E0 is evaluated to yield a pointer to the entry point of a function or routine. At this stage we are in a position to transfer control to the called procedure and the stack has the following form:

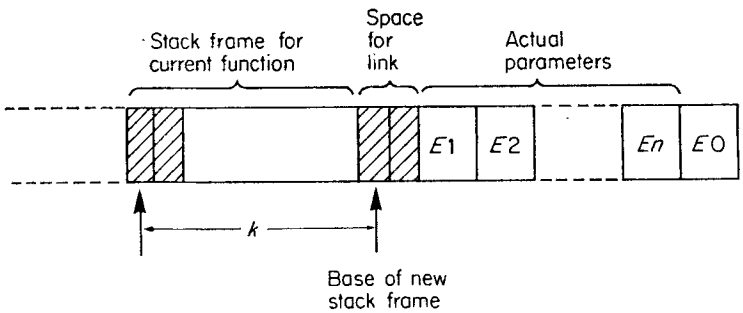


*Figure 2. The stack at the moment of call*

*k* is the distance between the old and the new stack frame pointers and is a constant depending only on the amount of stack used in the current function and this is known by

the compiler. The form of the link information is left completely open to give the implementer as much freedom as possible. The two stack cells allocated for the link are invariably optimal but can easily be changed by changing a constant in the first phase of the compiler. The actual entry is made using FNAP *k* or RTAP *k* depending on whether a function or routine is being applied, that is, whether a result is expected or not.

The entry point of a function or routine is marked by an entry statement of the form:

$$\text{ENTRY } k \text{ L}n \text{ K}_1 \text{ K}_2 \ldots \text{K}_k$$

where L*n* is the assembly parameter for the entry and $K_1$ to $K_k$ are the characters of the print name of the function or routine using the same character code as for strings. The entry statement is immediately followed by SAVE *n* which gives the code generator the initial stack frame size from which it can also deduce the number of formal parameters, namely $n-2$. The return from a function is caused by the statement FNRN and from a routine by RTRN.

These six statements can be defined more formally by a simple implementation such as the following:

| | | |
|---|---|---|
| FNAP *k* | means | K, L := *k*, M<br>*goto* P[S − 1]<br>M : S := *k* + 1<br>P[S − 1] := A |
| RTAP *k* | means | K, L := *k*, M<br>*goto* P[S − 1]<br>M : S := *k* |
| ENTRY *k* L*n* $K_1 \ldots K_k$ | means | L*n* : P[K] := P<br>P[K + 1] := L<br>P := P + K |
| SAVE *n* | means | S := *n* |
| FNRN | means | A := P[S − 1]<br>L := P[1]<br>P := P[0]<br>*goto* L |
| RTRN | means | L := P[1]<br>P := P[0]<br>*goto* L |

In this implementation it is assumed that there are machine registers K, L and A that can be used for communication between the calling and called routines.

## EXAMPLE

As an example we give a program and its corresponding OCODE form. It includes the definition of a recursive function which evaluates factorials and a *for* loop which will cause

it to be evaluated for integers in the range one to ten. The program is as follows:

```
GLOBAL $( START: 1; WRITEF: 76 $)
START: $(
LET F(N) = N = 0 -> 1, N*F(N-1)
FOR I = 1 TO 10 DO WRITEF("F(%N) = %N*N", I, F(I))
FINISH $)
```

WRITEF is a general purpose output routine which is defined in the library. Its first argument is a string which both gives the the format of the output and controls the printing of the remaining arguments. Within a format %N will cause an argument to be printed as a decimal integer, and in string constants *N represents the newline character. The call WRITEF("F(%N) = %N*N", 4, 24) will output the following:

F(4) = 24

The corresponding OCODE for this program is as follows:

```
STACK 2
INITGL 1 L1 LAB L1 STACK 2          initialize START
DATALAB L3 ITEML L2                 allocate cell for F
JUMP L4                             jump round the body of F
ENTRY 1 L2 70 SAVE 3                entry to F
LN 0 LP 2 EQ JF L6                  test if N = 0
LN 1 JUMP L5                        then load 1 or
STACK 3
LAB L6 STACK 5
LP 2 LN 1 MINUS LL L3 FNAP 3        F(N-1)
LP 2 MULT                           multiplied by N
LAB L5
FNRN                                return with the result
STACK 2
LAB L4 STORE                        end of declarations
LN 1 STORE JUMP L7                  initialize the FOR loop
LAB L8 STACK 5                      load arguments of WRITEF
LSTR 11 70 40 37 78 41
     32 61 32 37 78 10
LP 2
STACK 9 LP 2 LL L3 FNAP 7
LG 76 RTAP 3                        call WRITEF
LP 2 LN 1 PLUS SP 2                 increment I
LAB L7
LP 2 LN 10 LE JT L8                 loop again if I <= 10
STACK 2
FINISH
STACK 2
```

## THE CODE GENERATION OF OCODE

As has been seen OCODE consists of a set of very primitive statements separated into logical groups. The statements concerned with accessing variables are independent of those for arithmetic operations which are in turn distinct from those provided to handle flow of control. For almost any language one could design an intermediate object code similar in flavour to OCODE, and indeed for a number of implementations of BCPL itself, OCODE has been extended to include operators for floating point arithmetic, field selection in packed data structures, and for accessing external named variables.

