C. 1

Software Portability - Theory and Practice

by

Noel Kalicharan

B.Sc.(Hons), University of the West Indies, 1973.

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in the

Department of Computer Science

We accept this thesis as conforming

to the required standard

The University of British Columbia

June 1976

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 6th July 1976

ABSTRACT

Large amounts of time and money are currently being spent in transferring computer programs from one machine to another. It seems unlikely that the problem of transferring such programs efficiently will ever be completely solved. What can be done, however, is to program in such a way that transfer costs are minimized. The first part of this paper discusses some effective techniques which have been developed to facilitate the writing of mobile programs. The second part discusses two detailed applications of one of these techniques, the use of an Abstract Logical Language.

Contents

ACKNOWLEDGEMENTS

I wish to thank Dr. John Peck, my supervisor, for his many helpful suggestions which made the writing of this thesis possible, and Dr. Harvey Abramson for his time and patience in reading the thesis.

0.0 Introduction

There is a pressing problem that any designer or implementor of software must face today. It is one of obsolescence. The threat of obsolescence may manifest itself in more ways than one. For instance, the computer on which the software is running may become obsolete and have to be replaced, or the designer/implementor is moving and wants to take his software with him, necessitating a transfer to another installation. These threats can be alleviated if due consideration is given to the writing of the software in as machine-independent a way as possible. If not, many good pieces of software will die with the computer they ran on.

Software portability is of supreme practical importance since astronomical sums of money are currently spent in re-programming for new environments. It is an obvious waste of time and effort, if, in order to implement identical pieces of software on different computers, one has to start from scratch in each case.

It seems unlikely that the problem of transferring programs efficiently from one machine to another will ever be completely solved. It will always be the case that some software would need to be recoded for a new environment. What can be done is to program in such a way that transfer costs are minimized.

Many effective techniques have been developed which lend themselves to the writing of highly mobile programs. Some of these techniques are discussed in the sequel, and two portability exercises are presented.

1.0 Part 1 - THEORY

Before concentrating on the major techniques used in the production of portable software, let us dispense with two extreme, trivial cases.

Case 1 : No Solution

Warshall[25] argues that it is impossible to achieve true portability. Undoubtedly, this is true if our portability exercise is of the form: software S is implemented on one machine and then someone else wants an implementation for his machine. This new someone might have a machine, M, which was designed years after S was first implemented. M might be radically different from any machine produced up to that time. The point is that it may be impossible to achieve portability over all current machines and all that can be imagined in the future. But that is not to say that one should not try to achieve mobility over most current computers; and if the design is flexible enough, there is a good chance that the software can be ported to future machines.

Case 2 : No Problem

There is a growing trend towards computer networks. Here we have a number of different computers linked together in some way, so that, in principle, a user has access to all computers in a given network. Within a network the problem of portability is bypassed. For if some software is available on one machine,

then it is not necessary to implement it on another within the network. The portability problem only arises if one network wants a program from another network. It is not impossible to imagine the ultimate network in which all computers are linked together, but it is a bit far-fetched at the present time. Until such a network is viable, the portability problem will remain a thorn in the sides of computer programmers.

1.1 On Producing Portable Compilers

The introduction of high-level languages eased considerably the burden of writing programs in assembly or machine languages. But it created a problem. For each language introduced, a compiler or interpreter had to be written for it. Today there is a proliferation of languages - from special purpose ones designed to solve one particular class of problems, e.g., LISP for list processing, or Picture Description Languages for writing graphics programs, to general purpose ones like PL/1 or ALGOL W. It is no surprise, therefore, that with the possible exception of Operating Systems, compilers are probably the most important components of software today. Consequently it is of immense benefit to most people to have portable compilers around.

1.1.1 The UNCOL Concept or Daydreams in Retrospect

As far back as the nineteen fifties (see footnote), people were postulating ways to ease the burden of transferring software from one computer to another. The most enticing to emerge was the UNCOL (UNiversal Computer Oriented Language) concept[12].

The problem it proposed to solve was the following: given m machines and l languages, it is required to construct, for each language L and each machine M, a compiler which translates from L to the order code of M. In the straightforward scheme of doing things, i.e., write a separate compiler for each pairing of languages and machines, l * m pieces of complicated software must be written. In the UNCOL scheme of things, only l generators (none of which is any more difficult to write than any of the above compilers) and m translators (all of which would be much simpler to write than any of the above) need to be written. All generators would be written in UNCOL. Each would accept a program PL written in some language L, producing as output the UNCOL version of PL. Each translator translates from UNCOL to the order code of the machine M for which it was written. A translator can be written in the assembly language of M or in any other language available on M. Now a program written in L, say, can be compiled in two stages, L -> UNCOL and

---

Though the first published paper was in 1958, the concept was discovered by many independent persons since 1954. Some even claim that it might not be difficult to prove that "this was well-known to Babbage".

UNCOL -> machine language.

Thus we see that UNCOL reduces the requirement of l * m programs to writing l + m programs. In practice, l and m are much greater than 2, and hence, if the UNCOL concept could be realized, it would result in a marked savings in programming, debugging and documentation with the consequent sizable reduction in the cost of producing compilers.

Not only would it solve the l languages, m machines problem, but it can be adapted easily to accomodate a new language NEWL, or a new machine NEWM.

Notation : {A,B -> C} represents a program written in A, which accepts a program written in B and translates it to an equivalent program in the language C.

In the case of NEWL, all that is needed is A, {UNCOL, NEWL -> UNCOL}. (Presumably this would be provided by the formulator of NEWL, but it doesn't have to be). To implement NEWL on a machine OLDM, A is input to B, {OLDM-ML,UNCOL -> OLDM-ML} where OLDM-ML stands for the machine language of OLDM. The output is C, {OLDM-ML, NEWL -> UNCOL}. A program written in NEWL can then be compiled by inputting to C, and the output submitted to B.

In the case of NEWM , any existing machine OLDM using the UNCOL system may be used. All that is needed is a translator D, {UNCOL, UNCOL -> NEWM-ML}. D is input to {OLDM-ML, UNCOL -> OLDM-ML} to produce E, {OLDM-ML, UNCOL -> NEWM-ML}. D is then input to E to produce {NEWM-ML, UNCOL -> NEWM-ML}. The generators {UNCOL, Li -> UNCOL} can now be used to provide

{NEWM-ML, Li -> UNCOL}.

From the above discussion, it seems that the UNCOL concept was a very attractive one. Why, then, was it never realized? It was not a logical impossibility since Steel[19] provided a demonstration of the logical existence of a language with the properties of UNCOL. He also demonstrates the desirability and economic feasibility of UNCOL, or some similar proceeding.

The UNCOL dream was shattered by more mundane considerations. The universally accepted language suitable as a target for all compilers was never agreed upon by a significant proportion of the computing profession. Also logical possibility is not practical possibility. It is clear that UNCOL would have to be suitable as both a source and a target language. Naturally, this leads to the requirement that it possess a number of somewhat conflicting properties. It must be

(a) simple and general because it has to be implemented on every computer;

(b) efficient, rich and expressive because of prime importance is the ease and efficiency with which the algorithm for accomplishing the task can be programmed in UNCOL.

(c) problem-oriented, because it will be used for writing the generators {UNCOL, Lj -> UNCOL}.

Summarily, it must possess the best of the two worlds of high- and low-level languages. The creation of an UNCOL with such properties is a monumental task in software engineering.

1.1.2 Koster's Solution : the Compiler Compiler Approach

Koster[9] attempts to solve the UNCOL problem by a separation of the various functions of UNCOL. He proposes the design of a special-purpose high-level language, UNCOLH, say, which would have only one function - only compilers would ever be written in it. It must be simple, efficient and problem-oriented towards the writing of compilers. It may very well lack many of the conventional facilities present in today's high-level languages. In fact, CDL, the Compiler Description Language he has been using, lacks features such as an assignment, array-access and arithmetic operations.

Once UNCOLH is agreed upon (is it possible?) the problem of realizing the l * m translators {Mj, Li -> Mj} i=1{1}l, j=1{1}m, can be solved by constructing the collections {UNCOLH, Li -> Mj} and {Mj, UNCOLH -> Mj}. A compiler for Li on Mj is effected by inputting {UNCOLH, Li -> Mj} to {Mj, UNCOLH -> Mj}. Output is {Mj, Li -> Mj}. A program written in Li is then compiled in the usual way.

At first glance the above seems to be worse than the straightforward solution of writing l * m compilers, for now we need to write l * m + m translators. The benefit is that if UNCOLH is well chosen, it should be a lot easier to write {UNCOLH, Li -> Mj} than {Mj, Li -> Mj}; the m translators {Mj, UNCOLH -> Mj} should not present too much difficulty if UNCOLH is simple.

Further advantages accrue to this new approach. Most

noteworthy is that UNCOLH is never used a target language, but only as a source language. Hence less stringent conditions are imposed on it than on UNCOL proper. This is so because it is no longer necessary to be able to realize the concepts of all other languages in UNCOLH. Also the translation from Li to Mj is a one stage process and consequently there is a better chance of achieving efficiency.

Koster further explains that the size of the problem can be decreased using macro techniques. He points out that for a given language Lk, the translators {UNCOLH, Lk -> Mj} j=1{1}m, will show a marked similarity since the lexical and syntactic phases will be mostly common. The target machine orientation of the translator will be manifest mainly in the code-generation phase. He proposes to exploit this fact by using macro techniques for this last phase.

For each language Lk, the m translators {UNCOLH, Lk -> Mj} can be replaced by the single translator {UNCOLH, Lk -> MIMCk} where MIMCk is a sequence of machine independent macro calls. MIMCk can be regarded as the language of a high-level abstract machine well suited for realizing Lk. To realize Lk on a machine M, all that is needed is a collection of macros whose replacement text is the order code of M. The problem has been reduced to the construction of m translators {Mj, UNCOLH -> Mj}, l translators {UNCOLH, Li -> MIMCi} and l * m collections of macros. This last is a relatively small job and hence there has been an impressive reduction in the size of the problem. But, as is usual in Computer Science, this saving has not been

achieved without a price. Macro expansion is a relatively slow business and since the code generation module and the macro processor forms part of the final production compiler, it may, as a result, be slow.

1.1.3 Self-Compiling Compilers

A self-compiling compiler is one which is written in the language it compiles, e.g., writing an ALGOL 68 compiler in ALGOL 68. In a paper written in 1960, Masterson[10] describes the method and how it was used to implement the NELIAC language. So the idea is not new, but there aren't many self-compiling compilers available at present. This stems from the fact that not many languages are suitable for coding their own compiler. Very few people would support the idea of writing a COBOL compiler in COBOL and, because of its limited character manipulation facilities, FORTRAN is not the best language for writing a FORTRAN compiler. When the language is suitable, however, self-compiling can be a valuable aid to portability.

Before describing the method, some observations are in order. Much of the effort required to transfer a compiler is in the rewriting of the code-generator for the new machine - the size of the machine-independent parts of the compiler is irrelevant. This suggests that portability does not suffer significantly if the language includes higher level features such as conditional commands and the scope rules of identifiers,

whereas portability is enhanced mainly by reducing the more fundamental facilities of the language such as the variety of storage and data types, the complexity of the calling mechanism for procedures and the number of primitive expression operators. Wilkes[27] indicates that a programming language can be divided into two parts :

(i) The Outer Syntax - consists of those language features that are independent of the data being manipulated, e.g., a conditional or GOTO statement.

(ii) The Inner Syntax - consists of those language features concerned with the manipulation and declaration of data.

In Wilkes' terminology, the portability of a language is hardly affected by the size of its outer syntax, whereas portability is inversely proportional to the size of the inner syntax. Having observed that the portability of a compiler revolves around its code generator, we now describe how self-compiling can aid portability.

The compiler, for a language L, say, must be designed so that there is as rigid an interface as possible between the machine-independent (MI) parts and the code-generation module which generates machine code from some internal machine-independent form. The MI parts are coded in L. To effect a first implementation, one must either translate the compiler to machine code by hand, or use some sort of bootstrapping technique, in much the same way as Wilkes did in implementing WISP[26].

Now that we have a compiler for L on some machine M1, how

much effort must we expend to transfer it to another machine, M2? Since we must produce code capable of being executed on M2, the first task is to alter the code generator so that it generates code for M2. A potential problem arises at this point. Since the modified code-generator will be used as a part of each of the two compilers on M1 and M2 (see below), it is best coded in a machine-independent way, and presumably, the best choice is to use the language L. In the following we assume that this is the case.

Letting M10 and M20 represent the order codes of M1 and M2 respectively, we have the following translators :

A, {M10, L -> M10}, the given compiler;

B, {L, L -> M20}, the source code of the compiler with the new code-generator.

It is required to construct {M20, L -> M20}, a compiler for L on M2.

Construction

B is input to A to produce C, {M10, L -> M20}, a hybrid compiler which runs on M1 but produces code for M2. B is then input to C to produce {M20, L -> M20}, as was required.

The above process is simple and straightforward in theory, but there are practical problems to contend with. One might be that M2 is too small to accomodate the compiler, in which case overlaying techniques may have to be used. Also one would need to adapt to new operating system conventions and the like. It is these apparently trivial details, not the overall

organization of a porting project, that defeat many well-planned projects. One would be well-advised to give them serious consideration in the early stages of planning to have reasonable assurance of success.

