Part III — An Exercise in Machine Independence

9 The Use of Macro Processors in Implementing Machine-Independent Software

9.1 Introduction

There are at least three internationally accepted machine-independent high-level languages in which algorithms involving purely numerical manipulations can be encoded. Since compilers for one or more of these languages are available on most machines in the world and since it is a relatively easy (though by no means trivial) operation to transliterate between these languages, once an algorithm has been described in one of the languages it is readily available to anyone else who finds it useful. Moreover compilers for these algebraic languages are entirely satisfactory for the description of numerical algorithms.

However for algorithms involving a fair amount of non-numerical manipulations, and in particular for the description of software, suitable high-level languages are much less widely accepted and when compilers for these languages exist they tend to generate very slow, highly interpretive, object code for non-numerical features. Since software is heavily used it must be implemented in as efficient a manner as possible. Hence, because of the defects of high-level languages, nearly all software is coded in assembly languages or other machine-dependent languages, and if a piece of software has been implemented on one machine and it is required to implement it on a second machine, it has to be completely re-coded for the second machine. This involves a good deal of time and manpower. Various special-purpose languages for software writing have been designed to try to alleviate this problem but none of them is widely accepted and most either are machine-dependent or generate inefficient object code or both. Similarly general-purpose software-writing systems, such as the Compiler Compiler, suffer from both these defects.

9.2 Discussion of Machine-Independence

The importance of machine-independence can, however, be over exaggerated. A lot of software, particularly supervisors, is highly machine-dependent and hence there is no merit in trying to describe it in a machine-independent way. In fact the question of deciding whether the logic of a piece of software or, for that matter, a programming language, is machine-independent is always a tricky one. Strictly speaking it is impossible to say that the logic of any software is machine-independent; all that can be said is that it appears to work for all or nearly all the machines that are in existence at the present moment. Hence any discussion of machine-independence must necessarily be couched in rather inexact terms.

9.3 Simple Criteria for Machine-Independence

Two necessary, but not sufficient, conditions that software must satisfy to be considered machine-independent in its operation are:

a. The form of its input and output must be independent of the machine on which it runs (except for minor differences in character set, etc.).

b. It must be reasonably small, i.e. small enough to fit in the core storage of all but the smallest machines.

Compilers fail both these conditions. Firstly the output from a compiler should be (apart from very specialised exceptions) the object code of the machine on which it runs. Hence condition a) is not satisfied. Secondly compilers are normally large and the design of a compiler for a small machine will be completely different from the design of a compiler for the same language on a large machine. Hence condition b) is not satisfied. Therefore the most one can hope for in designing a compiler is to design parts of it in a machine-independent way and the same applies, *a fortiori*, to supervisors.

However the types of software that take streams of characters as input and generate streams of characters as output usually satisfy the above conditions and are usually machine-independent in their operation. Examples include editors, macro processors and processors for symbol manipulation languages. However even these will normally involve some operations that are best described in a machine-dependent way. Examples are I/O, type conversion and hashing functions.

9.4 The Problem

To sum up the preceding material, there exists a good deal of software (or parts of software) that is machine-independent in its operation, and it is desirable to describe it in a machine-independent language and to use a mechanical translation process to generate implementations for all the desired object machines. However existing compilers are not suitable for performing these translations for one or more of the following reasons:

a. The language that is compiled is not suitable for describing software.

b. Compilers are not available for many machines.

c. The generated object code is inefficient.

9.5 A Method of Solution

It is the purpose of this Part of the discussion to describe an alternative to the use of a pre-defined high-level language for the description of software. This alternative method removes all three of the above difficulties. The method is called a DLIMP, which stands for **D**escriptive **L**anguage **I**mplemented by **M**acro **P**rocessor.

Assume therefore that some software S is machine-independent and it is desired, or it is thought that it may be desired in the future, to implement it for several machines. If it is decided to accomplish this by a DLIMP using ML/I the procedure is as follows.

A *descriptive language* for S is designed. This will be represented as DL(S). DL(S) is a machine-independent language with semantics designed especially for describing S and with syntax designed to be translatable by ML/I into the assembly language of any machine and preferably into any suitable high-level language as well. Hence if, for example, S used a dictionary, the semantics of DL(S) would provide facilities for accessing dictionaries. Furthermore the data types in DL(S) would be those required for the description of S. Hence if S was implemented by list-processing techniques, DL(S) would have list data.

If it is desired to implement S on a machine M the first step in the process of implementation is to select an *object language*. The object language is some language for which a compiler or assembler exists for M and into which DL(S) can conveniently be translated

using ML/I. In practice the object language is normally the assembly language for M. When the object language has been chosen, *mapping macros* are written to make ML/I map DL(S) into the object language and these macros are used to map the logic of S as described in the language DL(S) into an equivalent description in the object language. This description in the object language is then compiled or assembled for M and this completes the implementation of S on M.

Note that the mapping of DL(S) into the object language can be performed on any machine for which an implementation of ML/I exists. It will often be convenient to perform the mapping at the installation where the designer of S works rather than on the object machine.

The technique described above has been used in practice, with S as ML/I itself, and most of the rest of this discussion is devoted to describing and evaluating this operation. Appendix B contains a complete implementor's manual for performing this operation. The language DL(ML/I) is called, simply, *L* and the operation of using ML/I to translate the logic of ML/I from L into some object language is called an *L-map*. L-maps have been performed for several object machines. When ML/I is first implemented on a machine the L-map must, of course, be performed on a different machine, but after an implementation of ML/I has been generated, this can be used to generate improved implementations for the same machine.

9.6 Advantages over Use of High-Level Languages

Three possible disadvantages of describing software in a high-level language were quoted earlier and it was claimed that a DLIMP overcame these. Now that the technique has been described this claim will be examined in detail.

The first disadvantage, the fact that the language may be unsuitable, clearly does not apply since a descriptive language is specially designed for the purpose it is to serve.

The second disadvantage, the fact that compilers may not exist for many machines, is overcome by the fact that any implementation of ML/I can be used to translate the descriptive language. There is no requirement that there be an implementation of ML/I on the object machine.

The third disadvantage is that compilers are liable to generate inefficient object code. However a descriptive language will be made to contain statements (or other syntactic classes) specially designed for performing the most heavily used operations in the logic of the software it describes and since each statement in the descriptive language will be mapped into specially tailored object code, the resultant implementation of the software will be quite efficient. In other words a DLIMP, being special-purpose, will generate better code than a general-purpose method. This is best illustrated by an example. Assume that the logic of some software uses data structures of a particular form. If the logic is encoded in a pre-defined high-level language, this language may have a general data structure facility but, being general, it is unlikely that it would be especially efficient for describing the particular data structure required. However if the software were implemented by a DLIMP the descriptive language would have statements specially designed for manipulating the particular kind of data structure required, and each of these statements would be mapped

into specially designed object code, which will perform its task in the most efficient possible way.

