Code compaction for minicomputers with INTCODE and MINICODE

J.E.L.Peck, V.S.Manis and W.E.Webb

University of British Columbia

1 Introduction

The INTCODE system, originated by M.Richards [R1], is a means for programming computers, at a low level, in a machine independent manner. The basic philosophy involves the design of a simple ideal machine, and the interpretation of this ideal machine on various pieces of hardware. The idea is not novel, but the INTCODE machine is particularly successful, in that it has been used for the transportation of a BCPL compiler from one computer to another.

A crucial part of the system is a compiler [R3] from BCPL to an assembler language for the ideal machine. The success of INTCODE lies in the ease with which it may be assembled and interpreted on, or translated to and run directly on, any piece of hardware.

Richards gives an assembler and interpreter [R2] for the ideal machine, with word size at least 24 bits, but does not show(1) how to deal with a machine whose word size is smaller, nor how to produce relocatable code. Since many minicomputers have a word size of less than 24 bits, and since relocatable code brings essential flexibility, a more general assembler and interpreter is needed.

In this paper we discuss

a) an assembler and interpreter which is universal, in the sense that its word size and character size are run time parameters,
b) the compaction of this universal machine code by using relative addressing,
c) the production of relocatable load modules,
d) the production of assembler code for some real hardware,
e) the humanizing of INTCODE to a more readable form which we shall call MINICODE, and

---

(1) In a private communication he suggests how to do it.

f) the didactic possibilities in MINICODE.

1.1 The System

The system works in the following manner. A program written in BCPL, e.g.,

```
global { start:1; writes:74 }
let start be writes("hello")
```

when fed to the BCPL-to-INTCODE compiler, produces the following INTCODE assembler code.

```
JL2
$ 1 LL499 SP4 LIG60 K2 X4 2 X22
499 C5 C72 C69 C76 C76 C79
G1L1
Z
```

This assembler code, together with a similar standard transput library is then fed to an assembler which produces relocatable load modules, a few lines of which (for word size = 16 and chacter size = 8) are

```
P 000000
 033407, 070000, 003406, 012004, 005074, 060002, 070004, 070026
 002510, 042514, 046117
G 000001  0+000001
.END
```

The octal code is then loaded and interpreted by an interpreter. Observe that, since a BCPL-to-INTCODE compiler, written in INTCODE, is available, this allows for the transportation of an interpretive BCPL compiler.

An alternative route, if a full BCPL compiler is not locally available, is to translate the INTCODE assembler code directly to the assembler language of some real hardware. This will, of course, produce faster executing, but possibly larger object code. A few lines of such automatically produced 370 assembler code from the given example are shown here.

```
...
 USING *,12
C0L1 L 12,=A(C0)
 USING C0,12
 LR B,A
 LA A,C0L499
 SRL A,2
 ST A,4*4(P)
 LR B,A
 L A,60
...
```

The remainder of the process can now be completed using the software provided by the manufacturer. Note that, in order to

generate more efficient code, a BCPL compiler for a specific machine (as opposed to the ideal machine we discuss) will normally bypass INTCODE.

1.2 Machine independence

It is important to observe that not all these steps need be done on the same machine. A transfer of INTCODE assembler source or of octal load modules from one machine to another may be made. This could be of importance in places where several minicomputers of different kinds are in use for supporting some large system. To have a universal programming system, it is then only necessary to write an INTCODE interpreter for each minicomputer. If some large computer is also available, then it could be used to produce the relocatable octal load modules for loading to each minicomputer. This means that the target machines might even differ in word length or use different integer arithmetic. Of course, the programmer would not be able to make assumptions about word length or arithmetic formats.

2 The ideal machine

Since the basis of this system is an ideal machine, it is important to describe it here, although much of this has already appeared elsewhere [R2]. It has a simple architecture. Its memory consists of a sequence of words, starting from the address 0, and increasing in steps of 1. The number of words of memory is not specified, except that the address field of an instruction will impose an upper limit. The number of bits per word (word size) is also not specified, except that 12 is probably the lower limit of practicality. The number of bits per character (character size) is not specified. The two values, word size and character size, which can be run-time parameters, are used by the library routines for packing characters into words.