1.1.3.1 BCPL - A Language with a Self-Compiling Compiler

BCPL[16] was designed as a tool for compiler writing and systems programming. A major design goal was that it should be inherently portable. Consequently, it has a small number of primitive facilities. For instance, it has one data type, three storage types, a very simple procedure calling mechanism and few expression operators. Its one data type makes it reasonable to allocate space for items in the run-time stack by the machine-independent part of the compiler. The code-generator is thus simplified and the portability of the compiler is improved. Since BCPL was designed specially for software writing, it should be quite suitable for encoding its own compiler.

The BCPL compiler first translates a source program into an intermediate object code, OCODE, designed specifically for this purpose. This intermediate code is then passed to the code-generation module which converts it to the order code of the real machine.

A prospective implementor of BCPL on machine, M, say, is supplied with the compiler in one of two forms, OCODE or INTCODE. The latter is a low-level assembly language designed

by Richards[18].

For the OCODE form, one needs to write a translator, OCT, from OCODE to the order code of M. OCT is first used to translate the OCODE version of the compiler to M's machine language, and then acts as the code-generation phase of the new compiler. OCT can be written in any way convenient to the implementor. For instance, it can be written in any suitable language available on M, and if one has access to an existing BCPL compiler, it can be written in BCPL. Even a macro processor with the appropriate macro definitions can be used to generate machine code from OCODE, but this is not recommended since macro processing is slow and the compilation time of a BCPL program would be adversely affected. Once you have a working version of the compiler, you can modify an existing optimising code-generator to produce better code for M.

If one is interested in getting a working version of BCPL quickly, the INTCODE kit of the compiler is recommended. Because of its simplicity, it is much easier to write an INTCODE translator than an OCODE one. Richards[18] gives a good summary of the advantages of this approach. Among them are:

1. Less knowledge and less work is required to construct the first bootstrap.
2. INTCODE is easier to learn and is more convenient to write or modify than OCODE.
3. The text of the INTCODE form of the compiler is more compact than the corresponding OCODE text, an important factor when using cards or paper-tape.

1.2 On Producing Portable Software for Non-Numeric Applications

Why is it that many compilers can produce fairly efficient code for programs that are mainly numerical in nature, but fare rather badly when it comes to non-numerical applications? The reason is that the numerical facilities in a language can be efficiently mapped into the machine instructions available on most computers, whereas it is difficult to map efficiently the non-numerical facilities because machines vary widely in the way they manipulate characters and data structures, and no set of common primitive operations would be universally efficient. So given the situation that most compilers generate inefficient code for non-numerical algorithms, a prospective implementor must decide in what language he must code his software. In addition to the above, encoding software in a pre-defined high-level language has further drawbacks. The primitive operations and data types provided by the language may not coincide with those required by the software, and hence it becomes necessary to represent the needed primitives in terms of those offered by the high-level language. The representation might, perforce, be awkward, leading to large inefficiency factors. The inefficiency results because it is difficult, if not impossible, to generate efficient code for all possible combinations of primitive operations. Another bete noire is that one may have to pay the price for facilities provided by the language, but not used in implementing one's software. Examples might be recursion, run-time storage organization or

dynamic limits on looping statements.

After weighing the above considerations, it is clear that using a pre-defined high-level language is not the way to go, for, in that case, you would be tailoring your software to suit a language - not the best policy to adopt for production software. So the original question remains - what language should you use? The answer is: tailor a language to suit your software - design an Abstract Logical Language (ALL) whose primitives are just the primitives your software needs!

The task of writing software is simplified considerably if it is written in a language specifically tailored to it. Not only does an implementation become easier, but it is efficient compared to what can be obtained by using a pre-defined high-level language. The efficiency results from the fact that the primitives of the language are just those that are needed, and the language need contain no extraneous statements not used in the coding of the software, so you do not pay a price for features you do not use. We will call such a language an Abstract Logical Language.

1.2.1 What's in a Name?

Other terms are used in the literature to express the same concept. Waite[22] calls it an Abstract Machine, his philosophy being that the operations provided in an Abstract Logical Language (ALL) can be viewed as the order code of an imaginary machine that is independent of all real machines. Brown[3] uses

the term DLIMP (Descriptive Language Implemented by Macro Processor), which specifically states the method to be used to realize an ALL on a real machine. We prefer the term Abstract Logical Language because we are more interested in the language aspect than the machine aspect, and we do not wish to state a priori how to realise an ALL on a real machine, even though a macro processor would be used most of the time.

Why "logical"? Two reasons, the second being more of a pun than anything else. The first is that the language must be the "logical" choice for that particular piece of software and secondly, the language is to be used for encoding the "logic" of the software.

The mnemonic ALL may be misleading because an Abstract Logical Language, LLA, say, is designed to implement a particular piece of software. Hopefully LLA will be suitable for encoding other pieces of software (though this should not be so a priori), but it certainly will not be appropriate for encoding all the software one needs to implement.

1.2.2 Design Criteria for an ALL

An Abstract Logical Language is an embodiment of the basic operations required to perform a particular task. Choosing an ALL for writing a given piece of software is very much an engineering problem. An excellent account of the actual design of an all ALL is given by Waite[22]. In deciding upon an ALL,

three considerations (enumerated by Waite) are of prime importance.

1. The ease and efficiency with which the algorithm for accomplishing the task can be programmed in the ALL.
2. The ease and efficiency with which the ALL can be realised on machines available currently and in the forseeable future.
3. The tools at hand for the realisation in 2.

Let us elaborate on these points. Assume that our aim is to design an ALL for some software S. Suppose the basic data items of S consist of integers, stacks, trees and character strings. Then the ALL must provide data structures in which these concepts can be expressed, and operations for manipulating them. For instance, stacking operations such as pushing and popping should be included.

One should also bear in mind the macro processor which will be used to expand the ALL, and design the syntax of the language accordingly. For example, in Waite's case, he wanted his abstract language to be expandable by SIMCMP, a very simple macro processor. According to him, this was his prime consideration and any clear choice would be resolved in its favour. Since SIMCMP could only accept single character parameters, all operations in the FLUB language (his ALL) had single characters (letters or digits) as operands, e.g., PTR A = B + 7, which means add the PoinTeR fields of registers B

and 7 and store the result in the pointer field of register A; or TO 72 IF PTR Y = D, with the obvious meaning. The macro definitions for these two would begin with PTR * = * + *. and TO ** IF PTR * = *. respectively.

Unless one is forced to choose otherwise, the macro processor should be one which works in free-mode (as opposed to a processor like GPM[20] which requires a warning marker to herald a macro call), as this is a valuable aid to flexibility. In this case, it becomes easier to cater for unusual object machines or languages, since a free-mode processor does not fix in advance what is a macro and what is not, and hence which parts of an ALL are to be changed on a mapping and which parts are not. This feature would be very useful in dealing with such annoying, trivial details such as label formats. Suppose that an ALL allowed statement labels to be identifiers of up to eight characters, and, for some reason, some software written in this ALL were to be expanded into its FORTRAN version, where only numeric labels are permitted. A macro pre-pass could examine all label declarations and generate a unique integer corresponding to each declaration. For example, if START, PROCEED and END were labels in the program, output from a pre-pass could be macro definitions which define START as 1, PROCEED as 2 and END as 3. These definitions can then be combined with the rest of the mapping macros for the final expansion.

The design of the language should proceed concurrently with the design of S. Typically, you should choose what you consider

suitable instructions and then start the coding of S. Perhaps you will find that you have omitted several key operations and included some irrelevant ones. Some data types may also need changing. In this case, you should modify your instruction set, write some more code, and, again, perhaps institute some useful changes. By a process of gradual refinement, you should arrive at the primitives best suited for expressing your algorithm.

1.2.3 Implementation Using an ALL

Suppose it is required to implement some software S in a portable manner. This can only be done if S is inherently machine-independent (MI). If the machine-dependent (MD) parts of S (e.g., input/output, hashing algorithms or modules depending on the internal representation of characters) are of comparable size to the MI parts, then the following technique will not produce satisfactory results. This is only to be expected since major parts of S would have to be coded by hand for each new machine. Let us assume, therefore, that at most 10% of S is machine-dependent.

First the logic of S must be separated into its MI and MD parts. There is no clear-cut distinction between these two and the dividing line may contain a certain arbitrariness about it. Yet, subjective as it may be, most people will agree on what features are, and are not, machine-dependent.

After the above separation is made, an Abstract Logical

Language, SALL, say, must be designed for S.

Next the MI logic is coded in SALL. As noted above, the MD logic must be coded anew for each new machine. The interface between the two can consist of direct GOTO statements. There is another approach which is best illustrated by an example. Suppose that S requires a conversion from integer to character representation. This is a machine-dependent task. You can solve it by introducing the statement CONVIC PAR in SALL. On a particular machine, this can be mapped into a hand-coded routine to do the actual conversion.

Now suppose we have coded the MD parts for a machine M, and we have the MI parts coded in SALL. To effect an implementation on M, we need to map the MI parts into the order code of M. A macro processor can be used to achieve this. Presumably S was designed to be translatable by some macro processor(s) the designer had in mind. Suppose the macro processor (MP) is available on a machine MAC. The following must be accomplished:

1. Write macros acceptable to MP to translate SALL into M's Assembly language.
2. Using MP - on MAC, remember - expand the MI parts of S. The result is an Assembly language program for M.
3. Input the MD parts with the result of 2 to M's Assembler. Output is software S in M's machine code.

If MAC is not the same as M, this approach has many disadvantages. First one must have access to both M and N. If M is remote from N, and the macros are written incorrectly the

first time (almost a certainty), steps 2 and 3 would have to be repeated (perhaps more than once) and the commuting between MAC and M would make this, at least, inconvenient, slow and perhaps, costly. Secondly, MAC must be capable of writing Assembly language for M. If this is not the case, it becomes necessary to do annoying translations at each iteration of 2 and 3. However, these problems can be overcome by using a system similar to Waite's Mobile Programming System (see experiment 1).

1.2.4 Overheads, Disadvantages and Limitations

There is a large overhead involved in producing portable software using an Abstract Logical Language. If a suitable ALL does not exist, then the implementor must design one. One should not knead one's software in order to use an existing language, for, in that case, you would be tailoring your software to suit a language rather than vice versa, and the efficacy of the technique would be lost.

Once the language has been designed, it is very important to complement it with a macro test program. The absence of such a test program makes it necessary for a later implementor to be familiar with the inner workings of the program in order to debug his macros. This may be very undesirable. To quote from Waite[22],

> "I cannot overstress the importance of a comprehensive macro test program in the successful implementation of a machine independent system. Macro coding errors can be very subtle, requiring hours of debugging to trace them down if the

> machine-independent program is the only test case. A conservative estimate based on our experience is that a lack of a good test program will increase by a factor of five the time required to complete an implementation when the author of the system is available to debug the macros. When he is not available, the task is almost hopeless."

There is also a minor overhead involved in organising the logic of the software in a machine independent way. Of course, for a first implementation the program needs to be debugged, and finally the portability method must be well-documented.

Apart from the large initial overhead in using an ALL to produce portable software, there are other disadvantages. One is that any language which can be compiled by a macro processor must necessarily be set at a fairly low level. If set at too high a level, the macro definitions become much more difficult to write, and one begins to lose the portability that the use of a macro processor was originally meant to provide. Thus the technique is unsuitable for those applications requiring portability, but where a general high-level language would be more apt than any low-level ALL. Also, when using an ALL, the generated object code is less efficient than would be produced by hand-coding in assembly language, but is generally more efficient than that produced by using a pre-defined high-level language.

1.2.5 Examples

Perhaps the most ambitious use of an Abstract Logical Language has been in the implementation of SNOBOL4[7], a text processing language. The reference gives a detailed description of the project. The ALL used to implement SNOBOL4 is called SIL and contains 130 different types of statements. Apart from minor changes in character representations, etc., SIL can be mapped by any reasonably powerful macro-assembler. However, this does not preclude the use of free-mode processors such as STAGE2 or ML/1.

The implementation of WISP[26], a simple general list-processing language, is another good example of the use of the above technique. But WISP is unique in that the ALL used to implement it is WISP itself. In fact, the experiment combines the techniques of using an ALL with that of bootstrapping a compiler written in the language it compiles.

2.0 Part 2 - PRACTICE

The rest of this paper deals with two experiments in portability, using the concept of an Abstract Logical Language.

2.1 Experiment 1

The first experiment concerns the implementation of STAGE2, using a full bootstrap, i. e., we assume no access to a running version of STAGE2. A very simple processor, SIMCMP[14], is implemented by hand, and then used to obtain a more complex one, STAGE2. STAGE2 was designed by William Waite. It is similar to an earlier macro processor, LIMP[21], also designed by Waite. STAGE2 provides all the features normally associated with a general macro processor[11]. Its input recognition procedure is language-independent, employing the LIMP type of scanning mechanism to recognize macro calls and isolate parameters. With each macro is associated a template. A template consists of fixed strings with gaps between them for the arguments. To recognize a macro call, each source line is compared with the template. If several templates match the source line, a notion of "closest fit" is used to resolve ambiguities. For example, the source line

PTR A = B + 0

would match the templates

PTR ' = ' + ' and

PTR ' = ' + 0.

In this case STAGE2 assumes that the source line matches the second template.

Conditional expansion, iteration and the like are provided by "processor functions" rather than by explicit program structure. One can also perform different parameter conversions. For a complete description of STAGE2, see Waite[23].

STAGE2 was designed as a tool for creating machine-independent software. To be effective, STAGE2 itself must be highly mobile. Hence the system in which it is embedded must require minimal support from any computer on which it is to be implemented. As it turns out, the only thing required from the target machine is an Assembler (which need not have any macro capability). With his Mobile Programming System (MPS), Waite solves the commuting and translating problems mentioned on page 19. His method enables all the work to be done on the object machine.

