Improving The Portability Of The BCPL Compiler

R. D. Eager

Submitted in partial fulfilment of the requirements for the degree of Master of Science at the University Essex.

CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Richard Bornat of the Computing Centre, University of Essex, who suggested the idea of INTCODE 2, and with whom I had many useful conversations about it.

The project would also have been impossible without the help of Pete Gardner, also of Essex,who wrote the PDP-10 code generator on which the INTCODE 2 code generator is based.

Mention must also be made of Dave Lyons, once again of Essex, who put forward several helpful suggestions, and Dr. Peter Brown, of the University of Kent, who has discussed various aspects of software portability with me during the past two years.

ABSTRACT

An interpretive machine code, called INTCODE 2, is described. It is intended for use in bootstrapping the BCPL compiler onto a new machine. Certain advantages are claimed over earlier interpretive methods, and these are discussed. A code generator for INTCODE 2 has been written, suitable for interfacing to the BCPL compiler currently in use at the University of Essex. The specification of an ANSI FORTRAN IV loader/interpreter system for INTCODE 2 is also given, and discussed in some detail.

INTRODUCTION

1. The BCPL language

BCPL is a general purpose programming language which was originally designed as a tool for compiler writing and other system programming applications. The name derives from that of the language on which it is based, the initials standing for "Basic CPL". CPL was a language developed by Cambridge and London Universities in the mid-sixties.

BCPL is a block structured language with some similarity to ALGOL 60, and like ALGOL, it has recursive functions. However, it has but a single data type - a bit-string which occupies a single memory cell - and no type checking is performed either at compile time or at run time. It contains several restrictions not found in ALGOL or most other high-level languages, the purpose of these restrictions being to allow more efficient code to be generated for any given program. For further details see {1}.

2. The portability of BCPL

BCPL was designed to be a portable, high-level "assembly language". It has a self-compiling compiler, since, as stated above, it was designed with compiler writing in mind. As such, it is an ideal language in which to write a BCPL compiler. One might even say that the writing of BCPL compilers is a major application of BCPL!

Due to the portability requirement, the method for compiling BCPL differed somewhat from that used for other languages. Programs were compiled first into an intermediate object code for a pre-defined abstract machine (called the O-machine), this object code being known as OCODE. The OCODE was then translated in a separate pass into machine code for the particular computer on which the program was to be run {2}. This method meant that if the BCPL compiler, or indeed any other program written in BCPL, was to be transferred to another machine (henceforth referred to as the "target" machine), all that needed to be available on the target machine was a code generator to convert the OCODE form of the program into the appropriate machine code. For bootstrapping purposes when producing a new implementation of the compiler, a simple non-optimising version of this code generator was comparatively easy to produce.

3. The OCODE abstract machine

To quote Martin Richards (one of its progenitors), OCODE was a "macro-like low-level language" {2}. It was thought that the final translation from OCODE to machine code could possibly be done by using a general purpose macro processor such as GPM {3} or, later on, ML/I {4}. To this end, OCODE was normally distributed to intending implementors in character form, although production systems usually used a binary form since this was faster and easier to process. It was, and still is, very difficult to write

macros to map OCODE into assembly language for a target machine, as OCODE statements may have a variable number of arguments, the last argument not being marked in any special way. The current state-of-the-art in macro processors does not extend to anything that can handle this kind of input, {5}. Instead, the number of arguments was specified as (usually) the first argument of the statement. I have, in fact, succeeded in mapping OCODE into assembly language for an ICL 4130, using ML/I as a mapping tool, but the resulting code was impossibly large and inefficient, although part of the blame must be attributed to the unsuitability of the target machine, which has only one accumulator and only one index register.

OCODE contained fifty-six different statements, each starting with a key word followed by the arguments. Spaces and new lines were treated as separators and had no other significance. The arguments could be either positive or negative integers, or labels. A label was specified by the letter L followed by a positive, non-zero integer.

OCODE expression evaluation made extensive use of a run-time stack, which, although necessary for the implementation of recursion, was also used to hold the Reverse Polish forms of expressions for evaluation. This made the job of writing a good optimising code generator to produce conventional machine code a decidedly non-trivial matter, especially as (for a new implementation of BCPL) it was often done by someone with little or no initial knowledge of BCPL, and OCODE leaned heavily on BCPL itself.

4. The INTCODE approach to improved portability

A simpler method of implementing BCPL was later devised. This was to distribute the compiler in the form of an interpretive machine code called INTCODE. To avoid later confusion, I shall refer to this as INTCODE 1 from now on, although it was never actually known as such.

INTCODE 1 was compact, and its assembler and interpreter were trivial to implement since the language consisted of only eight main types of statement {6,7}. However, it was not particularly readable, was still strongly dependent on the structure of BCPL, and required an input-output library for the target machine to be written in INTCODE 1. It did have the advantage that less initial knowledge of BCPL was required, and the task of writing the library was eased because most of it differed little from machine to machine.

5. The INTCODE 2 abstract machine approach

Both of the above methods of portability suffered from the defect that their abstract machine (which is a "stack" machine) bore little resemblance to most "real" machines (which are "register" machines), excluding machines with hardware stack operations such as the Burroughs B5000, the English Electric KDF9, and more recently, the ICL 2900 series. This made the code obscure and difficult to debug. However, the INTCODE 1 method in particular still finds favour among users of small machines, who use the interpreter as their main

BCPL system. The method gives a considerable advantage in the size of the code generator (about 4 to 6 times smaller than with the OCODE method) although the execution speed naturally suffers. The speed factor is of the order of a tenfold increase in time taken.

I thus developed a new interpretive machine code to aid transportation of the new University of Essex BCPL compiler. This compiler is of conventional form, with a production version generating code directly from the syntax tree rather than via some intermediate form. It has been written with portability in mind, insofar as some attempt has been made to separate the machine-dependent and machine-independent sections of the logic. I am of the opinion that this has only been partially successful.

The new interpretive machine code is known as INTCODE 2, which, though still capable of closely modelling the operations required by most BCPL programs, bears a close resemblance to the order code of a typical medium-sized machine. As such, it is quite unlike OCODE, and is much nearer, say, the intermediate language discussed in papers on the UNCOL concept, {8}. However, one does not get something for nothing, and the price paid in this case is an increase in the size and complexity of the interpreter, and an increase in the size of the program being interpreted. To partially offset this, INTCODE 2 has been designed with compactness of code being given priority over execution speed, as usually you can wait

longer for a program to run, but there is rarely any easy way of increasing the storage available on a machine beyond certain fixed hardware limits. The question of interpreter complexity is dealt with by supplying intending implementors with a ready-written interpreter (see below).

The INTCODE 2 code generator is primarily intended as an exemplary code generator which intending implementors are meant to use as a basis for writing a code generator suitable for the target machine. To make this possible, INTCODE 2 has to look like a typical order code for a medium-sized machine, to reduce the machine-dependence of what is, after all, a relatively machine-dependent sort of program. It is structured so that modifications, necessary because of some peculiarity of the target machine, are easily incorporated.

The word size of the INTCODE 2 abstract machine is not specified, but should be at least 16 bits. The implementation "kit" consists of the interpreter together with source and INTCODE 2 copies of the parser and the INTCODE 2 code generator. The interpreter is written in ANSI Standard FORTRAN IV, this being probably the most widely available high-level language, with input-output facilities which usually require the minimum of alteration to a program at different installations. Another argument for the use of FORTRAN is that, in general, FORTRAN programs compile into relatively efficient code due to the restrictive nature of the language. However,

if, for some reason, FORTRAN were not available on the target machine, it would not be too difficult to write the interpreter in some other language, due to the conventional nature of the INTCODE 2 abstract machine and the restricted format of an INTCODE 2 program. (The restriction on format has been included in deference to the requirements of Standard FORTRAN). This problem is unlikely to arise, though, as the widespread use of FORTRAN almost forces hardware designers to produce machines capable of processing it, and manufacturers' software departments to provide compilers for it. For a detailed account of Standard FORTRAN, see {9}.

6. Use of a macro processor as a tool for implementing INTCODE 2

I considered that it would be a good idea to make the format of INTCODE 2 suitable for input to a marco processor, to provide an alternative method of implementation. This method would involve writing macro definitions to map INTCODE 2 directly into some language acceptable to the target machine, and the use of a general purpose macro processor such as ML/I {4}, or STAGE2 {10}. The macro processor used would have to be capable of operating in free mode, that is, without a warning character preceding each macro call. This is necessary since the macros may be needed to adjust format, and in any case, if a warning character were used, it would have to be included as part of INTCODE 2, which is clearly undesirable since the actual character used would almost

certainly vary between different macro processors. Thus GPM would be unsuitable, but STAGE2 would be acceptable because it operates in free mode at all times, and so would ML/I since it can run in either free or warning mode, the choice being left to the user. A powerful macro assembler might also be used on some machines.

It soon became evident that suitability for macro processing conflicted with other requirements. One example of this is the method used for compiling forward references. The backward chaining technique is employed, and this kind of information is hard to pass through a macro processor and hence to an assembler program. The method used in OCODE, which is to generate unique numerical labels as they are required, is far more amenable to macro processing. This was not done originally because of the extra complexity involved in the loader.

Another snag with the macro-processor approach is the input-output library. This is incorporated in the interpreter in the standard INTCODE 2 system, but it would have to be hand-coded in the case of implementation by macro processor. In addition, the overall system would probably be very slow and unsuitable for use as even a temporary production system. This is because macro processors are notoriously slow programs, although against this must be set the fact that they are readily available on many machines due to their own inherent portability {11}.

I therefore decided to concentrate on implementation using an interpreter. Despite this, it would probably be

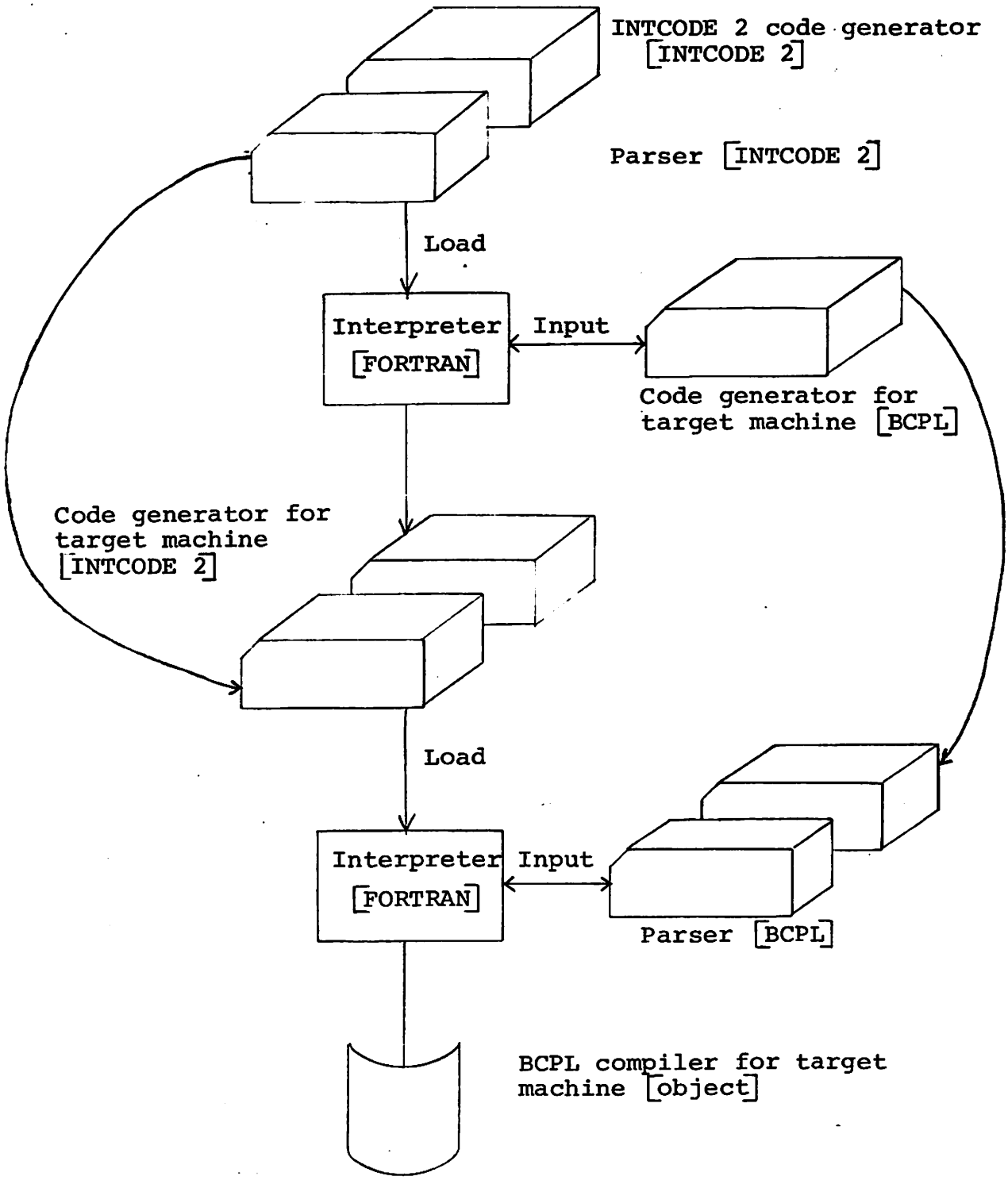
possible to modify the INTCODE 2 code generator to produce code suitable for input to a macro processor. This could well form the basis of a subsequent project.

7. The procedure for transfer of the compiler

The procedure for transfer of the compiler to a target machine, given the "kit" mentioned above, is as follows.

Once the interpreter has been modified to fit in with any local FORTRAN idiosyncrasies, the INTCODE 2 version of the compiler is loaded and debugged. This is the original parser and the original code generator for INTCODE 2. A code generator for the new machine is then written in BCPL and compiled, using the interpreted compiler, into INTCODE 2. The parser, as supplied in the "kit", and the new code generator are then loaded, producing a full compiler for the target machine, but running under the interpretive system. The parser and the new code generator (in source form) are then compiled using this, the result being a machine code version of the BCPL compiler for the target machine. A diagram of the procedure is given below.

This procedure has the advantage that the debugging loop for the new code generator encompasses only one machine (the target machine), thereby reducing the problem of inter-machine data exchange to a single occasion. The method is often known as "pulling", since the compiler is "pulled" onto the target machine with only initial assistance from the donor machine. There is evidence that this method works well for other pieces of software, for example



PROCEDURE FOR TRANSFER OF THE BCPL COMPILER USING INTCODE 2

the STAGE2 macro processor {10}. An example of the opposing technique ("pushing") is given by the implementation of the ML/I macro processor {11}.

Another advantage is that, as with INTCODE 1, there is no necessity for a new code generator to be written at once, since the INTCODE 2 system will function adequately (though probably very slowly) as a production system. This allows the implementor to get the "feel" of BCPL before writing the code generator.

8. Writing the new code generator

There is another justification for designing the INTCODE 2 abstract machine to appear much like a conventional machine. This is that the task of modification of the code generator for a new machine is a lot easier if the INTCODE 2 machine and the target machine are similar in architecture.