It is, therefore, the central contention of this discussion that *it is better to tailor the software-writing language to the software rather than vice versa.*

9.7 Advantages over Hand-Coding

The use of a DLIMP has many advantages over the method of describing software by means of, say, flow charts and then encoding it by hand for each object machine, though hand-coding will result in slightly more efficient object code. The main advantages of the use of a DLIMP are that it requires fewer man hours, it eliminates coding errors, it can be used to generate software for a newly manufactured machine with no software of its own and it makes trivial the upgrading of the generated software when a new version becomes available. This represents a fairly solid list of advantages and a DLIMP has its attraction even for describing machine-dependent software that is only required for one object machine. It may well be preferable to abandon assembly language coding in favour of the use of a good descriptive language that is both easy to read and easy to write, even allowing for the overheads of building up such a language. Indeed macro-assemblers have often been used in practice to accomplish this end.

9.8 Further Mappings

Now that one descriptive language, namely L, has been defined and mapped into several different object languages, it will be much easier to design future descriptive languages and map them into the same object languages. This is because, although many of the features of L are specially oriented to describing ML/I, L contains a kernel that will be applicable to any descriptive language. Thus if a descriptive language is required for describing some other software, this descriptive language can be derived from L by deleting the features of L that are only applicable to describing ML/I, and replacing them by features applicable to the software concerned. When mapping the new descriptive language into any of the object languages into which L has been mapped it will only be necessary to write mapping macros for the new features since the mapping macros common to L and the new descriptive language can be re-used.

In this context Wilkes' concept of inner and outer syntax [41] is very relevant. It will probably be possible for all descriptive languages to share the same outer syntax, and, with luck, some features of the inner syntax, arithmetic expressions for instance, can also be shared.

10 The Language L

This Chapter describes the language L and how it is used to describe ML/I.

The logic of ML/I is divided into two parts:

a. The *MI-logic*, which is the machine-independent part that can be mapped into each desired object language by means of an L-map.

b. The *MD-logic*, which is the machine-dependent part that requires hand-coding for each implementation of ML/I.

Typically the encoding of the MD-logic turns out to be one sixth of the size of the MI-logic. The operation of implementing ML/I by means of an L-map is represented diagrammatically in Figure 1.

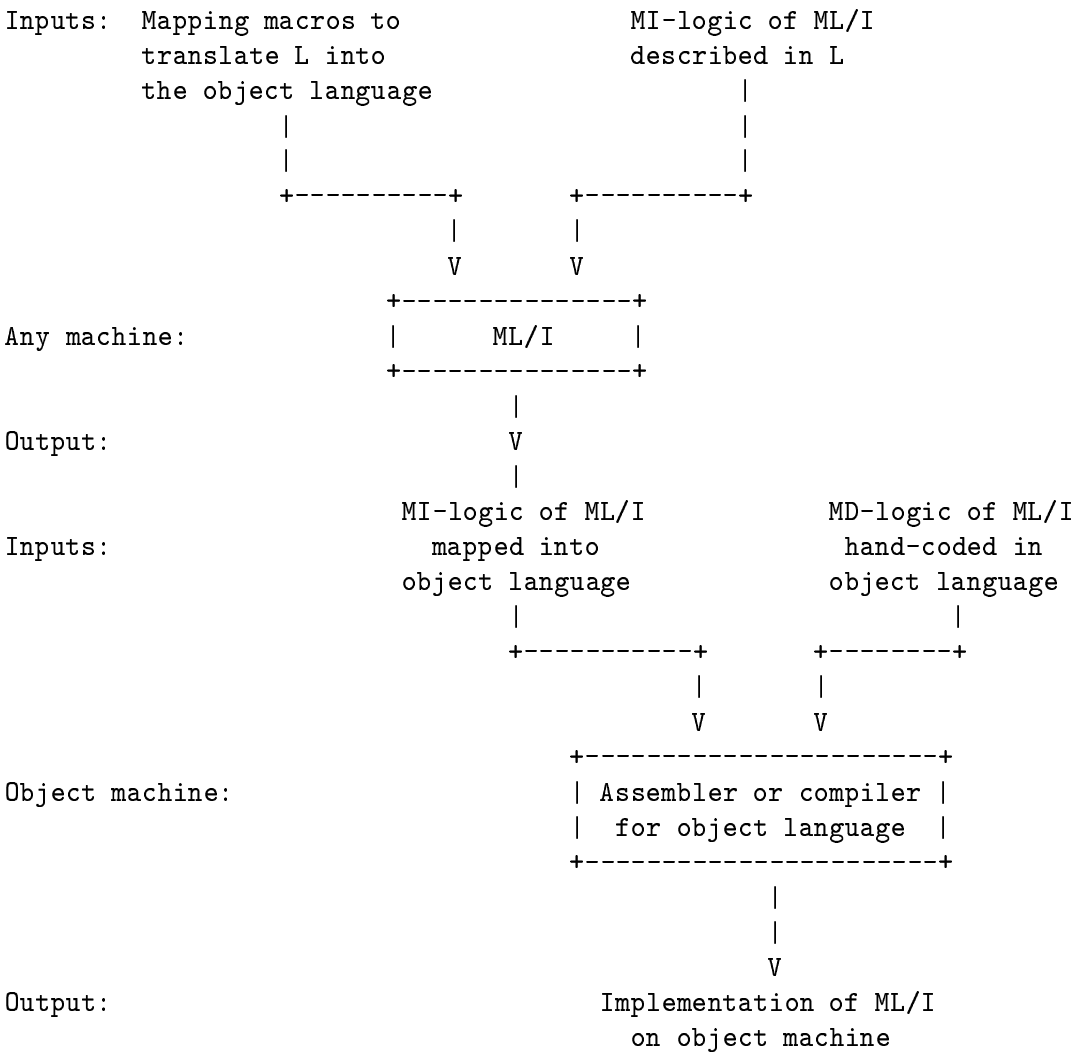

Figure 1

10.1 Basic Details of L

The full description of L appears in Appendix B and this Chapter will confine itself to describing the most important features of L.

L has been designed to be as readable as possible and furthermore it has been designed in such a way that it should be possible to map it into a suitable high-level language as well as any assembly language, so that if an object machine has a compiler for a high-level language that is suitable for describing the logic of ML/I, then it will be possible to map L into this language rather than into assembly language. Mapping into a high-level language would represent a "quick and dirty" way of implementing ML/I, unless, of course, the object machine possessed that rare animal, a high-level language suitable for software description that compiled into efficient code. In this latter case it would be "quick and clean" and hence would be a very desirable operation.

The most important characteristic of any language is the types of data that it allows. L has four data types, namely character, pointer, number and switch. The first three of these are self-explanatory. Switches are logical variables that can take on the value zero to seven. In each L-map data types are represented in the way most appropriate for the object machine.