The base of the MPS is a very simple macro-processor, SIMCMP[14]. It is available as an 89-statement program written in ASA Fortran. If Fortran is available on the object machine, implementation of SIMCMP is trivial. If not, the logic of SIMCMP is so simple that it can be translated by hand into assembly language in about 4-8 hours. Once SIMCMP is running, it is used to implement STAGE2, a much more powerful macro-processor. It should be emphasized that SIMCMP has one job - to translate FLUB, the Abstract Logical Language in which STAGE2 is written. Indeed, FLUB was designed to be translatable by SIMCMP. When STAGE2 is running, it can be used to generate a

more optimised version of itself. This is possible because the mapping macros can now be written, utilising the conditional facilities of STAGE2, to recognize special cases such as PTR A = 0 + 0, or recognizing when use can be made of special instructions such as BCTR REG,0.

The FLUB Language

FLUB (First Language Under Bootstrap) was designed specifically for the task of constructing a machine-independent macro-processor. A complete description of the characteristics and the rationale behind the design of the FLUB language is given by Waite[22]. The FLUB program which implements STAGE2 must conform to the restrictions imposed by SIMCMP, the only tool available for converting it to assembly language. Consequently, only single character parameters (and hence arguments) are allowed.

The basic data type of the language is the FLUB word, which consists of three fields -

1. the FLG field, which must be able to hold the integers 0 to 3.
2. the VAL field, which must be able to hold the largest character or the longest string length.
3. the PTR field, which must be able to hold the largest machine address.

The operations on these fields are as follows

FLG - test and/or set;

VAL - addition and subtraction; tests for equality of VAL

fields;

PTR - addition, subtraction, multiplication and division; tests for relative magnitude.

In any mapping of FLUB to a real computer, the way the FLUB word is mapped is of prime importance. One would be lucky to find a computer whose words are partitioned into three fields of the types constituting a FLUB word. Hence one-to-one mappings from FLUB to a real computer are very rare. Otherwise, there are two possible approaches -

1. pack the fields of the FLUB word into one or more words of the target computer;
2. use one target computer word for each field.

The first choice will minimize the space required to store information, but will (presumably) introduce large overheads to pack and unpack fields. The second choice will be wasteful of space, but one should expect faster execution.

To resolve the dilemma, FLUB has 36 "registers" named by the characters A-Z and 0-9. Each register has the same format as a FLUB word. Hence, in an implementation, if one target computer word is allocated for each register field, only 108 words will be so used. Since almost all FLUB operations take place on the registers, memory can be packed without introducing excessive overheads due to packing and unpacking.

2.1.1 Objectives

In order to survey the effects of different methods of expansion of the FLUB language, four versions of STAGE2 were implemented. The main features to be studied were

(a) the effect on speed and size when (i) the FLUB registers and memory are packed and (ii) the registers are kept in unpacked form but memory is packed. In the unpacked form, a FLUB register was represented using three 370 machine words - one for each of the FLG, VAL and PTR fields. In the packed form, one 370 machine word was used for the PTR field and one for the FLG and VAL fields - each occupying a half-word.

(b) the effect on speed and size when special cases are optimised and use is made of special instructions. For example, in a straightforward expansion of VAL C = A + 0, the VAL field of A is added to the VAL field of 0 and the result stored in the VAL field of C. In an optimised version, the VAL field of A is simply transferred to the VAL field of C. As an example of using a special instruction, consider VAL W = X - 1; this can be recognised as a special case of subtraction - decrementing by one - hence use can be made of the BCTR GR,0 instruction to accomplish the subtraction.

2.1.2 Implementations

Let us denote the FLUB version of the machine-independent part of STAGE2 as STAGE2.FLUB.

Version 1 was a straightforward expansion of STAGE2.FLUB by SIMCMP, with no attempt at optimisation. No trouble was taken to recognise special cases. (It is possible to do so, albeit awkwardly, with SIMCMP, by writing macros with specialised templates, e.g. PTR ' = ' + 0). The macros were written in the simplest, most direct way acceptable to SIMCMP. The only object was to get a running version of STAGE2. The FLUB registers and FLUB memory were held in packed form. The macros for Version 1 are given in Appendix A.

Version 2 was similar to Version 1 in that no optimisation was attempted. However, the FLUB registers were held unpacked, while the FLUB memory was packed. The macros for Version 2 are given in Appendix B.

One may wonder why no version was implemented with both the registers and memory being unpacked. While such an implementation should be faster (no packing and unpacking has to be done), it is very wasteful of space, and would be impractical for most purposes.

Version 3 was produced using the running versions of STAGE2. The macros were written as STAGE2 macros, utilising the greater power of this macro-processor. Specifically, extensive use was made of the conditional and unconditional branching facilities of STAGE2 to recognize special cases and hence

produce a more optimised mapping than either of versions 1 or 2. Further optimisation was possible by holding the frequently used constants, 0, 1 and 8 (the number of 370 address units - bytes - making up a FLUB word in memory) in general registers. When a macro call using one of these constants was encountered, the register-register instruction was generated instead of the register-memory equivalent. Since the former instructions are faster and occupy only half a word, we should expect some improvement in size and efficiency. As in Version 1, the FLUB registers and memory were held in packed form. The macros for Version 3 are given in Appendix C.

Version 4 was similar to Version 3 in that special-cases optimisation was performed. However, the FLUB registers were held unpacked, while the FLUB memory was packed. The macros for Version 4 are given in Appendix D.

As an indication of the relative size performance of these versions, the following figures show the number of IBM 370 Assembly Language statements generated for the two macro test programs FLT1 and FLT2. FLT1 and FLT2 consists of 735 and 536 FLUB statements respectively.

	Version 1	Version 2	Version 3	Version 4
FLT1	1851	1851	1440	1440
FLT2	1288	1328	1012	1052

As expected, the optimised versions produced less code than the unoptimised ones. But it can be easily seen that the

"unpacked registers, packed memory" versions generated more statements than their "packed registers, packed memory" counterparts. This is readily explained by an examination of the STO and GET macros for the different versions. V1 and V2 produce the same number of statements for all macros except STO and GET. (Note that this applies only to the number of statements, not the statements themselves). Hence the difference in the amount of code produced by V1 and V2 is dependent on the number of STO and GET statements in the source program. Since FLT1 contains none of these statements, the number of assembly statements produced is the same for V1 and V2. Similar remarks apply to V3 and V4.

2.1.3 Notes on Mapping

A problem arose with the mapping of the LOC '' statement. In effect a LOC '' statement labels the next source statement. The most natural way to map LOC 98, say, is to map it into

```
LL98    ANOP
```

where ANOP is the assembler pseudo-op meaning "no operation". Unfortunately, the name field of ANOP must be a sequence symbol (one starting with a period). But if we translate LOC98 to

```
.LL98    ANOP
```

then we cannot branch to it by BZ .LL98, say, since the latter is an illegal 370 Assembler statement. The problem can be solved by making an entry in the STAGE2 memory whenever a

LOC '' statement is encountered. Then whenever any other macro is to be expanded a check is made to see if a LOC '' statement was just processed. If so, the appropriate label is attached to the first statement in the expansion of the current macro, and the memory is cleared. This would work fine, except that extra statements would have to be included in every macro definition to test whether or not a label should be attached to the first statement generated by the macro. This is not very appealing, because of the extra work in writing the macros, plus the more important fact that macro processing would be slowed considerably. A cruder, but much simpler, solution is to generate a "useless" instruction

```
LL98    LTR   TEM,TEM     USED AS A NOOP
```

The above instruction, which just clears a temporary working register, will not affect the flow of the program. Its only purpose is to provide an instruction to which a label can be attached. Of course, now for every label in the source program we have an extra instruction. Hence whenever we branch to a label, we execute one more instruction than we need to. However, once the macros are debugged, a simple procedure can be written to examine the assembler listing, deleting the "useless" instructions and attaching their labels to the following statements. The following procedure will do the job:

```
        BALR  12,0
        USING *,12
        ENTER
LOOP    SCARDS  LINE1,(4),EXIT=EOF    READ A LINE
        CLC     LINE1(2),=C'LL'       DOES IT HAVE AN 'LL' LABEL?
        BNE     OUTPUT                IF NOT,WRITE THE LINE
        SCARDS  LINE2,(4),EXIT=EOF    ELSE READ NEXT LINE
```

```
**                                          IF ENDFILE, GOTO EOF
        MVC     LINE2(4),LINE1              ATTACH LABEL TO NEW LINE
        SPUNCH  LINE2,(4)                   WRITE IT OUT
        B       LOOP
OUTPUT  SPUNCH  LINE1,(4)                   WRITE LINE WITHOUT
**                                          'LL' LABEL
        B       LOOP
EOF     EXIT
LINE1   DS   80C
LINE2   DS   80C
        END
```

2.1.4 Results of Experiment 1

Let us divide STAGE2 into two parts - the invariant and variant. The invariant part consists of those pieces common to all the versions. The variant part consists of the assembly code equivalent to STAGE2.FLUB. It is variant because the number of assembly language statements generated depends on the macro-processor and the macro definitions used to expand STAGE2.FLUB.

2.1.4.1 Space Measurements

The breakdown of the core storage occupied by the invariant part of STAGE2 follows. The figures indicate the numbers of words on the IBM 370/168.

416 - generalised I/O package, including the line buffer and four channel buffers.

270 - initialization code plus constant definitions.

16004 - data space for FLUB registers and memory.

Total Size of invariant part - 16690 words.

Notation: Let Vi represent Version i, i=1,4.

The statistics for the variant part of STAGE2 are as follows. The figures indicate the number of assembly language statements (generated from 983 FLUB statements) after the "useless" instructions have been deleted. Since the same number is deleted for each version, the comparison is not affected.

V1	V2	V3	V4
2330	3048	2052	2770

When assembled, the number of words of memory occupied by the variant part for the different versions were

V1	V2	V3	V4
2284	3004	1988	2710

At a glance, it is evident that the "unpacked registers, packed memory" versions are significantly larger than their "unpacked registers, packed memory" counterparts. Specifically, V2 is 32% longer than V1 and V4 is 36% longer than V3. As expected, the unoptimised versions are longer than the optimised ones; V1 is 15% longer than V3 and V2 is 11% longer than V4. We conclude that the optimised version with packed registers provide the most compact implementation.

But there are other considerations which can influence one's decisions; consider the following table, where STG2.V refers to the size of the invariant part and % refers to the percentage of the total size that STG2.V occupies.

	V1	V2	V3	V4
STG2.V	2284	3004	1988	2710
TOTAL	18974	19694	18678	19400
%	11.5	15.3	10.6	14.0

Even in the most unoptimised case, V2, STG2.V is only 15.3% of the total size. Hence any real saving in space would come from a reduction in the work space allocated to STAGE2, if that is feasible. This is highlighted by the fact that though the variant part of V3 is 15% smaller than that of V1, say, the total size of V3 is 98% that of V1.

2.1.4.2 Time Measurements

Next we did some time measurements.

Notation: Let Vi represent Version i, i=1,4.

Each version was given four tasks to perform, and each task was to be done four times. To eliminate the differences in times given by the operating system when the same program is run at different times, corresponding times were obtained by running the different versions at approximately the same time via a

batch job. To further eliminate differences due to operating system overheads, the time to perform one of the tasks was taken as the average of the four times obtained for that task.

The tasks were as follows:

1. Executing the standard test data for testing a STAGE2 implementation.
2. Translating LOWLTEST (see experiment 2) to 370 Assembler. LOWLTEST consists of 775 source statements and each version produced 1243 assembler statements.
3. Translating ALGEBRA (see experiment 2) to 370 Assembler. ALGEBRA consists of 1097 source statements and each version produced 1450 assembler statements.
4. Translating STAGE2 to 370 Assembler using the macros for Version 3. STAGE2 consists of 983 source statements and each version produced 2146 assembler statements.

The results are shown in Table 1. The times shown are in seconds of CPU time. A is task 1, B is task 2, C is task 3 and D is task 4.

Comments on Table 1

Not only do the "unpacked registers, packed memory" versions occupy more space than the "packed registers, packed memory" versions, they are also slower, as evidenced by comparisons between V1 and V2, and V3 and V4. This is somewhat

surprising since one would normally expect the unpacked versions to be faster. However, for the IBM 370/168, this apparent contradiction can be explained. In the packed form, even though the registers are compressed to fit into two words, no "unpacking" (in the sense of loading and shifting bits around) had to be done. This is so because one can use the half-word instructions on the 370 to access the FLG and VAL fields. Also, the 370 has two useful instructions, Load Double and Store Double, which one can use to manipulate two words at a time. These instructions enabled the STO and GET macros (the major factors in determining size) to be implemented efficiently, since in each case, all that is involved is a transfer of two full-words from one part of the 370 memory to another.

When the registers are unpacked (occupying three full-words) and memory is packed (occupying two full-words), one cannot implement STO and GET by a simple transfer of a block of information from one part of memory to another. So the above contradiction is explained by another contradiction - that one has to unpack fields when the registers are unpacked but not when they are packed.

An interesting case is the comparison between V1 and V4. In all cases, V1 is significantly faster than V4. Hence we can conclude that the local optimisation obtained is more than nullified by the extra overheads due to the unpacking necessary to transfer information between the FLUB memory and the FLUB registers.

Comparison between V3 and V1 shows that local optimisation

produced slightly better code. The improvement is not drastic, however. This is explained by the fact that in the tasks performed, STAGE2 spent most of its time in the I/O routines, and in the transfer of information between the FLUB memory and registers. The optimisation should be more noticeable if the macro definitions involved more numerical computation.

