Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1973

A Study of Decompiling Machine Language into High-Level Machine Independent Languages

Barron Cornelius Housel

Report Number:
73-100

Housel, Barron Cornelius, "A Study of Decompiling Machine Language into High-Level Machine Independent Languages" (1973). *Department of Computer Science Technical Reports.* Paper 2.
https://docs.lib.purdue.edu/cstech/2

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

A STUDY OF DECOMPILING MACHINE LANGUAGES INTO HIGH-LEVEL MACHINE INDEPENDENT LANGUAGES

Barron Cornelius Housel III
Purdue University
CSD TR 100

A STUDY OF DECOMPILING MACHINE LANGUAGES INTO HIGH-LEVEL MACHINE INDEPENDENT LANGUAGES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Barron Cornelius Housel III

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 1973

ACKNOWLEDGEMENTS

I offer thanks to Professor M. H. Halstead for his inspiration, guidance and patience during the course of this research project. For their financial support, I thank the IBM Corporation. I also thank my colleague, Frank Friedman, for his constructive criticism and comments. Finally, I thank my wife Ann for her constant help and support.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT vii

CHAPTER 1 - BACKGROUND AND GENERAL CONSIDERATIONS 1

Introduction 1
Background of Decompiling 4
Considerations in Decompilation 15
Overview of This Research 28

CHAPTER 2 - DETERMINING THE CONTROL FLOW GRAPH 33

Separating Data From Instructions 33
The Block Detection Method 43

CHAPTER 3 - INTERMEDIATE TEXT GENERATION AND COMPRESSION 54

Overview of the Translation Process 59
"IMTEXT" Description and MIX-IMTEXT Translation 65
Determining Busy Status of Variables 73
The IMTEXT "Compression" Algorithm....................79

CHAPTER 4 - FINDING PROGRAM LOOPS 87

Definitions 87
The Algorithm 90
Analysis Constraints 97
Block Levels 101

CHAPTER 5 - DETERMINING DISJOINT ARRAYS VIA ANALYSIS OF LOOPS 103

The "VALUE-SET" of a Variable
Individual SCR (loop) Analysis 118
A Structure for Representing Nested Loops 128

Computing Extents of Indexed References 131
A Heuristic Approach 146

CHAPTER 6 - TARGET LANGUAGE GENERATION 151

Data Declarations 152
Arithmetic Expression Translation 169
Translation of Control Statements 178

CHAPTER 7 - EXPERIMENTAL PROCEDURE AND RESULTS 186

Implementation 190
Performance Considerations 191
Test Case Editing 194

CHAPTER 8 - EXTENSIONS AND CONCLUSIONS 197

Subroutines 197
Self-Modifying Code 199
Conclusions 204

REFERENCES 207

APPENDICIES 210

Appendix A: Summary of the MIX Machine 210
Appendix B: Test Case Results 214
Appendix C: Performance Data 257

VITA 259

LIST OF TABLES

Table Page

6.A Unstructured Storage Element Mappings 160

6.B Default Attributes for Structured Data 163

7.A Summary of Test Case Editing 194

LIST OF FIGURES

Figure Page

1.A The Decompilation Process 29

2.A Control Flow With Indexed Jumps 52

4.A Control Flow With Nested Loops 92

4.B Irreducible Graph 98

4.C Tangent Loops 100

5.A "VALUE-SET" Example Program 109

5.B Initial C-graph Representation 110

5.C C-graph Reduced One Level 111

5.D C-graph Reduced Two Levels 112

5.E Nested Iterative Loop Example Program 133

5.F C-graph for VALSET(XR5,10,INITIAL,SAVE,CGP) 136

6.A Expression Tree for an n-tuple 174

6.B "DO-group" Translation 180

7.A Experimental Decompilation Procedure 187

Appendix
Figure
C.1 Predicted and Observed Decompile Times 260

ABSTRACT

Housel, Barron Cornelius. Ph.D., Purdue University, August, 1973. A Study of Decompiling Machine Languages into High-Level Machine Independent Languages.

This dissertation describes techniques for translating restricted classes of machine or assembly language into high-level machine independent languages. This translation process, called "decompilation", is intended to be largely, but not entirely automatic.

A systematic methodology is developed for the decompiling process. Initially, the source program is mapped to an abstract representation consisting of the program control flow graph and an intermediate text form of the program statements. A sequence of transformations is performed on this representation, in effect, raising the level of the program. Sequences of computations are combined to form high level statements in the target language and redundant transfers and temporaries are eliminated. Finally, a formatted, structured target language program is generated.

Among the significant features of this study are algorithms for detecting program loops and their interrelationships, the detection of data structures, and program simplification.