The writing of the new code generator is not a trivial operation, and in the past it has been customary to modify an existing code generator rather than to write one from scratch. The INTCODE 2 code generator has been written so that it is easy to modify for different target machines. All machine-dependent sections have been kept together as far as possible, and thus the task of writing the new code generator for the target machine is reduced to that of rewriting certain parts of the INTCODE 2 code generator, and changing certain compile-time constants. It is here that the necessity for an exemplary code generator is evident; the changes required are mainly in header files, not the main program body. These header

files define selector ("bit-picking") formats for packing information into a machine word, and clearly they need modifying for machines with differing word sizes. They also define machine-dependent constants such as word size, number of bits occupied by a character, etc.

9. Layout of the following Chapters

Chapter 1 describes the INTCODE 2 machine language in some detail, and explains the reasons behind its eventual form. Chapter 2 gives an account of the INTCODE 2 code generator, in particular the methods used to make it as exemplary a program as possible.

Since the code generator on which the INTCODE 2 code generator is based was still in the development stage, and INTCODE 2 along with its code generator were still being designed, the interface between the parser and the code generator was somewhat fluid. This meant that writing and debugging the code generator took much longer than anticipated, and it was not, therefore, possible to write and test the interpreter in the time available. The interpreter was fully specified in all important aspects since it is closely related to the structure of INTCODE 2 itself. Chapter 3 thus contains a specification of a probable form of the interpreter, while Chapter 4 concludes by, among other things, examining the usefulness of this method of compiler portability compared to other methods in current use.

CHAPTER 1 - THE INTCODE 2 ABSTRACT MACHINE

1.1 Design requirements

The INTCODE 2 "abstract machine" had to satisfy several important design constraints. It had to be a fairly conventional register machine, in order that its code generator could be used as a model for subsequent code generators on various target machines. It also had to have a small word size, so that a wide range of target machines could be catered for. To this end, it was decided to make the word size a variable quantity, with the proviso that it had to be at least 16 bits in order to accommodate all the fields of an instruction. The number of accumulators was also made variable for similar reasons, the minimum number allowed being three.

The restriction to a 16-bit word proved to be a major design headache, since it was necessary that most instructions should occupy only one word. This naturally produced many problems when it came to designing methods of accessing storage, since far too few bits were available in a single word for specifying a reasonable number of different storage cells. Bit fields were also required within an instruction word for specifying such things as address indirection, floating point operations, stack addressing, accumulator numbers, and the presence of immediate (sometimes known as "literal") operands.

It was decided at a fairly early stage that the INTCODE 2 machine should have either address indexing or indirection capabilities, but not both. This was partially

due to the above-mentioned problem of how to specify everything in a single word, but mainly to the fact that since not all real machines in common use have both, the code generator (which is meant to be exemplary - see Section 5) would require considerable modification to use only indexing or indirection. Some machines, for example the ICL 1900 series allow only some of the general purpose registers to be used for indexing purposes, and this complicates the task of register allocation.

It is, however, fairly simple to model indirection using one index register, so the course adopted was to allow indirection to a single-level only, and to incorporate no index registers in the INTCODE 2 machine architecture. This also had the advantage of requiring only one bit in an instruction word to indicate indirection of the specified address.

Another design decision was the number of accumulators to be incorporated in the INTCODE 2 machine. The final decision was affected by the difficulty of using an accumulator 0; there are two reasons for this. The first is that the original PDP-10 code generator does not use accumulator 0 in any of the code it compiles, and when accumulator 0 is passed as a parameter between its internal routines, it has a special meaning. This will be briefly explained. The register allocation is done by a routine called NEXTREG, taking a single parameter, which may be a positive or negative accumulator number or the value zero. Zero means that any free accumulator is acceptable, a negative number means that any free accumulator, except the one specified by the absolute value of

the parameter, is acceptable, and a positive value requests that the accumulator specified by the parameter is to be allocated if it is free. The effects of this convention range over the whole code generator, and there was not sufficient time to change all the routines depending on it. The second reasons for not using an accumulator 0 is due to the expected form of the INTCODE 2 interpreter system. If the interpreter (see Chapter 3) were written in FORTRAN and it modelled the accumulators as store locations, an accumulator 0 would require special treatment because FORTRAN arrays do not have zero elements.

To minimise accumulator dumping during expression evaluation, at least two accumulators were required. It was finally decided to use three, since two bits were required in any case for the accumulator field of an instruction, if accumulator 0 was not to be used. One way round this might have been to have the interpreter mapping each accumulator reference into a number one larger, then two accumulators could have been specified by only one bit.

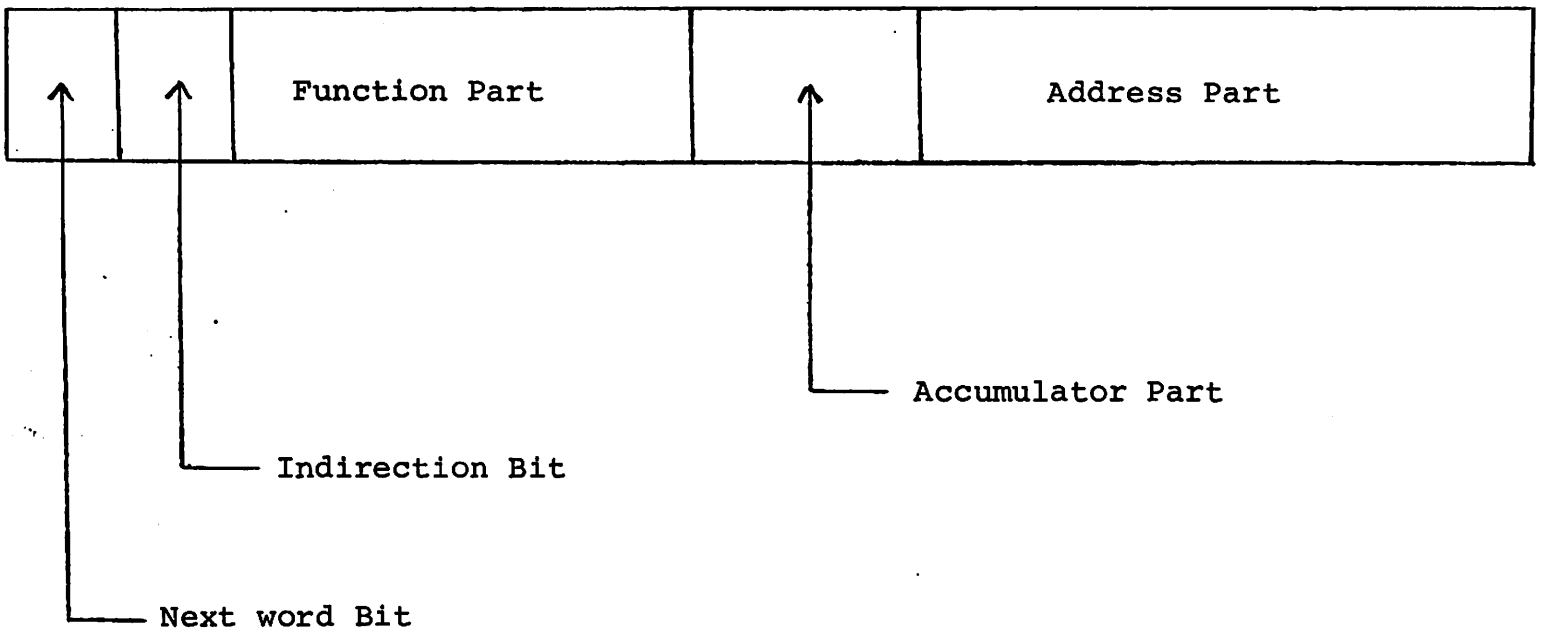
The above decision does not affect the exemplary nature of the code generator as a whole, and it is easy to change the appropriate parts to use less accumulators if this is required when writing a production code generator.

Another important aspect of the INTCODE 2 machine design was the variable number of words that an instruction may occupy. Multi-word instructions are necessary in order to correctly specify addresses which exceed the capacity of

the address part of a single-word instruction. This includes (for all practical purposes) every jump instruction. It was necessary to earmark a bit in the instruction word to specify whether the instruction was continued in the following word. Three-word instructions are the largest that occur, and these are identified by the fact that they are all "byte selection" operations.

It was found that a reasonable set of instructions totalled about fifty-five in number. (Many instructions perform similar functions with differing types of operands). Clearly, therefore, six bits were required for an operation code field in an instruction word. This left six bits for the address part of an instruction, assuming the "worst case" of a 16-bit machine. A diagram of the word layout is given below.

The six-bit address field is useless for most applications. A "stack" variant on all relevant instructions was thus allowed, since most operations in a BCPL program tend to be on local variables located in the current stack frame. Use of more than one operation code, to specify different types of operand for the same operation, allowed this to be included without using any more bits in an instruction word. Thus, it is possible, even on a 16-bit machine, to access at least the first 63 variables in the current stack frame, using only a one-word instruction. In addition an immediate (or "literal") operand is possible, if sensible, with a value of up to 63 with a 16-bit one-word instruction. This value of 63 obviously increases if a larger word size is used.



LAYOUT OF AN INTCODE 2 INSTRUCTION WORD

It was still necessary to specify variations on an instruction, such as the fact that some operands were to be treated as floating-point numbers. Although many BCPL compilers do not offer floating point facilities, I felt that an exemplary compiler should cater for them, as otherwise a form of "inbreeding" sets in where the compiler's "offspring" do not have such facilities either. Usually the only way to include them afterwards is to code them in machine code and patch them in, never a good idea at the best of times. (This is discussed further in {12}, Chapter 2.2). However, floating point operations do not feature extensively in the compiler, and I considered it acceptable that such operations should always occupy two words. Thus, the address part of the first word of a two-word instruction is used to store information on such instruction variants. ("Waste not, want not" - Proverb).

A final problem was the exact treatment of the accumulators. Should they be treated as low-address store locations, as on the DEC PDP-10 and the ICL 1900, or should they appear as hardware non-addressable registers such as are found on, say, the DEC PDP-11? A compromise was eventually worked out where they are effectively treated as store locations, but an entirely separate set of operation codes is used to specify inter-accumulator operations.

1.2 The final form of the INTCODE 2 abstract machine

Such difficulties as those detailed above caused the development of the INTCODE 2 machine to become an

essentially iterative process, despite the most careful initial design. For instance, I did not realise at first that use of the accumulators as store locations would require special treatment, not only for the reasons given above, but also because any instruction referencing an accumulator as a store location would have occupied two words! Similarly, the difficulty in the use of accumulator 0 did not come to light until I studied the code of the PDP-10 code generator in detail.

1.2.1 General description of the machine

The INTCODE 2 machine is a conventional single-address stored program register machine, with a store consisting of equal sized cells addressed by consecutive integers. Program and data are not mixed within a given load module, and no code modification takes place at run time apart from that given by the label and procedure assignments allowed by BCPL itself. As such, it is similar to the BCPL "machine" and most modern word machines. However, a problem arises when dealing with byte machines such as the IBM 360, where the addresses of adjacent words do not differ by one. (In this case, the value is four, since there are four bytes to a 360 word). This has always presented an implementation problem for BCPL, and the usual solution is to store an address divided by four, then convert to a byte address (i.e., multiply by four) whenever an operation such as vector application is required. The INTCODE 2 machine is insulated from such worries since its store, as

modelled by a FORTRAN interpreter, is of conventional form. The implementor of a new code generator needs to be wary of this point.

The word size of the INTCODE 2 machine is implementation dependent, but as stated in the Introduction, it should be at least 16 bits in order to accommodate all the fields of an instruction. As stated above, the number of accumulators should not be less than three, but whether more are used is again up to the implementor.

There is one auxiliary register, the link register, (L). This is used to hold the return link following a call to a routine, being set by the jump to the routine. Its contents are stored in the current stack frame by the entry code of the actual routine. Splitting up the routine call into two distinct parts has the advantage that should, say, a jump to location zero occur, (perhaps due to not initialising a variable used to hold any entry point) some idea of the source of the undefined jump is available for debugging purposes.

Mention must also be made of the stack pointer, (P). This points at the base of the current stack frame. It is altered by operations such as routine calls, which create a new stack frame, and returns from routines, which restore a previous stack frame. It is referenced implicitly by all instructions with a "stack" operand.

1.2.2 Format of an INTCODE 2 instruction

The format of an INTCODE 2 instruction, as it appears in store, is now discussed. This format may

seem rather complex, but this is defended on the grounds that it is the only (?) way of packing so much information into a machine word which may be only 16 bits wide. In fact, the method used is quite logical and easy for the interpreter to decode.

An INTCODE 2 instruction word (or the first word in the case of a multi-word instruction) consists of five parts. They are: function part, accumulator part, address part, next word bit, and indirection bit. These are covered in detail below.

1.2.2.1 The Function Part

This is a six-bit field specifying the particular operation to be performed. The various possible operations are discussed in a later section and summarised in Appendix A.

This part of the instruction, in conjunction with the next word bit, governs the interpretation of the address part.

1.2.2.2 The Accumulator Part

The accumulator part of an instruction is variable in size, depending on the number of accumulators used. It specifies the number of the accumulator which is to be operated on during the execution of the instruction. For some instructions, such as unconditional skips, its value is immaterial and undefined. However, it is used in one case to convey extra information about the instruction, in order to economise on operation codes.

Since the minimum number of accumulators allowed is three, the minimum width of this field is two bits.

1.2.2.3 The Address Part

This field also has an implementation-dependent width. Its size is equal to the word size less all the space required for the other fields. For example, on a 16-bit machine, using three accumulators:

Function	:	6 bits
Accumulator	:	2 bits
Next word	:	1 bit
Indirection	:	1 bit
Total ...	:	10 bits

Therefore, six bits remain for the address field. In fact, the size of the address field must be at least the size of the accumulator field plus four bits, which means in practice that the configuration of a machine word given above is the only possible one for a 16-bit machine. The extra four bits are used in the "accumulator operate" instructions - see section 1.2.3.1 below.

The exact meaning of the address part depends on the operation code and the status of the next word bit. The various possibilities are outlined in section 1.2.3

1.2.2.4 The Next Word Bit

The function of the next word bit is simply to indicate whether the word occupying the location

immediately after the word containing the operation code, forms part of the instruction, (i.e. whether the operand is specified by the first word of the instruction). A zero in this position means a one-word instruction, a one means that two words are used.

A byte selection operation uses a further word in either case, to specify the selector to be applied. The interpreter knows about this extra word because it only occurs with the byte selection operations, and is thus processed automatically by the section of the interpreter which deals with such operations.

1.2.2.5 The Indirection Bit

The indirection bit is a single bit which specifies whether the effective address, as calculated, is to be replaced by the contents of the storage cell specified by such an address. A one in this position indicates that the replacement is to take place, and a zero that it is not.

1.2.3 The use of the address part in an INTCODE 2 instruction

The exact meaning of the address part in an INTCODE 2 instruction depends on the setting of the next word bit, and also, on the operation code. It is convenient to split the cases up into those where the next word bit is a zero and those where it is a one.

1.2.3.1 Use of the address part when the next word bit is a zero

There are three possibilities. If an immediate form

of an instruction is sensible, there are two main operation codes for that instruction. For one of these, the address part forms an unsigned integer which is treated as an immediate operand. The other interprets the address part as a reference to a storage cell at the given offset in the current stack frame. A method thus exists for referencing, in a one-word instruction, variables at low stack effects.