L allows for variables, constants, and indirectly addressed data (i.e., data accessed by means of a pointer) of each data type. There are polymorphic operators in L but the data types of operands can be ascertained during an L-map by examining the last two letters of the identifiers, etc., used to represent the operands, so that there is no need to relay this information from declaration to point of use. Thus switch variables have names like INSW, DELSW, etc., and pointer variables have names like FFPT, DELPT, etc. L is in this respect similar to FORTRAN II since FORTRAN II uses the first character of variable names to indicate their data type.

L allows data of type number or of type pointer to be combined into arithmetic expressions, though it was found that only the addition and subtraction operators were necessary. L also has logical and comparison operators.

The MI-logic of ML/I is divided into several separate units called *SECTIONs*. In particular one SECTION contains exclusively declarations of variables and a pair of SECTIONs contain exclusively definitions of the tables of data used in the MI-logic. These tables of data contain the names and delimiters of the operation macros and some information about them. The two SECTIONs containing these tables are called collectively the *data SECTIONs*. The remaining SECTIONs contain exclusively executable statements.

L allows a set of declarations of variables to be combined into a contiguous *block*. A block of variables is a concept somewhat similar to the *structure* found in COBOL. There are no arrays of variables in L.

Most of the executable statements in L are simple statements, written one to a line. There are however two executable compound statements; one is an IF statement similar to that of ALGOL and the other is a statement for following down a chain and executing a given set of statements for each link of the chain.

In addition L contains statements for controlling the layout of the object language and facilities for comments. Each of these factures can, if desired, be ignored on an L-map if the readability of the object code is not considered important by the implementor.

Figure 2 contains a short extract from the description of the MI-logic of ML/I in L, showing two of the subroutines in the MI-logic, in order to give the reader some flavour of the language. If the meanings of the statements are not intuitively obvious, reference can be made to Appendix B for a full explanation.

```
SUBROUTINE GTATOM

        IF LEVEL = 0 & SKVAL GE 0 & FFPT-SPT = OF(LCH) THEN
                SET FFPT = INFFPT
                SET SPT = FFPT-OF(LCH)
        END
        CALL ADVNCE EXIT ENTEXT
        IF IND(SPT)CH = '$' THEN /-OVP-/SET LINECT = LINECT+1
        SET IDPT = SPT
        SET IDLEN = OF(LCH)
        CALL MDTEST(SPT)PT EXIT GT3
[GT2]   CALL ADVNCE EXIT ENDID
        CALL MDTEST(SPT)PT EXIT ENDID
        SET IDLEN = IDLEN+OF(LCH)
        GO TO GT2

//END OF IDENTIFIER//

[ENDID] SET SPT = SPT-OF(LCH)
[GT3]   RETURN FROM GTATOM

ENDSUB


SUBROUTINE LUDEL(PARPT) EXIT //DELIMITER NOT FOUND//

        CHAIN FROM PARPT EXIT MACERR
                CALL CMPARE(CHANPT+OF(LNM))PT EXIT LDCNT
                GO TO LDSUC
[LDCNT] ENDCH
        EXIT FROM LUDEL
[LDSUC] RETURN FROM LUDEL

ENDSUB
```

Figure 2

10.2 The Mapping of L

L contains thirty different kinds of statement, and mapping macros need to be written for all of these statements for each L-map. It has been found in practice that the mapping

macro for a statement in one L-map often has much the same form as the mapping macro for the same statement on another L-map, and so each L-map tends to involve a little less work for the implementor than the previous L-map. In addition to the mapping macros for statements, it is necessary to write mapping macros for machine-dependent constants, for indirect addresses and for arithmetic expressions. The writing of these last two requires a fair amount of skill and aptitude in using ML/I. Typically it takes about six weeks to write and debug a set of mapping macros. It is usually convenient, though not logically necessary, to perform L-maps using three separate passes (see Chapter 8 of Appendix B). The descriptions of the L-maps performed so far are presented in Chapter 12.

In order to give the reader an idea of what mapping macros look like, two examples will be given of the mapping macro for the simplest statement in L, the GO TO statement, which has the form

```
GO TO label
```

The standard conventions observed in the ML/I User's Manual (see Section 2.11 of Appendix A) will be used in these examples. The L-maps concerned are described in Chapter 12.

Example 1

Mapping macro for the L-map into PDP-7 Assembly Language:

```
MCDEF GO WITHS TO
AS<      JMP ~A1.
>;
```

Example 2

Mapping macro for the L-map into IIT for Titan (where label names are mapped into numbers on a prepass):

```
MCDEF GO WITHS TO
AS<      121          127      0        ~A1.
>;
```

10.3 Features of L That Specially Aid Machine-Independence

In many ways L is very similar in form to ordinary high-level languages, but it does contain two special features that are relevant to its use as a language to aid machine-independence. These features are its *constant-defining macros* and its *statement prefixes*. Each of these is explained below.

10.4 Constant-Defining Macros

The logic of ML/I involves many machine-dependent numbers and markers. These are represented in L by *constant-defining macros*. On each implementation of ML/I these constants are mapped into appropriate values for the machine concerned. The most heavily used constant-defining macro is the OF macro, which is described in detail in Section 3.3.1 of Appendix B. The OF macro is used to define constants that depend on the number of units of storage occupied by the various data types used in the logic of ML/I so that the offsets of items on stacks are represented correctly. Thus on IBM System/360 a pointer would

occupy four units of storage (i.e., four bytes), and if the value of a pointer was placed on a stack the stack pointer would need to be increased by four. On a completely word-oriented machine, on the other hand, a pointer would normally occupy only one unit of storage. (The amounts of storage occupied by data of the various types for an implementation are determined by the way in which the statements of L that declare or manipulate data are mapped.)

The sample of the MI-logic that was shown in Figure 2 contains several uses of the OF macro. It also includes an example of the constant-defining macro that denotes character constants. This macro has a single quote as its name and a single quote as the closing delimiter of its one argument. (See Section 3.3.2 of Appendix B for a fuller description of this macro.)

10.5 Statement Prefixes

The most necessary feature of any exercise in machine-independence is flexibility. In the next Chapter the advantages of ML/I in this respect are discussed. However when the language L was designed a further element of safety was added by making L itself flexible by means of *statement prefixes*. These are used to add extra information, of interest in only one L-map (or perhaps in only a small proportion of L-maps), to statements in L without upsetting other L-maps. Statement prefixes are normally used as an aid to generating optimal code for a particular machine or machines. An example follows.

10.6 Example of a Statement Prefix

Assume that it is desired to map L into the assembly language of a machine M. Assume further that the order code of M includes a special instruction which can be used to increase the contents of a storage location by a constant provided that the contents of that storage location is positive, whereas in the non-positive case it is necessary to use a sequence of several instructions. (A rather similar situation to this arises in practice withe the "Load Address" instruction on IBM System/360.) Now the MI-logic of ML/I contains statements such as

```
SET IDLEN = IDLEN + 1
```

and it would be useful if the mapping macros for the SET statement for M could be given the information on statements such as the one above as to whether IDLEN was always positive. This would enable optimal code to be generated in the positive cases.