	RUN	V1	V2	V3	V4
A	1	.41	.42	.40	.43
	2	.44	.49	.45	.47
	3	.42	.44	.38	.43
	4	.43	.45	.39	.44
	Ave	.42	.45	.40	.44
B	1	3.92	4.22	3.71	4.23
	2	3.86	4.40	3.85	4.59
	3	3.90	4.25	3.65	4.29
	4	4.00	4.45	3.74	4.40
	Ave	3.92	4.33	3.74	4.40
C	1	4.24	4.73	4.16	4.79
	2	4.47	4.83	4.35	4.98
	3	4.30	4.60	4.10	4.70
	4	4.42	4.75	4.16	4.69
	Ave	4.36	4.73	4.19	4.79
D	1	4.23	4.49	3.99	4.58
	2	4.03	4.45	4.03	4.87
	3	4.23	4.70	4.00	4.67
	4	4.19	4.50	3.99	4.50
	Ave	4.17	4.53	4.00	4.65

Table 1

2.2 Experiment 2

After STAGE2 was successfully implemented, it was used to effect the implementation of a symbolic logic package, ALGEBRA. ALGEBRA is a program designed for students and research workers in Boolean algebra and other kinds of logic. Its spectrum of uses range from frivolous applications to game-playing to the analysis of switching networks. It is best used in conversational mode. A complete description of ALGEBRA is given by Brown and Lowe[5] and Brown[4]. We will outline the main features below.

2.2.1 Outline of ALGEBRA

Each line of data fed to ALGEBRA must be a statement. There are four types of statements: the OPERATOR statement, the TABLE statement, the TRY statement and the NULL statement. A comment may be appended to any statement by preceding it with a semi-colon, e. g.

```
OPERATOR >    ; THIS IS THE IMPLIES OPERATOR
```

A comment can occupy a whole line if the line starts with a semi-colon. Only the first two letters of a statement are used to determine its type, e. g. OPXXX + is the same as OP + is the same as OPERATOR +.

Definitions

1. A SYMBOL is used to represent the name of a value, operator or variable. It must be either
   (a) a NAME SYMBOL which is an arbitrarily long sequence of one or more letters or digits, or
   (b) a PUNCTUATION SYMBOL which is a single character that is NOT any of a letter, digit, comma, tab, space, semi-colon, left parenthesis, right parenthesis or the newline character.

Examples of valid symbols are A, 7, ATOM45, AVERYLONGSYMBOLNAME, +, -, |.

Examples of invalid symbols are SPA CE (contains a blank), ;, -> (not a single character).

2. A VARIABLE is any symbol that is not the name of an existing operator or value.
3. An EXPRESSION is a series of operators with values or variables as operands.

During the evaluation of an expression, operators of highest precedence are performed first. Subject to this, operators are evaluated from left to right. Parentheses can be used to override precedence.

The following sample terminal session should give one an idea of how the system operates. The statements prefixed with ">" are typed by the program. The others are typed by the user.

```
# RUN ALGEBRA.O T=1
```

```
 # EXECUTION BEGINS
>  ALGEBRA - VERSION 1    ; THESE TWO LINES ARE PRINTED WHEN AL-
                          ; GEBRA IS FIRST ENTERED. THE SYSTEM IS
>  PLEASE ENTER VALUES    ; NOW EXPECTING THE SYMBOLS TO BE USED
   TRUE FAL(E             ; AS VALUE NAMES. IF AN INVALID SYMBOL
>  ERROR - ILLEGAL SYMBOL; IS USED THE REQUEST IS REPEATED. ONCE
                          ; ESTABLISHED, VALUE NAMES CANNOT BE
>  PLEASE ENTER VALUES    ; CHANGED DURING THE COURSE OF A RUN -
   TRUE FALSE             ; YOU MUST START AGAIN FROM SCRATCH IF
                          ; YOU WISH TO USE DIFFERENT VALUE NAMES
                          ; ANY NUMBER OF VALUES MAY BE ENTERED.
                          ; AFTER THE VALUES HAVE BEEN DECLARED,
                          ; INPUT STATEMENTS ARE EXECUTED.
;
;
   OPERATOR ¬ ; THIS IS THE "NOT" OPERATOR
              ; ANY SYMBOL THAT IS NOT THE NAME OF A VALUE MAY
              ; BE USED AS THE NAME OF AN OPERATOR. OPERATORS
              ; CAN BE DEFINED AT ANY POINT OF A RUN, NOT
              ; NECESSARILY AT THE BEGINNING. OPERATORS CAN
              ; BE REDEFINED, THE NEW DEFINITION OVERRIDING THE
              ; OLD. EACH OP STATEMENT IS FOLLOWED BY A SERIES
              ; OF QUESTIONS.
;
;
>  UNARY OR BINARY ? ; STATE THE TYPE OF OPERATOR
   UMARY
>  **INPUT ERROR. MUST BE "UNARY" OR "BINARY"
                     ; OR ANY SYMBOL BEGINNING WITH UN OR BI -
>  UNARY OR BINARY ? ; ONLY THE FIRST TWO LETTERS ARE USED.
   UNARY             ; THE NOT OPERATOR IS UNARY
                     ;
>  PRECEDENCE ?      ; STATE THE PRECEDENCE OF THE OPERATOR
   1000
>  **INPUT ERROR. MUST BE BETWEEN 0 AND 999
>  PRECEDENCE ?      ; THE PRECEDENCE OF AN OPERATOR IS IMPORT-
   -1                ; ANT ONLY IN RELATION TO THE PRECEDENCE
                     ; OF OTHER OPERATORS. FOR EXAMPLE, A PRE-
                     ; CEDENCE OF 10 IS THE SAME AS THAT OF 900
                     ; IF ALL OTHER OPERATORS HAVE PRECEDENCE
                     ; LESS THAN 10.
>  **INPUT ERROR. MUST BE BETWEEN 0 AND 999
>  PRECEDENCE ?
   99
                     ;
>  ¬ TRUE?           ; NEXT THE USER IS ASKED TO DEFINE THE
   FALS              ; VALUE OF THE OPERATOR FOR ALL POSSIBLE
                     ; VALUES OF ITS OPERANDS.
>  **INPUT ERROR. MUST BE ONE OF THE DEFINED VALUES
>  ¬ TRUE?
   FALSE
>  ¬ FALSE?
   TRUD
>  **INPUT ERROR. MUST BE ONE OF THE DEFINED VALUES
```

```
>  ¬ FALSE?
   TRUE
   OP &  ; DEFINE THE "AND" OPERATOR
>  UNARY OR BINARY ?
   BUNARY
>  **INPUT ERROR. MUST BE "UNARY" OR "BINARY"
>  UNARY OR BINARY ?
   BINARY
>  PRECEDENCE ?
   49
>  TRUE & TRUE?
   TRUE
>  TRUE & FALSE?
   FALSE
>  FALSE & TRUE?
   FALSE
>  FALSE & FALSE?
   FALSE
   OP |  ; THE "OR" OPERATOR
>  UNARY OR BINARY ?
   BIN
>  PRECEDENCE ?
   9
>  TRUE | TRUE?
   TRUE
>  TRUE | FALSE?
   TRUE
>  FALSE | TRUE?
   TRUE
>  FALSE | FALSE?
   FALSE
   OP >  ; IMPLY OPERATOR
>  UNARY OR BINARY ?
   BIN
>  PRECEDENCE ?
   1
>  TRUE > TRUE?
   TRUE
>  TRUE > FALSE?
   FALSE
>  FALSE > TRUE?
   TRUE
>  FALSE > FALSE?
   TRUE
   ; END OF OPERATOR DEFINITIONS. IF WE NEED MORE WE CAN ALWAYS
   ; ADD THEM LATER.

   TABLE ¬X > (Y & ¬Z)  ; THE TABLE STATEMENT IS A REQUEST TO
                        ; PRINT THE VALUES OF THE EXPRESSION FOR
                        ; ALL POSSIBLE COMBINATIONS OF VALUES OF
                        ; ITS VARIABLES. ANY NUMBER OF VARIABLES
                        ; ARE PERMITTED. IF NECESSARY, VARIABLES
                        ; WILL BE TRUNCATED TO SIX CHARACTERS IN
                        ; THE TABLE HEADING.
```

```
>  X       Y       Z       : VALUE
>  ------  ------  ------  :------
>  TRUE    TRUE    TRUE    :TRUE    ; THIS TABLE IS PRINTED
>  TRUE    TRUE    FALSE   :TRUE    ; BY ALGEBRA.
>  TRUE    FALSE   TRUE    :TRUE
>  TRUE    FALSE   FALSE   :TRUE
>  FALSE   TRUE    TRUE    :FALSE
>  FALSE   TRUE    FALSE   :TRUE
>  FALSE   FALSE   TRUE    :FALSE
>  FALSE   FALSE   FALSE   :FALSE
>
   TRY (X & Y) > (¬X | Y); THE TRY STATEMENT IS AN ABBREVIATED
                          ; FORM OF THE TABLE STATEMENT. IF THE
                          ; EXPRESSION HAS THE SAME VALUE V, SAY,
                          ; FOR  ALL  POSSIBLE  COMBINATIONS  OF
                          ; VALUES OF ITS VARIABLES, THE RESULT V
                          ; IS PRINTED. OTHERWISE,
                     ; "VALUE DEPENDS ON VALUE(S) OF VARIABLE(S)"
                          ; IS PRINTED.
>  TRUE

   TRY (A | B) > B
>  VALUE DEPENDS ON VALUE(S) OF VARIABLE(S)


; Let us use the above definitions to solve the following pro-
; blem.  we know the following three facts:
;
; 1.  If Jones did not meet Smith last night, then either
;     Smith was the murderer or Jones is lying.
;
; 2.  If Smith was not the murderer, then Jones did not meet
;     Smith last night and the murder took place after midnight
;
; 3.  If the murder took place after midnight, then either
;     Smith was the murderer or Jones is lying.
;
; The question is: was Smith the murderer?
;
; We can answer the question as follows:
;
; Let p represent "Jones met Smith last night"
; Let q represent "Smith was the murderer"
; Let r represent "Jones is lying"
; Let s represent "The murder took place after midnight"
;
; Fact 1 can be expressed as  ¬p > (q | r)
; Fact 2 can be expressed as  ¬q > (¬p & s)
; Fact 3 can be expressed as  s > (q | r)
;
; The above question can now be posed thus:
; Is it always the case that
;  (¬p > (q | r))  &  (¬q > (¬p & s))  &  (s > (q | r)) > q
```

```
; is true, regardless of the truth values of p, q, r and s?
; The answer is given by the following statement:

   TRY (¬P > (Q | R)) & (¬Q > (¬P & S)) & (S > (Q | R)) > Q
>  VALUE DEPENDS ON VALUES OF VARIABLES
;
; Hence the conjunction of the above facts do not imply
; that Smith was the murderer. Let us examine the table.
;

   TABLE ((¬P > (Q | R)) & (¬Q > (¬P & S)) & (S > (Q | R))) > Q

>  P       Q       R       S       : VALUE
>  ------  ------  ------  ------  :------
>  TRUE    TRUE    TRUE    TRUE    :TRUE
>  TRUE    TRUE    TRUE    FALSE   :TRUE
>  TRUE    TRUE    FALSE   TRUE    :TRUE
>  TRUE    TRUE    FALSE   FALSE   :TRUE
>  TRUE    FALSE   TRUE    TRUE    :TRUE
>  TRUE    FALSE   TRUE    FALSE   :TRUE
>  TRUE    FALSE   FALSE   TRUE    :TRUE
>  TRUE    FALSE   FALSE   FALSE   :TRUE
>  FALSE   TRUE    TRUE    TRUE    :TRUE
>  FALSE   TRUE    TRUE    FALSE   :TRUE
>  FALSE   TRUE    FALSE   TRUE    :TRUE
>  FALSE   TRUE    FALSE   FALSE   :TRUE
>  FALSE   FALSE   TRUE    TRUE    :FALSE
>  FALSE   FALSE   TRUE    FALSE   :TRUE
>  FALSE   FALSE   FALSE   TRUE    :TRUE
   FALSE   FALSE   FALSE   FALSE   :TRUE
>
; The value "FALSE" appears in only one line in the last
; column - a narrow escape indeed for Smith!
```

The machine-independent part of ALGEBRA was originally coded in L, a high-level Abstract Logical Language (ALL) used by Brown for implementing his ML/1 macro processor[2]. The L version has been mapped into LOWL, an ALL set at a much lower level than L. Brown[4] gives a complete description of LOWL.

2.2.2 Outline of LOWL

LOWL statements have a format similar to assembly language. They are written one to a line, and each consists of an optional label followed by a mnemonic operation code followed by a number of arguments separated by commas. Each instruction has a fixed number of arguments - if a certain argument is not applicable in a given case, it is indicated by an X. The argument list is null for some instructions. Arguments that are literal character strings are enclosed in quotes. The following are some sample LOWL statements:

```
      NB     'THESE ARE SAMPLE LOWL STATEMENTS'
      DCL    HOLD
OK    LAL    1
      STV    HOLD,X
      ALIGN
      GOSUB  TEST,X
      CAL    0
      GOEQ   OK
INCR  BUMP     HOLD,2
      BSTK
```

LOWL consists of a <u>kernel</u>, the statements of which are used by all software encoded in it, plus extensions oriented towards each individual piece of software. For example, ALGEBRA needs two extension statements, the RMESS and the QMESS statements, in addition to the 60 statements in the kernel.

Almost all statements in LOWL involve a storage address, and all assignments, comparisons and arithmetic operations are done via three registers, A, B and C.

A is the numerical accumulator.