In order to demonstrate the techniques developed, a decompiler was written to translate Knuth's MIX assembly language into PL/1. A number of published algorithms were decompiled and verified by executing the target language programs.

CHAPTER 1

BACKGROUND AND GENERAL CONSIDERATIONS OF DECOMPILING

INTRODUCTION

In the context of computing systems, compilation is generally thought of as the translation process whereby a program written in a high level language is translated into a target machine or assembly language suitable for execution on the target machine. Intuitively decompilation can be viewed as the inverse of this process; that is, the translation of machine or assembly language into a high level language. This has been the context of decompiling in previous work.

Decompilers were written as early as 1960 (Halstead,1962), and yet the literature has been conspicuously void of articles describing the technology developed in this area. One possible explanation is that they were unsuccessful. However, this can be discounted since there is documentation (Halstead,1967) of at least one decompiling effort undertaken by the Lockheed

Corporation which was commercially successful. It is also known that the IBM Corporation (IBM,1967) and smaller software firms have developed successful decompilers for translating IBM Autocoder into Cobol. A more probable explanation of the lack of technical contribution in the area is the following. First of all, as alluded to above, most of the existing decompilers have been developed commercially. The packages have been proprietary in nature and the technology has remained in the "trade secret" status. This is analogous to the early history of compilers before compiler writing techniques became generally known. It was not until the academic community took interest in artificial language translation that the compiler writing technology became widely published and developed. A second conjecture which would explain a lack of published knowledge in the area is the lack of generality of the technology developed in the specific instances of decompiler development. A typical commercial decompiler is concerned with translating a specific machine or assembly language into a specific target language at least cost. Many of the implementation techniques were machine and language dependent and would not be appropriate as general contributions to the area. In other words these implemenatations can be classified as "ad hoc".

Another argument would suggest that with the trend toward

higher level languages for all aspects of computing that decompiling is a temporary technology, thus a rigorous treatment of the subject is not worthwhile. Although higher level languages are definitely coming of age, it is generally conceded that they do not as yet meet all the requirements of the computing community and may never completely do so, especially in cases where time and space are very critical as in process control applications or small memory mini-computing systems. In addition there are millions of dollars worth of machine language programs still in existence whose life is far from over. For example, many IBM 1401 applications are still being run in emulation mode on the latest IBM 370 models. Therefore, decompiling appears to be a worthwhile pursuit in the area of program conversion alone for the foreseeable future.

Recently the goals of decompilation have been expanded with the result that interest has been considerably increased in this area. If the definition of decompilation is expanded to mean the translation of <u>any</u> lower level (not just machine languages) to some higher level language, the technology becomes open ended. This more general concept of decompiling paves the way not only for program conversion from one hardware system to another, but provides means for automatic evolution from one language to another. This has been demonstrated in a minor way by the SIFT/LIFT (Allen

et. al.,1963) packages which convert Fortran II to Fortran IV. The need here is obvious when one considers the fact that language sophistication has increased many fold over that of Fortran, and yet Fortran is still the most widely used scientific language today.

In addition to being a valuable program conversion tool, decompiling may offer a valuable tool in program documentation. To be able to increase the level of abstraction of a machine language program to a representation in a higher level language would greatly increase the clarity of the program logic and ease the task of reprogramming and maintenance.

Based on the above considerations, it seems that the development of general principles and methodologies in the area of decompiling is both timely and relevant.

BACKGROUND OF DECOMPILING

Historically the primary goal of decompilation has been that of program conversion. With the rapid advance in hardware technology, the ability to automatically transfer a machine or assembly language program from one machine to another would have the obvious economic advantage of eliminating the reprogramming problem. With this in mind it is interesting to note some of the alternatives to

decompiling which have been attempted in order to achieve automatic program conversion.

Alternatives to Decompiling for Automatic Program Conversion

Two of the most common choices are emulation and simulation. Emulation has proven quite popular and successful. With this technique, however, it is not possible to take full advantage of the hardware since the instructions of the source machine must be interpretively executed by microprograms. Also, this overhead is incurred every time the programs are run. Another disadvantage is that users are not encouraged to upgrade to the latest technology. Also, many machines lack emulation capability. Lichstein (1969) gives a good treatment on the applicability of emulation.