The obvious solution to this is to change L by introducing a new statement called, say, "SETPOSITIVE" to deal with cases such as the one above. However this solution is unsatisfactory since a change in L would upset all other existing L-maps. Instead the problem is overcome by adding a prefix, "POS" say, to the SET statement and enclosing this within the delimiters "/-" and "-/" to make it a *statement prefix*. Thus the above SET statement would be written, in the case where IDLEN was always positive, as

```
/-POS-/ SET IDLEN = IDLEN + 1
```

The addition of statement prefixes does not upset existing L-maps since every L-map is required to delete statement prefixes by means of the following ML/I "skip":

```
MCSKIP / WITH - - WITH /;
```

This all-embracing skip can be overridden for each of the individual statement prefixes that it is desired to recognise on a particular L-map but it ensures that any new statement prefixes introduced into the logic of ML/I after the L-map was written do not upset the L-map when it is performed again on the new logic.

Hence the only problem for an implementor who wants a new statement prefix is to persuade me, as the writer of the MI-logic of ML/I, to add the statement prefix in the appropriate places. If I consent, the required information is available to the implementor who requested it and in any future L-map where it may be found useful.

Figure 2, which was presented earlier in this Chapter, contains an example of a statement prefix called "OVP" which is used for a slightly different purpose to the one suggested above; OVP means that arithmetic overflow is possible.

11 Comparison with Other DLIMPs

It is the purpose of this Chapter to describe some other DLIMPs that have been performed, using macro processors other than ML/I, and to discuss some of the advantages in using ML/I to perform a DLIMP. Firstly the considerations that go into the design of a descriptive language will be discussed, since this is of relevance to all DLIMPs.

11.1 The Design of a Descriptive Language

When designing a descriptive language it is a matter of judgement as to how complicated it should be made. The amount of effort required to write a set of mapping macros depends on the complication of the descriptive language.

At one extreme a very simple descriptive language could be designed, for example a language with semantics the same as a Turing machine. This would be very easy to map but would result in hopelessly cumbersome and inefficient object code.

At another extreme the descriptive language could be made to involve very complicated operations and/or a very large number of operations, in which case, although the description of the software to be implemented would be made simple and/or short and the resulting object code efficient, the mapping macros would require a considerable effort to write and the writing of them would be a little different from hand-coding the entire software. In the absolutely extreme case the entire software would be defined by the replacement of one macro.

However if some sensible middle course is taken and some reasonable balance is made between object code efficiency and ease of transfer, a DLIMP can be a very useful and time-saving operation.

11.2 Other DLIMPS

In addition to the implementing of ML/I by an L-map, a number of other DLIMPs have been performed, though only one of them has been described in the literature. These DLIMPs are described in the next sections.

11.3 Implementing of SNOBOL

The implementing of the symbol manipulation language SNOBOL4 has been performed using the Macro-FAP and similar macro-assemblers. The descriptive language used consists of 130 different types of statement, each corresponding to a FAP macro.

The notation of FAP can be converted, with a little editing, into the notation of most other macro-assemblers. Hence in order to implement SNOBOL (or rather most of SNOBOL, since some hand-coding is always necessary in this kind of exercise) for a given object machine, the implementor converts the description of the SNOBOL processor from FAP notation to the notation of the object machine's macro assembler and then writes 130 macros to define the machine code that is to replace each of the statements in the descriptive language. Using this technique SNOBOL implementations have been completed

or are being performed for the following machines: IBM 7090, CDC 6600, IBM System/360, Titan.

However the system is still under development and, as yet, no published description exists.

11.4 Implementing of Meta-Assemblers

Ferguson [11] has described a descriptive language called METAPLAN that has been used to describe meta-assemblers. In order to implement a meta-assembler to run on a given object machine the first step is to make an existing meta-assembler capable of acting as an assembler for the object machine. When this has been done macros are written to map METAPLAN into the assembly language of the object machine. These macros are run on an existing meta-assembler or on a specially written macro processor. They perform the mapping of the logic of a meta-assembler in METAPLAN to an equivalent description in the assembly language of the object machine. This is then assembled by the meta-assembler that has been made to act as an assembler for the object machine. Full details of this work have not been published (though Ferguson has been kind enough to send me a rather more detailed account of the operation than appears in his original paper), but the technique has been used to implement meta-assemblers for a number of machines. It can be seen that Ferguson's technique is very similar in concept to an L-map but it is rather more specialised in that it requires that the object language be a meta-assembly language whereas there is no such restriction on an L-map. However Ferguson points out that it is quite a trivial operation to make an existing meta-assembler capable of assembling for almost any machine.

11.5 DLIMPs Using WISP

It is not, perhaps, generally realised that the WISP compiler is simply a macro processor. It so happens that most users of the WISP compiler use a built-in set of standard forms (i.e., macros) which have come to be known as the WISP language. However the user of the WISP compiler not only can supplement these built-in standard forms but he can throw them all away and replace them by a completely different set. Hence the WISP compiler is a suitable vehicle for performing DLIMPs and, indeed, several DLIMPs have been performed using it, including:

a. The implementing of the WISP compiler using the WISP language as descriptive language.

b. The implementing of LIMP using an extended version of the WISP language as descriptive language.

c. The implementing of AMBIT (a symbol manipulation language) using a modification of the WISP language as descriptive language.

Case a) is the only DLIMP that has been described in the literature and Wilkes' account of it [40] is well worth reading. In this DLIMP the descriptive language was developed as much as a language in its own right as a language for describing the WISP compiler. However, if a language can satisfactorily serve such a dual purpose it is all to the good.

The emphasis of Wilkes' article is as much on the bootstrapping aspect, i.e., the use of an existing WISP compiler to generate a more powerful WISP compiler for the same machine, as on the transfer from machine to machine.

Cases b) and c) are, however, more perfect examples of DLIMPs since in these cases the descriptive language was developed specially to describe the software concerned.

11.6 Comparison with Other Methods

It is a feature of a DLIMP that it can be used for any machine-independent software and it places no constraints on the logic of the software to be implemented. Software implemented by a DLIMP runs completely independently of the macro processor used to implement it. Relative to other automated methods of software implementation a DLIMP is fairly unambitious in that the implementor is left to do a good deal of work for himself; he gets no help in the design of the logic of the software and he has to write mapping macros for each object language. In order to set DLIMPs in perspective it is worth mentioning some more specialised, more ambitious, techniques for software implementation.

Firstly there are a number of techniques where a general-purpose processor is made to *act as* the software it is desired to implement. Under this heading come syntax-directed compilers and Halpern's [18] proposed use of a macro processor as a general-purpose compiler. In systems such as these the implementor is relieved of some of the work of designing the logic of the software he wants to implement, but the logic of the software is constrained to work on the principles round which the general-purpose processor has been designed.