The machine has five registers called A, B, G, P and C, where,

- A and B are accumulators used for executing dyadic operations,
- G and P are index registers, G pointing to a global array (a sequence of memory words) and P pointing to the stack (another sequence of memory words),
- C is the program counter (containing the address of the next instruction).

At run time there are three independent areas of storage in use, the global vector, the executable code plus constant data and the stack. In the algorithms of the MINICODE users guide [P], the storage sequence is in the order just stated, but this is an arbitray choice. If one does not mind a sacrifice in flexibility, one may even dispense with the G-register, by locating the global vector in some fixed memory area. On a machine with base-page addressing, e.g., PDP-15 or HP2100, this might be most convenient.

2.1 Machine operations

The machine has eight operations; these are

operation	code
Load	(0)
Store	(1)
Add	(2)
Jump	(3)
True jump	(4)
False jump	(5)
K call a procedure	(6)
eXecute	(7) .

Each machine instruction has one operand which is interpreted sometimes as an integral value and sometimes as an address. When an instruction is analyzed, this operand may be modified by one or more of three flags called the I, P and G flags. For example, L1 means load 1 into the A register (after moving the content of the A register to the B register); LP3 means load the address of the third word of the present environment. LIG13 means load the content of the 13th global word to the A register.

The operand modification is as follows:

a) if the P flag is on, then add the content of the P register to the operand,
b) if the G flag is on, then add the content of the G register to the operand,
c) if the I flag is on, then fetch a new operand from an address which is the old operand (indirect addressing).

It is important to note that the P and G flags cannot both be on. Also the G register does not change during program execution whereas the P register changes. The latter points to the most recent environment on the stack.

2.2 The instruction repertoire

The instruction repertoire is described below, where it is assumed that d is the value of the operand after index modification and relative and indirect address calculation, if any, has been completed. When an instruction is fetched, then the C register is first incremented by one so that it points to the next instruction.

0) Ld (Load d) copies the content of the A register to the B register and then loads the operand d into the A register.
1) Sd (Store d) copies the content of the A register to the word at address d.
2) Ad (Add d) adds d to the content of the A register leaving the sum in the A register.
3) Jd (Jump to d) places d in the control register C.

4) Td (jump to d if True) places d in the control register C if the content of the A register is -1.
5) Fd (jump to d if False) places d in the control register C if the content of the A register is 0.
6) Kd (call in an environment of length d) places the content of P at position (P)+d (stack link), places (C) at (P)+d+1 (return address) places (P)+d in P (new environment) and places the content of A in C (address of procedure).
7) Xd (eXecute the operation number d). The operation specified is executed using registers A and B. Usually the result is placed in A. For example, X8 adds the content of B to the content of A, and X5 multiplies the content of A by the content of B.

Further details of the register operations (there are about 30) can be found in section 4.2. One of them, X23, introduces a case statement.

3 Implementation

Implementation of the system involves the writing of an assembler and an interpreter, each of which is only a modest effort. Both of these are given in BCPL [P, R2] and in other languages. If a BCPL-to-INTCODE compiler is available, then only the interpreter need be written for any new machine.

3.1 The instruction format

The implementation of the ideal machine described by Richards [R2] uses an instruction format in which each instruction occupies one word of memory. For each ideal machine

Instruction format

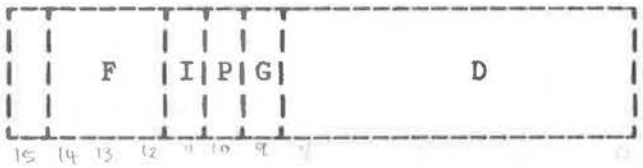

Figure 3.1

instruction then

a) 3 bits are used for the instruction code F,
b) 1 bit is used for the I flag,
c) 1 bit is used for the P flag,
d) 1 bit is used for the G flag, and
e) the remaining bits are used for the operand D.

For interpretation, these may be placed where one pleases in the word. The scheme shown in figure 3.1 is a common choice, and is used in the algorithms given in the MINICODE users guide [P].

If one insists that at least 14 bits are needed to represent the address in a machine of reasonable size, then, since the operand D is sometimes an address, it would seem that at

least 20 bits are required per word in the ideal machine. This is more than one has on many minicomputers, so other schemes should be considered.