Some instructions exist in INTCODE 2 for which an immediate operand is meaningless (e.g., an accumulator store instruction). In this case there is only one operation code for that instruction, and the address part is always interpreted as a stack offset.

The third case is rather a special one. There are three operation codes in INTCODE 2 known as "operate" instructions. They do not reference any storage cell, and thus the address part is available for specifying a multitude of functions. One code is used for the more conventional "operate" instructions such as "return from routine", but the other two are used for inter-accumulator operations. Four bits are used to specify which operation to perform, and the remainder specify which accumulator is to be used as operand. This allows a total of 32 accumulator-to-accumulator operations, and accounts for the restriction to three accumulators on a 16-bit machine.

1.2.3.2 Use of the address part when the next word bit is a one

Since the next word bit is a one, the address part is not used to specify the operand, this function being performed by the following word in store. It is, thus, available for conveying additional information in the form of flag bits. These are as follows:

(1) A bit signifying that the following word is to be treated as an immediate operand. This provides a method of referencing large constants which is more economical on storage than the more usual method of storing them separately (a saving of one word for each different constant used).

(2) A bit signifying that the value given in the following word is to be treated as a reference to some storage cell at the given offset in the current stack frame. It covers the case when a one-word instruction is insufficient for this purpose.

(3) A bit which states that the operation is to be performed in floating point mode if this is meaningful, e.g. floating point addition instead of integer addition. It may be combined with either of the previously mentioned bits, although floating point 'immediates' are best avoided due to the variations in representation of floating point numbers on various machines.

The remaining bits in the address part are unused at present. If none of the bits are set, the instruction is treated as a normal integer operation with the next word specifying a direct store address.

1.2.3.3 Instruction Groups

From the above, it can be seen that the exact interpretation of an instruction word and address depends on the particular operation code. It is convenient to split the instructions into seven groups with similar legal operand combinations. These groups are as follows:

GROUP 1 - Instructions that can have no immediate variant.

GROUP 2 - Instructions that may have an immediate variant.

GROUP 3 - Stack variants of instructions in GROUP 2 (i.e., the single word form refers to a stack offset rather than an immediate operand).

GROUP 4 - Conventional "operate" instructions, where the function part contains code 00 (OP4).

GROUP 5 - Accumulator "operate" instructions corresponding to instructions in GROUP 1 and some instructions in GROUP 2. The function part contains code 01 (OP5).

GROUP 6 - More accumulator "operate" instructions, corresponding to instructions in GROUP 2 not covered by GROUP 5. The function part contains code 02 (OP6).

GROUP 7 - INTCODE 2 pseudo-operations (codes greater than 63).

1.2.4 Instructions available on the INTCODE 2 machine

It is not proposed to go into great detail here about all the instructions which are available, since many are found on most conventional modern machines. Brief descriptions, will, however, be given, with fuller discussion of the more "interesting" instructions. Appendix A should be consulted for a complete list.

1.2.4.1 Accumulator load instructions

Instructions included: LDA, LMA, LCA, LNA, LAA, LFA.

These are perfectly conventional instructions apart from LAA (Load address of operand) and LFA (Load and convert to and from floating point format).

LAA is necessary since it is not possible to use an ordinary load instruction with an immediate operand, as the operand may be a cell in the current stack frame, in which case its address will not be known until run time. This could be circumvented by always using the two-word form of an instruction and setting the 'stack' and 'immediate' flag bits, but I considered it worthwhile to include a method which allowed single-word forms of such an instruction, since BCPL often requires vector addresses (which are addresses in the current stack frame) to be calculated and stored. In addition, I felt that it was not a good idea to use an ordinary load immediate operation to specify an address load, since although these operations are identical on most machines, there are one or two where this is definitely not the

case. For example, the DEC PDP-10 has only an address load instruction, although in some cases this is equivalent to, and may be treated as, an immediate load.

LFA converts the contents of a cell between the formats for fixed point and floating point numbers. If it is specified as a floating point operation (i.e., its floating point flag bit is set) it loads the contents of the storage cell specified by the effective address into the appropriate accumulator, and converts it to floating point format. Otherwise its effect is to load the word as before, but to assume that it represents a floating point number which is to be converted to fixed point. It is used for implementing the unary operators FIX and FLOAT, which are incorporated in the compiler.

1.2.4.2 Operations on store locations

Instructions included: CLS, STA, AOS, SOS.

Apart from STA (Store accumulator) which is obviously essential, all of these instructions are included purely for optimising purposes, to reduce the number of instructions executed and the amount of store occupied by the program itself. It is a simple matter to modify the INTCODE 2 code generator if the target machine does not possess such instructions, when writing the production code generator for the target machine. This is because the routine which compiles code using these operations is largely table driven, so that most of the work is concerned with just altering the table.

1.2.4.3 Jump instructions

Instructions included: JMP, JOT, JOF.

JMP is an unconditional transfer of control to the location specified by the effective address. Since no accumulator need be specified, the accumulator field is used to say whether or not the return link is to be placed in L to be picked up by a routine entry instruction. (ENT, see section 1.2.4.9). The link is only placed in L if the accumulator field is non-zero.

It could be argued that it would be simpler to store the link in L on execution of every jump, but this complicates the situation if tracing is to be included in INTCODE 2 at some later stage. It might be preferable to redefine this operation so that a zero means "do not store the link" (so that tracing would not occur), a one means "store the link for indirect jumps only" (for routine application and traced GOTOs) and a two means "always store the link" (for a full trace).

JOT and JOF are conditional transfer instructions using a truth value hold in the specified accumulator. They could not, therefore, be traced using the above method, since the accumulator field is already in use. BCPL has a convention that a word with all the bits set to ones means "true" and a word with all the bits set to zeros means "false". The interpretation of any other value is normally left to the implementor, but I felt that it would be reasonable to perform the control transfer only if the appropriate condition is exactly satisfied. I would expect the writer of the inter-

preter to continue this convention when implementing JOT and JOF.

1.2.4.4 Shift instructions

Instructions included: SHL, SHR, ROL, ROR, ASL, ASR.

All of these instructions should be self-explanatory, as they correspond exactly to the BCPL logical shift, logical rotate, and arithmetic shift operators. The number of bits shifted is specified by the address part of the instruction, which may be an immediate operand in the case of a shift whose distance is known at compile time.

1.2.4.5 Logical operations

Instructions included: AND, IOR, NQV.

The logical operations perform the obvious bit-wise logical functions. An equivalence instruction (EQV) was originally included in addition to the not-equivalence (NQV) but was eventually omitted due to lack of spare operation codes, on the grounds that it was probably the least used of those operation codes which could be described with existing INTCODE 2 instructions. (This is certainly true as far as the compiler itself is concerned). An equivalence operation is compiled into not-equivalence followed by logical inversion using LCA (Load complement).

1.2.4.6 Arithmetic operations

Instructions included: ADA, SBA, MUL, DIV, REM.

These should all speak for themselves except possibly

REM, which is a "remainder" operation. It is not strictly necessary, but reduces the amount of code needed for remaindering, which occurs in several places in the compiler. Once again, it was felt that separate operation codes ought to be provided for logically separate operations, and that to compile REM as an auxiliary function of DIV would not be a good idea.

1.2.4.7 Conditional skip operations (memory reference)

Instructions included: SLE, SLS, SNE, SEQ, SGR, SGE.

This group of instructions is a set of skips which test the contents of an accumulator against the contents of the effective address, and skip the next instruction in sequence if the condition holds. It should be noted that the "program counter" in the interpreter may have to be incremented by one, two or even (in the case of a following selector operation) three words in order to produce the right effect.

1.2.4.8 Skip operations (non-memory reference)

Instructions included: SKP, SKC, SOT, SOF.

SOT and SOF are skip versions of JOT and JOF, once again testing a truth value in an accumulator. SKC skips unconditionally and clears the accumulator, allowing very compact code to be compiled for conditional expressions. As it is not very likely that the target machine will possess such an instruction, it can be argued that this detracts from the exemplary nature of the compiler. This is unfortunately one of those cases where there are conflicting requirements. (Ease of rewriting vs. size of an

INTCODE 2 program).

SKP is a simple unconditional skip, about which no more need be said.

1.2.4.9 Routine entry code and routine exit instructions

Instructions included: ENT, RTS.

These two instructions may be thought of as "macro-instructions" since they perform all the operations required at the entry to and exit from a routine, such as preserving the link (on entry) and adjusting the runtime stack pointer to point at the appropriate stack frame. The amount by which the stack pointer needs to be adjusted is given by an extra word placed after the JMP instruction that called the routine.

Two cells are reserved at compile time, at the base of every stack frame for storing link information.

1.2.4.10 Program termination instructions

Instructions included: FIN, HLT.

The FIN instruction corresponds exactly to the BCPL command FINISH. Its function is to terminate the program run and return control to the system after the interpreter has closed input-output, etc.

HLT is designed to be an abortive termination instruction. The operation codes have been arranged in such way that a word consisting of all zeros will appear to be a halt instruction, thus providing a small degree of protection should a program "run amok". It is expected that the interpreter would set all unused locations in the

machine "store" to zero before commencing execution of a program.

1.2.4.11 Error trap instructions

Instructions included: ERR.

If an error is detected during compilation, and the error is of such a type that grossly erroneous code would be generated, the compiler plants this instruction, followed by four parameter words for the error trap service routine which should be included in the interpreter. These parameter words specify the error number, the original line number on which the error occurred, and the addresses of two strings which give the name of the source file being compiled and a suitable error message. The interpreter is expected, on execution of this instruction, to print out appropriate comments on a monitoring channel, then to either terminate execution or return control to the program following the error trap code.

1.2.4.12 Byte manipulation instructions

Instructions included: LDB, DPB.

Strictly speaking, these instructions perform load and store operations. They are used for implementing the BCPL SELECTOR and BYTE operators. Since their use introduces some machine-dependence into a program, they are avoided in the distributed version of the compiler, which only uses full-word selectors and bit fields. They are thus treated separately here. They have an extra word after the one or two making up the actual instruction,

this specifying the word and field to be operated on. The format of the extra word is given in Appendix A.

There is a special feature incorporated in the INTCODE 2 code generator which effectively allows full-word SELECTORS to be defined without prior knowledge of the word size. A bit field width of -1 is interpreted as the width of a word on the machine for which code is being compiled. This idea could be extended, with other negative numbers meaning other pre-defined fields such as the first byte in a word, second byte, etc.

The compiler recognises the case of a selector with a full-word bit field, and compiles conventional load and store operations under these conditions. Space is saved by this, and because the distributed compiler will contain only full-word selector operations, there should be no occurrences of the INTCODE 2 byte selection operations in the distributed code.

Selector operations are useful in the compiler, as they allow the packing of data to be varied by altering SELECTOR definitions, usually held in a header file. This means that, say, the format of a tree mode can be varied to suit a particular target machine, with minimal work by the implementor.

1.2.5 INTCODE 2 pseudo-operations

INTCODE 2 also contains several pseudo-operations which are included to allow information to be passed to the loader/interpreter system. Since they are never actually packed into a word on the target machine, their

codes can be numerically higher than the maximum value for an operation code. (i.e., greater than 63).

The various pseudo-operations available are discussed below. Their format is identical to that for a genuine INTCODE 2 operation, so that the loader need not be concerned with which type of operation it is currently processing.

1.2.5.1 Segment start and end operations

Operations included: SEG, ESG.

The SEG pseudo-operation signifies to the loader that a new segment of code is about to be loaded. The ESG operation signals the end of a segment.

Both of these operations have a six-character field which contains the first six characters of the name of the segment, so that the loader may use this to output suitable monitoring information if desired. The ESG operation additionally terminates any list of GLOBAL initialisation requests introduced by the GLB pseudo-operation.

1.2.5.2 Data specification pseudo-operations

Operations included: VAL, NUL.

The address field of these operations simply specifies a value (which may be a relocatable quantity) for use by the loader in a manner which depends on context. The VAL pseudo-operation generates a data word in store, whereas NUL is merely used to specify some value for use by the loader.

1.2.5.3 String specification operation

Operations included: STR.

Since character codes differ from machine to machine, a character string is represented by the characters themselves, with suitable modifications to allow such items as carriage returns and line feeds to be included. The argument to the STR pseudo-operation specifies the number of characters in the string, subsequent 80-character records of INTCODE 2 containing all the characters of the string, padded with trailing spaces to form a complete 80-character record in all cases. (This is a requirement of Standard FORTRAN). The actual number of 80-character records is determined by the loader from the number of characters in the string as specified by the address part of the STR pseudo-operation.

The procedure for "special" characters is to replace them by an "escape" character (asterisk) followed by a code letter, a similar method to that employed in BCPL itself. Naturally, asterisks also require such treatment in this case. Each character is counted only once when forming the argument to STR at compile time. (i.e. the asterisks are not included in the character count).

1.2.5.4 GLOBAL initialisation request

Operations included: GLB.

This pseudo-operation is included mainly for historical reasons. It is used to specify the initial values of elements in the global vector, originally used in BCPL for inter-segment communication but now super-

seded by EXTERNALS. Its argument gives the number of the highest global referenced within the segment, and it is followed by pairs of NUL operations giving the number of a GLOBAL and the address within the segment to which it is to be initialised. (It is not possible in BCPL to initialise a GLOBAL to a compile-time constant).

1.2.5.5 External definition request

Operations included: EXT.

The EXT pseudo-operation is used to make known to the loader any external names which are defined within the current segment. Its address part contains the value of the symbol, and a six-character field specifies the symbol itself. (External symbols are restricted to six characters in length although they may be equivalenced inside a segment to a name of any length up to the permitted maximum string length for the implementation. This is 127 in the distributed version).

1.2.5.6 Fixup chain specification

Operations included: CHN.

Since the code generator produces code on a single pass over the syntax tree, forward jumps and references to static variables are chained through the second word of the instruction involved. (It should be obvious that all such instructions are two-word ones). The chain is terminated by a chain link of zero, and its head is defined by the CHN pseudo-operation, with the address of the "top" element of the fixup chain in its address part.

CHN is effectively a directive to the loader to fix up all locations on the chain to the address of the next word to be loaded. The compiler always creates chains through the second word of an instruction in order to simplify matters for the loader.

1.2.6 Some points about the implementation of INTCODE 2

The format of the routine call has deliberately been left as open as possible in much the same way as it was for OCODE. This is because it is essential in BCPL for routine calls to be compiled into efficient code, and this is best done by leaving the exact form of a call to the implementor. Naturally, this refers mainly to the rewritten code generator for the target machine, but advantages are also apparent for the interpreted version, since the overhead of instruction fetching and decoding is reduced to one instruction in each case. It also makes for more compact INTCODE 2 programs.

It is easy for an implementor to omit the code dealing with floating point operations, when writing the code generator. This is because both sets of machine operations have the same operation code, but a different variant is used for the floating point set. This facility is required if the target machine cannot represent floating point numbers in a single word, since BCPL only manipulates single words.

1.2.7 The format of INTCODE 2, as seen by the loader/interpreter

A program compiled into INTCODE 2 appears to the loader as a series of fixed length records. The actual length depends on the word size of the target machine, as this sets a limit on the maximum magnitude of numbers that may be input. All fields of the record consist of decimal integers except for the last one, which is a six-position character field. If this field is not used in a particular instruction, it is padded with spaces so that all records appear the same length. This is a requirement of Standard FORTRAN. The character field is used mainly for specifying external names to the loader, but it is also used for passing over the name of the segment at the beginning and end of the load, for possible monitoring purposes.