Secondly the well known technique of "writing a compiler in itself" should be mentioned. The best-known example of this is provided by NELIAC [20]. In order to transfer a compiler to a new machine it is necessary to change the compiler itself in order to make it generate code for the new machine. Thus transferable compilers are organised in such a way that this change is relatively easy to make. Apart from this special consideration the writing of a compiler in itself is merely an example of describing software in a pre-defined language.

These two techniques have been mentioned merely to point out their differences with DLIMPs and they will not be considered in the rest of this discussion. Since both of the techniques have entirely different objectives from DLIMPs, there is no point in trying to compare their relative merits with DLIMPs.

11.7 Advantages of using ML/I for a DLIMP

It can be seen that there is nothing conceptually new in the idea of a L-map since several other DLIMPs have been performed, some of them before ML/I was even conceived of. However the difference between one DLIMP and another lies in the macro processor used to perform it. It is felt that ML/I offers considerable advantages in this respect in terms of both flexibility and power, and this makes ML/I a practical tool for converting a wider range of software to a larger number of machines. These claims about the flexibility and power of ML/I will be discussed in the rest of this Chapter. However the best way to judge claims such as these, especially in the field of machine-independence, which is notorious for glib unsubstantiated claims, is by the results. In the field of machine-independence,

projects often seem sound and well-planned but fail because of a number of trifling details overlooked in the overall plan. The results of L-maps using ML/I are presented in Chapter 12 and it is these that really prove the success of the technique.

11.8 Flexibility

As has been said, the most important feature that a scheme for machine-independence must possess is flexibility. Each object machine and each object language has its own individualistic quirks and one can never find out the troubles that will arise in an implementation for a particular machine until one actually starts working on that implementation.

ML/I offers considerable flexibility because it is notation-independent and because it allows a free choice as to which features of the descriptive language need mapping in a particular implementation and which can be left unchanged. This second point is worthy of example. In the language L names of labels and variables are represented by identifiers of six or fewer characters. On most L-maps that have been performed these names were left unchanged since such identifiers were legal in the object language. However in one L-map the object language only permitted five-character identifiers for names of labels and in another case the object language was purely numerical and all identifiers had to be mapped into numbers. In each of these two cases it was possible to write macros for ML/I to make it capable, on a pre-pass, of recognising all those identifiers that required alteration and generating macro definitions that would make the required changes on a subsequent pass.

The flexibility of ML/I is borne out by the fact that on all the L-maps that have been performed it has never been necessary to make any modifications to ML/I or to the compiler or assembler for the base language. (In the only other published description of a DLIMP [40], the assembler for at least one of the object machines, the Elliott 803, required special modification. This was because the 803 assembler requires labels to follow rather than to precede the statement to which they are attached. ML/I could perform this transposition quite easily.)

11.9 Power

It is claimed that ML/I permits the use of much more powerful descriptive languages than has previously been possible, and it is hoped that a glance at Figure 2 of Chapter 10 will convince the reader that the language L is akin to a high-level language in that it is easy to read and easy to write. In fact L has been mapped with comparative ease into a high-level language (PL/I) whereas no attempt has been made to map any other descriptive language into anything other than an assembly language.

Two facilities of ML/I are especially relevant in giving it extra power, namely its delimiter structures and the allowing of macro calls within macro calls. The use of delimiter structures leads to features of L such as its elaborate IF statement and its arbitrarily long arithmetic expressions. The facility for nested macro calls in ML/I makes it possible for arithmetic expressions in L to occur within many different types of statement. In WISP, and other logically similar macro processors, this sort of elaboration is impractical because it would be necessary to enumerate every possible form of each statement; in the case of L, this would require hundreds or even thousands of standard forms. In fact WISP seems

to be very suitable for list-processing applications, where it has traditionally been found acceptable to use statements of a very simple syntax, but in applications where arithmetic is involved WISP is less suitable. Of course the whole point about WISP is that it is *intended* to be a very simple tool rather than a sophisticated one.

The ML/I facility for nested macro calls was found useful when designing L for many other purposes in addition to arithmetic expressions. Statements in L which include other statements are the executable compound statement, like the IF statement, and the statements for declaring "blocks" of variables. Macros in L which occur within other statements include the constant-defining macros and the macro for indirect addressing.

ML/I also allows the use of several different data types within L though there are other macro processors in which this would be possible.

The fact that ML/I allows, for the same mapping effort as other macro processors, a descriptive language involving much more complicated statements means that ML/I will generate much more efficient object code since it is well known that in macro-generated code the worst inefficiencies occur at boundaries between statements. To illustrate this point, it would be much easier to generate efficient code from:

```
SET X = -Y + Z - C + 1
```

than from the equivalent statements:

```
SET TEMP = -Y
SET TEMP = TEMP + Z
SET TEMP = TEMP - C
SET TEMP = TEMP + 1
```

Unfortunately there have been no published figures for the efficiency of code generated by other DLIMPs but it is felt that the figures quoted in the next Chapter for the L-maps that have been performed are very good ones.

12 Results of Transfers

All current implementations of ML/I have been generated by L-maps. Initially, versions of ML/I were hand-coded for both Titan and the PDP-7 and these versions were used to perform the first L-maps. (If the operation of implementing ML/I had originally been planned as a bootstrapping operation, it would only have been necessary to code one version by hand.) Each of these two hand-coded versions was written before ML/I had been fully developed and the Titan version, especially, lacked many of the features that make ML/I especially suitable for performing an L-map.

In this Chapter the four L-maps that have been completed are described in some detail and the two L-maps that are under development are more briefly mentioned. These two incomplete L-maps have reached the stage where, from the evidence of similar L-maps, all the main difficulties have been found and overcome. Hence it is reasonable to assume that the completion of these projects, though requiring a good deal of time and effort, would not reveal any logical difficulties.

Before considering the individual L-maps, it is necessary to mention a few general points about them.

12.1 Inefficiency

An attempt has been made to measure the degree of inefficiency in size and speed of the object code generated on each L-map. The inefficiency is measured by comparing the macro-generated code with the code that would result if a competent programmer were given the description of the MI-logic of ML/I in L and told to code it by hand in the object language. It is assumed that this programmer uses no programming tricks nor any techniques dependent on a knowledge of the innermost internal workings of the logic of ML/I since both of these should be discouraged in all software writing because of maintenance problems.

Note that the use of the machine-independent language L to describe the logic of ML/I introduces some inefficiency. For example the original hand-coded version of ML/I for the PDP-7 made use of the spare bits in a word containing a pointer, but there is no place for this sort of thing in a machine-independent description of the logic. However this source of inefficiency, which in any case is very small, has not been allowed for in the measures of inefficiency that are quoted, since these measures are only intended to give the extra inefficiency introduced by the use of macro techniques rather than hand-coding in the implementing of machine-independent software.