OCODE relies on its simplicity to be effective as a bootstrapping tool since otherwise the initial code generator for a new machine would be too difficult to write. But it is important to note that it may also be compiled into efficient code using an optimizing code generator and in this section we explore to what extent this is possible.

There are two main areas which affect the quality of compiled code: global organization and local optimization. Global organization is concerned with general design decisions such as how many of machine registers are to be allocated for special purposes and how many to be left for use as temporary work space, and how to implement the recursive calling mechanism. It also concerns such factors as the internal representation of data, the organization of the stack and the allocation and addressing of variables. While local optimization deals with the selection of appropriate machine instructions and allocation of temporary work space within the constraints already imposed. Good global organization is often a better way of achieving efficient code than any amount of local optimization. For instance on a machine such as ATLAS where there are a very large number of index registers, one can, by permanently allocating some of them to certain common constants, arrange that simple statements such as:

$$X := X+1; \quad Y := 5-Y; \quad Z := Z \wedge 7$$

can each be compiled in one instruction. Another example of good global organization is to pass the first few parameters of a call on central machine registers since this will reduce the amount of code in calling sequences and has the additional advantage of providing a cleaner interface with the machine code library. This use of index register on ATLAS accounts for a saving of nearly 1200 instructions out of about 8000 in the compiled form of the first phase of the BCPL compiler.

It is, of course, desirable to have a reasonable level of local optimization and this can be achieved by delaying the generation of code as long as conveniently possible and checking for special cases. This is best done by simulating the state of the runtime stack and only compiling instructions when it becomes necessary to simplify the simulation. The simulation that is used for most BCPL code generators including those for the IBM 7094, 360, ATLAS and XDS Sigma 5 is fairly simple and has proved to be quite satisfactory.

In all these code generators, each simulated item is represented by a node which holds the following information:

(1) The runtime address relative to P of the stack location that this node is simulating.
(2) A numerical value, variable address or machine register.
(3) An integer offset.
(4) A Boolean marker to indicate possible indirection.

With such a node we can represent stacked items corresponding to such BCPL expressions as the constant 83, the value of the variable $V$, the value $V+83$ or the array element $V!83$ (which represents the contents of the cell whose address is $V+83$). In addition one can

represent values held in machine registers. Stacked items for which there is no simulating node are assumed to be already in their true storage location and so can be addressed directly relative to P. One advantage of this form of simulation is that it is relatively machine independent and hence much of a code generator for one machine can be used in a code generator for another.

## CONCLUSION

BCPL has now been transferred to between ten and twenty different machines using OCODE as the interface between the two halves of the compiler. The time taken to complete a transfer is very variable depending strongly on the computing facilities available, and usually takes between three and five months if the implementer has no previous knowledge of BCPL and no access to the donor machine, but it may take as little as three or four weeks in ideal conditions. Much depends on how long it takes to design and implement the interface with the operating system; this can be very little if one chooses to provide the user with only a few very primitive facilities, but it is usual to give him a more powerful interface and this necessarily requires much more work.

## REFERENCES

1. P. J. Brown, 'Macro processors and software implementation', *Comput. J.* **12**, 327–331 (1969).
2. P. J. Brown, 'The ML/1 macro processor', *Communs Ass. comput. Mach.* **10**, 618–623 (1967).
3. W. M. Waite, 'The mobile programming system: STAGE 2', *Communs Ass. comput. Mach.* **13**, 415–421 (1970).
4. R. J. Orgass and W. M. Waite, 'A base for a mobile programming system', *Communs Ass. comput. Mach.* **12**, 507–510 (1969).
5. W. D. Maurer, 'The compiled macro assembler', *Proceedings of the Spring Joint Computer Conference*, **34**, 89–93 (1969).
6. M. Richards, *The BCPL Reference Manual*, Technical Memorandum 69/1, Computer Laboratory, Cambridge, 1969.
7. M. Richards, 'BCPL: A tool for compiler writing and system programming', *Proceedings of the Spring Joint Computer Conference*, **34**, 557–566 (1969).