B is the index register.

C is the character register.

All statements that use the A or B registers have numerical operands, and those that use the C register have a single character as operand. The statements in the kernel can be classified roughly as follows:

a. statements for declaring variables;

b. data definition statements for defining table items;

c. load statements - load a value into one of the registers A, B or C;

d. store statements;

e. arithmetic and logical statements - these include addition, subtraction, multiplication and ANDing;

f. compare and branching statements;

g. statements for defining and branching into and out of subroutines;

h. stacking and block moving statements;

i. I/O statements;

j. comment and layout statements.

2.2.3 ALGEBRA Coding

The coding is divided into two parts - the machine-independent (MI) part, which is coded in LOWL and must be mapped into the assembly language of the object machine, and the machine-dependent (MD) part which needs to be coded by hand

for each implementation.

The MD-logic consisted of some initialization code plus a set of short subroutines. The initialization code performs the following tasks:

1. reserves storage for the stacks used by LOWL, and sets pointers to point to the start and immediately beyond the end of the reserved area;
2. initializes the subroutine stack;
3. sets up the handling of interrupts during a run of ALGEBRA;
4. initializes the output buffer;
5. prints an introductory message;
6. branches to the label BEGIN in the MI-logic.

The subroutines required were the following:

MDLINE - reads a line of input and sets a pointer to point at the first character of the buffer; checks if end-of-file was detected.

MDQUIT - closes all I/O (outputs an incomplete line in the output buffer), and returns to the Operating System (not to its point of call).

MDERCH, MDRCH, MDQCH - outputs the character in the C register on the message, results and questions stream, respectively - in this case, the file attached to the MTS logical I/O unit SPRINT.

2.2.4 Mapping the MI-logic

The major part of the project was concerned with the mapping of the LOWL version of ALGEBRA into the assembly language of the IBM 370/168, using STAGE2 as the mapping tool. A number of problems arose, mainly due to the restrictions imposed by the 370 Assembler on the format of statements it would accept, and the argument and template matching conventions of STAGE2.

2.2.4.1 Source Format Problems

In the following ◻ represents a blank.

The most troublesome problems arose because the Assembler does not allow blanks in the argument list of an operation code, e.g.

```
L    GR8,◻HOLD
```

is invalid. Typically, these problems arose because the LOWL listing from which we worked was in free format, in the sense that an arbitrary number (but at least one) of blanks separated the various fields of an instruction, with zero or more blanks at the end of the instruction. For example, consider the following macro definition for the LAV (Load A from a Variable) statement:

```
'□LAV□□□','|
%10 L     #GRA,%20%F1$
$
```

If the source statement was

□□□□□LAV□□□FFPT,X

the generated instruction would be

□□□□□L   #GRA,FFPT

which is all right, but if the source statement was

□□□□□LAV□□□□FFPT,X

the generated instruction would be

□□□□□L   #GRA,□FFPT

which is an invalid assembler statement. If the template was written with 4 blanks between the V and the second parameter flag, then the generated instruction for the second statement would by valid, but now the first would not match the template.

The only way to solve the problem was to write a procedure which would accept the free-format listing and produce a fixed-format listing, with a known fixed number of blanks between the various fields. The following procedure would accomplish this:

```
INLENGT   EQU   4              SET UP REGISTERS
CHAR      EQU   5
INPTR     EQU   6
OUTPTR    EQU   7
          BALR   12,0          SET UP BASE REG
          USING  *,12
          ENTER
NEXTLN    SCARDS  INBUFF,(INLENGT),EXIT=EOF  READ A LINE
**                                  GOTO EOF IF END-FILE
          MVC   OUTBUFF(1),BLANK    BLANK THE OUTPUT
          MVC   OUTBUFF+1(79),OUTBUFF  BUFFER.
          CLI   INBUFF,C'*'         IS IT A COMMENT?
          BE    OUTCOM              YES, WRITE IT OUT
          SR    INPTR,INPTR         INIT BUFFER POINTERS
```

```
         LA     OUTPTR,5
GETLAB   IC     CHAR,INBUFF(INPTR)      GET THE LABEL, IF ANY
         CLM    CHAR,1,BLANK            A BLANK ENDS THE
         BE     SKPBLNK                 LABEL FIELD
         STC    CHAR,OUTBUFF(INPTR)     MOVE NON-BLANK TO OUTBUFF
         LA     INPTR,1(INPTR)          INCREASE IN-BUFFER PTR
         B      GETLAB
SKPBLNK  LA     INPTR,1(INPTR)
         CR     INPTR,INLENGT           WRITE LINE IF IT HAS
         BE     OUTPUT                  BEEN PROCESSED
         IC     CHAR,INBUFF(INPTR)      THIS SECTION OF CODE
         CLM    CHAR,1,BLANK            SKIPS OVER THE BLANKS
         BE     SKPBLNK                 BETWEEN FIELDS.
         LA     OUTPTR,6(OUTPTR)
STCHAR   STC    CHAR,OUTBUFF-1(OUTPTR)
         LA     INPTR,1(INPTR)
         CR     INPTR,INLENGT           AN END-OF-LINE CAN END
         BE     OUTPUT                  THE ARGLIST, IF ANY
         IC     CHAR,INBUFF(INPTR)
         CLM    CHAR,1,BLANK            A BLANK ENDS THE
         BE     SKPBLNK                 OPCODE OR ARGLIST
         LA     OUTPTR,1(OUTPTR)
         B      STCHAR
OUTPUT   SPUNCH  OUTBUFF,(OUTPTR)
         B      NEXTLN
OUTCOM   SPUNCH  INBUFF,(INLENGT)
         B      NEXTLN
EOF      EXIT
BLANK    DC     C' '
INBUFF   DS     CL80
OUTBUFF  DS     CL80
         END
```

The templates could now be written with the appropriate number of blanks between parameter flags. This solved most of the problems. However, in certain cases, a trailing blank still caused headaches. An example was the LAM (Load A Modified) statement. The macro definition originally used was

```
'□LAM□□□□□'|
%10 L     #GRA,%20(#GRB)$
$
```

If the input statement was

```
L25□□□□LAM□□□□□8
```

with no trailing blank, the instruction generated would be

```
L25□□□□L    #GRA,8(#GRB)
```

which is all right. But if the statement was

```
L25□□□□LAM□□□□□8□
```

the generated instruction would be

```
L25□□□□L    #GRA,8□(#GRB)
```

which is an invalid Assembler statement. The problem could have been solved by using the above procedure which also chops off trailing blanks. (The one used originally did not do so). Another solution was to use some character to indicate the end of a source line, as is customary when writing programs using the FLUB language. But since the number of statements (4) causing such problems were few, it was easier to introduce another macro definition with the template

```
'□LAM□□□□□'□|.
```

It should be remarked that STAGE2 looks for an expression which is balanced with respect to parentheses to match with a parameter flag. This caused a problem with the mapping of the CCL (Compare C with a Literal) statement. The definition was

```
'□CCL□□□□□'|
%10□CLI   #CHARLOC,C%20%F1$
$
```

This works well for statements like

CCL 'A' or CCL '0'.

But when it comes to

CCL '(' or CCL ')',

STAGE2 would not match the unbalanced parenthesis with the parameter flag, and hence the source statement was considered a mismatch with the given template. As a result, the source statements

CCL '(' and CCL ')',

were punched out instead of

CLI #CHARLOC,C'(' and CLI #CHARLOC,C')'.

In our case, since there were only two instances to consider, we simply translated them by hand to the 370 Assembler instructions.

2.2.4.2 The OF Macro

One of the more difficult tasks of the mapping was the handling of the OF macro. Using this macro is one way to represent numerical constants. It uses the submacros

LCH - the number of storage units used to represent a character;

LNM - the number of storage units used to represent an integer;

LICH - 1/LCH.

The OF macro is of the form OF(argument) where "argument" is one of

a. N*S+S

b. N*S-S

c. S+S

d. S-S

e. S

where N stands for any positive integer and S for any of the submacros. Examples of the OF macro are

OF(LNM+LNM)

OF(LCH)

OF(3*LCH+LNM).

It is used, for example, in

LAL OF(LNM+LNM)

i.e., load A with the value 8 (if LNM is 4).

On the IBM 370/168, LCH is 1 and LNM is 4. A special STAGE2 macro was defined

```
' EQUAL '|
%F3$
$
```

This macro inserts its first argument in the STAGE2 memory and gives it the value of the second argument, evaluated as an arithmetic expression. For example, after the statement

LCH EQUAL 1

is encountered, LCH and 1 can be used interchangeably.

Now consider the definition of the OF macro:

```
'OF(')'|
%10%24%30%F1$
$
```

The whole line is copied as is, except that OF(expression) is replaced by the value of "expression".

Let us illustrate the workings of this macro via the BUMP

instruction. The format of the BUMP statement is

```
BUMP     variable,expression
```

and it increases the value of its first argument by its second argument. For example,

```
BUMP     HOLD,4
```

increases the value of HOLD by 4. The BUMP macro definition is as follows:

```
' BUMP     ','|
%10 LA     #GRT,%30$
           A      #GRT,%20%F1$
           ST     #GRT,%20%F1$
$
```

Consider the input statement

```
L10    BUMP     STKPT,OF(3*LCH+LNM)
```

The first statement generated by the above definition is

```
L10    LA    #GRT,OF(3*LCH+LNM)
```

But since the first line in the body of the definition does not end with %F1, the generated line is rescanned to see if it matches any given template. It turns out that it matches the template of the OF macro. That definition evaluates the expression 3*LCH+LNM and finally the line

```
L10    LA    #GRT,7
```

is generated. The full output from the above call of BUMP is

```
L10    LA    #GRT,7
       A     #GRT,STKPT
       ST    #GRT,STKPT
```

2.2.4.3 Mapping of Registers

The numerical registers A and B were mapped into two general purpose registers. However, the character register C was held in memory. It was designated by #CHARLOC. Since the 370 instruction set contains a number of memory-memory and memory-immediate instructions, this choice for C enabled a much more concise mapping of ALGEBRA than if C was mapped to a general register. For example, consider the LOWL statement

```
CCI   CPTR
```

which compares the character in the C register with the character pointed at by CPTR. In the implementation, the above statement was mapped to

```
L     #GRT,CPTR
CLC   #CHARLOC(1),0(#GRT)
```

However, if the general register, #GRC, say, was used as the C register, the generated statements would be something like

```
L     #GRT,CPTR
IC    #GRT,0(#GRT)
N     #GRT,=X'000000FF'
CR    #GRC,#GRT
```

Even this assumes that the high-order 24 bits of #GRC are always zero. The AND instruction could be eliminated if #GRT represents a register whose high-order 24 bits are always guaranteed to be zero. In either case, holding C in memory enables fewer statements to be generated. The same observation holds for almost all the other instructions involving the C

register.

2.2.4.4 Mapping the Output Statements - MESS, RMESS and QMESS

The three output statements were implemented the same way. The biggest problem was encountered in dealing with newlines - as represented by a "$" within the argument. For instance, the statement

MESS '**INVALID STATEMENT$'

should cause

**INVALID STATEMENT

to be printed on the output device, whereas

MESS '**INVALID STATEMENT'

should simply place the message in the output buffer without printing it (unless the buffer overflows). This was accomplished by passing the argument string to a routine which sequenced through the string, taking the appropriate action depending on whether or not a "$" was encountered. If the current character was a "$", then the output buffer was written on the output device. Otherwise, the output buffer pointer was increased by one. If this now exceeded the length of the buffer, the buffer was written. Otherwise the current character was stored in the buffer.

2.2.5 Further Notes on Implementation

The debugging of the macros was facilitated by a macro test program, LOWLTEST. The complete text of this program is given by Brown[4].

Subroutine calls are handled by a stack mechanism. When the subroutine is entered, the return address is pushed on the stack, and when the routine is about to be exited, the stack is popped, giving the return address.

The program accepts interrupts at any time. For instance, if a large table is being output, the user can interrupt it without losing any information concerning values or operator definitions. If no values have been defined when the interrupt occurs, a return to the Operating System is taken.

A complete listing of the macros finally arrived at is given in Appendix E. For those macros implemented by subroutine calls, the subroutines are given in Appendix F.

2.2.6 Conclusions

The mapping tool, STAGE2, was adequate but not ideal. A powerful macro assembler might have avoided the format problems mentioned earlier, because the LOWL statements are of the form that is readily acceptable to such processors. Also bypassed would be the problem that STAGE2 had in translating the CCL '(' statement. On the other hand, it may not have been

able to handle the call of the OF macro within another macro call. Problems may also have arisen due to restrictions imposed on what characters could appear in the replacement text of macro bodies. For example the IBM 370 Assembler does not allow one to generate a comment (a line starting with *) from within a macro.

Bibliography

1. Brooker, R. A. And Morris, D. - A General Translation Program for Phrase Structure Languages, Jour. ACM,9,1, pp. 1-10 (1962).

2. Brown, P. J. - The ML/1 Macro Processor, Comm. ACM,10,10, pp. 618-623 (1967).

3. Brown, P. J. - Using a Macro Processor to Aid Software Implementation, Computer Journal,12,4, pp. 327-331 (1969).

4. Brown, P. J. - Macro Processors and Techniques for Portable Software, Wiley & Sons (1974).

5. Brown, P. J. And Lowe, J. D. - A Computer Program for Symbolic Logic, Bulletin IMA,7,11, pp.320-322 (1971).

6. Ferguson, D. E. - The Evolution of the Meta-assembly Program, Comm. ACM,9,3, pp. 190-193 (1966).

7. Griswold, R. E. - The Macro Implementation of SNOBOL4, Freeman, San Francisco (1972).

8. Halpern, M. I. - Towards a General Processor for Programming Languages, Comm. ACM,11,1, pp. 15-25 (1968).

9. Koster, C. H. - Portable Compilers and the UNCOL Problem, IFIP Working Conference on Machine Oriented Higher Level Languages, Trondheim, 1973.