Note further that each measure of inefficiency only applies to the particular L-map that was performed and a second L-map into the same object language might have entirely different characteristics. The degree of inefficiency is, in fact, partly a measure of the skill and effort put into writing the mapping macros.

12.2 Time Taken

In the descriptions that follow of L-maps, an estimate is given of the approximate number of macro calls that has been needed to accomplish this mapping. Current implementations of ML/I on Titan and the PDP-7 proceed at about two to three thousand calls per minute, and

it will be found that the average L-map into an assembly language, involving the translation of the entire MI-logic of ML/I, which consists of about two thousand L statements, takes about twenty minutes computing time on either of these machines.

In addition an estimate is given, for each L-map, of the number of man-weeks needed to write and debug the mapping macros. Normally one or two undetected bugs in the mapping macros still remain when a mapping is performed, but in all L-maps so far performed it has been easy to correct these errors by hand-editing of the generated object code without recourse to a second mapping. However it takes an extra week to a fortnight over and above the times quoted to check out the generated object code for the MI-logic and its interface with the MD-logic. The PL/I L-map was exceptionally bad in this respect and required twenty days of debugging on the object machine.

On two L-maps, those into IIT and PLAN, some time was saved by encoding the data SECTIONs by hand rather than by macro mapping. This is because the data SECTIONs are very short and require several special mapping macros, and, at least in the short term, it is quicker to code them by hand than to write the extra mapping macros. In fact hand-coding is recommended in Appendix B.

Apart from the data SECTIONs the entire MI-logic has been mapped by macro replacement on every L-map. In the description of each L-map an estimate is given of the number of lines needed to specify the mapping macros, each line normally being a single object language instruction, a call of an ML/I operation macro or a call of some intermediate macro such as one of the macros described in Chapter 8 of Appendix B.

12.3 Representation of Data Types

The most important decision to be taken when planning an L-map is how the four data types of L are to be represented on the object machine. The way this is done is therefore specified in the descriptions of L-maps that follow.

In all three L-maps so far performed for word machines (i.e., machines where the smallest addressable unit is a word rather than a character) all four data types have been represented in the same way and each has occupied a single word of storage (or, in the case of Titan, a half-word since this is the smallest addressable unit). In each of these machines it would have been possible to pack more than one character to a word (or half-word) but this was never done because the resultant increase in size and slowness of the object code would more than offset the gain in compactness of the data. The only packing that has ever been done is on the character string constants used in error messages.

12.4 Acknowledgements

I would like to thank the following people, each of whom has made a major contribution to an L-map:

- Mrs. Heather Brown and Mr. Richard Gray of Cambridge University Mathematical Laboratory.
- Mr. John Nicholls and Mr. Peter Seaman of IBM (UK) Laboratories.
- Mr. Stuart Clelland and Mr. Ian Duncan of Ferranti.

- Mr. Arthur Adams of Glaxo Laboratories.
- Mr. Lorne Bouchard of the University of Essex.
- Mr. Brian Chapman of Honeywell.

I have been extremely lucky in the quality of the people I have found to assist me and am extremely grateful to all those mentioned above and to a multitude of others who have assisted me in smaller ways on the various L-maps.

12.5 The PDP-7 Assembly Language L-map

The Project

L was mapped by me into PDP-7 Assembly Language. This mapping has been performed several times during the development of ML/I and L. Both Titan and the PDP-7 have been used at various times to perform the mapping. The macro-generated PDP-7 implementation of ML/I has been working satisfactorily since March 1967 with a new version being released in August 1967.

The Machine

The PDP-7 is a word machine with 8K 18-bit words, one accumulator and a very simple order code. It represents about the smallest possible configuration on which ML/I can usefully be implemented.

Because of the smallness of the machine, the emphasis of the PDP-7 L-map was on generating object code that was as concise as possible rather than as fast as possible. Thus the object code was designed to be very highly subroutined.

Representation of Data Types

All types of data were stored in a single word. Numbers were stored in two's complement form.

Difficulties

As an object machine the PDP-7 presented no great problems. Its very simple order code was an asset. Furthermore the PDP-7 Assembler performed fairly well as an object language. Its two main deficiencies were that it requires negative numerical constants to be represented in one's complement form rather than two's complement form and that it contains no usable facility for character string literals or character string data constants.

The Mapping

The PDP-7 L-map was the first L-map to be performed, and, although some of the mapping macros have been updated, the overall organisation has never been changed from its original state. From the experience gained from the PDP-7 and Titan L-maps improved mapping techniques have been developed and if the PDP-7 L-map were now re-planned, starting from scratch, the mapping could be performed much more quickly and easily.

The PDP-7 L-map was performed in four passes of which the last was simply an optimiser. The optimiser managed to eliminate about forty redundant instructions from the object code. Altogether Ml/I executed 85,000 macro calls in performing the PDP-7 L-map. About one third of these were taken up by the optimiser and about one sixth in converting character string literals into numerical values. An effort was made to make the object code look fairly neat and comments in L were copied over to the object code.

Overall it required about 700 lines to specify the mapping macros and it took about a month to write and debug them.

The Object Code

The generated object code consisted of about 3,500 words, of which about 2,950 were instructions and the remainder were declarations and data. Thus it required an average of 25 macro calls to generate each word of object code. The MD-logic for the PDP-7 was relatively large, being nearly 1,500 words. This was because the PDP-7 has no supervisor and hence all the I/O routines and interrupt routines had to be included in the MD-logic and also because the PDP-7 implementation included a lot of extra facilities, including:

a. The ability to communicate with the user via the teleprinter.

b. The ability to dump itself out on paper tape for later re-use.

c. Its own loader.

d. The ability to perform I/O in either Titan flexowriter code or ASCII.

The object code generated by the PDP-7 L-map represents about the ultimate in object code efficiency likely to be achieved in an L-map. To test the efficiency of the code certain parts of the MI-logic of ML/I were encoded by hand with the same design objectives as the macro-generated code, namely to be as concise as possible, and it was found that the macro-generated code was about 3% inefficient in size and speed. The speed inefficiency is relatively unimportant since the PDP-7 implementation is normally I/O bound.

These results are considerably better than are likely to be achieved on many other L-maps. The reason for the efficiency of the PDP-7 L-map is that the PDP-7 has so few instructions that there is little scope for clever coding by hand.

Conclusion

The PDP-7 L-map was highly successful and represented a very rare achievement: an exercise in machine-independence that resulted in only trivial inefficiencies in the generated object code.

12.6 The IIT L-map for Titan

The Project

L was mapped into IIT, the non-symbolic assembly language for Titan. The mapping was performed by H. Brown of the staff of the Cambridge University Mathematical Laboratory, and the resultant implementation has been working satisfactorily since June 1967. The mapping was performed using an early, rather inadequate, version of ML/I on Titan, which had been coded by hand.