The first record in each segment (even before the SEG pseudo-operation) is a FORTRAN FORMAT specification, to be used for reading in the rest of the segment in the correct format. This reduces the amount of code that needs to be changed in the loader/interpreter when it is transferred to the target machine. The same record also contains information, in pre-defined columns, on the number of accumulators used, word size, etc.

The only exceptions to the above format are the arguments to the STR pseudo-operation. Strings are always read in as 80-column records, the actual number of records read being determined from the argument to the preceding STR pseudo-operation. The last record is

padded with spaces to 80 columns if necessary.

1.2.8 Character codes

One potential implementation problem arises in connection with the different character codes used on various machines. Since all character comparisons within a BCPL program (and there are, of course, lots of such comparisons in the compiler) are effected by comparison with a cell containing the actual character code, the distributed version of the compiler in INTCODE 2 will assume that all character codes are the same as on the machine it was compiled on. The solution to this problem is fortunately quite simple.

The internal character code is ASCII, which is fairly universal as character codes go. Any input-output done by the interpreter is converted from or to the appropriate code, and when the production compiler is running, the conversion code is easily removed.

1.2.9 Library linkage

Lastly, we consider the communication between a program being interpreted and its input-output library. The compiler requires relatively few library routines, since most of the functions normally performed by BCPL libraries (such as numerical input-output) are included in the compiler itself. The routines required are mainly low-level ones for character manipulation and input-output.

On completion of loading, the loader will have

several undefined external references (either GLOBALS or EXTERNALS). All references to these, as long as they are routine calls, are expected to be replaced by suitable spare operation codes from the standard "operate" set, of which there are about 55 even on a 16-bit machine. References to static variables in the library are best processed by leaving space for all such variables before loading, this being necessary in order that the library routines themselves may "know" where such a variable may be found.

The above method depends on there being no more than 55 library routines. Although this is acceptable if the system is used only as a compiler implementation technique, it means that there may be restrictions when using it as a production system. It can be argued that the restriction disappears on any larger machine (due to the increased number of spare "operate" codes), and in any case, a program requiring so many library routines, would in all probability be so large as to preclude running on an interpretive system unless practically unlimited time is available.

If it is planned to use the interpreter as a production compiler, some increase in speed could be obtained by coding critical routines within the compiler in FORTRAN or even machine code, then linking them to the main program by using more of the undefined "operate" codes. This approach has some similarity to that used in the "LISP" system {13}. A further increase in speed could be obtained by writing the interpreter in machine code.

CHAPTER 2 - THE CODE GENERATOR FOR INTCODE 2

2.1 Design objectives for the INTCODE 2 code generator

There were two main points to bear in mind while the INTCODE 2 code generator was being designed and written. The first was that the code generator should be, as much as possible, an exemplary program which could be used as a model by intending implementors writing a code generator for particular target machines. In other words, the INTCODE 2 code generator had to be as machine-independent as possible.

The second design criterion was more obvious; the code generator should produce reasonably efficient code which was also as compact as possible.

I decided, at a very early stage, to base the INTCODE 2 code generator on the new one being developed for the Essex PDP-10. Perhaps "decided" is not quite the right word; it was practically obligatory to do so, since the parser and library system were tailored to this code generator. It is, in fact, planned to include the INTCODE 2 code generator as part of the Essex BCPL system, in the form of an alternative second pass overlay which may be invoked on request.

From the above, it may be seen that the design of the PDP-10 BCPL code generator was an important factor in the design of the INTCODE 2 code generator. A few words about it are thus in order at this stage.

2.2 The PDP-10 BCPL code generator

This section refers only to the version of the code generator which is now in operation at Essex, not the earlier versions depending on OCODE.

The input to the code generator is in the form of a syntax tree in the store of the machine. The kernel of the program is a recursive tree-walking routine, which calls service routines to perform translation of the various types of tree node. As such, the program is quite well structured, and this is consistent with the intention that it should be an exemplary program.

In fact, I found that it was not entirely exemplary, and although certain minor things were changed at my suggestion (such as a library call which depended on a large word size) some other undesirable aspects remained. One example of this was given in section 1.1 (the register allocation function). Others will now be discussed.

The routine for translating a SWITCHON block in BCPL basically compiles a jump over the code of the block, then translates the block, finishing by compiling tests for the various case conditions and the jumps back into the body of the block. As the block itself is translated, the code generator places the values of the various CASE labels into a temporary table. At the end of the block, it examines this table and evaluates (among other things) the density of occurrence of the CASE values. It then invokes a complicated recursive algorithm to determine the best way of performing the

tests for this set of CASE labels. The result depends on the number of instructions which would be compiled in each case, and may compile a skip chain or an indexed jump table. Such a routine is clearly machine-dependent and should not really appear in a supposedly "exemplary" program which purports to be machine-independent. Since, however, it would be possible to modify it for different machines, it would perhaps be a good idea to include it in the source code of the INTCODE 2 code generator as a comment, although this was not actually done.

Another area of potential problems was that of variable (or dynamic) selector application. In the original form of BCPL, a field selector was constrained to be a compile-time constant. The new compiler allows selectors to be set up at run time, although this may produce inefficiency in the case of, say, full-word selectors better implemented by simple load and store instructions. One advantage is that selectors may be used for indexing bit fields in tree nodes (for instance), thus allowing packed tree nodes to be used at a considerable saving in space. This method was used in the production version of the PDP-10 BCPL compiler.

It is difficult, though not usually completely impossible, to make such an area machine-independent, but is very awkward if you are trying to do it for a 16-bit machine. One possibility would be to restrict the value of the "offset" field to, say, 127, but this is not very useful since most applications of dynamic selectors are usually in cases where a selector con-

taining an actual address is applied to the constant zero. Machine-independence (and simplicity) were achieved by restricting selectors to be compile-time constants, as in previous BCPL compilers.

From the above, it is clear that in addition to the obvious modifications required to make such a code generator produce INTCODE 2, further changes were needed to remove the non-exemplary sections where possible.

More difficulties were incurred by the fact that the PDP-10 code generator and its associated programs were still under development when the INTCODE 2 version was being written and tested. It is difficult to hit a moving target!

2.3 A general description of the INTCODE 2 code generator

The design of the INTCODE 2 code generator proceeded in parallel with the design of the INTCODE 2 abstract machine, since the two are closely linked.

Because the original PDP-10 code generator was still under development, some of the bugs encountered had not previously been discovered in that version. In other words, I "inherited" some bugs with the basic design! This was useful to the writer of the PDP-10 code generator but not to me as it proved very time-consuming to trace errors which were not introduced by myself.

In general, the INTCODE 2 code generator is much like the PDP-10 version. Forward references to STATIC variables and labels were eventually handled by backward

chaining, although another method was considered at one stage. This would have been to use the solution used in the OCODE machine, which is to allocate a unique integer for each new reference, and then to output this integer in a special field in the output code. On reflection, this would probably have been a better method, since it would have made it easier to macro process INTCODE 2 and it would have been possible to incorporate conditional loading facilities into the loader. However, this would have been achieved at the expense of extra complexity in the loader, and (less important) extra storage used during the loading operation. The storage problem would not be acute since if only that much store were available, there would probably have been insufficient for the compiler to operate correctly when loaded. This applies even if the loader and interpreter are split into two separate programs (see section 3.1), since both would probably be of similar size.

One feature which was incorporated into the INTCODE 2 code generator was parameterisation of the code-planting routines so that the "best" code could be compiled for any given target machine. It is possible to change the word size, character size (in bits) and the number of accumulators for a particular realisation of the INTCODE machine on a target computer. As explained in Chapter 1, there is no choice with a 16-bit machine due to the small word size, but there are several possibilities with larger word sizes. For instance, a

24-bit machine could use anything between three and sixty-three accumulators, with a corresponding instruction word address part ranging from ten bits down to six bits in width! It has not been possible to test the code generator for any number of accumulators other than three, because of lack of runtime facilities to communicate with the program. (All options are handled by the parser, which was still being developed). However, various word sizes were tried, to see how much they affected the size of a compiled program in INTCODE 2. Further details of these test runs will be found in Chapter 4.

As stated before, the INTCODE 2 code generator is intended to be an exemplary BCPL code generator, which can be easily modified to produce a code generator for a target machine. Although this has not been completely realised because of the non-exemplary nature of the code generator on which it was based, the most important points requiring attention are covered in Appendix C.

The code generator will not be discussed in lengthy (and boring) detail, but the more interesting aspects will be covered in the following sections. A listing and copy of the program is available on request from the Computing Centre, University of Essex.

2.4 The code generator in detail

The INTCODE 2 code generator is basically, as stated above, a recursive tree-walker. There is a fair amount of local code optimisation but no attempt at global optimisation as it is not usually a good idea for system programs (which BCPL is used for writing) to

do things "behind the programmer's back". In any case, the use of the BCPL indirection (or RV) operator frequently precludes any attempts at optimisation.

There were a number of very interesting and useful routines in the PDP-10 code generator, and most of these are included in the INTCODE 2 version. Sadly, some of them are definitely not machine-independent (and thus not exemplary) and have had to be omitted. An example of such a routine is that used for handling SWITCHON blocks (see section 2.2).

The routines of interest which were used are now discussed, together with some others written specially for the INTCODE 2 code generator.

2.4.1 Routines included from the PDP-10 code generator

2.4.1.1 The tree-printing routine. (Print-tree)

Print-tree was copied unaltered from the PDP-10 code generator. It is a recursive routine which prints out a "picture" of all or part of the syntax tree in a quite readable format. It is invaluable while debugging a code generator, and for elucidating some of the more obscure semantic errors encountered during a regular compilation.

It is, unfortunately, a space-consuming routine, especially on small machines, since it contains a lot of strings which need to be packed into store locations. It could thus be omitted if an implementor were worried

about the space occupied by the entire compiler/interpreter system.

It may be useful to note at this point that a further saving in space could be made by replacing all of the lengthy error messages incorporated in the compiler by null strings. Since an error number is output in any case, they may be omitted at only slight inconvenience to the user, resulting in a fairly substantial saving in store.

2.4.1.2 The tree-weighting function. (Mark)

This function "walks" over the syntax tree prior to code generation determining the complexity of evaluating each subnode. It allows code to be generated that evaluates the "heaviest" arm first, thus minimising the amount of accumulator dumping required during the evaluation of complex expressions. An example of the kind of code produced for expressions involving function calls is given below (in mnemonic form for clarity).

The full meaning of the notation used is explained in Appendix A. It is expected that the tree-weighting function will be included in most production code generators. The INTCODE 2 code generator contains an exact replica of the routine in the PDP-10 code generator.

BCPL code:

```
EXTERNAL £(FOO £)        // FOO IS AN EXTERNAL FUNCTION
LET A, B, C, D = 0, 1, 2 3
A := B + FOO(C * D)
```

INTCODE 2 equivalent:

```
CLS:      0,000002          Store zero in A
LDA:  L   1,000001          Load value 1 into acc. 1
STA:  S   1,000003          Store in B
LDA:  L   1,000002          Load value 2
STA:  S   1,000004          Store in C
LDA:  L   1,000003          Load value 3
STA:  S   1,000005          Store in D
LDA:  S   1,000004          Load C into acc. 1
MUL:  S   1,000005          Multiply by D
STA:  S   1,000008          Store in what will be the next
JMP:  I   1,000000* FOO     stack frame, and jump to FOO
VAL:      0,000006          Amount by which stack adjusted
ADA:  S   1,000003          Add B to result in acc. 1
STA:  S   1,000002          Store in A
```

2.4.1.3 Evaluation of constant expressions. (E, Evalcconst and Evallconst)

This set of routines is used for evaluating constant expressions. E and Evalcconst handle compile-time constant expressions, while Evallconst is responsible for evaluating expressions which assume a constant value at load time, such as addresses of STATIC variables, etc. It is always

useful if constant expressions can be evaluated at compile-time or load-time, but BCPL also makes extensive use of this facility to enable conditional compilation of selected sections of program. The way in which this is done will now be explained.

When the code generator is translating commands of the form:

1) IF E DO C

2) UNLESS E DO C

3) TEST E THEN C1 OR C2

and expressions of the form:

4) E → E1, E2

where C and E represent arbitrary commands and expressions respectively, it sees if the expression E can be evaluated completely at compile time. If so, then the code compiled is only for the part of the conditional which can be executed. Thus, if E were TRUE in the above examples, the code compiled in each case would be:

1) C

2) Nothing

3) C1

4) E1

If E were FALSE, the code compiled would be:

1) Nothing

2) C

3) C2

4) E2

There is one exception. If the code not being compiled contains a label, then it may still be possible for

it to be executed after a GOTO command elsewhere, so it must be compiled anyway.

This technique provides a useful means of incorporating optional debugging code into a BCPL program. Other uses for optional compilation are the inclusion of code to produce statistics about the program itself. An example of the latter may be found in the MEASURING code in the INTCODE 2 code generator. If the MANIFEST constant MEASURING is TRUE, code is compiled to work out the percentage of one-word instructions in the total code produced in a given run. If it is false, no code is compiled, this being the normal state of affairs.

A compile-time constant may be of any complexity so long as any identifiers it contains are declared as MANIFEST constants, and the operators LV and RV are not used. A load-time constant allows the use of LV under certain circumstances which are too involved to define here. It is only allowed in conjunction with an identifier which refers to a STATIC or GLOBAL variable, or some EXTERNAL value. The rules which apply are basically a restricted version of those applied by loader programs when evaluating relocatable expressions. (For an example of such rules, see {14}).

Evalcconst and Evallconst are included essentially unchanged from the PDP-10 code generator. E is included in a modified form which calculates the correct selector value for selector expressions (depending on the word size of the target machine), and which simulates the BYTE operation for compile-time BYTE applications.

2.4.1.4 Evaluation of logical expressions. (Loadlogop)

This routine is included in a form which is much the same as the version in the PDP-10 code generator. It has been modified where necessary to accommodate the INTCODE 2 instruction set, thus removing peculiarities which were present because of the PDP-10 order code.

Its basic function is to load a logical operand into an accumulator, whereas the OCODE version loaded a truth value onto the stack. However, it goes to considerable trouble not to produce redundant code, and the code it does produce is arranged so that no redundant tests are performed.

Some examples of the action of Loadlogop under various circumstances may be useful; these follow.

2.4.1.4.1 Logical true or false operations between two variables

A typical case of this is given by the following fragment of BCPL:

TEST A LOGAND B THEN C1 OR C2

A is first evaluated. Since this is a logical true or false operation, if A is false there is no point in evaluating B since the expression as a whole must be false. Control passes immediately to C2 if A is found to be false.

A similar situation arises in the case of a logical OR operation, except that control is immediately passed to C1 if A is found to be true.

2.4.1.4.2 Logical true or false operations between two expressions

An example of this is:

IF (A - B) LOGAND FOO(X) THEN C

Regardless of the relative complexity of the expressions (the call of FOO is considered the more complicated), the components of the logical expression are evaluated from left to right. This means that the programmer can always predict what will happen in a particular case.