10. Masterson, K. S. - Compilation for Two Computers Using NELIAC, Comm. ACM,3,11, pp. 607-611 (1960).

11. Mc Ilroy, M. D. - Macro Extensions of Compiler Languages, Comm. ACM,3,4, pp. 214-220 (1960).

12. Mock, O. et al - The Problem of Programming Communication with Changing Machnes - a Proposed Solution, Comm. ACM,1, Aug. pp. 12-18, Sept. pp. 9-15 (1958).

13. Newey, M. C., Poole, P. C. And Waite, W. M. - Abstract Machine Modelling to Produce Portable Software - a Review and Evaluation, Software - Practice and Experience,2, pp. 107-136 (1972).

14. Orgass, R. J. and Waite, W. M. - A Base for a Mobile Programming System, Comm. ACM,12,9, pp. 507-510 (1969).

15. Poole, P. C. And Waite, W. M. - Machine Independent

Software, Proceedings of the ACM 2nd Symposium on Operating System Principles, Princeton, New Jersey (1969).

16. Richards, M. A. - BCPL - A Tool for Compiler Writing and System Programming, AFIPS Conference Proceedings,34, pp. 557-566 (1969).

17. Richards, M. A. - The Portability of the BCPL Compiler, Software - Practice and Experience,1, pp. 135-146 (1971).

18. Richards, M. A. - Bootstrapping the BCPL Compiler Using INTCODE, IFIP Working Conference on Machine Oriented Higher Level Languages, Trondheim, 1973.

19. Steel, T. B. - UNCOL: the Myth and the Fact, Annual Review in Automatic Programming,2, pp. 325-344 (1961).

20. Strachey, C. - A General Purpose Macrogenerator, Computer Journal,8,3, pp. 225-241 (1967).

21. Waite, W. M. - A Language Independent Macro Processor, Comm. ACM,10,7, pp. 433-440 (1967).

22. Waite, W. M. - Building a Mobile Programming System, Tech. Rep. 69-2, Graduate School Computing Center, U of Colorado, Boulder, colo. USA.

23. Waite, W. M. - The STAGE2 Macro Processor, Tech. Rep. 69-3, Graduate School Computing Center, U of Colorado, Boulder, colo. USA.

24. Waite, W. M. - The Mobile Programming System: STAGE2, Comm. ACM,13,7, pp.415-421 (1970).

25. Warshall, S. - Language Independence?, Incompatibility, Infotech State of the Art Report 9, pp. 233-244 (1972).

26. Wilkes M. V. - An Experiment with a Self-compiling Compiler for a List Processing Language, Annual Review in Automatic Programming, 4, pp. 1-48 (1964).

27. Wilkes M. V. - The Outer and Inner Syntax of a Programming Language, Computer Journal,11,3, pp. 260-263 (1968).

Appendix A

The following are the SIMCMP mapping macros used for producing STAGE2, Version 1.

```
.'$'0
FLG ' = '.
       LH    TEM,LIST+'21*8$
       STH   TEM,LIST+'11*8$
$
VAL ' = PTR '.
       L    TEM,LIST+4+'21*8$
       STH   TEM,LIST+2+'11*8$
$
PTR ' = VAL '.
       LH    TEM,LIST+2+'21*8$
       ST    TEM,LIST+4+'11*8$
$
GET ' = '.
      L     TEM,LIST+4+'21*8$
      LD    EVENR,LIST(TEM)$
      STD   EVENR,LIST+'11*8$
$
STO ' = '.
      L    TEM,LIST+4+'11*8$
      LD   EVENR,LIST+'21*8$
      STD   EVENR,LIST(TEM)$
$
TO ''.
     B    LL'10'20$
$
STOP.
     EXIT$
$
TO '' IF FLG ' = '.
      CLC    LIST+'31*8(2),LIST+'41*8$
      BE    LL'10'20$
$
TO '' IF VAL ' = '.
      CLC    LIST+2+'31*8(2),LIST+2+'41*8$
      BE    LL'10'20$
$
TO '' IF PTR ' = '.
      CLC    LIST+4+'31*8(4),LIST+4+'41*8$
      BE    LL'10'20$
$
TO '' IF FLG ' NE '.
      CLC    LIST+'31*8(2),LIST+'41*8$
      BNE    LL'10'20$
$
TO '' IF VAL ' NE '.
      CLC    LIST+2+'31*8(2),LIST+2+'41*8$
```

```
      BNE   LL'10'20$
$
TO '' IF PTR ' NE '.
      CLC   LIST+4+'31*8(4),LIST+4+'41*8$
      BNE   LL'10'20$
$
TO '' IF PTR ' GE '.
      L     TEM,LIST+4+'31*8$
      C     TEM,LIST+4+'41*8$
      BNL   LL'10'20$
$
TO '' BY '.
      LA    TEM,L'00+4$
      ST    TEM,LIST+4+'31*8$
L'00  B     LL'10'20$
$
RETURN BY '.
     L     RTN,LIST+4+'11*8$
     BR    RTN$
$
LOC ''.
LL'10'20   SR    TEM,TEM    USED AS A NOOP$
$
END PROGRAM.
     END$
$
VAL ' = ' + '.
      LH    TEM,LIST+2+'21*8$
      AH    TEM,LIST+2+'31*8$
      STH   TEM,LIST+2+'11*8$
$
VAL ' = ' - '.
      LH    TEM,LIST+2+'21*8$
      SH    TEM,LIST+2+'31*8$
      STH   TEM,LIST+2+'11*8$
$
PTR ' = ' + '.
      L     TEM,LIST+4+'21*8$
      A     TEM,LIST+4+'31*8$
      ST    TEM,LIST+4+'11*8$
$
PTR ' = ' - '.
      L     TEM,LIST+4+'21*8$
      S     TEM,LIST+4+'31*8$
      ST    TEM,LIST+4+'11*8$
$
PTR ' = ' / '.
      L     EVENR,LIST+4+'21*8$
      SRDA  EVENR,32$
      D     EVENR,LIST+4+'31*8$
      ST    ODDR,LIST+4+'11*8$
$
PTR ' = ' * '.
      L     ODDR,LIST+4+'21*8$
```

```
      M     EVENR,LIST+4+'31*8$
      ST    ODDR,LIST+4+'11*8$
$
VAL ' = CHAR.
      L    PAR1,MINUS2$
      BAL    RTN,ICHIO$
      STH    RTNVAL,LIST+2+'11*8$
$
CHAR = VAL '.
      LH    PAR1,LIST+2+'11*8$
      BAL   RTN,ICHIO$
      STH    RTNVAL,LIST+'11*8$
$
READ NEXT '.
     LH    PAR1,LIST+2+'11*8$
     BAL   RTN,IREAD$
     STH   RTNVAL,LIST+'11*8$
$
WRITE NEXT '.
     LH    PAR1,LIST+2+'11*8$
     BAL   RTN,IWRITE$
     STH   RTNVAL,LIST+'11*8$
$
REWIND '.
     L      RTNVAL,TWO$
     STH    RTNVAL,LIST+'11*8$
$
MESSAGE '''' TO '.
      SPRINT    M'10'20'30'40,L'10'20'30'40$
      SR        RTNVAL,RTNVAL$
      STH       RTNVAL,LIST+'51*8$
$$
```

Appendix B

The following are the SIMCMP mapping macros used for producing STAGE2, Version 2.

```
.'$'0
FLG ' = '.
       L   TEM,LIST+'21*12$
       ST  TEM,LIST+'11*12$
$
VAL ' = PTR '.
       L   TEM,LIST+8+'21*12$
       ST  TEM,LIST+4+'11*12$
$
PTR ' = VAL '.
       L   TEM,LIST+4+'21*12$
       ST  TEM,LIST+8+'11*12$
$
GET ' = '.
      L    TEM,LIST+8+'21*12$
      LH   TEM1,LIST(TEM)$
      ST   TEM1,LIST+'11*12$
      LH   TEM1,LIST+2(TEM)$
      ST   TEM1,LIST+4+'11*12$
      L    TEM1,LIST+4(TEM)$
      ST   TEM1,LIST+8+'11*12$
$
STO ' = '.
      L   TEM,LIST+8+'11*12$
      L   TEM1,LIST+'21*12$
      STH TEM1,LIST(TEM)$
      L   TEM1,LIST+4+'21*12$
      STH TEM1,LIST+2(TEM)$
      L   TEM1,LIST+8+'21*12$
      ST  TEM1,LIST+4(TEM)$
$
TO ''.
     B   LL'10'20$
$
STOP.
     EXIT$
$
TO '' IF FLG ' = '.
      CLC   LIST+2+'31*12(2),LIST+2+'41*12$
      BE   LL'10'20$
$
TO '' IF VAL ' = '.
      CLC   LIST+6+'31*12(2),LIST+6+'41*12$
      BE   LL'10'20$
$
TO '' IF PTR ' = '.
      CLC   LIST+8+'31*12(4),LIST+8+'41*12$
```

```
      BE   LL'10'20$
$
TO '' IF FLG ' NE '.
      CLC   LIST+2+'31*12(2),LIST+2+'41*12$
      BNE   LL'10'20$
$
TO '' IF VAL ' NE '.
      CLC   LIST+6+'31*12(2),LIST+6+'41*12$
      BNE   LL'10'20$
$
TO '' IF PTR ' NE '.
      CLC   LIST+8+'31*12(4),LIST+8+'41*12$
      BNE   LL'10'20$
$
TO '' IF PTR ' GE '.
      L     TEM,LIST+8+'31*12$
      C     TEM,LIST+8+'41*12$
      BNL   LL'10'20$
$
TO '' BY '.
      LA    TEM,L'00+4$
      ST    TEM,LIST+8+'31*12$
L'00  B     LL'10'20$
$
RETURN BY '.
     L     RTN,LIST+8+'11*12$
     BR    RTN$
$
LOC ''.
LL'10'20   SR    TEM,TEM    USED AS A NOOP$
$
END PROGRAM.
     END$
$
VAL ' = ' + '.
      L    TEM,LIST+4+'21*12$
      A    TEM,LIST+4+'31*12$
      ST   TEM,LIST+4+'11*12$
$
VAL ' = ' - '.
      L    TEM,LIST+4+'21*12$
      S    TEM,LIST+4+'31*12$
      ST   TEM,LIST+4+'11*12$
$
PTR ' = ' + '.
      L     TEM,LIST+8+'21*12$
      A     TEM,LIST+8+'31*12$
      ST    TEM,LIST+8+'11*12$
$
PTR ' = ' - '.
      L     TEM,LIST+8+'21*12$
      S     TEM,LIST+8+'31*12$
      ST    TEM,LIST+8+'11*12$
$
```

```
PTR ' = ' / '.
      L     EVENR,LIST+8+'21*12$
      SRDA   EVENR,32$
      D     EVENR,LIST+8+'31*12$
      ST    ODDR,LIST+8+'11*12$
$
PTR ' = ' * '.
      L     ODDR,LIST+8+'21*12$
      M     EVENR,LIST+8+'31*12$
      ST    ODDR,LIST+8+'11*12$
$
VAL ' = CHAR.
      L    PAR1,MINUS2$
      BAL   RTN,ICHIO$
      ST    RTNVAL,LIST+4+'11*12$
$
CHAR = VAL '.
      L    PAR1,LIST+4+'11*12$
      BAL   RTN,ICHIO$
      ST    RTNVAL,LIST+'11*12$
$
READ NEXT '.
     L    PAR1,LIST+4+'11*12$
     BAL   RTN,IREAD$
     ST    RTNVAL,LIST+'11*12$
$
WRITE NEXT '.
     L     PAR1,LIST+4+'11*12$
     BAL   RTN,IWRITE$
     ST    RTNVAL,LIST+'11*12$
$
REWIND '.
     MVC   LIST+'11*12(4),TWO$
$
MESSAGE '''' TO '.
     SPRINT    M'10'20'30'40,L'10'20'30'40$
     SR    RTNVAL,RTNVAL$
     ST    RTNVAL,LIST+'51*12$
$$
```