3.2 Single and double word instructions

For computers with a small word size, an alternative layout for an instruction is as in figure 3.2. Here one uses a single

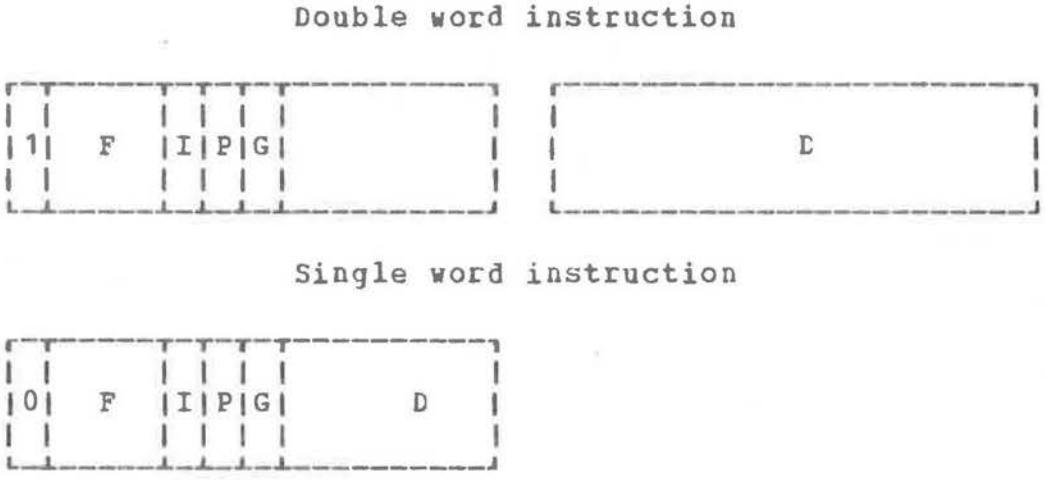

Figure 3.2

word if the operand, D, can fit into what remains of a single word, otherwise two words are used. Of course, an extra bit (it could be the sign bit) in the first or only word of an instruction is used to distinguish these two formats.

3.3 Memory size

Another problem with small computers is memory size. All too often it happens that an important program will just about fit into memory. The possibility of compacting code as much as possible is therefore an important consideration, even though one may have to work hard to accomplish it.

Experience shows that a large number of memory references within the code, in particular for jumps, are a short distance forward or backward from the address of the current instruction. This suggests that the use of relative addressing, i.e., relative to the address of the current instruction, an address contained in register C at the time of fetching, will significantly reduce the number of instructions which must occupy two words. Relative addressing is a well known technique and is available, for example, on the IBM 1130, the NOVA and the PDP-11. The last machine has a single-word instruction, Branch, which will jump to a location within ±128 words of the current one, and a double-word instruction, Jump, which will transfer control to any cell in memory. The saving with this relative addressing technique may be as much as ten percent. While this is not spectacular, if it makes the difference between having a crucial program on the computer or not, then the effort to pro-

duce such code may be well worthwhile.

3.4 Relative addressing

But why is this production of relatively addressed code an effort? Well, INTCODE contains no provision for relative addressing (and should not contain it, in order to maintain machine independence), so the job must be done automatically by the assembler. Moreover, one cannot easily relativize a piece of machine code while assembling it, since one does not know some addresses unless a special pass is made over the assembler code to locate all those assembler instructions which are labelled. But even this extra pass could only be a first step in the process, because making some address references relative will compact the code and this, in turn, will allow some other address references to be made relative. Complete compaction of code by making addresses relative, wherever possible, is therefore an iterative process which must be repeated until no further compaction can be done.

With this in mind, and considering that the code for the minicomputer can be produced on a large computer, it is probably easier to produce nonrelative code, in the first instance, and then to massage and compact the code after it has been produced. But this involves complexities similar to those encountered in garbage collection and is therefore sufficient reason for describing at least one solution here. Close inspection of the octal load module in the introductory section will reveal that it was produced in this way. For a discussion of a similar problem of code compaction, see a paper by D.L.Richards [R4] (another Richards!).