Simulation is the process of modeling the source machine on the target machine by writing an interpreter on the target machine in macro machine code, which interpretively executes the source machine instructions. This method has most of the drawbacks of emulation and in addition it is considerably less efficient. The implementation cost of a simulator is not excessive, however, and this approach is feasible for infrequently run programs or where no other alternative is available. It should be noted, however, that complete simulation for all source machine programs

is in general not possible because of timing considerations and other disparities between the two machines. The techniques of emulation and simulation are means whereby the target machine is constrained to create the environment of the source machine, allowing the source machine programs to run on the target machine without change. Other program conversion techniques attempt to translate the source machine programs to a form executable in the unconstrained new environment of the target machine. Since decompiling falls in the latter category, it is of primary interest to study various techniques which attempt program conversion by automatic translation. Besides decompilers, direct machine language to machine language (MLs to MLt) translators and assembly language to assembly language (ALs to ALt) translators have been implemented.

Gunn (1962) describes an effort aimed at converting a Mercury machine language into one that would run on the Orion computer, and Opler et. al. (1962) document a translator which attempted to convert IBM 705 machine language programs (binary) to equivalent IBM 7074 code. Neither of these efforts led to known practical results. Two packages of this type which achieved success commercially are EXODUS which is marketed by Computer Sciences Corporation, and the LIBERATOR which was developed by Honeywell Corporation. EXODUS translates IBM 1401

Autocoder to IBM 360 assembly language, while LIBERATOR converts IBM 1401 Autocoder to the Honeywell 200 series machines. Perhaps the success enjoyed here could be attributed to high compatability between the source and target machine instruction sets.

An effort (Olsen,1965) which was successful despite some major differences between the source and target machines was the conversion of Philco 2000 codes into equivalent IBM 7094 FAP programs. This translator accepted both binary and assembly language (TAC) programs as input; binary programs which did not have a corresponding TAC deck were disassembled to create one. Translation to FAP was done symbolically. The binary decks were used to map all symbolic references to absolute addresses in order to define the program flow and detect data usage and external references. The translator would map the Philco 48 bit word into two IBM 7090 36 bit words when necessary. All 48 bit data were mapped directly into 36 bit words at the expense of computational precision. Conversion of indexed jumps, hollorith data, and data tables had to be done manually. Approximately 98 percent of 4600 assembly or binary statements were translated correctly. This efficiency is quite encouraging; however, the sample was somewhat restricted in that these programs were machine coded subroutines for scientific applications and probably

involved considerable arithmetic expression computation and relatively straightforward machine language programming.

Several attempts have been made at symbolic translation from the assembly language of one machine to that of another. One approach used by Dellert (1965) to translate IBM 7090 code to that of the IBM 7040, was to supply a set of macros to convert the incompatible instructions to equivalent IBM 7040 sets of instructions or calls to the appropriate subroutines if I/O is involved. This translator handled 85-90 percent of the required translation. It was noted by Dellert, however, that the success of this approach, like that of Honeywell's LIBERATOR, was due to high instruction set compatability. The IBM 7090 and IBM 7040 have the same word size and similar instruction subsets.

Meta-assembly (Graham,Ingerman,1965) is another approach to symbolic translation between assembly or machine codes. The idea here is to have a generalized assembly program in which the input and output rules are supplied. The input and output of this assembler are considered to be streams of bits. These streams are subdivided into "lines". The translation rules describe how lines of the input are mapped into corresponding lines of output. For reprogramming purposes the input and output would be the machine or assembly languages of the source and target machines

respectively. Graham and Ingerman (1965) describe a meta-assembler specifically designed for reprogramming, which was being implemented by Westinghouse Corporation. The final results of this project were not reported.

Some instances of direct machine to machine translation have been commercially successful. Despite this, however, the technology of direct machine to machine translation appears to have a limited future. The resulting translation is as equally machine dependent as the original, which results in a separate translator having to be written each time conversion to a different machine is necessary. Furthermore, the implementation cost of such packages appears to be prohibitive unless either the two machines have very similar architectures or the scope of the application (i.e. domain of source programs) is sufficiently restricted.

Previous Decompiling Efforts

The term "decompiling" was first coined in connection with a decompiling project at the Navy Electronics Laboratory (Halstead,1962). Maurice Halstead, Herman Englander, and Joel Donnely demonstrated the feasibility of decompilation by implementing a decompiler to translate machine code for the Remington Rand Univac M-460 computer into an extended version of Neliac (D-Neliac). Their first

decompiler was operational in the summer of 1960. Subsequent decompilers were written for other CDC and Univac machines as well as the IBM 709X series of computers. These decompilers did not in general achieve 100 percent translation. It is shown in a later section that this is an infeasible goal. Up to 98 percent translation was ultimately realized by some of the Neliac decompilers. The first reference to decompilers in the open literature is found in the book Machine Independent Computer Programming, published in 1962.