Appendix C

The following are the STAGE2 mapping macros used for producing STAGE2, Version 3.

```
.'$'0 (+-*/)
IF ' = ' SKIP '.
'F50$
$
IF ' NE ' SKIP '.
'F51$
$
SKIP '.
'F4$
$
FLG ' = '.
IF '20 = 0 SKIP 3$
IF '20 = 1 SKIP 4$
IF '20 = 2 SKIP 5$
SKIP 6$
      STH  #ZERO,LIST+'18*8'F1$
SKIP 6$
      STH  #ONE,LIST+'18*8'F1$
SKIP 4$
      STH  #TWO,LIST+'18*8'F1$
SKIP 2$
      LH   TEM,LIST+'28*8'F1$
      STH  TEM,LIST+'18*8'F1$
$
VAL ' = PTR '.
      L   TEM,LIST+4+'28*8'F1$
      STH  TEM,LIST+2+'18*8'F1$
$
PTR ' = VAL '.
      LH   TEM,LIST+2+'28*8'F1$
      ST   TEM,LIST+4+'18*8'F1$
$
GET ' = '.
      L    TEM,LIST+4+'28*8'F1$
      LD   EVENR,LIST(TEM)'F1$
      STD  EVENR,LIST+'18*8'F1$
$
STO ' = '.
      L   TEM,LIST+4+'18*8'F1$
      LD  EVENR,LIST+'28*8'F1$
      STD  EVENR,LIST(TEM)'F1$
$
TO ''.
     B   LL'10'20'F1$
$
STOP.
     EXIT'F1$
```

```
$
TO '' IF FLG ' = '.
      CLC   LIST+'38*8(2),LIST+'48*8'F1$
      BE   LL'10'20'F1$
$
TO '' IF VAL ' = '.
      CLC   LIST+2+'38*8(2),LIST+2+'48*8'F1$
      BE   LL'10'20'F1$
$
TO '' IF PTR ' = '.
      CLC   LIST+4+'38*8(4),LIST+4+'48*8'F1$
      BE   LL'10'20'F1$
$
TO '' IF FLG ' NE '.
      CLC   LIST+'38*8(2),LIST+'48*8'F1$
      BNE   LL'10'20'F1$
$
TO '' IF VAL ' NE '.
      CLC   LIST+2+'38*8(2),LIST+2+'48*8'F1$
      BNE   LL'10'20'F1$
$
TO '' IF PTR ' NE '.
      CLC   LIST+4+'38*8(4),LIST+4+'48*8'F1$
      BNE   LL'10'20'F1$
$
TO '' IF PTR ' GE '.
      L     TEM,LIST+4+'38*8'F1$
      C     TEM,LIST+4+'48*8'F1$
      BNL   LL'10'20'F1$
$
TO '' BY '.
      LA   TEM,L'00+4'F1$
      ST   TEM,LIST+4+'38*8'F1$
L'00  B    LL'10'20'F1$
$
RETURN BY '.
     L    RTN,LIST+4+'18*8'F1$
     BR   RTN'F1$
$
LOC ''.
LL'10'20   SR    TEM,TEM    USED AS A NOOP'F1$
$
END PROGRAM.
     END'F1$
'F0$
$
VAL ' = ' + '.
IF '30 = 0 SKIP 5$
IF '20 = 0 SKIP 9$
      LH   TEM,LIST+2+'28*8'F1$
      AH   TEM,LIST+2+'38*8'F1$
      STH  TEM,LIST+2+'18*8'F1$
SKIP 6$
IF '20 = 0 SKIP 2$
```

```
      MVC  LIST+2+'18*8(2),LIST+2+'28*8'F1$
SKIP 3$
      STH  #ZERO,LIST+2+'18*8'F1$
SKIP 1$
      MVC  LIST+2+'18*8(2),LIST+2+'38*8'F1$
$
VAL ' = ' - '.
      LH   TEM,LIST+2+'28*8'F1$
IF '30 NE 1 SKIP 2$
      BCTR  TEM,0'F1$
SKIP 1$
      SH   TEM,LIST+2+'38*8'F1$
      STH  TEM,LIST+2+'18*8'F1$
$
PTR ' = ' + '.
IF '30 = 0 SKIP 8$
IF '20 = 0 SKIP 12$
      L    TEM,LIST+4+'28*8'F1$
IF '30 NE 7 SKIP 2$
      AR   TEM,#EIGHT'F1$
SKIP 1$
      A    TEM,LIST+4+'38*8'F1$
      ST   TEM,LIST+4+'18*8'F1$
SKIP 6$
IF '20 = 0 SKIP 2$
      MVC  LIST+4+'18*8(4),LIST+4+'28*8'F1$
SKIP 3$
      ST   #ZERO,LIST+4+'18*8'F1$
SKIP 1$
      MVC  LIST+4+'18*8(4),LIST+4+'38*8'F1$
$
PTR ' = ' - '.
      L    TEM,LIST+4+'28*8'F1$
IF '30 NE 7 SKIP 2$
      SR   TEM,#EIGHT'F1$
SKIP 1$
      S    TEM,LIST+4+'38*8'F1$
      ST   TEM,LIST+4+'18*8'F1$
$
PTR ' = ' / '.
      L    EVENR,LIST+4+'28*8'F1$
      SRDA  EVENR,32'F1$
      D    EVENR,LIST+4+'38*8'F1$
      ST   ODDR,LIST+4+'18*8'F1$
$
PTR ' = ' * '.
      L    ODDR,LIST+4+'28*8'F1$
      M    EVENR,LIST+4+'38*8'F1$
      ST   ODDR,LIST+4+'18*8'F1$
$
VAL ' = CHAR.
      L   PAR1,MINUS2'F1$
      BAL   RTN,ICHIO'F1$
      STH   RTNVAL,LIST+2+'18*8'F1$
```

```
$
CHAR = VAL '.
      LH    PAR1,LIST+2+'18*8'F1$
      BAL   RTN,ICHIO'F1$
      STH    RTNVAL,LIST+'18*8'F1$
$
READ NEXT '.
     LH    PAR1,LIST+2+'18*8'F1$
     BAL   RTN,IREAD'F1$
     STH   RTNVAL,LIST+'18*8'F1$
$
WRITE NEXT '.
     LH    PAR1,LIST+2+'18*8'F1$
     BAL   RTN,IWRITE'F1$
     STH   RTNVAL,LIST+'18*8'F1$
$
REWIND '.
     L      RTNVAL,TWO'F1$
     STH    RTNVAL,LIST+'18*8'F1$
$
MESSAGE '''' TO '.
     SPRINT    M'10'20'30'40,L'10'20'30'40'F1$
     STH       #ZERO,LIST+'58*8'F1$
$$
```

Appendix D

The following are the STAGE2 mapping macros used for producing STAGE2, Version 4.

```
.'$'0 (+-*/)
IF ' = ' SKIP '.
'F50$
$
IF ' NE ' SKIP '.
'F51$
$
SKIP '.
'F4$
$
FLG ' = '.
IF '20 = 0 SKIP 3$
IF '20 = 1 SKIP 4$
IF '20 = 2 SKIP 5$
SKIP 6$
      ST  #ZERO,LIST+'18*12'F1$
SKIP 6$
      ST  #ONE,LIST+'18*12'F1$
SKIP 4$
      ST  #TWO,LIST+'18*12'F1$
SKIP 2$
      L   TEM,LIST+'28*12'F1$
      ST  TEM,LIST+'18*12'F1$
$
VAL ' = PTR '.
      L   TEM,LIST+8+'28*12'F1$
      ST  TEM,LIST+4+'18*12'F1$
$
PTR ' = VAL '.
      L   TEM,LIST+4+'28*12'F1$
      ST   TEM,LIST+8+'18*12'F1$
$
GET ' = '.
      L    TEM,LIST+8+'28*12'F1$
      LH   TEMP,LIST(TEM)'F1$
      ST   TEMP,LIST+'18*12'F1$
      LH   TEMP,LIST+2(TEM)'F1$
      ST   TEMP,LIST+4+'18*12'F1$
      L    TEMP,LIST+4(TEM)'F1$
      ST   TEMP,LIST+8+'18*12'F1$
$
STO ' = '.
      L   TEM,LIST+8+'18*12'F1$
      L   TEMP,LIST+'28*12'F1$
      STH TEMP,LIST(TEM)'F1$
      L   TEMP,LIST+4+'28*12'F1$
      STH TEMP,LIST+2(TEM)'F1$
```

```
      L    TEMP,LIST+8+'28*12'F1$
      ST   TEMP,LIST+4(TEM)'F1$
$
TO ''.
      B    LL'10'20'F1$
$
STOP.
      EXIT'F1$
$
TO '' IF FLG ' = '.
      CLC    LIST+2+'38*12(2),LIST+2+'48*12'F1$
      BE    LL'10'20'F1$
$
TO '' IF VAL ' = '.
      CLC    LIST+6+'38*12(2),LIST+6+'48*12'F1$
      BE    LL'10'20'F1$
$
TO '' IF PTR ' = '.
      CLC    LIST+8+'38*12(4),LIST+8+'48*12'F1$
      BE    LL'10'20'F1$
$
TO '' IF FLG ' NE '.
      CLC    LIST+2+'38*12(2),LIST+2+'48*12'F1$
      BNE    LL'10'20'F1$
$
TO '' IF VAL ' NE '.
      CLC    LIST+6+'38*12(2),LIST+6+'48*12'F1$
      BNE    LL'10'20'F1$
$
TO '' IF PTR ' NE '.
      CLC    LIST+8+'38*12(4),LIST+8+'48*12'F1$
      BNE    LL'10'20'F1$
$
TO '' IF PTR ' GE '.
      L      TEM,LIST+8+'38*12'F1$
      C      TEM,LIST+8+'48*12'F1$
      BNL    LL'10'20'F1$
$
TO '' BY '.
      LA    TEM,L'00+4'F1$
      ST    TEM,LIST+8+'38*12'F1$
L'00  B     LL'10'20'F1$
$
RETURN BY '.
      L     RTN,LIST+8+'18*12'F1$
      BR    RTN'F1$
$
LOC ''.
LL'10'20    SR    TEM,TEM    USED AS A NOOP'F1$
$
END PROGRAM.
 .    END'F1$
'F0$
$
```

```
VAL ' = ' + '.
IF '30 = 0 SKIP 5$
IF '20 = 0 SKIP 9$
      L   TEM,LIST+4+'28*12'F1$
      A   TEM,LIST+4+'38*12'F1$
      ST  TEM,LIST+4+'18*12'F1$
SKIP 6$
IF '20 = 0 SKIP 2$
      MVC  LIST+4+'18*12(4),LIST+4+'28*12'F1$
SKIP 3$
      ST  #ZERO,LIST+4+'18*12'F1$
SKIP 1$
      MVC  LIST+4+'18*12(4),LIST+4+'38*12'F1$
$
VAL ' = ' - '.
      L   TEM,LIST+4+'28*12'F1$
IF '30 NE 1 SKIP 2$
      BCTR  TEM,0'F1$
SKIP 1$
      S   TEM,LIST+4+'38*12'F1$
      ST  TEM,LIST+4+'18*12'F1$
$
PTR ' = ' + '.
IF '30 = 0 SKIP 8$
IF '20 = 0 SKIP 12$
      L    TEM,LIST+8+'28*12'F1$
IF '30 NE 7 SKIP 2$
      AR   TEM,#EIGHT'F1$
SKIP 1$
      A    TEM,LIST+8+'38*12'F1$
      ST   TEM,LIST+8+'18*12'F1$
SKIP 6$
IF '20 = 0 SKIP 2$
      MVC  LIST+8+'18*12(4),LIST+8+'28*12'F1$
SKIP 3$
      ST   #ZERO,LIST+8+'18*12'F1$
SKIP 1$
      MVC  LIST+8+'18*12(4),LIST+8+'38*12'F1$
$
PTR ' = ' - '.
      L    TEM,LIST+8+'28*12'F1$
IF '30 NE 7 SKIP 2$
      SR   TEM,#EIGHT'F1$
SKIP 1$
      S    TEM,LIST+8+'38*12'F1$
      ST   TEM,LIST+8+'18*12'F1$
$
PTR ' = ' / '.
      L    EVENR,LIST+8+'28*12'F1$
      SRDA  EVENR,32'F1$
      D    EVENR,LIST+8+'38*12'F1$
      ST   ODDR,LIST+8+'18*12'F1$
$
PTR ' = ' * '.
```

```
      L     ODDR,LIST+8+'28*12'F1$
      M     EVENR,LIST+8+'38*12'F1$
      ST    ODDR,LIST+8+'18*12'F1$
$
VAL ' = CHAR.
      L    PAR1,MINUS2'F1$
      BAL    RTN,ICHIO'F1$
      ST    RTNVAL,LIST+4+'18*12'F1$
$
CHAR = VAL '.
      L    PAR1,LIST+4+'18*12'F1$
      BAL   RTN,ICHIO'F1$
      ST    RTNVAL,LIST+'18*12'F1$
$
READ NEXT '.
     L    PAR1,LIST+4+'18*12'F1$
     BAL   RTN,IREAD'F1$
     ST   RTNVAL,LIST+'18*12'F1$
$
WRITE NEXT '.
     L    PAR1,LIST+4+'18*12'F1$
     BAL   RTN,IWRITE'F1$
     ST   RTNVAL,LIST+'18*12'F1$
$
REWIND '.
     MVC  LIST+'18*12(4),TWO'F1$  SET ERROR RETURN CODE
$
MESSAGE '''' TO '.
     SPRINT    M'10'20'30'40,L'10'20'30'40'F1$
     SR        #ZERO,#ZERO'F1$  RESET GR1 USED BY SPRINT
     ST       #ZERO,LIST+'58*12'F1$
$$
```