3.5 The assembly process

The initial pass of the assembler on each segment of INTCODE produces octal code with nonrelative addresses. To do this it uses an array with one entry for each label definition in the segment, an array for global values, and an area of storage for storing the code that is produced. An instruction is assembled into one word if its unmodified operand is an integral value which can be held in the D part and the sign bit is set to 0; otherwise, two words are used, the operand is placed in the second word, and the sign bit of the first word is set to 1. At this stage, instructions which reference a location usually occupy two words. Since the machine code produced can contain either instructions with memory references or data words with memory references, and since there is no way to tell the difference between an instruction and a data word, the assembler also keeps an array of pointers, one for each instruction or datum which contains a memory reference. If it is not an instruction, this pointer is stored negatively.

The assembler operates by reading a segment of INTCODE and assembling it into memory. Next, it applies the massaging process described below, and then writes out the resulting code. Prior to massaging, the memory looks as in figure 3.5.

3.6 Code massaging

After a segment of machine code has been produced, the last array of pointers described above is used to help in the compaction process. This is achieved, essentially, as in the

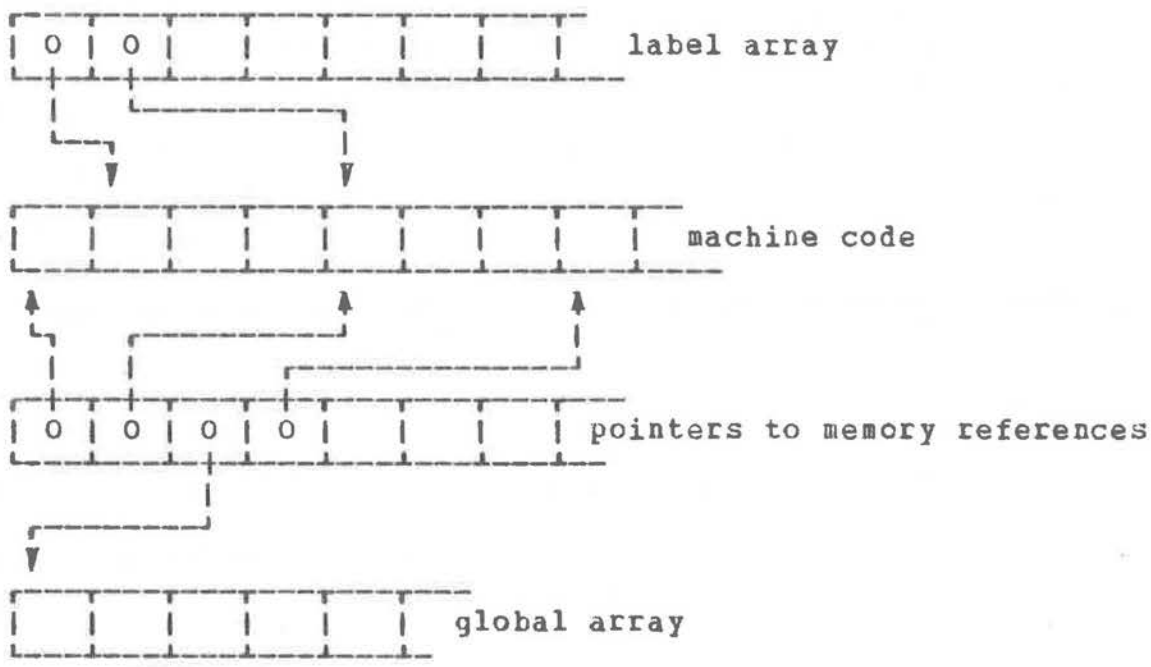

Figure 3.5

following BCPL routine:

```
let compact() be
  { done := true
    relate()
    if done return
    adjust()
    shift() } repeat
```

where 'relate()' modifies the boolean variable 'done'. Thus, 'compact' consists of three parts, 'relate', 'adjust', and 'shift'.

The procedure 'relate' examines each instruction of the code to determine whether an address can be made relative or not. If this is possible, it sets 'done' to false, computes the relative address, sets the R (relative) flag in the instruction (see below) and sets the following word (which originally contained the address), to -1 to indicate that there is a "hole" which may be eliminated later.