The Machine

Titan is a large word machine with 48-bit words and an extensive, rather inhomogeneous, order code. It is possible to address 24-bit half-words. Titan has 90 general-purpose registers (i.e., accumulators and/or index registers), called *B-lines*, that are available to the general user.

Representation of Data Types

All types of data were stored in a single 24-bit B-line or half-word.

Difficulties

The Titan L-map probably presented more difficulties than any two others put together.

Titan has a very large order code and it takes a lot of work to design macros that pick the right instruction for the right purpose. If advantage were not taken of Titan's large range of special-purpose instructions a program would be very inefficient.

However most of the troubles of the Titan L-map arose because of the inadequacies of IIT. (This situation has recently improved and Titan now has an adequate symbolic assembler.) Because IIT is purely numerical, all names in L had to be mapped into numbers. Other troubles were the lack of character string literals and difficulties arising from the fact that some constants had to be represented as a decimal integer coupled with an octal fraction.

The Mapping

The Titan L-map was performed on an old version of ML/I while L was still being developed. The Titan mapping macros have never been updated to conform with the latest developments of L and ML/I.

The mapping, which was accomplished in four passes, required 75,000 macro calls. The mapping macros took two months to write and debug and it took 700 lines to specify them. (Hopefully all these figures would be cut by a third or more if the Titan L-map was repeated using the latest version of ML/I and mapping into a symbolic assembly language.)

The Object Code

The MI-logic of ML/I required 2,430 words, of which 2170 were macro-generated instructions and 260 were the hand-coded data SECTIONs. The MD-logic required only 370 words, making a total of 2,800.

The generated object code for Titan was about 5% wasteful in size and 25% inefficient in speed. In order to improve the speed the most heavily used part of the MI-logic, the basic scanning loop, was re-coded by hand. This involved hand-coding 50 instructions and cut the speed inefficiency down to 5%. It was rather fortunate that the number of variables required in the logic of ML/I was just fewer than the number of B-lines available on Titan and so the B-lines were used quite efficiently.

Conclusion

The Titan L-map was not quite such an outstanding success as the PDP-7 but it was felt by the implementor to be a much better method of implementation than coding by hand. Compared with Titan software that has been entirely coded by hand ML/I has been remarkably free of bugs and the inefficiencies of speed and size are trivial compared with Titan software generated by other artificial means, for instance by the Compiler Compiler.

12.7 The PL/I L-map

The Project

L was mapped by me into the high-level language PL/I for the IBM System/360 (or any other machine with a PL/I compiler). The resultant implementation on System/360 has been working since January 1968.

This project was attempted because PL/I has been claimed to be a suitable language for software writing and it was of interest to compare the results of an L-map into PL/I with the results of assembly language L-maps.

The Machine

Since PL/I is a machine-independent language the characteristics of the object machine are not very relevant.

Representation of Data Types

PL/I contains all the data types of L but unfortunately it was not possible to make use of these since there is no way, at least no straightforward way, of defining in PL/I the stacks involving dynamically mixed data types that are required by the logic of ML/I. Instead all the data types of L had to be represented as integers in PL/I, pointers being represented as indices to an array and characters being converted to numerical representation on input.

Difficulties

The interesting thing about the PL/I L-map was that many of the features of L requiring a lot of work in assembly language L-maps were trivial in the PL/I L-map. However there were a few features of L where the reverse applied, notably the statements for communicating with the linkroutine (see Section 4.1.2 of Appendix B), the statements for declaring "blocks" of variables (see Section 5 of Appendix B) and the encoding of the data SECTIONs. (The data SECTIONs would have been even more difficult to code by hand than by macro replacement.) However overall the PL/I L-map was much easier than any assembly language L-maps have been.

The Mapping

The PL/I L-map was performed mostly on Titan but part of the work was done on the PDP-7. The job was done in three passes (plus an extra pass through the data SECTIONs) and required 25,000 macro calls. The machine time taken to perform the mapping of L into PL/I was much less than that taken to compile the resultant PL/I program. Furthermore a good deal of the mapping time was spent on the non-essential task of controlling the layout of the object program, in particular indenting statements to make the program more readable.

It required about three weeks to write and debug the mapping macros and altogether their specification only occupied 250 lines. A few replacements were left for the PL/I macro processor to perform in order that these could be altered at Winchester, where the object machine was, rather than at Cambridge. This turned out to be a wise decision. The PL/I L-map was, in fact, the first L-map where the object machine was a large physical distance from the implementor's installation. Because of communication problems the debugging of the object code proceeded rather slowly, but thanks to the excellent work of P. Seaman at Winchester, the object code was working after twenty days. A large number of the errors were due to my lack of familiarity with PL/I. As a result of this experience, it was decided in future, when performing an L-map for a physically distant machine, to map a short test program before attempting to map the entire MI-logic of ML/I. (It would, of course, have been sensible to have planned it this way in the first place.)

The Object Code

The PL/I object code for the MI-logic of ML/I consisted of 1,750 PL/I statements and the MD-logic, which was hand-coded in PL/I, added 250 statements to this. In some cases

a sequence of several statements in L mapped into a single statement in PL/I although this did not apply in the case of executable statements. It took 18 minutes to compile the object code on a System/360 Model 40 (i.e., about the same time as it takes Titan to perform an L-map into assembly language) and the compiled code occupied 80,000 bytes (i.e., 20,000 words) of storage. This is about five times larger than would be expected if the L-map had mapped into System/360's assembly language rather than PL/I. Moreover the compiled code was extremely slow in execution, being ten to twenty times slower than would be expected from an assembly language implementation.

These huge inefficiencies were not due to the use of macro mapping since if I had hand-coded the MI-logic of ML/I into PL/I the object code would have been very little different. However some of the inefficiencies were due to my lack of knowledge of PL/I and in particular to my lack of appreciation that a subroutine call in PL/I involves a very heavy time penalty. Many features of L were mapped into subroutine calls in PL/I when in-line code would have been much better. Moreover it would have been a good idea to replace the calls of the shorter subroutines in the MI-logic of ML/I by in-line code, a job that can easily be accomplished by macros. Unfortunately it is not a trivial matter to perform another L-map for PL/I since the existing implementation is so slow that it would take several hours to generate a new version of itself and if one of the Cambridge machines is used for the job the resultant output needs to be keypunched by hand because the Cambridge machines produce paper tape output and the Winchester machine only accepts card input.

Improvements such as those suggested above would probably double or treble the speed of the PL/I implementation but large inefficiencies would still remain. This is due to the fact that the PL/I compiler is still rather crude and the PL/I language is not very suitable for describing the logic of ML/I with the result that many operations have to be done in a rather clumsy way. This, of course, bears out the contention made in Chapter 9 that software should not be coded in a pre-defined high-level language.