2.4.1.4.3 Logical expression evaluation

The above procedures are still used if an actual value is required from a logical operation, rather than some transfer of control. The logical result (always true or false, all ones or all zeros) is placed in an accumulator for storage or further manipulation. Even if the component parts of the expression are not "true" or "false" as represented by all ones or all zeros, the result is of such a form.

The above examples show that Loadlogop is a practically indispensable routine. It is fairly machine-independent, but really requires the target machine to have conditional skip operations rather than (or in addition to) conditional branch instructions. It would, however, be no great job to modify it for a machine where these operations were not provided.

2.4.1.5 Assignment operations. (Trnupdate, Update, Trnass, Ass)

The assignment routines are worthy of mention since

they incorporate a fair degree of local optimisation. Some of this optimisation was originally linked to a PDP-10, but was easily changed by altering one table and a few odd lines of code.

There are two main areas of optimisation. The first is that of assigning zero, and any other value which may be assigned in a single machine operation. Tests are made for this, and the appropriate machine order generated instead of the two orders (load, then store) usually needed.

The second case is that where an assignment operation effectively reduces to an operation on a single location. (Disregarding the simple assignment case mentioned in the preceding paragraph). Adding one to a variable comes into this category, being compiled into an add-one-to-store instruction.

As previously mentioned, when discussing conversion of this set of routines for the INTCODE 2 machine, it is simple to cater for the particular cases which can be optimised on the target machine, by altering one table.

2.4.2 Routines specially written for the INTCODE 2 code generator

2.4.2.1 The SWITCHON block processing routines. (Casetest, Switch and others)

As was stated earlier, the SWITCHON translation routine incorporated in the PDP-10 code generator was not exemplary, and was, therefore, of no use for the INTCODE 2 code generator. A completely new routine was thus written,

this having one or two advantages over the original one. These are discussed below.

Casetest is the routine that processes CASE labels and places them into a sorted list. A CASE label is allowed (in the new compiler) to have a range of values associated with it, this being a convenient shorthand. An example of this is:

```
CASE 3 ... 5: C (which is exactly equivalent to CASE 3:
                                                CASE 4:
                                                CASE 5: C  )
```

The case "range" is compiled into a single range test (four INTCODE 2 instructions) whereas the separate case labels are compiled into three separate tests and jumps. (Six INTCODE 2 instructions).

This would have been the state of affairs if the SWITCHON processing code from the PDP-10 code generator had been used, suitably modified. The newly written routines go one better than this by trying to merge CASE labels into CASE ranges if they have contiguous ranges of values and they label the same point in the code. They will, for example, convert this piece of BCPL:-

```
    CASE 3 ...  5:
    CASE 7 ... 11:
    CASE 6:       (command)
```

to the equivalent in a single CASE range:-

```
    CASE 3 ... 11:   (command)
```

The former takes ten instructions when compiled, the latter uses only four.

2.4.2.2 The routine for compiling SELECTOR operations. (Loadanddumpbyte)

The routines which performed this function in the PDP-10 code generator were most definitely not exemplary, as they leaned very heavily on the PDP-10 hardware byte manipulation operations. As such, they were useless for the INTCODE 2 code generator.

The PDP-10 code generator also allowed selectors to be defined dynamically. The original practice in BCPL was to allow SELECTORs to only be defined as compile-time constants, and this was the course adopted for INTCODE 2. It is to be hoped, in any case, that SELECTOR operations will not be used with a portable version of the compiler, due to their inherent inefficiency (especially when interpreted) and their dependence on word size.

The new routines, as included in the INTCODE 2 code generator, is intended as a model for implementors to use as a starting point if they decide to implement SELECTORs on their own machines. It ensures that full-word selector operations are treated as the more compact, faster, word size independent, ordinary load and store operations.

The code generator also includes a special method of defining full-word selectors when the word size of the machine is not known. This has been covered in section 1.2.4.12, but to recap, it basically treats a selector field width of -1 as if it were the word size of the machine. (e.g. SELECTOR -1:0:3 is compiled as SELECTOR 16:0:3 for a 16-bit machine, but as SELECTOR 24:0:3 for a 24-bit machine).

2.4.2.3 The code planting routine. (Plant)

This routine is probably the most important one in the INTCODE 2 code generator, and it contributes more than any other to the essentially machine-independent nature of the program as a whole. The reasons for this will now be explained.

It will be recalled from Chapter 1 that the format of an INTCODE 2 instruction is, to put it politely, contorted. This is necessary in order that INTCODE 2 may be used on machines with small word sizes, but it also places a great burden on the code generator if it is to produce the most compact code in all cases.

However, as far as all the other routines in the code generator are concerned, it is of no concern whether the operand of a given instruction is a stack cell, a static cell or an accumulator. Although, for instance, there are in reality three different operation codes for loading an accumulator with varying types of operand, to the code generator at large there only appears to be one.

The routine Plant is called to plant an actual instruction, as defined by its parameters. These are: the operation code, the operand of the instruction (or a pointer to its symbol table entry), a set of bits specifying variants on the basic instruction (such as "immediate" or "indirect"), an accumulator number, and a relocation flag. From this information it generates a standard form of the instruction, the correct address replacing the pointer into the symbol table if necessary. It then checks to see if the instruction can be converted into a single-word form,

either by coding it as an accumulator-to-accumulator instruction or as an ordinary single-word instruction with a stack address or an immediate value as an operand. Such a conversion is always made if at all possible.

The address part of an operate instruction is then set to the correct value for the particular function required, and the instruction is output. An option is included to list the code produced on a monitoring device. This is arranged so that the mnemonics printed are those in Appendix A, with an extra symbol to indicate the type of address relocation, and a six-character field which corresponds to the INTCODE 2 field use for external references, etc.

As far as a prospective implementor is concerned, most of Plant can be replaced with a routine for producing machine code suitable for the target machine, and few other changes of a major nature are required. If the functions of Plant were scattered over the entire code generator, this would not be possible.

2.5 Comments on sundry features of the code generator

2.5.1 Compilation of the GLOBAL feature in BCPL

Code is included in the INTCODE 2 code generator for processing GLOBAL definitions, although it is to be hoped that these will be avoided by the alternative use of the newer EXTERNAL method of linking separately compiled segments. Despite this, it was felt that the code should at least be included so that sufficient information is

available to include GLOBALs in a production code generator, should they be needed to ensure compatibility with earlier BCPL systems.

GLOBALs are treated as a special case of EXTERNALs. More information appears in the loader/interpreter specification in Chapter 3.

2.5.2 EXTERNAL prefixes

The PDP-10 code generator includes a feature to allow the use of six-character names, for library routines, which would be unlikely to occur in normal use. This is done by allowing an optional field to appear in an EXTERNAL declaration, immediately before the opening section bracket. This field has to be either an identifier or a string, and the characters thus specified are concatenated with each name defined within the EXTERNAL declaration, to produce the corresponding external name, after truncation to six characters. The six-character name is a limitation of the loader in use at Essex, and the string is allowed as an alternative to an identifier so that the user can use such characters as "%" within his external names.

Since this feature costs little in space or compilation time, it was included in the INTCODE 2 code generator. It is included only as a basis for such a facility on a new implementation, and is not used in the compiler itself. It would, perhaps, be a good idea to include it as a comment in the compiler source code, or place it under the control of the conditional compilation system, so that it could be easily included or omitted as desired.

2.5.3 Machine code instructions

The PDP-10 code generator incorporated provision for including machine code instructions in a BCPL program, these being introduced by section brackets composed of dollar and square bracket rather than dollar and parenthesis.

Since the parser used with the INTCODE 2 code generator is the same one as that used with the PDP-10 code generator, machine code instructions are accepted by the parser. When the INTCODE 2 code generator attempts to generate code from the appropriate portion of the syntax tree, an error message is output at present, although little modification would be needed to include facilities generating such code. I cannot see much use for this, especially in a code generator intended primarily for bootstrapping purposes.

Implementors should have minimal trouble writing a suitable routine for their own machine. The format of a machine code tree node is easily deduced from the code of the parser.

CHAPTER 3 - THE INTCODE 2 LOADER AND INTERPRETER

3.1 Introduction

As explained earlier, the loader/interpreter system was not actually written, due to lack of time. Since its design is heavily dependent on the form of the INTCODE 2 "machine" and the INTCODE 2 code generator, a possible specification can now be given. It should be emphasised that since it has not actually been written, this is only a specification.

It is suggested that the loader/interpreter should be written in ANSI Standard FORTRAN, to ensure maximum ease of transference from machine to machine. All input-output should be done by the FORTRAN program using the FORTRAN input-output system, as this is relatively machine-independent and requires little or no alteration by implementors for various target machines. The only problem with using FORTRAN as the language for the loader/interpreter is the lack of character operations in FORTRAN, for manipulating strings and identifiers. This problem can be alleviated by mapping all characters into integers on input, and applying the reverse process on output. Such a mapping function has to be written afresh for each different target machine, but since it will only occupy about ten FORTRAN statements (on average) this is no great hardship.

3.1 General specification of the loader/interpreter system

The "store" of the INTCODE 2 machine would be

represented by an array with two equivalenced names. (Or possibly three, see section 3.3.1 below). One of these names would be of type REAL, the other of type INTEGER. This allows correct accessing of each "memory cell" without unwanted type conversion by the FORTRAN system.

It is anticipated that the loader would occupy about 10K, and the compiler itself around 20-25K (about 10-25% larger than the PDP-10 version). If the functions of loading and interpreting were separated, only the interpreter would be in store at the same time as the compiler and its runtime stack. This stack would need to be of the order of 15K in size. If the interpreter occupied about 15K also, then the maximum store requirement at any one time would be about 50-55K. (This may be a bit on the pessimistic side, but it was stated in the Introduction that this is not an implementation method suitable for use on machines with restricted store). To this figure must be added any space needed by the operating system of the machine, if it has one!

It thus seems that it would be a good plan to separate the loader and interpreter into two separate programs. In practice, this could work as follows; the loader would first be run to load the program (usually the compiler) into the "store" array. A series of unformatted WRITE statements (a Standard FORTRAN construct) would then be executed to write the contents of the array to some intermediate storage medium, probably disc or magnetic tape. The loader would then be replaced by the interpreter, which would have an identical "store" array

defined. It would proceed to initialise this by executing a series of unformatted READs from the storage medium.

This technique provides an extra bonus; a "core image" of the program is available on backing store, which eliminates having to re-run the loader for each run of the program. This is especially useful because I found it practically impossible to make the INTCODE 2 code generator produce restartable code, without introducing a great deal of additional complexity into the loader.

3.2 The loader program

The first function of the loader would be to initialise all pointers, counts, etc. and to set all the elements of the "store" to zeros. (Making them HLT instructions). I suggest that the external reference linkage table be kept at the top end of this array, rather than in a separate array, to minimise wastage of space.

The loader would then read in the first line of a segment, this being the FORMAT specification telling it how to read in the subsequent code. The same record would also contain parameters such as the number of accumulators, word size, etc.

The main body of the segment could then be read in, checking for pseudo-operations and performing the appropriate action in each case. (See section 3.2.1 below). In the case of an operation which generated a word in store, it would be necessary to examine the "next word" bit on each instruction line, to ascertain whether the instruction was to occupy one or two words. The actual data storage could

then take place. A problem here is how to code all the bits of the instruction in a single word, since integers would probably be held in twos complement form or something similar. The best way would probably be to code the entire instruction word as a positive integer, disregarding the next word bit, then to negate the result if the next word bit were set. This would also make for easy recognition of a one-word instruction at runtime, since the array element holding it would always be negative.

After the word had been stored, it would be necessary to check the relocation flag for the instruction. It does not matter if the instruction word would be negative, since a little thought will show that a one-word instruction would never need to be relocated. A zero relocation flag value means that no relocation is required, whereas a one means "add the address of the first instruction of this segment to the word now being loaded". Usually the word affected is really the next one in store, as pointed out above, except for the VAL pseudo-operation which generates data in the store.

A relocation flag with a value of two or three has a similar meaning to a value of zero or one respectively, except that the word thus relocated references an external symbol whose six-character name is given in the character field of the instruction. It is necessary, in this case, to search the external symbol table at the top of "store". If the symbol is undefined, it is entered in the table, and noted so that it may be fixed

up later. If it is already defined, its value is added to the address part of the instruction being loaded. This method of adding the value to a location rather than completely replacing the contents of the location means that a GLOBAL may be treated as a special EXTERNAL (with external name ".G"), with its offset in the global vector being placed into the address part of an instruction at compile time.

The end of the load would be signalled by the end of file on the input medium. On some FORTRAN systems, this can be detected by the program, using special system functions or a modified form of READ statement. Standard FORTRAN makes no provision for such things, so the loader would have to recognise a special end-of-file record. This could consist of an extra pseudo-operation, introduced either by the code generator, on request, or by the implementor when creating the INTCODE 2 source file. It would be useful, for debugging purposes, to print some form of storage map giving perhaps the location of the global vector, the address of each EXTERNAL, and the bounds of the storage area occupied by each program segment. This would greatly assist in locating bugs in the production code generator, which are bound to occur.

The loader should also be able to process the various INTCODE 2 pseudo-operations. Probable loader action for each operation is now outlined.

3.2.1 Loader action on reading an INTCODE 2 pseudo-operation

There are eight INTCODE 2 pseudo-operations

which should be recognised by the loader. These are defined in Chapter 1, but the expected action of the loader upon encountering them is described below.

3.2.1.1 The CHN pseudo-operation. (Fixup chain)

The chain pointed to by the address field should be fixed up to the address of the next free location in store. No fresh data is generated in the store, and no table entries made.

3.2.1.2 The EXT pseudo-operation. (External symbol definition

This defines the six-character symbol specified by the character field of the operation. The value of the symbol may be an absolute or relocatable quantity, and is given by the address field of the operation. Relocation is indicated in the usual way by the relocation field. The value may even be the address of some as yet undefined external location.

The loader action would be to fixup any previous references to this symbol, then enter it in the external symbol table together with its value, flagged as "defined". No data would be generated in "store".

3.2.1.3 The VAL pseudo-operation. (Data word)

This operation means that the next free word in store should be initialised to the value specified by the address field. The free location pointer should naturally be updated.

The specified quantity may be an absolute or relocatable value. In the case of an absolute value, it may be negative.

3.2.1.4 The STR pseudo-operation. (String definition)

The next free location in store should be set to the length of the subsequent string, given by the address field of the operation. Following locations would then be filled with the string itself. The string would be defined on the next record (or records in the case of a long string) and should be packed as economically as possible. Since the compiler is "told" the number of bits which a character will occupy, and the word size of the target machine, the correct number of locations will have been allocated to hold the string.

3.2.1.5 The GLB pseudo-operation. (GLOBAL initialisation list)

The address field of this pseudo-operation contains the number of the highest GLOBAL which is referenced in the segment. The loader should note this value, and change it only if subsequent segments reference higher-numbered GLOBALS. At the end of loading, this number is needed so that the correct amount of space may be allocated for the global vector.

The records following the GLB pseudo-operation are grouped in pairs. They are all NUL pseudo-operations, which like GLB generate no data in store, at least directly. The first item in each pair is the offset of

some element in the global vector, and the second is the value to which it is to be initialised. The loader should note these pairs, and use them to set up the correct values in the global vector at the end of the load.