The procedure 'adjust' now examines every address reference, whether relative or not, and counts the number of holes between it and the word it references. This number is then subtracted from the address (whether relative or not), if it is a

forward reference, and added to it if it is a backward reference.

The procedure 'shift' now shifts the code to the left in order to remove the holes. At the same time, it adjusts the non-relative addresses by the amount of the shift and corrects the sign bit. The whole process is then iterated until the procedure 'relate' finds no more addresses which can be made relative, i.e., it no longer sets the boolean variable 'done' to false.

In all of this, special care must be taken with addresses in the global vector, since these do not get shifted, although they must be adjusted. Examination of the code for this compaction [P] will reveal that it is the trickiest part of the assembly process. Nevertheless in some applications compactification may be well worthwhile.

It is interesting to note that, to the best of our knowledge, the assemblers for most minicomputers which exhibit instruction formats similar to our ideal machine, do not perform the compaction discussed above (the INTERDATA assembler is an exception). Instead, the programmer is forced to guess whether or not a single-word instruction will do the trick.

3.7 The relative flag

Of course, if one uses relative addressing, then another bit, the R bit, must be sacrificed to indicate this. However, since the use of the P, G and R flags is mutually exclusive, one may superimpose flags in the following manner.

000	no flags
x10	P flag
x01	G flag
x11	R flag (relative address)
1xx	I flag

The layout of the first word of an instruction may then be pictured as follows.

3.8 The load module

Since it is desirable to produce relocatable octal code, the array of pointers which were used in the compaction process, and which were continually adjusted during it, may now be used to determine which words require a relocation factor from the loader. To do this, the octal load module of each section is produced after compaction of its code has been completed. The module is preceded by an appropriate loading address, relative

to zero, and each word needing relocation is preceded by the characters '0+'. The values to be loaded in the global array are produced at the end of the section. The example in the introductory paragraph, 1.2, shows a load module consisting of one section followed by its (only) global value. Of course this is not the most concise representation for a load module. However, this module simplifies debugging and does not cost much more.

3.9 Comments on performance

Some remarks on speed seem appropriate here. The interpreter for most small machines would execute between 10 and 20 instructions for each INTCODE instruction. At first sight, this may seem to be an intolerable slowdown. Yet many programs which run on minicomputers spend most of their time waiting for input or output operations to complete. Viewed in this light, the decrease in speed may not even be noticeable. Of course, in many non-real-time minicomputer applications, we may not even care whether the program runs in one minute or two, a luxury not available on a large machine. Also, if on may run more complex programs now, because INTCODE is more compact than a machine's native order code, then the slowdown in speed may still be tolerable.

In any case, the simplicity of the interpreter shown in the users guide [P] suggests that it may be microprogrammed fairly easily on a machine with such a facility. Such a microprogrammed interpreter might reasonably be expected to run at about the speed of the interpreters provided by the manufacturers.

4 The assembler language

The assembler language, as accepted by the algorithms of the users guide [P], has

executable instructions,
storage reservation instructions,
pseudo instructions, and
pragmats (run-time option settings).

For the executable instructions, the mnemonics are L, S, A, J, T, F, K and X, as explained in section 2.1. These may be followed by an optional I, indicating indirect addressing, and then by an optional P or G, indicating index register modification of the operand. This is followed by the operand proper, which is either a non-negative integer or a label reference. A label reference is the letter L followed by a small non-negative integer (<500). Note that there is no ambiguity between the use of the letter L for load, and the use of the same letter in a label reference. Examples of executable instructions are L13, LG13, LIP6 and LL499. Defining occurrences of labels are just non-negative integers, e.g., 499, and both executable and storage reservation instruction may be preceded by one or more defining occurrences of a label.

There are three storage reservation instructions, each

using one of the the letters D, C or G. Dn indicates a data word with value n, where n is an integer, and DLn indicates a data word storing the address of label number n. Cn indicates storage of a character whose ASCII or EBCDIC value is n. GnLm indicates storage of the address of label number m at the n-th global word.

A pseudo-instruction which indicates the end of a control section is Z. Label references may not refer to label numbers of other control sections.