These Neliac decompilers processed machine language programs in object deck format. Only the entry point and the extent (in core) of the program were required. It was assumed that the entire program (all subroutines etc.) was contained in the input to the decompiler. Once the instruction blocks and data areas were found, the data areas were flagged according to data type (arrays, simple variables, initialization, etc.) and a Neliac "noun list" or set of data declarations was generated. Subsequently, translation rules were applied for mapping combinations of the machine language instructions into Neliac statements.

The Neliac family of decompilers serves as the most successful model of decompilers found in the literature. However, another decompiler which attained moderate success is described by Sassaman (1966). This decompiler translates

IBM 709X assembly language (i.e. MAP,FAP) into Fortran. Several restrictions were placed on this decompiler which greatly eased the decompilation analysis. No attempt was made to handle indirect addressing or self-modifying code. In addition, the decompiler did not attempt to translate into Fortran those things which Fortran was not designed to handle such as bit handling and partial word processing. Although these restrictions seem severe, the population of programs to be translated were primarily engineering and scientific applications involving algebraic algorithms, which did not require the above capability. Since the input was symbolic text and no code modification was allowed, the instructions and data are easily identified by a linear scan of the text. The major emphasis was made in the translation of arithmetic expressions and iterative loops and in providing the user with the ability to edit the resulting translation to correct discrepancies. Like the Neliac effort, no attempt was made to realize total translation in general.

IBM's "Autocoder to Cobol Conversion Aid Program" (ACCAP) (IBM,1967) is another example of a commercial decompiler. Like the previous examples, complete translation is in general not attempted. This translator produces in effect a carbon copy of the original Autocoder program. However, since core to core moves are allowed in the source machines

(i.e. IBM 1401/1440/1460/1410/ and 7010), the direct mapping does not involve mapping intermediate loads and stores to and from registers as is done in scientific machines. In typical Autocoder programs much time is spent moving data fields and sorting because these programs are business data processing oriented. Usually only elementary computations are performed; trying to simplify expressions, therefore, would not be excessively fruitful. In fact floating point computations are not even converted. If the original Autocoder programs use IBM IOCS, ACCAP converts the I/O to the equivalent IBM/360 counterpart. Again, however, a direct mapping is done, often yielding inefficient results. Much consideration is given to providing elaborate cross referencing between the original Autocoder and the resulting translation in order to provide the user with ample documentation for manually completing and refining (optimizing) the translation. The description manual (IBM,1967) outlines a number of limitations of ACCAP. For example, address modification is not handled except for storing an address in a jump instruction, and conversion of subscripted references is not guaranteed. In conclusion, the biggest problem with this package would appear to be the inefficiency of the resulting translation, and the restrictions of the translation rules. The one to one mapping plus the inefficiencies of typical Cobol compilers result in the Cobol program occupying an average of 2.1

times the core storage of the original program. No figures are available on the percentage of code translated. One would expect a considerable amount of manual optimization to be necessary before the program would be ready for production. Since code modification is not permitted in general, and the input is symbolic, no attempt is made to analyze the program globally via flow analysis.

A recent and interesting decompiling project is the "PILER" system (Barbe, 1970). This system is much more ambitious than any of its predecessors in that it attempts to provide translation for a large (not universal) class of source-target language pairs. To achieve this, a machine dependent interpreter is written for each source machine which translates the source machine instructions into a general intermediate "micro-form" text. The bulk of the decompilation analysis is performed on this text resulting in the generation of a higher level intermediate text similar to those employed by compilers. A language "converter" is then called to process this text and generate statements in the desired target language. Flow analysis, loop analysis, and data analysis are performed on the input program (micro-form text). Like other efforts, total translation is not always possible and communication is provided to the user via a flow chart which describes the program in terms of its logical instruction blocks as

determined in the flow analysis. The user can manually alter the flowchart at intermediate points of the translation. The project is still in the research and development stage; no performance figures have yet been given.

The above approach deserves some discussion. First of all, the micro-form intermediate text must be general enough to handle the description of many different, possibly unknown, instruction sets. Thus, the resulting micro-form text is often at a lower level than the original, since frequently several micro-form instructions must be generated for one macro machine instruction, resulting in a loss of information. Using this approach worst case would require that the micro-form code be capable of simulating the macro instruction. Since the micro-form instruction repertoire is so general, the analyzer must examine many options and recombine groups of these instructions in order to generate the intermediate text at a higher level. Also, it is not clear that the higher level text is suitable for translation to any compiler language. For example, the intermediate text of a Fortran compiler would presumably be different from that for a block structured language such as Algol or PL/1. Perhaps more desirable would be a "decompiler generator" system, which given a description of the source-target pair, would produce a tailored decompiler for that

pair, thus obviating the excessive overhead of generalized translation for every program processed. It is the opinion of this author that more theoretical research is needed to understand the basic principals of decompiling before attempting to develop a more general system.