There is at most one occurrence of GLB in each segment. The list of NUL pairs is terminated in all cases by the ESG pseudo-operation. (See section 3.2.1.7).

3.2.1.6 The NUL pseudo-operation. (Null operation)

This is used simply for passing, to the loader, information which is not intended to generate data in store. The information is passed in the address field of the operation.

One example of its use is in a GLB list, see section 3.2.1.5 above.

3.2.1.7 The SEG and ESG pseudo-operations. (Start and end of segment)

SEG and ESG respectively mark the start and end of a segment of INTCODE 2. They contain information, in their character fields, which is the first six characters of the name of the segment. This is useful, as it enables the loader to print out monitoring information while loading, and to give a segment name in diagnostics if an error is detected. (E.g. store full).

The ESG pseudo-operation has the additional function of terminating a list of GLOBAL initialisation pairs

following a GLB pseudo-operation, if these are present.

3.2.2 Library linkage

At the end of loading, the loader should examine its list of undefined external references. Those that correspond to library routines would be "plugged" to the appropriate point in the runtime system (in effect, the interpreter) by the use of spare "operate" instructions (see section 1.2.9). Those refering to library variables are fixed up to the addresses of these variables, which would probably occupy some pre-defined low addressed area of the "store". Any outstanding references would then be flagged as user errors.

This method has the advantage that the user is able to over-ride the library routines with his own routines, since only those external references which are unsatisfied at the end of loading get linked into the library. However, if this convenience can be sacrificed, loading may be speeded by using a method employed in the FASTLINK loader for the IBM 360. In this system, all the library routines are defined at the start of loading, and references to the library can, therefore, be linked in as they are loaded. The method is claimed to considerably speed loading when a conventional library search is involved, although the difference would probably not be as much with the INTCODE 2 loader since a backing store library is not used. In any case, speed is not of paramount importance unless INTCODE 2 is to be used as a production system.

Further details of FASTLINK may be found in {14}, Chapter 5.

3.3 The interpreter

An idea of how the interpreter program should look is now given. This may be used as a starting point for anyone who might be involved in writing it.

The first thing the interpreter should do is to read the "core image" produced by the loader into its "store" array, using unformatted READ statements, (assuming that the loader and interpreter are separate programs).

It should then set up certain runtime parameters such as the number of accumulators, the number of characters per word, etc., so that the library routines will work correctly. Initialisation of the library should be the next job, finishing with setting the program counter to the starting address of the program, also passed as a parameter from the loader. Execution of the interpreted program could then commence.

The various functions of the interpreter program are now discussed in detail.

3.3.1 Setting of runtime parameters in the interpreter

If the interpreter is to be capable of running on any machine, there are certain items of information that will not be available to it until runtime. For instance, the number of accumulators used may differ even among INTCODE 2 programs compiled for the same machine (see section 2.3).

This would be given to the interpreter as a parameter from the loader.

The number of characters per word is required in order that library routines handling strings may function correctly (for instance, Packstring and Unpackstring).

One other point is worth mentioning. It is very difficult to include operations for "bit-picking" in a FORTRAN program, but these are needed to implement such operations as AND, IOR, etc. There are two possible solutions. The first is to unpack the words being operated on into two arrays of type LOGICAL, then perform the operation bit by bit, finishing by packing the result back into the appropriate accumulator (since the size of the arrays would not be known until the word size of the target machine were known, it might be useful to use part of the runtime stack, after equivalencing another name to allow LOGICAL operations). This is clearly a very time-consuming process, but one which is reasonably machine-independent.

The second method is something of a "fiddle". Most FORTRAN systems implement the FORTRAN logical AND operation (used on LOGICAL variables and expressions) as a bitwise AND operation, so that it actually produces the right effect if applied to variables of other types. To get round the problem of type checking, the "store" array once again needs to have a third equivalenced name of type LOGICAL if this method is used.

To save the implementor from having to worry whether or not the second (faster) method will work on his

machine, I suggest that the decision of which method to use is left to the interpreter at runtime. By the use of suitable test cases, it could discover if a bitwise AND were used or not, and set a flag to define the method by which the INTCODE 2 logical AND operation was to be interpreted. Similar methods could be employed to implement inclusive OR operations, logical not-equivalence, and logical inversion. It would not be necessary to implement all four of these operations separately, because the interpreter could invoke the rules of Boolean algebra which define each operation in terms of others. (See any book which discusses Boolean algebra, e.g. {15}).

3.3.2 The interpret loop

This is the most crucial portion of the loader/interpreter system, and it would be worth spending some time coding it efficiently, as although execution time is not a particularly important aspect in a system intended primarily for bootstrapping, a small saving in execution time here will probably result in a disproportionate overall saving, (because the main interpret loop is executed for each and every INTCODE 2 instruction obeyed).

Coding of the sections which simulate the various INTCODE 2 instructions is a fairly trivial operation, so little will be said about it. There are, however, one or two things worth mentioning about the rest of the code.

3.3.2.1 Instruction decoding

It is imagined that instruction decoding would proceed roughly as follows, assuming that the various fields of the instruction had been separated out after determining (perhaps from the sign of the instruction word) whether the instruction was a one-word or two-word version.

A set of logical variables describing the variants on the instruction would be set to FALSE. The number of the operation code would be examined to determine to which group it belonged (see section 1.2.3.3). Control would then be passed to a section of code devoted to evaluating the effective address of that group of instructions, and extracting and checking the variant information, storing it in the appropriate logical variables. It is worth noting that the validity of an instruction need not be checked if the INTCODE 2 code generator is working correctly (as it should be!) but it is probably wiser to incorporate such checking. It would certainly be required if machine code (i.e., INTCODE 2) blocks were included in the source program and the INTCODE 2 code generator had been modified to accept them.

After checking, a piece of code common to all instruction decoding would evaluate the effective address on the basis of the immediate bit and indirection bit settings. Depending on the instruction group, one of several computed GOTO statements would then be performed,

jumping to the appropriate section of the interpreter for simulating that particular instruction. The program counter would, at this stage, have been incremented to point to the succeeding instruction in store, the increment being determined by the instruction size.

All the sections of simulation code would end with a GOTO to the beginning of the interpret loop in readiness for executing the next instruction.

3.3.2.2 Runtime checking

The interpreter should maintain certain checks on a program whilst it is being interpreted. Some of the more important checks are outlined in this section.

3.3.2.2.1 Store address checking

Checks should be made on all references to "store" locations to ensure that they are within the bounds of the storage array. It is not a good idea to leave this to the FORTRAN system since some systems don't do any checking!

I originally intended to make the code portions of INTCODE 2 programs reside in a write-protected area of store. The protection would be provided by checks in the interpreter, and would prevent erroneous programs from accidentally overwriting themselves or the FORTRAN runtime system. This plan had to be abandoned because the variables associated with each program segment are loaded with, and are adjacent to, that segment. Write protection of many small areas would require complexity in the interpreter, whereas an alternative method of storage allocation would greatly increase the complexity of the loader.

3.3.2.2.2 Jumps to invalid addresses

The value of the "program counter" should be monitored in order to trap jumps to invalid addresses (including jumps to location zero). If such a jump occurs, the interpreter should print out the value of L (the link register) and any other useful debugging information. The contents of L are particularly useful if used in conjunction with a storage map (see section 3.2), in determining the source of the erroneous jump.

3.3.2.2.3 Execution of the INTCODE 2 halt instruction, HLT

An INTCODE 2 halt is caused by a jump to some location containing the value zero (see section 1.2.4.10). It indicates that the program is in error, and monitoring information, similar to that given after an invalid jump, should be printed out by the interpreter. The value of L is once more a useful debugging aid.

One possible improvement to INTCODE 2 would be to define the operation code zero as an illegal operation. At present, operation code zero signifies an 'operate' instruction with the particular function indicated by the value of the address part. If the spare operation code octal 77 were used for this, then defining an illegal code zero would mean that jumps to locations containing low numbers would be trapped as illegal. Most variables tend to hold low values compared to the maximum possible even in a 16-bit word, so this could be quite a useful enhancement.

3.3.3 Character input-output

It should be remembered that characters are expected to be in ASCII code when fed to the compiler. The input-output system in the interpreter should thus perform conversion between external and internal character codes.

3.3.4 The runtime library

The interpreter should contain the runtime library. This provides input-output facilities, and also many other things such as string packing and unpacking routines. It is intended that all the machine-dependent routines required by the compiler are included in this library, hopefully greatly reducing the amount of recoding needed to get the system going on a new machine.

Because the library incorporates such machine-dependent sections, it would need information about the machine it was running on, and the version of the INTCODE 2 (number of accumulators, etc.), that it was loaded with. Such information would already be known to the interpreter, and would include such items as word size, character size, and so on.

CHAPTER 4 - SUMMARY AND CONCLUSIONS

I hope that the preceding Chapters show that the INTCODE 2 approach to portability is probably a reasonable one for implementing the BCPL compiler on a new machine. The main problems likely to be encountered are now discussed.

4.1 Speed of execution of a program in INTCODE 2

The speed of the system is not of major importance, which is probably a good thing since it is likely to be quite slow! It may even be unusably slow, but one can only speculate at present since the loader/interpreter has not been written, and thus no figures are available. However, execution time will be considerably greater than that of a program using INTCODE 1 and its implementation mechanism. The overheads of actually compiling into INTCODE 2 as opposed to the order code of a conventional machine (such as the PDP-10) appear to be very small, although more time is naturally spent in the code-planting routine, due to the complexity of representation of an INTCODE 2 instruction. The increase in execution time is an effect of the trade-off between practical considerations and the necessity of an exemplary code generator. Perhaps this trade-off is unwarranted and a separate example code generator should have been written - it is pointless to speculate further until the loader/interpreter is written and the method is used.

4.2 Size of the interpreter and an INTCODE 2 BCPL compiler

Another big problem is likely to be the size of the

interpretive version of the compiler. This is another trade-off, for much the same reasons as above. A "macro-like" order code for the INTCODE 2 machine would have resulted at a considerable saving in storage, but a code generator for such an order code would have borne little resemblance to a code generator for a "real" machine.

Rough estimates of the size of the system were given in section 3.1, and these assume a size overhead of about 10-25% compared to the space occupied by the PDP-10 compiler on a PDP-10. The figures depend on the word size of the target machine, and to give some idea of what may be expected, sample values will now be given. The figures quoted are for three different word sizes, also for the PDP-10 code generator producing code for the PDP-10, to provide a basis for comparison. The program being compiled in all cases was the entire machine-independent section of the INTCODE 2 BCPL compiler. The machine-dependent parts would normally be incorporated as part of the interpreter, and need not concern us here.

Relative sizes of INTCODE 2 code generator for differing word sizes

Type of compiled code	No. of words occupied	% of one-word instructions	% size overhead compared to machine code for a PDP-10
Machine code for PDP-10	20370	-	-
INTCODE 2 for a machine with a 36-bit word	22826	60.0%	12.1%
INTCODE 2 for a machine with a 24-bit word	23632	60.0%	16.0%
INTCODE 2 for a machine with a 16-bit word	25323	55.3%	24.3%

All the figures quoted above for INTCODE 2 are for a machine with three accumulators, whereas the figure for the PDP-10 is for code using up to 10 accumulators. I do not think this makes any significant difference since there are no extremely complicated pieces of code in the compiler which require so many accumulators.

I should point out that the PDP-10 has a 36-bit word, so that the overhead figure of 12.1% for "36-bit" INTCODE 2 can be considered as a measure of the relative efficiency (in terms of store used) of expressing the compiler algorithm in INTCODE 2 as opposed to the equivalent machine code. This figure seems very favourable, and is better than that originally hoped for.

The figures for the percentage of one-word instructions were obtained by using code written into the INTCODE 2 code generator under the control of the compile-time constant MEASURING. A production compiler would normally set this to FALSE to inhibit the statistics.

It is interesting to note that a law of diminishing returns takes effect somewhere between the 16-bit and 24-bit word sizes, and no further improvement is obtained when a 36-bit word is used. If more time had been available it would have been interesting to find out the exact point at which this occurs. I suspect that it is around the 18-bit or 19-bit stage, since this word size would allow nearly all the constants used in the compiler to be loaded into an accumulator in a one-word instruction, and nearly all the references to variables on the stack to be expressed in a single word as well. Jump instructions always occupy two words, of course, as do references to STATICs and EXTERNALs.

The decrease in the size of the 36-bit version over the 24-bit version is entirely due to the greater number of characters that may be packed into the larger word. A saving in space could be obtained by removing the bulk of

the error message strings, and just printing out an error number instead.

Some parts of the compiler actually occupy less space when expressed in INTCODE 2 than they do when compiled into PDP-10 machine code. This suggests that INTCODE 2 is a good language for describing at least some of the constructs encountered in BCPL programs, i.e. it is a good descriptive language for BCPL. Of particular importance here are such instructions as ENT, the routine entry instructions, which performs several operations such as storing a return link and adjusting the stack. It does in one word what the PDP-10 version requires several words to do. However, the figures quoted above are for PDP-10 code which includes tracing facilities, and this naturally increases the amount of code compiled for routine entries.

There is another factor which increases the amount of storage required by the INTCODE 2 compiling system. This is that the INTCODE 2 system is inefficient in its use of work areas. All tree nodes are held in an unpacked form (i.e. each word contains only one piece of information). Thus, the amount of store required for compiling reasonably sized BCPL programs is greater than that required by, say, the PDP-10 compiler. There is not much that can be done about this, especially if the byte selection operations are to be avoided.

4.3 Writing the new code generator

A major consideration in the evaluation of the usefulness of the INTCODE 2 implementation mechanism is whether

or not the compiler, as it stands, is exemplary. I feel that not much can be done for the parser section, but the code generator is a different story. Although as much machine-dependence (dependence on the INTCODE 2 abstract machine) as possible has been removed, there are inevitably some areas which will still need rewriting for particular target machines. I hoped to be able to supply a set of instructions of the form: "change line --- to agree with the ----- on your machine" ... etc., but this isn't really practical since there are points which will be important on some machines but not on others. I believe that the need for such changes has been kept to a reasonable minimum, and that they will not prove difficult to implement. The full picture will only emerge when (or if) someone uses the system to produce a new BCPL implementation.

As yet, there is not much advice that can be given to prospective implementors. This is because the system is untried, and the area of most immediate concern to an implementor (the loader/interpreter) has yet to be written, and is thus a relatively unknown quantity. Given a good FORTRAN system on the target machine, I do not envisage any major problems as long as sufficient store is available. The only changes required to the loader/interpreter would be constants describing the target machine, such as word size, etc., and a routine for conversion of character codes on input and output. The word size could be passed from the INTCODE 2 on the first line of such segment, as suggested in section 3.2, and this would reduce the amount of work required of the implementor, though only to a small degree.

Once the above modifications have been carried out, there is no reason why a perfectly good BCPL compiler, with all the facilities available on the PDP-10 implementation, should not be produced.

4.4 Implementation of INTCODE 2 by macro processor

At this point I should like to discuss something first mentioned in section 6 of the Introduction. This is the implementation of INTCODE 2 by means of a macro processor. Brown (see {11}) calls this method a DLIMP - a Descriptive Language Implemented by Macro Processor.