The assembler given in the users guide [P] allows for the inclusion of pragmats. These pragmats may appear in the INTCODE source as an asterisk followed by the pragmat item. Alternatively, a sequence of pragmat items may be passed as a parameter string when the assembler is loaded. The details of these pragmat items may be found in the users guide. It is sufficient to say here that they specify the following things:

word size (in bits), character size (in bits), internal character coding (ASCII or EBCDIC), listing of the source, suppression of code compaction, insertion of special instructions at entry points, e.g., the PDP11 trap instruction, and the generation of run time tracing instructions.

Comments may appear and consist of the symbol "/" and then everything up to the newline character. A dollar symbol, $, may be used at a procedure entry. It allows generation of the trap instruction discussed above. If this precedes a defining label, then the load address of that label may be displayed as a comment in the octal load module. This is useful for debugging purposes. The use of a dollar symbol at a procedure entry also helps a translator to recognize an entry place in a load module.

4.1 Strings and the case statement

Character strings are generated as sequences of storage reservation instructions, e.g., Cm Cn1 Cn2 ... Cnm, where m is the number of characters in the string and n1, ..., nm are the ASCII or EBCDIC values of those characters. To be of any use, this should be preceded by a label. Characters are packed into words by the assembler in accordance with the values in pragmats following *W and *C, e.g., *W16 and *C8 will mean two characters per word.

Case statements are generated by the BCPL-to-INTCODE compiler in the form X23 Dn DLd Dv1 DLd1 ... Dvn DLdn, where n is the number of cases, d is the default label number, v1, ... , vn are the case values, and d1, ... , dn are the corresponding case label numbers. The assembler given in the users guide will accept all of this, digest it, and then produce whatever code is less wasteful of storage, i.e., a linear search if the values are widespread and an indexed jump otherwise. This optimization might be done, instead, by the compiler.

4.2 MINICODE - a humanized INTCODE.

The INTCODE language was designed to be compact and easy for a machine to translate or assemble. It was not intended for use as a programming language for humans. Despite this, its simplicity, and the simplicity of the machine on which it runs, allow one to read it quite easily. This tempts one to determine whether small changes in the assembler will bring further improvements in readability. One obvious candidate is a character string. For example, it is easier to read

```
499:C"F(%N) = %N*N"
```

than

```
499 C11 C40 C37 C78 C41 C32 C61 C32 C37 C78 C42 C78
```

and the additional work for the assembler is minimal. For character constants it is easier to read

```
L'S' SP7 L'I' SP8 L'D' SP9
```

than it is to read

```
L83 SP7 L73 SP8 L68 SP9 .
```

For the execute instruction it is easier to read

```
LIP2 LIP3 X+ SP4
```

than it is to read

```
LIP2 LIP3 X8 SP4 .
```

The improvements suggested here, all of them easy to add to the assembler, have led to a new more general assembler language which we shall call MINICODE. The users guide [P] shows a version of the BCPL compiler which translates to MINICODE (except for character constants).

With readability in mind, the assembler given in the users guide will accept either the original INTCODE or the new MINICODE which includes INTCODE. This opens up the possibility that it might be understood more easily as a primitive assembler language by humans. The symbols chosen for the register execute operations in MINICODE are as follows:

X1	X!	dereference register A
X2	XN	arithmetic negation
X3	X¬	logical negation
X4	XR	return from procedure
X5	X*	multiply
X6	X/	divide a:=b/a
X7	X/*	remainder a:=b/*a
X8	X+	add
X9	X-	subtract a:=b-a
X10	X=	equal
X11	X¬=	not equal
X12	X<	less than a:=b<a
X13	X>=	greater equal a:=b>=a
x14	X>	greater than a:=b>a
X15	X<=	less equal a:=b<=a
X16	X<<	shift left a:=b<<a
X17	X>>	shift right a:=b>>a
X18	X/|	and
X19	X|/	or
X20	XE	equivalent
X21	X¬E	not equivalent

```
X22   XF    finish
X23   X?    Case
```