In spite of the objections raised above, the concept of an intermediate text proves to be quite useful and is used in this study, although with a different rationale. While the intermediate text developed here may provide a basis for translation of more than one machine language, its primary function is to provide an abstraction of the original program which is amenable to program analysis and reorganization. One key distinction between the text developed here (IMTEXT) and that of the PILER micro-form code is that of level. The source to IMTEXT mappings are generally one to one. The properties and operators of IMTEXT are described in chapter 3.

CONSIDERATIONS IN DECOMPILATION

The results of the decompiling efforts described in the previous section suggest that decompiling is in general an incomplete process. While it is theoretically possible to decompile an arbitrary program, assuming the entire program is available and all data dependencies are resolved,

it is generally conceded that it is economically infeasible to do so.

Perhaps the main problem is that the technology in decompiling has not been sufficiently developed. What is needed is a more general approach which can be employed to translate arbitrary sequences of machine instructions into a more abstract representation which would be suitable for translation into a reasonable target language. The approach taken in the past for handling translation has been to classify the most common types of code sequences (e.g. arithmetic expressions) and provide the appropriate translation rules. Source code sequences which violated the translation rules of a given decompiler required manual translation. If it was learned by experience that a particular situation occurred frequently enough, then the decompiler could be extended to handle it as another "special case". If the above approach is attempted to achieve total translation, then due to the vast number of instruction sequence combinations, the number of "special cases" would become <u>very large</u>, and an exorbitant number of translation rules would have to be implemented. Such an approach would not be economically sound.

Another factor affecting the degree of translation is the target language. If it can easily express many of the machine functions (e.g. shift or mask), then a low level

translation (approaching one to one) can be done for a small instruction sequence (even 1) in cases where the high level translation rules fail. One might conjecture, however, that as the level of the target decreases, then the machine dependency of the resulting translation tends to increase. In other words one is sacrificing the level of abstraction of the result in order to achieve a more complete translation.

A decompiler is written to translate machine language programs for a specific machine M to a specific target language T. Thus, given an arbitrary machine language program P(M), the decompiler, D(M,T), must translate it into an "equivalent" program P(T). For reasons discussed above one would expect that given a practical decompiler D(M,T), then legal programs P(M) could always be written which would not comply with the translation rules built into the decompiler.

The Target Language

One of the basic considerations in writing a decompiler is the target language. As was shown in the description of the Sassaman decompiler, if the target language is too restrictive, some instructions of M may be untranslatable except by direct simulation of the instruction. In these instances the machine language is in a sense a "higher

level" language than the target language. Neliac proved to be a suitable target language for several reasons. Neliac was a self-compiler and was therefore easily extended to accommodate desirable features necessary for decompiling, such as bit handling and indirect addressing (Halstead, 1967). Furthermore, being a self-compiler, Neliac could easily be bootstrapped to run on and generate code for the target machine in order to recompile the decompiled program for the new system. Neliac also allowed for computation involving program labels and absolute addresses, which simplified mapping the machine language into Neliac. The level of the decompiled code was low, but it was largely machine independent, thus satisfying the goal of program transferrability.

The question arises: what features does a language have to have in order to be an "ideal" target language? Involved in answering this, of course, is the goal sought by decompilation. If program conversion is the only goal a language like Neliac might be close to ideal. If documentation is the aim, however, one would like to decompile to as high a level as possible in order to expose the logic of the program.

A suitable target language should permit decompilation to various levels of translation. For example, the first version may produce a low level of translation of P(M),

while subsequent versions would produce successively higher levels as the sophistication of the decompiler increased. The language should be flexible, allowing for a variety of data structures and data types; also, the scope of the language should be broad enough to allow functions common to machine languages, when necessary, such as bit and partial word handling. Hopefully, an ideal target language would be extendable in order to incorporate convenient constructs which were not considered a priori to the language selection. Unlike Neliac, most commonly used languages lack this capability.

The goal may be to translate programs written for a sequential machine to run efficiently on a parallel or pipeline machine. Such a goal may require a "very high" level language such as "Aiken Dynamic Algebra" (Noonan,1971).

Perhaps this "ideal" language has yet to be developed. Therefore, the decompiler designer must choose his target language based on his goals, and the practical constraints of his computing environment. PL/1 was chosen as the target language for this research for several reasons. First, PL/1 is representative of current advanced algebraic languages and the problems encountered may be generally relevant. Secondly, PL/1 is a large language and one would expect a rich selection of translation rules. It will be

seen that PL/1 has some deficiencies for certain kinds of decompilation.