Implementation by a DLIMP requires that the software to be implemented is expressed in a suitable language, usually tailored to the piece of software itself. Hence the term "descriptive language". We have already seen that INTCODE 2 is a reasonably good descriptive language for BCPL programs, so the method is certainly applicable to INTCODE 2, although it would be necessary to make minor changes in format and alter the method of compiling forward references. The question remains as to whether the method has any significant advantages over the interpretive approach.

To my mind, the advantages of a DLIMP are, in this case, as follows. The implementor is given more control over the generated code, since in this case machine code for the target machine would be generated directly from INTCODE 2. The resulting program would be more economical on storage (although the problem of unpacked work areas would remain). This is because few (if any) two-word

instructions would be needed, and the implementor could make use of any special facilities accorded by the order code of the target machine. The interpreter would not be needed, although it would be necessary to provide a run-time library, including some form of input-output system. This could still be written in FORTRAN, and linked to the machine-code main program (most FORTRAN systems allow machine code subprograms).

On the debit side, the task of implementation is increased somewhat. The implementor must write macro definitions to map INTCODE 2 into some form acceptable to the target machine (usually machine code), and these must be debugged. Tests on such macros can be difficult to write, especially if the macros attempt any more than the simplest optimisation. The answer here is obviously to avoid trying to optimise the code, since this is only a bootstrap method. There is always time to be clever when the compiler is actually up and running!

Another snag is that, if a macro processor or a powerful macro assembler is not available on the target machine, one must be provided. A suitable choice would be STAGE2 {10} or possibly ML/I {4}, although the former is preferable because it is faster to implement and uses a pre-written FORTRAN input-output package which could conceivably be modified for use as the basis of a run-time library {18}. I have used this package, and found it rather awkward to use, but I suspect that this may have been due to inadequate documentation rather than to any inherent flaw in the package itself.

If a DLIMP were used (the language being INTCODE 2 of course), it would probably be advantageous to set INTCODE 2 at a slightly higher level. This would allow slightly more optimisation, and in general, reduce the number of redundant operations. Among others, Brown {16} and Poole {17} give fuller details of the technique. Both have used the method with great success. They also suggest the idea of a hierarchy of descriptive languages, so that the implementor may choose the one most suited for implementation on the target machine. One practical illustration of the merits of a high level descriptive language is given by the recent implementation of ML/I on a Burroughs B6700, which does not have an "assembly language" as such. Programs on this machine are usually written in a much extended dialect of ALGOL, and any attempt to implement ML/I using its low-level descriptive language (LOWL) would have resulted in an impossibly large and slow program. It was necessary to use the high-level descriptive language for ML/I, called L. This provided a suitable means of implementation, although macros for high level mappings are in general harder to write and debug. The resulting version of ML/I was thus written (eventually) in Burroughs "Extended ALGOL". A similar problem may arise with the ICL 2900 series of machines, which are again devoid of a machine code, at least as far as users are concerned.

It can be seen that it would be very difficult to implement the BCPL compiler on such machines using a DLIMP. The INTCODE 2 method is really no easier when one

comes to write the code generator. The answer is perhaps that you should not try - Burroughs ALGOL provides an adequate system programming tool without recourse to BCPL!

4.5 A brief evaluation of INTCODE 2

Lastly, I shall consider the merits (or otherwise) of the INTCODE 2 approach to portability.

I think that it was probably a mistake to limit the INTCODE 2 machine to a 16-bit word, since the maximum addressable store on a 16-bit machine is only 64K, and this is not much more than the minimum necessary to run the loader/interpreter and the compiler. A 24-bit lower limit would allow a much more rational instruction format, and multi-word instructions could probably be eliminated altogether. It remains to be seen whether the method is viable on a 16-bit machine.

The use of a pre-written FORTRAN input-output package is, I think, a good idea, since implementors of many pieces of portable software tend to spend more time on this aspect than on any other part of the work. If this time can be reduced at all, so much the better.

Overall, then, I think that the implementation method could be improved by modifying the code generator to produce a slightly higher level language, using a macro processor for the initial implementation, and leaving the machine dependent operations to a FORTRAN package supplied as part of the implementation "kit". The other items in this "kit" would include an INTCODE 2 copy of the entire compiler, a source listing of parser and code generator,

and perhaps a copy of STAGE2 for target machines without a working macro processor or macro assembler. Such a method would be applicable to a wider range of machines, because less store would be needed, and there would probably be a saving (in the long run) in the amount of machine time needed for the transfer, even allowing for the voracious "CPU time appetites" of macro processors.

INTCODE 2 then, seems to be a method with possibilities. The best solution would probably be to offer the implementor a choice: INTCODE 2 plus macro processor for more economical use of storage and (possibly) better eventual code. Only time will tell which is best.

REFERENCES

1. Richards, M. (1973). "The BCPL Programming Manual". Computer Laboratory, University of Cambridge.

2. Richards, M. (1971). "The Portability of the BCPL Compiler". Software Practice and Experiences, Vol. 1, pp. 135-146.

3. Strachey, C. (1965). "A general purpose macro-generator". Computer Journal, Vol. 8, pp. 225-241.

4. Brown, P.J. (1967). "The ML/I macro-processor". C.A.C.M., Vol. 10, No. 10, pp. 618-623.

5. Brown, P.J. (1971). "A survey of macro-processors". Annual Review in Automatic Programming, Vol. 6, Pergamon Press, Oxford. pp. 37-88.

6. Richards, M. (1972). INTCODE - An Interpretive Machine Code for BCPL". Computer Laboratory, University of Cambridge.

7. Richards, M. (1973). "Bootstrapping the BCPL Compiler Using INTCODE". Computer Laboratory, University of Cambridge.

8. Steel, T.B. (1963). "UNCOL - The Myth and the Fact". Annual Review in Automatic Programming, Vol. 2, Pergamon Press, Oxford. pp. 325-344.

9. National Computing Centre. "Standard FORTRAN Programming Manual". SBN 85012 063 2.

10. Waite, W.M. (1970). "The Mobile Programming System: STAGE2". C.A.C.M., Vol. 13, No. 7, pp. 415-421.

11. Brown, P.J. (1969). "Using a macro-processor to aid software implementation". Computer Journal, Vol. 12, pp. 327-331.

12. Brown, P.J. "Macro Processors". Wiley-Interscience, ISBN 0 471 11005 1.

13. McCarthy, J. et al "LISP 1.5 Programmer's Manual". M.I.T. Press, ISBN 0 262 13011 4.

14. Barron, D.W. "Assemblers and Loaders". MacDonald-Elsevier, SBN 356 02682 5.

15. Aleksander, I. "Introduction to logic circuit theory". Harrap, SBN 245 50375 7.

16. Brown, P.J. (1972). "Levels of language for portable software". C.A.C.M. Vol. 13, No. 12, pp. 1059-1062.

17. Poole, P.C. (1971). "Hierarchical Abstract Machines". Symposium on Software Engineering held at Culham Laboratory, H.M.S.O., London, pp. 1-9.

18. Waite, W.M. (1970). "Building a mobile programming system". Computer Journal, Vol. 13, pp. 28-31.

APPENDIX A - Description of the INTCODE 2 machine

A.1 The INTCODE 2 instruction set

This section lists, in symbolic form, the actual operations performed by each INTCODE 2 instruction. The terminology and notation used are as follows:

AC The accumulator number as specified by the accumulator part.

E The result of the effective address calculation. E is either the address part of a one-word instruction, or a full word in the case of a two-word instruction.

PC The "program counter" of the INTCODE 2 machine.

K The number of words occupied by the next instruction.

(X) The word contained in accumulator X or location X.

A→B The quantity A replaces the quantity B. eg. :

(AC) + (E) → (AC)

means the word in accumulator AC plus the word in location E replaces the word in AC.

and The Boolean operator "AND".

ior The Boolean operator "inclusive OR".

xor The Boolean operator "exclusive OR".

not The Boolean operator "complement" (logical negation)

+,- The arithmetic operators for addition and subtraction.

*,/ The arithmetic operators for multiplication and division.

rem The arithmetic operator for remaindering.

abs The arithmetic unary operator for absolute value (magnitude).

For convenience, the instructions may be considered in seven groups, including pseudo-operations. These groups are as follows:

Group 1 - Instructions that can have no immediate variant.

Group 2 - Instructions that may have an immediate variant.

Group 3 - Stack variants of instructions in Group 2.

Group 4 - Conventional "operate" instructions (using OP4 - see below).

Group 5 - Accumulator "operate" instructions (using OP5 - see below).

Group 6 - More accumulator "operate" instructions (using OP6 - see below).

Group 7 - INTCODE 2 pseudo-operations (codes greater than 63).

GROUP 1

Code (octal)	Mnemonic	Description
00	OP4	Group 4 "operate" instruction.
01	OP5	Group 5 "operate" instruction.
02	OP6	Group 6 "operate" instruction.
03	LAA	E → (AC)
04	AOS	(E) + 1 → (E)
05	SOS	(E) - 1 → (E)
06	JMP	if (AC) = 0 then (PC) → (L); E → (PC)
07	JOT	if (AC) = true then E → (PC)
10	JOF	if (AC) = false then E → (PC)
11	LDB	Byte load - see Section 1.2.4.12 and A.2 below.
12	DPB	Deposit byte - see Section 1.2.4.12 and A.2 below.

Code (octal)	Mnemonic	Description
13	STA	(AC) → (E)
14	CLS	0 → (E)
GROUP 2		
15	SHL	Shift (AC) left E places.
16	SHR	Shift (AC) right E places.
17	ASL	Arithmetic shift (AC) left E places.
20	ASR	Arithmetic shift (AC) right E places.
21	ROL	Rotate (AC) left E places.
22	ROR	Rotate (AC) right E places.
23	AND	(AC) and (E) → (AC)
24	IOR	(AC) ior (E) → (AC)
25	NQV	(AC) xor (E) → (AC)
26	LDA	(E) → (AC)
27	LFA	if floating point variant, FLOAT (E) → (AC) else FIX (E) → (AC)
30	LMA	abs (E) → (AC)
31	LNA	-(E) → (AC)
32	LCA	not (E) → (AC)
33	ADA	(AC) + (E) → (AC)
34	SBA	(AC) - (E) → (AC)
35	MUL	(AC) * (E) → (AC)
36	DIV	(AC) / (E) → (AC)
37	REM	(AC) rem (E) → (AC)
40	SLE	if (AC) le (E) then (PC) + K → (PC)
41	SLS	if (AC) ls (E) then (PC) + K → (PC)

Code (octal)	Mnemonic	Description
42	SEQ	if (AC) eq (E) then (PC) + K → (PC)
43	SNE	if (AC) ne (E) then (PC) + K → (PC)
44	SGE	if (AC) ge (E) then (PC) + K → (PC)
45	SGR	if (AC) gr (E) then (PC) + K → (PC)

GROUP 3

All the instructions in Group 3 are variants of the instructions in Group 2. The difference is that the one-word version of a Group 2 instruction has an implied immediate operand, whereas the corresponding Group 3 one-word instruction refers to a location offset in the current stack frame. Operation codes are from 46 to 76, and refer to variants of codes 15-45 respectively.

GROUP 4

All operation codes in Group 4 have OP4 (code 00) in the function part of the instruction. The address part (they are always one-word instructions) contains an integer which specifies the exact operation.

Code in address part	Mnemonic	Description
00	HLT	Halt the program run.
01	SKC	0 → (AC); (PC) + K → (PC)
02	SKP	(PC) + K → (PC)
03	SOF	if (AC) = false then (PC) + K → (PC)
04	SOT	if (AC) = true then (PC) + K → (PC)

Code in address part	Mnemonic	Description
05	ERR	Source error trap - see Section 1.2.4.11.
06	RTS	Perform exit from routine.
07	ENT	Perform routine entry sequence.
10	FIN	Terminate program run (FINISH).

In the INTCODE 2 code generator, these instructions are represented internally by codes 200-210 (octal).

GROUP 5

The Group 5 instructions are "operate" instructions similar to those in Group 4, except that they are variants of instructions in Groups 1 and 2 which may reference an accumulator as operand. They exist to provide a method of referencing an accumulator in a one-word instruction. The code used in the address part for each instruction in Groups 1 and 2 is as follows, some instructions being omitted if an accumulator variant is not sensible:

Code in address part	Mnemonic
00	JMP
01	JOT
02	JOF
03	LDB
04	DPB
05	STA
06	CLS

Code in address part	Mnemonic
07	SHL
10	SHR
11	ASL
12	ASR
13	ROL
14	ROR
15	AND
16	IOR
17	NQV

In the INTCODE 2 code generator, these instructions are represented internally by codes 300-317 (octal).

GROUP 6

The Group 6 instructions provide accumulator variants of instructions from Group 2 not included in Group 5. Their codes are as follows:

00	LDA
01	LFA
02	LMA
03	LNA
04	LCA
05	ADA
06	SBA
07	MUL
10	DIV
11	REM

12	SLE
13	SLS
14	SEQ
15	SNE
16	SGE
17	SGR

In the INTCODE 2 code generator, these instructions are represented internally by codes 400-417 (octal).

GROUP 7

Group 7 contains the INTCODE 2 pseudo-operations. Their internal codes (used in the INTCODE 2 code generator) and a brief description of their action are given below. For fuller details Section 1.2.5 should be consulted.

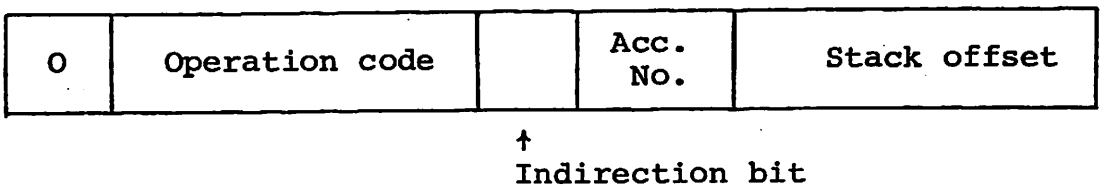
Internal code	Mnemonic	Description
500	DAT	Generate data-purely internal.
501	GLB	Global variable definition.
502	SEG	Start of segment.
503	ESG	End of segment.
504	VAL	Generate initialised data word.
505	CHN	Fixup chain-loader directive.
506	NUL	Null operation - data to loader.
507	STR	String definition.
510	EXT	External definition.

A.2 Format of an INTCODE 2 instruction word

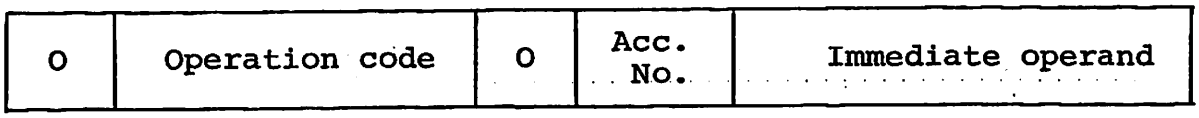
See also, Section 1.2.2.