4.3 An example

To illustrate the use of MINICODE we give an example in which the comments show the original BCPL source. Observe that it is now easier to follow what is happening. Indeed, it would be possible to use such an example to show the relationship between a high level language and the low level code into which it translates.

```
JL3
                          /    LET HANOI(N, S, I, D) BE
$ 1:LIP2 L0 X<= FL5 XR    /    $( IF N <= 0 RETURN
                          /    HANOI(N-1, S, D, I)
  4:LIP2 L1 X- SP8 LIP3 SP9 LIP5 SP10 LIP4 SP11 LIL2 K6
                          /    WRITEF("MOVE %N FROM %C TO %C*N",
                          /         N, S, D)
  LL499 SP8 LIP2 SP9 LIP3 SP10 LIP5 SP11 LIG76 K6
                          /    HANOI(N-1, I, S, D)
  LIP2 L1 X- SP8 LIP4 SP9 LIP3 SP10 LIP5 SP11 LIL2 K6
  XR                      /    $)
 3:JL6                    /    LET START() BE
$ 5:7:LIG70 K2 SP2        /    $( LET N=READN()
  LIP2 L0 X<= FL8 XF      /    IF N<=0 DO FINISH
                          /    HANOI(N, S, I, D)
  8:LIP2 SP5 L'S' SP6 L'I' SP7 L'D' SP8 LIL2 K3
  JL7 XR                  /    $) REPEAT
 6:XF
2:DL1
499:C"MOVE %N FROM %C TO %C*N"
G1L5
 Z
```

5 The interpreter

The loader-interpreter is fairly simple, involving about two hundred lines of BCPL code. It has already been written in several languages, as the listings in the users guide show.

5.1 Implementation of the interpreter

Implementation of the loader and interpreter is not difficult. In the first instance it requires only two routines to be supplied by the user, viz., those which read and write one character. One may then choose a version in one of the languages listed in the users guide.

Pragmats may be supplied to the interpreter either on comment lines, i.e., after a semicolon, in which case each pragmat item is preceded by an asterisk, or as a parameter string on the system run command. These pragmat items allow one to control

word size, character size, internal character coding (ASCII or

EBCDIC) and various run time tracing features. For the details see the users guide [P].

5.2 The library routines

There is a minimal set of library routines consisting of procedures written originally in BCPL. These can be kept as INTCODE source and assembled with each program, or kept as separate pre-assembled octal load modules to be loaded concurrently with the program. In the minimal library there are just two routines which are primitive in the sense that they communicate with the operating system and must therefore be provided by the implementer. These are the BCPL character input and output routines, 'rdch()' and 'wrch(c)'.

System routines may be added to the library by coding them in the language of the interpreter as additional execute instructions. For example, if we wish to introduce a function called 'f', then the BCPL source of the library should contain (assuming global 33 is available),

```
global { execute:33 }
let f(a, b) = execute(n, a, b)
```

where n is the number of the execute instruction, Xn. When this library is translated to INTCODE, only the following additional hand coded line need be added:

```
$100 LIP4 LIP3 XIP2 X4 G33L100
```

(assuming that label 100 is available). This hand coded fragment transfers the second and third parameters to the A and B registers, where the function 'f' may, or may not, use them. Any value delivered is left in the A register. Examination of the library in the users guide should make this process clear.

Further primitives may be provided if one wishes to handle files or service interrupts, but the two given are sufficient for the definition of the other standard input and output routines of the usual BCPL library. Two of the routines 'putbyte(s,i,byte)' and 'getbyte(s,i)' help in the packing and unpacking of characters into words. These are written to use machine defining constants as follows:

```
manifest { chars.per.word=4; char.size=8; char.mask=#XFF
  ch.p.w.m.1=1 }

let putbyte(s, i, byte) be
  { let j = i / chars.per.word
  and shift = (ch.p.w.m.1 - i rem chars.per.word) * char.size
  let mask = not (char.mask << shift)
  and char = (byte & char.mask) << shift
  s!j := (s!j & mask) | char }

and getbyte(s, i) =
  (s!(i/chars.per.word) >>
```

(ch.p.w.m.1 - i rem chars.per.word)*char.size) & char.mask

and are included in the library to be used in the first instance. Eventually more efficiency can be achieved by treating these two also as primitives which are supplied by the implementer.