Some Difficult Problems

Since total translation of P(M) to date has been considered economically infeasible, it is of interest to investigate some of the difficulties.

Self-modifying Code/Separating Data From Instructions

Clearly one such problem is that of self-modifying code. Self-modifying code makes the task of separating instructions from data more complex, because data locations are usually determined by recording all the data references (loads and stores) within the program. Checks must be made to determine whether any of the data references occur within code segments. If so, these locations are flagged and further analysis is necessary to achieve the proper translation.

In a worst case situation where the program structure is time dependent, translation of self-modifying programs may require a simulated execution of the source program. This approach was viewed by Opler (1962) as dynamic translation. He used this approach in his translator, incurring large implementation cost; the results were marginal. Every effort should be made to achieve static

decompilation; that is, the analysis of only the original program structure.

Fortunately, some common uses of self-modifying code can be analyzed statically. For example, the store of a return address in a subsequent jump instruction is handled easily. Code which modifies the address part of an instruction reference can often be detected as a type of array subscripting.

The problem becomes more interesting when considering self-modifying code in general. If it is assumed that modified instructions do not subsequently alter other instructions, then a general solution in the context of static decompiling should be possible. Further discussion of self-modifying code is given in chapter 8.

Indexed Jumps

In order to discover the relationships between instructions and data, sophisticated decompilation requires a global flow analysis of the program. This implies finding all the control flow paths incurred because of transfer or jump instructions. Clearly the transfer locations of an indexed jump instruction are not readily detected. This problem involves determining the possible values of the index either heuristically or analytically. This will be discussed in depth in chapter 2.

Idiomatic Expressions and Programmer "Tricks"

Gaines (1965) defines an idiomatic expression as "a sequence of instructions which form a logical entity, and which cannot be derived by considering the primary meaning of the instructions". For example, incrementing the exponent of a binary floating point number to effect multiplication by powers of two is an idiomatic expression. The problem here is to recognize and classify these frequently used idioms. Considered in their context, they can usually be translated unambiguously.

The difference between a programmer trick and an idiom is primarily the frequency of usage. When a programmer is trying to optimize a section of code for either time or space, he may use the instruction repertoire in a nonstandard way to save a few machine cycles or words of core storage. The "tricks" used here usually are specific to the particular program and lack the generality of an idiom. These procedures frequently employ self-modifying code and call for manual translation in decompiling in order to produce an efficient translation.

Hardware Dependencies

One major factor to consider is whether or not the output of the source program is hardware dependent. For example, if a compiler is correctly decompiled and executed on the

target machine, it would still produce code for the original machine. Some programs, such as hardware diagnostic routines, are completely hardware dependent and their decompilation for conversion purposes would not be a consideration. Also, such things as I/O and character conversion require special consideration. A source machine language program which operates under a primitive operating system may do its own buffering for I/O, while in the target language this might be handled automatically. In order to achieve the best translation the decompiler would have to translate the I/O of the source program into the much higher level I/O statements of the target language. While this may be possible most previous efforts have relied on manual conversion of I/O.

The storage structures of data in the source machine are another vital consideration. In direct machine to machine translators much effort was expended in determining how the storage elements of the source machine would be mapped into the elements of the target machine. In decompiling, the objective is to abstract the storage structure to a machine independent data structure. The storage structure of the recompiled program in the target machine will probably be different. For example, if the word lengths differ, the precision of computations may be affected, as was seen in the Philco 2000 to IBM 7094

translator. Another example is that the layout of arrays may be altered. However, the same differences might occur if a program written in Fortran were moved between the two machines. Differences in hardware capability are always a consideration, regardless of the language of the original source computer program.

Decompiler-User Communication

Since practical decompilation is an incomplete process, and in some cases even an erroneous one, it is vital for the decompiler to interface well with the user. This problem is not conceptually difficult; however, it requires careful consideration in the design stage of any decompiler. This was brought out in all the previous decompiling efforts. Diagnostics should be stated whenever the translation is doubtful or incomplete. If possible, the names in the translated program should be correlated with those of the original program. The user should also be given the ability to make changes to intermediate results. Depending upon how exhaustive the decompiler is, it should be able to interface with the user in order to request additional information as needed, such as the ranges of data dependent variables. In short, the decompiler-user interface should facilitate the flow of relevant information, when necessary, in order to achieve complete translation as efficiently as possible.

The Economics and Efficiency of Decompiling

In regard to the economics of using decompilation for automatic program conversion, the implementation cost of producing a decompiler along with several other factors must be considered. The cost of implementing a decompiler which will (in general) translate some X percent of the code, plus the cost of completing the decompilation manually (100-X percent) for the total population of programs must be weighed against the expense of total reprogramming.