A.2.1 One-word instructions

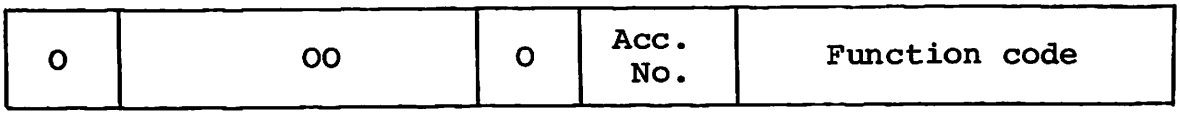
A.2.1.1 Group 1 or Group 3



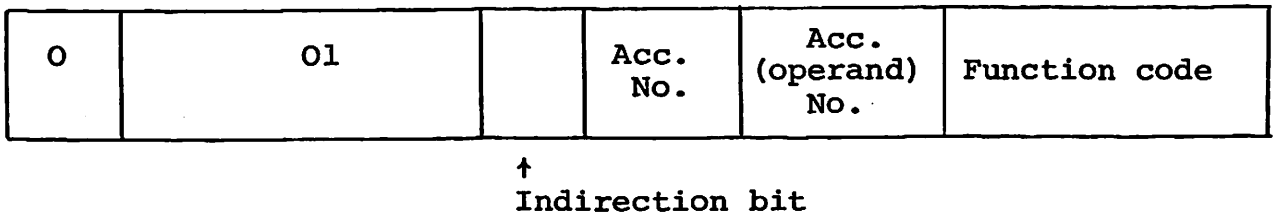
A.2.1.2 Group 2



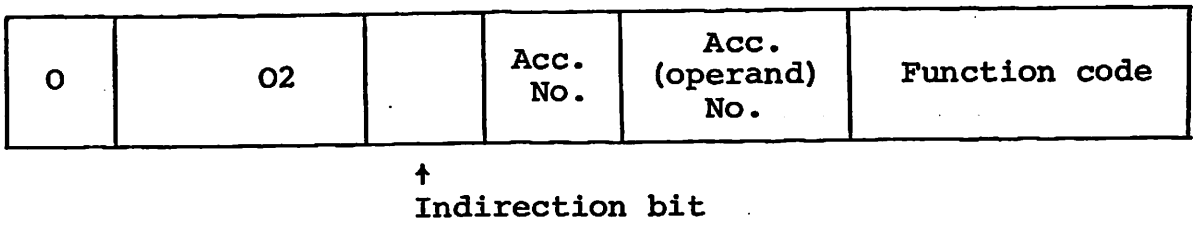
A.2.1.3 Group 4



A.2.1.4 Group 5

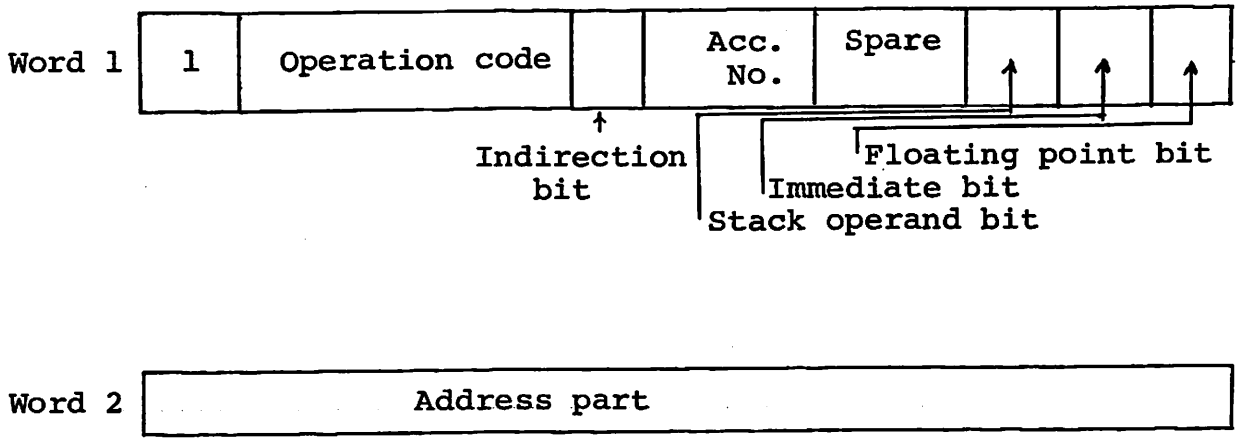


A.2.1.5 Group 6



A.2.2 Two-word instructions

All relevant groups (Groups 1-3)



A.3 Format of an INTCODE 2 byte selector word

Although byte selectors are best avoided as far as possible due to their inherent machine-dependence (on word size), they are sometime useful. The format of a byte selector as used by the INTCODE 2 system is now given. In the absence of hardware selector operations, this format could be used by intending implementors of byte selectors.

The byte selector word appears after the instruction word which references it, or after the second word in the case of a two-word instruction.

Field width	Field position	Word offset

The field width element should be large enough to specify the word size as an integer, since the field width may be as much as a full word. The position element should be one less than this, since it states the number of bits to the right of the specified field, and for a meaningful selector this cannot exceed the word size minus one. The word offset field uses whatever space is left in the word.

For example, on a 16-bit machine:

Field width: possible values are 0-16: element size 5 bits

Field position: possible values are 0-15: element size 4 bits

Word offset: space left is 7 bits, so maximum offset is 127.

A.4 Format of the human-readable INTCODE 2

INTCODE 2 is essentially a string of numbers for input to the loader/interpreter, and as such it is not very readable. The code generator thus includes an option to print out INTCODE 2 in mnemonic form, using the mnemonics defined in A.1 above. The output shows the relocatable address of each instruction or data word, relocation markers, external references and pseudo-operations. A typical instruction serves to illustrate the format:

```
000023' LDA:I    1,000453* FOOBAZ
```

The first column is the relocatable address of the first

word of the instruction. This is followed by the mnemonic for the instruction itself (load accumulator) with accompanying variants separated from the mnemonic by a colon to improve readability. Two numbers follow, separated by a comma. The first is the accumulator number, the second the contents of the address field (the address part of a one-word instruction, or the second word of a two-word instruction). A relocation flag occupies the next column. This flag may be a space (no relocation), a single quote (relocation by the address of the first location occupied by the segment) or an asterisk (external reference). The last field is only present for an external reference, and is a copy of the six-character field in the "real" INTCODE 2 version of the code.

The variants all appear in a particular column allocated to each one, and consist of a single letter. The key to these letters is:

I - indirection bit is set.

L - immediate (or Literal) instruction.

S - operand is in the current stack frame.

F - instruction is the floating point version.

A - instruction is an accumulator "operate".

Obviously, not all combinations are meaningful.

A.5 Format of the machine-readable INTCODE 2

It may be useful to briefly explain the format of INTCODE 2, as seen by the loader/interpreter, in a little more detail.

An INTCODE 2 instruction is read as a single FORTRAN record of fixed length. (See Section 1.2.7). This record contains several decimal numbers describing the instruction, in fields whose width depends on machine-dependent parameters. The function of these fields (left to right) is as follows:

Field	Description
1	Store address of instruction (for checking only).
2	Value of next word bit (0 or 1).
3	Operation code.
4	Indirection bit (0 or 1).
5	Immediate operand indicator. (0 for non-immediate, 1 otherwise).
6	Stack variant bit. (0 if not stack variant, 1 otherwise).
7	Floating point instruction variant indicator. (0 for normal instruction, 1 for floating point).
8	Accumulator number.
9	Operand value.
10	Relocation indicator. (0, 1, 2 or 3).
11	Six-character field for external references, etc.

An example of machine-readable INTCODE 2 may be found in Appendix D.

APPENDIX B - USING THE INTCODE 2 CODE GENERATOR

This Appendix is intended to give information on running the INTCODE 2 code generator. Unfortunately, the information available is somewhat limited, since it is concerned almost exclusively with the various options which can be used, and these depend on the environment in which the program is running.

The method of passing "switches" to the PDP-10 INTCODE 2 code generator was still being decided on when the project finished, and the method of specifying options to the interpreted version would depend on the interpreter library, which has not been written. I shall thus describe the action of the switch variables named within the code generator. These names can easily be linked to the appropriate external option specifications.

Switch name	Action (if TRUE)
BATCH	Batch working is assumed, for instance all error messages are sent to the listing file.
CODEING	INTCODE 2 is to be produced. A FALSE switch value gives a semantic check but no code.
LISTING	A source listing is produced.
NOWARNINGS	Error class messages only are produced.
CODELIST	The INTCODE 2 produced is listed in symbolic form (see Appendix A) on the monitoring file, or listing file if BATCH.
NAMELIST	The names used in the compiled program are listed on the monitoring file, or listing file if BATCH.

Switch name	Action (if TRUE)
CHECK	Only a syntax check is performed.
TREELIST	A picture of the parse tree is sent to the listing file.

There are also three "switches" which take a numerical value. These are as follows:

Switch name	Description
PASSNO	This switch is present in the PDP-10 code generator to select various second pass overlays, one of which is the INTCODE 2 generator. Currently, the value required for INTCODE 2 is 12 (decimal).
OPTLEVEL	This switch selects whether any tracing or stack checking is to be incorporated into the compiled code. It currently has no effect on the running of the INTCODE 2 code generator since tracing is not supported.
ERRORMAX	The value of this switch is the maximum number of errors that may occur before a compilation is abandoned. The default value on the PDP-10 version depends on the setting of BATCH.

The PDP-10 INTCODE 2 code generator is run by calling the BCPL compiler in the normal way (by ".R BCPL"), then requesting the INTCODE 2 code generator as the second pass overlay. This is currently done by typing the switch "/P:12" to the compiler.

At the end of compilation, the INTCODE 2 code generator prints the number of words occupied by the compiled code, including space occupied by STATIC variables, etc. The value is given in both octal and decimal.

APPENDIX C - NOTES ON WRITING A CODE GENERATOR FOR THE TARGET MACHINE

This Appendix is meant to give a little assistance to implementors to whose lot it falls to write a code generator. It attempts to point out the parts of the INTCODE 2 code generator which will need modification on most machines. It does not set out to be an exhaustive list, so "caveat implementor"!

The various files making up the code generator are discussed separately; each needs its own particular modifications.

File TRANS0

Change some of the initialisation code; remove the error messages relating to INTCODE 2.

File TRANS1

No alterations.

File TRANS2

No alterations.

File TRANS3

Routine DECLSTAT - allocation of space for STATIC variables.

File TRANS4

Write a version of TRNMCODE suitable for the target machine order code (if required).

File TRANS5

No alterations.

File TRANS6

No alterations.

File TRANS7

Remove references to floating point operations if they are not to be supported; alter the code in E for selector and byte operations; alter the code for load time constants if necessary; remove routines SELECT and BYTEAP.

File TRANS8

No alterations unless jump tables are to be used; the code will work as it is for an initial implementation.

File TRANS9

Change TRNJUMP to allow for method of routine call on target machine; alter the code of LOADLOCAL if there is no "clear store" instruction.

File TRANSA

Alter the operation code tables.

File TRANSB

Alter the code of LOADLOGOP if the target machine cannot do a "jump on true" and a "jump on false"; alter operation code table for conditional operations; supply an alternative to the FALSESKIP routine.

File TRANSC

Rewrite OPONLEAF to cater for the order code of the target machine.

File TRANSD

No alterations.

File TRANSE

Rewrite to LOADANDDUMPBYTE for the hardware byte selection operations on the target machine (if any).

File TRANSF

Alter the code of UPDATE if target machine does not have a "clear store" operation; alter the table in ASS to allow for the available operation codes, then carefully check ASS for necessary changes.

File TRANSG

Complete rewrite to except for SETGLOBAL and possibly SETEXTERNAL.

File TRANSH

Rewrite PLANT, as this is completely dependent on INTCODE 2; check all the other routines for necessary changes.

File TRANSI

Rewrite to suit the target machine order code.

The above information gives the implementor a start; the best way to continue is to read the code itself.

APPENDIX D - EXAMPLES OF INTCODE 2 PROGRAMS

This Appendix shows typical code for fragments of BCPL programs, and the output from the INTCODE 2 code generator in both machine-readable and human-readable form.

Example 1 shows a simple BCPL function, the INTCODE 2 equivalent in human-readable form, and the INTCODE 2 equivalent in the form used for input to the loader/interpreter.

Example 2 is a longer piece of code. Amongst other things it demonstrates the treatment of EXTERNALs in INTCODE 2. It is a complete segment, and as such includes the SEG and ESG statements. It may be noted that these statements assume the name of the file holding the segment to be SEGNAM. A machine-readable version has not been included.

```
LET FRED(A, B) = VALOF
$( LET X, Y = NIL, 0
   X := (Y EQV X) LOGAND A
   RESULTIS B / X
$)
```

EXAMPLE 1 - BCPL fragment

```
000000' ENT:       0,000000
000001' CLS:   S   0,000005
000002' LDA:   S   1,000005
000003' MOV:   S   1,000004
000004' LCA:     A 1,000001
000005' AND:   S   1,000002
000006' STA:   S   1,000004
000007' LDA:   S   1,000003
000010' DIV:   S   1,000004
000011' RTS:       0,000000
```

EXAMPLE 1 - Human-readable INTCODE 2

```
00000 0 000 0 0 0 0 00 00007 0
00001 0 012 0 0 0 0 00 00005 0
00002 0 047 0 0 1 0 01 00005 0
00003 0 046 0 0 1 0 01 00004 0
00004 0 002 0 0 0 0 01 00020 0
00005 0 044 0 0 1 0 01 00002 0
00006 0 011 0 0 1 0 01 00004 0
00007 0 047 0 0 1 0 01 00003 0
00008 0 055 0 0 1 0 01 00004 0
00009 0 000 0 0 0 0 00 00006 0
```

EXAMPLE 1 - Machine-readable INTCODE 2

```
EXTERNAL $( FOO ; MARY $)   // MARY IS DEFINED IN
                            // ANOTHER SEGMENT

LET FOO() BE
$( STATIC $( A = 0 $)
   LET B, C = 23, 6
   C := MARY + B / C
   A := C
   FOR I = 0 TO A DO B := B + 1
   IF MARY = 0 RETURN
   MARY := A
$)
```

EXAMPLE 2 - LCPL segment (above)

EXAMPLE 2 - Human-readable INTCODE 2 (below)

```
000000' SEG:        0,000006   SEGNAM
000002' ENT:        0,000000
000001' LDA: L      1,000027
000002' STA:  S     1,000002
000003' LDA: L      1,000006
000004' STA:  S     1,000003
000005' LDA:  S     1,000002
000006' DIV:  S     1,000003
000007' ADD:        1,000000* MARY
000011' STA:  S     1,000003
000012' LDA:  S     1,000003
000013' STA:        1,000000
000015' LDA:        1,000014'
000017' STA:  S     1,000005
000020' CLS:    A   0,000001
000021' JMP:        0,000000
000023' STA:  S     1,000004
000024' AOS:  S     0,000002
000025' LDA: L      1,000001
000026' ADA:  S     1,000004
000027' CHN:        0,000022'
000027' SGR:  S     1,000005
000030' JMP:        0,000023'
000032' LDA:        1,000000* MARY
000034' SNE: L      1,000000
000035' RTS:        0,000000
000036' LDA:        1,000016'
000040' STA:        1,000000* MARY
000042' RTS:        0,000000
000043' FIN:        0,000000
000044' VAL:        0,000000'
000045' CHN:        0,000037'
000045' VAL:        0,000000
000046' EXT:        0,000044' FOO
000046' ESG:        0,000000   SEGNAM
```