6 Didactic possibilities

The teaching of computer science often begins with some high level language, in order to study algorithms, and then continues with the examination of machine architecture using the assembler language of the computer at hand. Very often these two things are not closely related, in particular, because the compiler for the high level language is as a closed book to the student and usually also to the instructor. Moreover, many modern computers are quite complex, and it often happens that, in teaching machine architecture, the basic principles which should be taught become bogged down in a morass of complex detail. In addition, the high level languages supplied by the manufacturer, such as FORTRAN and BASIC, do not lend themselves to the teaching of structured programming, nor to instruction in such basic principles as stack manipulation and recursion. What should be done about it?

The answer seems to lie in the choice of a simple machine from which the basic ideas are derived naturally. From what has been said above, it is clear that the INTCODE machine may have interesting possibilities. Some may argue that this is not the real world, and that teaching toy computers is not effective. In answer to this, it may be said that INTCODE is surprisingly close to the structure of some minicomputers, and that, with the great surge in the use of minicomputers, INTCODE may be closer to most of the real world than the structure of some large complex machine.

Consider then an introductory course in computer science using the INTCODE ideal machine. The student first studies the ideal machine, its simple construction and its modest set of operations. He then studies some simple programs written in MINICODE. These could well be the MINICODE versions of the decimal input and output routines 'readn()' and 'writen(i)' of BCPL. Remember that these routines are written in terms of just two primitives, the character input and output routines 'rdch()' and 'wrch(c)'. The emphasis here would be on how machines work rather than on how to program them. Study of existing well written algorithms would be the first step in the inculcation of good programming habits.

At this stage the student would understand the ideal machine and how it works and would be able to read, but perhaps not write, MINICODE. It is then time to introduce the high level language BCPL. This can be done by looking at the BCPL versions of the same input and output routines. Now one may branch out into the task of writing other algorithms in BCPL.

Since both the assembler and the interpreter are written in BCPL, the study of algorithms could soon take these as examples. The whole process would then become clear. There would be no mystery. The student would understand what an assembler is, what an interpreter is, and he could study the algorithms for assembly and interpretation for himself. He could then understand that there is a compiler from BCPL to INTCODE, for he could examine and check the INTCODE produced from his favourite program. The curious could also study the BCPL-to-INTCODE compiler, which is written in BCPL. A detailed examination of it would not likely be made in an elementary course, although parts of it could be studied with profit.

Consider then the advantages. In one short course, the student will have seen

a) machine architecture,
b) assembler language,
c) a good high level language,
d) the construction of an assembler,
e) the construction of an interpreter,
f) and, for the adventurous, an inside look at an interesting compiler.

A further advantage is that the basic machine is stack oriented, so that the ideas of recursion are immediately clear and natural rather than being some strange mystery that one has to struggle with at a later stage.

6.1 MINICODE and the high schools

Both the assembler and the interpreter are easy to implement even on a small machine. This can make one independent of the manufacturer's software at an early stage. These last facts are of some critical importance when one considers that a large part of introductory computer science is now moving down to the high schools, where it is natural that the computer to be chosen is a minicomputer, and where the manufacturer's software will have undue influence. The MINICODE system, as outlined above, now offers a way in which the essence of computer science can be taught, all within one programming system.

References

[P] J.E.L.Peck, The MINICODE system users guide, U.B.C., Vancouver, 1975.

[R1] M.Richards, Bootstrapping the BCPL compiler using INTCODE, Cambridge University, August 1973.

[R2] M.Richards, INTCODE -- An interpretive machine code for BCPL, Proceedings of the IFIP Trondheim Conference on Machine Oriented Higher Level Languages, North-Holland, 1973.

[R3] M.Richards, BCPL - A tool for compiler writing and system

programming, Spring Joint Computer Conference, 1969, pp. 557-566.

[R4] D.L.Richards, How to keep the addresses short, Comm. Assoc. for Comp. Mach., Vol 14 (1971) pp. 346-349.

[S] Stoy,J.E. and Strachey,C., OS6 - An experimental operating system for a small computer, The Computer Journal, 15, Nos 2 and 3 1972.

BEST-PRINTER CO. LTD.
VANCOUVER, B.C.