Halstead (1970) defines the economic success of a decompiler in terms of a "figure of merit" which is defined as the percent of the reprogramming costs which have been eliminated. He cites experience to the effect that, given a decompiler which translates some X percent of the code, the amount of effort needed to extend the decompiler to handle one-half of the remaining code is equal to the implementation effort already expended. For example, from data available from the Lockheed decompiling effort, it has been estimated (Halstead,1970) that if 8 units of effort are required to implement a decompiler which translates 92 percent of the source code automatically, then there is still 40 percent of the reprogramming work left to be done manually. If 8 additional units are spent on improving the decompiler then it will translate 96 percent of the code and 24 percent of the conversion effort must be done

by hand.

In Neliac decompiling efforts, it was found that over 98 percent of the code was converted, and it was estimated that decompilation eliminated only 90 percent of the reprogramming work. Sassaman (1966) states that 90 percent of the code was converted to Fortran automatically by decompilation, but, of course, his figure of merit would be considerably less.

The efficiency of decompiled programs is most easily expressed by a percentage of increase of core (in comparable units) over the original. Comparison of execution speeds between the original program and the decompiled program is difficult because of the many diverse characteristics between the source and target computing systems. It should be recognized, however, that if the decompiled translation is not tuned for the configuration of the target machine, then exceedingly poor execution efficiency may result. This was brought out in the discussion of the ACCAP translator. The Neliac decompiled programs realized an average core increase of one-third. The mean increase of core storage for the IBM's ACCAP converted programs was 110 percent over the original with a range of 10 to 210 percent for the sample tested. Of course, several factors affect this figure such as the level of decompilation, the target language, and the efficiency of the target language's

compiler on the target machine. As one might expect the execution time required for the actual decompilation process is much greater than the time customarily required for the compilation process. It has been estimated (Halstead, 1970) from experience that decompiling requires a factor of 50 more execution time than a one pass compilation for a fairly large program. However, this is a minor consideration when one considers that a program is decompiled only once.

OVERVIEW OF THIS RESEARCH

The philosophy of using decompilation as a program conversion tool is that of mapping the machine language up to a less machine dependent representation in some target language and then recompiling or mapping the result down to the target machine representation. Interestingly enough this same approach is used in the decompilation process itself as depicted by figure 1.A.