Appendix E

The following are the STAGE2 macros used in the mapping of the LOWL version of ALGEBRA.

```
|'$%0 (+-*/)
'OF(')'|
%10%24%30%F1$
$
IF ' = ' SKIP '|
%F50$
$
' EQUAL '|
%F3$
$
SKIP '|
%F4$
$
' DCL     '|
%F1$
22222222  DS   F$
$
' CON     '|
%10 DC   F'%20'$
$
' CON     ' |
%10 DC   F'%20'$
$
' EQU     ','|
%F1$
22222222  DS   F$
$
' IDENT     ','|
%F1$
22222222  EQU   333333333$
$
' NCH     '|
%10 DC   AL.8(%20)%F1$
$
' NCH     ' |
%10 DC    AL.8(%20)%F1$
$
' STR     '|
%10 DC  C%20%F1$
$
' LAV     ','|
IF %30 = R SKIP 2$
IF %30 = R  SKIP 1$
%10 L    #GRA,%20%F1$
$
' LBV     '|
%10 L     #GRB,%20%F1$
```

```
$
' LAL      '|
IF %20 = 0 SKIP 3$
IF %20 = 0  SKIP 2$
%10 LA     #GRA,%20$
SKIP 1$
%10 LR     #GRA,#ZERO%F1$
$
' LCN      '|
%10 MVI     #CHARLOC,%20%F1$
$
' LAI      ','|
IF %30 = R SKIP 3$
IF %30 = R  SKIP 2$
%10 L      #GRT,%20%F1$
           L      #GRA,0(#GRT)%F1$
$
' LCI      ','|
IF %30 = R SKIP 3$
IF %30 = R  SKIP 2$
%10 L      #GRT,%20%F1$
           MVC    #CHARLOC(1),0(#GRT)%F1$
$
' LAA      ','|
%10 LA     #GRA,%20%F1$
$
' STV      ','|
%10ST     #GRA,%20%F1$
$
' STI      ','|
%10 L      #GRT,%20%F1$
           ST     #GRA,0(#GRT)%F1$
$
' CLEAR      '|
%10 ST     #ZERO,%20%F1$
$
' AAV      '|
%10 A      #GRA,%20%F1$
$
' ABV      '|
%10 A      #GRB,%20%F1$
$
' AAL      '|
%10 LA     #GRT,%20$
           AR     #GRA,#GRT%F1$
$
' SAV      '|
%10 S      #GRA,%20%F1$
$
' SBV      '|
%10 S      #GRB,%20%F1$
$
' SAL      '|
%10 LA     #GRT,%20$
```

```
          SR     #GRA,#GRT%F1$
$
' SBL     '|
%10 LA    #GRT,%20$
          SR     #GRB,#GRT%F1$
$
' MULTL     '|
%10 LA    #GRT,%20%$
          MR     #GRAE,#GRT%F1$
$
' ANDV     '|
%10 N     #GRA,%20%F1$
$
' ANDL     '|
%10 LA    #GRT,%20$
          NR     #GRA,#GRT%F1$
$
' BUMP     ','|
%10 LA    #GRT,%30$
          A      #GRT,%20%F1$
          ST     #GRT,%20%F1$
$
' CAV     ','|
%10 C     #GRA,%20%F1$
$
' CAL     '|
IF %20 = 0 SKIP 4$
IF %20 = 0  SKIP 3$
%10 LA    #GRT,%20$
          CR     #GRA,#GRT%F1$
SKIP 1$
%10 LTR   #GRA,#GRA%F1$
$
' CCL     '|
%10 CLI    #CHARLOC,C%20%F1$
$
' CCN     '|
%10 CLI   #CHARLOC,%20%F1$
$
' CAI     ','|
%10 L     #GRT,%20%F1$
          C      #GRA,0(#GRT)%F1$
$
' CCI     '|
%10 L     #GRT,%20%F1$
          CLC    #CHARLOC(1),0(#GRT)%F1$
$
' GO     ',',',','|
%10 B     %20%F1$
$
' GOEQ     ',',',','|
%10 BE    %20%F1$
$
' GONE     ',',',','|
```

```
%10 BNE    %20%F1$
$
' GOGE      ',',',',|
%10 BNL    %20%F1$
$
' GOGR      ',',',',|
%10 BH     %20%F1$
$
' GOLE      ',',',',|
%10 BNH    %20%F1$
$
' GOLT      ',',',',|
%10 BL     %20%F1$
$
' GOPC      ',',',',|
%10 CLI    #CHARLOC,C'A'%F1$
           BL     %20%F1$
$
' GOND      ',',',',|
%10 CLI    #CHARLOC,C'0'%F1$
           BL     %20%F1$
           IC     #GRA,#CHARLOC%F1$
           N      #GRA,=X'0000000F'%F1$
$
' GOADD      '|
%10 L      #GRA,%20%F1$
           LA     #GRT,*+8%F1$
           B      #GOADD%F1$
$
' GOSUB      ',',|
%10 BAL    #GRRTN,%20%F1$
$
' SUBR      ',',',|
%20      LA     #GRPTR,4(#GRPTR)%F1$
         C      #GRPTR,#RSTKSZ%F1$
         BH     #STKERR%F1$
         ST     #GRRTN,#RTNSTK(#GRPTR)%F1$
IF %30 = X SKIP 1$
         ST     #GRA,%30%F1$
$
' EXIT      ',',|
%10 L      #GRRTN,#RTNSTK(#GRPTR)%F1$
           S      #GRPTR,#FOUR%F1$
IF %20 = 1 SKIP 2$
           B      (%20-1)*4(#GRRTN)%F1$
SKIP 1$
           BR     #GRRTN%F1$
$
' LAM      ' |
%10 L      #GRA,%20(#GRB)$
$
' LAM      '|
%10 L      #GRA,%20(#GRB)$
$
```

```
' LCM      ' |
%10 MVC    #CHARLOC(1),%20(#GRB)$
$
' LCM      '|
%10 MVC    #CHARLOC(1),%20(#GRB)$
$
' ALIGN'|
%10 BAL    #GRRTN,#ALIGN%F1$
$
' CSS'|
%10 L      #GRPTR,#MINUS4%F1$
$
' FSTK'|
%10 BAL    #GRRTN,#FSTK%F1$
$
' BSTK'|
%10 BAL    #GRRTN,#BSTK%F1$
$
' CFSTK'|
%10 BAL    #GRRTN,#CFSTK%F1$
$
' UNSTK      '|
%10 LA     #GRA,%20%F1$
           BAL    #GRRTN,#UNSTK%F1$
$
' FMOVE'|
%10 BAL    #GRRTN,#FMOVE%F1$
$
' BMOVE'|
%10 BAL    #GRRTN,#BMOVE%F1$
$
' PRGST      '|
** THIS IS THE START OF THE MI-CODE OF %20%F1$
$
' PRGEN'|
           END    #START%F1$
%F0$
$
' NB      '|
*%10    %20%F1$
$
' MESS      '|
%10 BAL    #GRA,*+6+((L'##%00+1)/2*2)    AROUND CONS%F1$
           DC     Y(L'##%00)     LENGTH%F1$
##%00      DC     C%20%F1$
           BAL    #GRRTN,#WRITE%F1$
$
' QMESS      '|
%10 BAL    #GRA,*+6+((L'##%00+1)/2*2)    AROUND CONS%F1$
           DC     Y(L'##%00)     LENGTH%F1$
##%00      DC     C%20%F1$
           BAL    #GRRTN,#WRITE%F1$
$
' RMESS      '|
```

```
%10 BAL    #GRA,*+6+((L'##%00+1)/2*2)    AROUND CONS%F1$
           DC     Y(L'##%00)     LENGTH%F1$
##%00      DC     C%20%F1$
           BAL    #GRRTN,#WRITE%F1$
$$
LNM EQUAL 4
LCH EQUAL 1
LICH EQUAL 1
```

Appendix F

The following procedures complement the mapping macros given in Appendix E. They represent those LOWL statements which are mapped into subroutine calls.

```
**
** THIS PROCEDURE ALIGNS THE CONTENTS OF THE
** A REGISTER TO THE NEXT FULLWORD BOUNDARY.
**
#ALIGN     LR     #GRTE,#GRA
          SRDA   #GRTE,32
          D      #GRTE,#FOUR
          LTR    #GRTE,#GRTE     IS IT ALREADY ALIGNED ?
          BZ     #OKA            I.E. IS REMAINDER ZERO?
          L      #GRT,#FOUR      IF NOT, ADD (4-REMAINDER)
          SR     #GRT,#GRTE      IF NOT, ADD (4-REMAINDER)
          AR     #GRA,#GRT       TO THE A REGISTER
#OKA      BR     #GRRTN
**
** THIS PROCEDURE STACKS A ON THE FORWARDS STACK.
** FFPT POINTS TO THE FIRST FREE STACK LOCATION.
**
#FSTK      L     #GRT,FFPT
           ST    #GRA,0(#GRT)
           LA    #GRT,LNM(#GRT)    INCREASE STACK POINTER
           C     #GRT,LFPT         IF FORWARDS STACK PTR MEETS
           BNL   ERLSO             BACKWARDS STACK PTR, THE
**                                 STACK SPACE IS USED UP.
           ST    #GRT,FFPT
           BR    #GRRTN
**
** STACK C, AS REPRESENTED BY #CHARLOC, ON THE FORWARDS STACK.
**
#CFSTK     L      #GRT,FFPT
           IC     #GRTE,#CHARLOC
           STC    #GRTE,0(#GRT)
           LA     #GRT,LCH(#GRT)    CHECK FOR STACK
           C      #GRT,LFPT         OVERFLOW, AS IN
           BNL    ERLSO             #FSTK.
           ST     #GRT,FFPT
           BR     #GRRTN
**
** STACK A ON THE BACKWARDS STACK.
**
#BSTK      L      #GRT,LFPT
           S      #GRT,#FOUR     DECREASE BACKWARDS STACK
           C      #GRT,FFPT      PTR; STACK OVERFLOW IF IT
           BNH    ERLSO          MEETS FORWARDS STACK PTR.
           ST     #GRT,LFPT
           ST     #GRA,0(#GRT)      STACK A.
           BR     #GRRTN
```

```
**
** UNSTACKS NUMBER AT THE TOP OF THE BACKWARDS AND
** STORES IT IN A VARIABLE, WHOSE ADDRESS IS IN A.
**
#UNSTK      L       #GRT,LFPT
            L       #GRB,0(#GRT)
            ST      #GRB,0(#GRA)     #GRA CONTAINS A(V)
            A       #GRT,#FOUR    INCREASE BACKWARDS STACK PTR
            ST      #GRT,LFPT
            BR      #GRRTN

#GOADD       M       #GRAE,#FOUR
             AR      #GRA,#GRT
             BR      #GRA
**
** PERFORMS A CHARACTER-BY-CHARACTER MOVE FROM SOURCE
** (POINTED AT BY SRCPT) TO DESTINATION (POINTED AT BY
** DSTPT), STARTING WITH THE FIRST CHARACTER AND ENDING
** WITH THE LAST. THE NUMBER OF CHARACTERS TO BE MOVED
** IS GIVEN BY THE CONTENTS OF A.
**
#FMOVE      L     #GRT,SRCPT
            L     #GRTE,DSTPT
#NXTMOV     IC    #GRB,0(#GRT)
            STC   #GRB,0(#GRTE)
            LA    #GRT,1(#GRT)
            LA    #GRTE,1(#GRTE)
            BCT   #GRA,#NXTMOV
            BR    #GRRTN
**
** THIS IS SIMILAR TO FMOVE ABOVE, EXCEPT THAT THE
** MOVE STARTS WITH THE LAST CHARACTER AND ENDS
** WITH THE FIRST.
**
#BMOVE      L       #GRT,SRCPT
            L       #GRTE,DSTPT
            AR      #GRT,#GRA
            AR      #GRTE,#GRA
#NXTMOV1    BCTR    #GRT,0
            BCTR    #GRTE,0
            IC      #GRB,0(#GRT)
            STC     #GRB,0(#GRTE)
            BCT     #GRA,#NXTMOV1
            BR      #GRRTN
**
** THIS PROCEDURE WORKS IN CONJUNCTION WITH #WRCH
** BELOW. THE CHARACTER STRING TO BE WRITTEN IS PASSED
** TO IT. REMEMBER THAT THIS STRING MAY OR MAY NOT
** CONTAIN THE NEWLINE CHARACTER("$"). IT PASSES
** THE CHARACTERS ONE AT AT TIME (VIA THE C REGISTER)
** TO WRCH WHICH DETERMINES WHAT ACTION SHOULD BE TAKEN.
**
#WRITE     ST    #GRRTN,#SAVADR1
           LH    #GRB,0(#GRA)
```

```
          LA    #GRA,2(#GRA)  ADDRESS OF STRING
          SR    #GRAE,#GRAE     INDEX TO STRING
#NEXTCH   IC    #GRTE,0(#GRAE,#GRA)
          STC   #GRTE,#CHARLOC   STORE CHAR IN C
          BAL   #GRRTN,#WRCH
          LA    #GRAE,1(#GRAE)   INCREASE STRING PTR
          BCT   #GRB,#NEXTCH
          L     #GRRTN,#SAVADR1
          BR    #GRRTN
**
** THIS PROCEDURE TAKES DIFFERENT ACTIONS DEPENDING
** ON THE CHARACTER IN THE C REGISTER AND THE
** CURRENT LENGTH OF THE OUTPUT BUFFER.
**
#WRCH     CLI     #CHARLOC,C'$'   WRITE LINE IF C CONTAINS
          BZ      #NEWLINE        $  OR
          CLI    #CHARLOC,NLREP   THE NEWLINE CHARACTER(X*'05*')
          BZ      #NEWLINE
          LH      #GRT,#OUTLEN
          LA      #GRT,1(#GRT) ELSE STORE CHAR IN THE
          IC      #GRTE,#CHARLOC  NEXT AVAILABLE LOCATION
          STC     #GRTE,#OUTBUFF-1(#GRT)  IN BUFFER.
          C       #GRT,=F'132'       WRITE BUFFER IF FULL
          BH      #OUTPUT
          STH     #GRT,#OUTLEN       STORE NEW BUFFER LENGTH
          BR      #GRRTN
#OUTPUT   STH     #GRT,#OUTLEN
#NEWLINE   ST    #GRRTN,#SAVADR
           SPRINT   #OUTBUFF,#OUTLEN   WRITE BUFFER ON DEVICE
           L     #GRRTN,#SAVADR
           LA     #GRT,1     INITIALIZE BUFFER LENGTH
           STH    #GRT,#OUTLEN
           BR     #GRRTN
**
```