Conclusion

This project produced an implementation of ML/I that is so large and slow that its usefulness is very limited. The mapping could have been made to produce rather faster code, but overall the project has shown that PL/I at present has severe limitations as a software writing language.

The fact that L can be mapped into a high-level language has been proved and this may prove useful if a suitable software writing language with an efficient compiler exists for a future object machine.

12.8 The PLAN L-map for ICT 1900

The Project

L was mapped into PLAN, the assembly language for the ICT 1900 series, by R.G. Gray, a student for the Diploma in Computer Science at Cambridge University. I supervised the project and wrote a few of the mapping macros. At the time of writing, the mapping of the MI-logic has been performed but the debugging of the MD-logic and its integration with the MI-logic has not yet been completed. The object machine is at the University of Essex and L. Bouchard, a research student there, has written the MD-logic and taken responsibility for the 1900 side of the project.

An earlier, independent, PLAN L-map was started by S. Clelland, a vacation student with Ferranti at Dalkeith. He completed about three-quarters of the work involved during his eight weeks with the company. This was a very commendable achievement as he was working under severe difficulties, for example: no initial knowledge of ML/I and little experience of PLAN, difficult and remote machine access, poor documentation of L — the implementor's manual was not ready at the time of his work — and, above all, no ready access to anyone familiar with ML/I or the techniques used in an L-map.

The results of Clelland's L-map became available to Gray about half way through his own work and enabled him to improve and correct some of his mapping macros. Furthermore Gray used Clelland's encoding of the data SECTIONs, which was done by hand.

The Machine

The 1900 series is a range of word machines. A word consists of 24 bits. Each machine has eight accumulators of which three can also be used as index registers. Gray's L-map was for a 16K machine.

Representation of Data Types

All data types of L were stored in a full word of storage, because, although the 1900 has some instructions for dealing with characters packed four to a word, the set of such instructions is not sufficiently comprehensive to be useful; there is, for example, no direct way of comparing individual characters in packed form.

Difficulties

In general the 1900 was found to be a fairly good object machine. Its MOVE instruction and its instructions with literal operands were very useful.

However PLAN was very disappointing and compared very unfavourably with assemblers on similar machines. Even the PDP-7 Assembler is in many ways more powerful than the most elaborate version of PLAN. Defects of PLAN as an object language for an L-map included: no character string literals, no symbolic register names, the restriction of names of labels to five characters, inexact documentation and a number of petty restrictions indicative of poor organisation in planning the PLAN assembler.

The Mapping

It required 29,000 macro calls to perform the 1900 L-map. It took six weeks to write and debug the mapping macros, the specification of which occupied 520 lines. The mapping was performed on Titan.

Note that the above figure of 29,000 represents a considerable improvement over earlier assembly language L-maps.

The Object Code

The MI-logic of ML/I occupied 3,950 words on the 1900 of which 550 words were declarations and data and the remainder were executable instructions. The MD-logic, which has not been fully debugged yet, occupies about 400 words.

The macro-generated object code is about 12% inefficient in speed and size. This is mainly due to the fact that the eight accumulators are not used in an optimal manner.

Conclusion

The 1900 L-map seems to be another successful one though, as would be expected, the object code is rather less efficient than for the PDP-7.

It is pleasing to see that, as a result of experience gained from previous L-maps, it has been possible on this L-map to cut down considerably on the amount of machine time needed to perform the mapping. This bears out the claims made previously that the PDP-7 and Titan L-maps could, if re-written, be made much faster.

12.9 The IBM System/360 Assembly Language L-map

The Project

I started an L-map into the assembly language of the IBM System/360 (or any of the compatible machines of other manufacturers). This project was shelved after about two-thirds of the mapping macros had been written in favour of the PL/I L-map described earlier because it was felt that the PL/I project was of more research interest.

The Machine

System/360 is a range of compatible machines where the store is addressable either in terms of 8-bit "bytes" or in terms of 32-bit words. System/360 contains most of the facilities of both character machines and word machines; in particular a good set of character manipulation instructions is available. System/360 has 16 general-purpose registers.

Representation of Data Types

Characters and switches were stored in one byte and numbers and pointers in one word.

Difficulties

Both the machine and its assembler were excellent for an L-map and no serious difficulties were encountered although the following minor difficulties arose:

a. The data types in L were treated in different ways since they occupy different units of storage.
b. System/360 has several different instruction formats.
c. The indirect addressing macro was rather clumsy because System/360 requires certain data fields to be aligned to word boundaries.

The first of these is really an asset rather than a deficiency, but it does necessitate some extra work in writing mapping macros.

Object Code

The object code would probably be fairly inefficient in speed and size, say about 40%, in view of the fact that it would be difficult to make optimal use of the sixteen general-purpose registers or of the wide variety of instructions available on System/360.

Conclusion

An L-map into System/360 assembly language is feasible and, in view of the fate of the PL/I implementation, desirable.

12.10 The EASYCODER L-map for Honeywell 200

The Project

B. Chapman of Honeywell is working, with a little help from me, on an L-map into EASYCODER, the assembly language of the Honeywell Series 200 machines. He has been

working on this project very much as a part time activity and work has now been suspended while he visits America.

The Machine

The Honeywell 200 is a character machine. Storage is addressable in units of 6-bit characters. The sizes of data fields are defined by means of "word marks" and "item marks" within the data itself rather than by the instructions that manipulate the data. Numerical values can be defined in either character or binary form. The machine has no accumulator but six index registers.

Representation of Data Types

Characters were represented in one character of storage and the remaining data types in three characters. Numerical values were represented in binary form and word marks were placed at the left hand end of data fields, as is standard.

Difficulties

The machine is very complicated and hence presented problems in generating efficient code. EASYCODER, however, was very satisfactory as an object language. There were difficulties due to the design of L, rather than the machine, and these are noted below.

The Object Code

The object code will be very much less efficient than would result if the logic of ML/I had been specially planned for the Series 200. This is because certain features of L do not fit in very well with the machine. In particular the lengths of data fields in L are defined explicitly rather than by special markers, and pointers in L point at the left-hand sides of the fields they designate whereas it is more convenient on Series 200 to point at the right.

Conclusion

The Series 200 shows up some limitations of L as a machine-independent language. However an L-map is clearly feasible.

12.11 Summary of Results

An L-map into the average assembly language takes about six weeks to perform and the whole project of implementing ML/I by an L-map can be completed by one skilled person in two months. The inefficiency in size and speed of the generated code varies between almost nothing and as much as 50% with an average of about 20%. The object code should be free of bugs (in as much as any code is) and, once one L-map has been done, it will be a trivial matter to implement further versions of ML/I.

Although the logic of ML/I could be coded by hand in a month or so, it would take very much longer than this to debug the resultant code and the implementor would have the unenviable task of finding out about all the details of the workings of the logic of ML/I.

Hence an L-map is a very advantageous way of implementing ML/I. If the technique is extended and used to implement other software the advantages multiply and multiply.