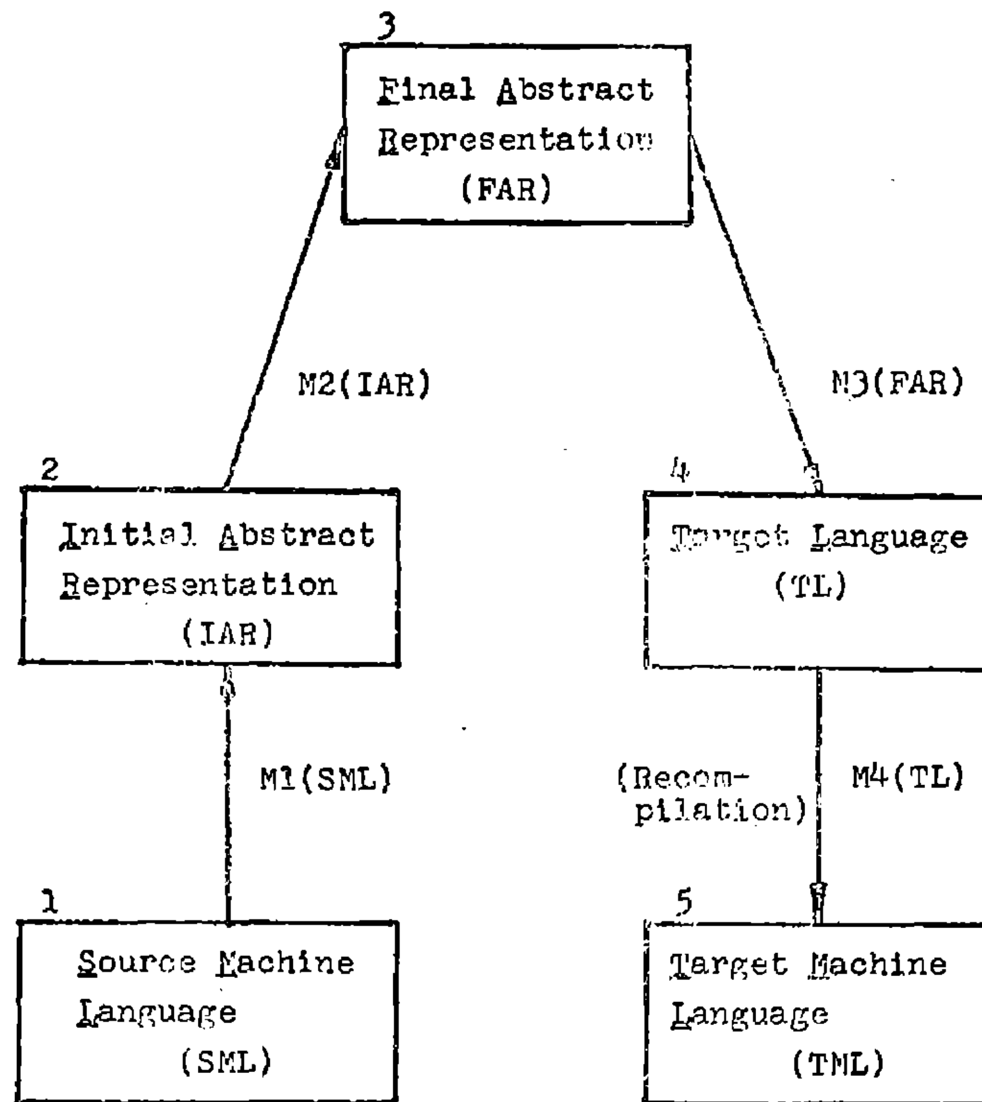


Figure 1.A - The Decompilation Process

The abstract representations in blocks 2 and 3 consist of an intermediate text representation of the instructions and data in conjunction with the control flow graph of the program. The mapping M1(SML) involves separating data from instructions, forming the control flow graph, and generating an initial abstract representation of the instructions and data. M2(IAR) concerns applying program analysis techniques in order to detect the data structures and simplify and reorganize the program. The decompiler described in this study can be viewed as consisting of the mappings M1, M2, and M3.

In order to gain insight into a limited number of the more interesting problems of decompiling and to demonstrate the feasibility of the proposed solutions, a decompiler was implemented. The chosen source and target languages of the decompiler are the MIX assembly language (MIXAL) and PL/1 respectively. MIXAL is the assembly language for the MIX machine developed by Knuth (1969) for pedagogical purposes. Several factors contributed toward choosing MIXAL. By design, MIX has many of the features of typical second generation machines, thus providing a fairly general representative machine language for typical decompiling applications. Also, the language should be fairly well known because of the widespread distribution of Knuth's book: <u>Fundamental Algorithms Vol. 1.</u> The MIX assembly

language was chosen instead of the machine language as a matter of convenience in test case preparation, and to illustrate some of the documentation benefits of decompiling by the correlation of the MIXAL symbols of the input program with the generated PL/1 symbols. In general the results developed in this thesis are applicable for decompiling either machine or assembly languages.

Major Phases

The major decompiler is divided into three separate phases:

1. MIXAL partial Assembly – the input is a MIXAL program and the output is a partially assembled text and symbol table. All symbolic addresses are mapped to their equivalent machine addresses, however, the opcodes are left in symbolic form. It is necessary to map the symbolic text to the address space of the machine in order to separate data areas from instructions. This is also necessary for the detection of specific data storage structures within the data areas. Once these storage structures (e.g. linear array) are found and classified, they can be translated into equivalent PL/1 data structures (i.e. data declarations).

2. ANALYZER - this program reads the partially assembled text generated in phase 1 and ultimately produces an intermediate text in the form of tables and 3-tuples suitable for translation into the target language (PL/1). This phase constitutes the bulk of the decompiler, and the description of the algorithms used therein comprise the major part of the thesis. Separating data from instructions, loop analysis and data analysis are primary functions performed by ANALYZER.

3. PL1GEN - this phase reads the tables generated by ANALYZER and produces syntactically correct code for the IBM PL/1-F compiler. Some simplification is done in this phase to combine statements and reorganize the program.

CHAPTER 2

DETERMINING THE CONTROL FLOW GRAPH

SEPARATING DATA FROM INSTRUCTIONS

Decompilation basically involves the analysis of the data and instructions of the source program and their interrelations. Therefore, given the memory extent and entry point, the initial function in the decompiling process is to analyze the original source program in order to identify the data and instructions. Where self-modifying code occurs, a core location serves as both. Furthermore, to effect sophisticated decompilation analysis it is necessary not only to identify the program's instructions, but also to determine how the instructions are related in their execution sequences (i.e. the control structure). The precise role of the control structure in the analysis will be made clear in the discussions of some of the algorithms. Generally the control structure is used in analyzing the program in a global way in order to discover various characteristics of the program's data and instructions.

A linear sequence of instructions contains a sequence of instructions which occupy contiguous locations in core memory. Starting with the first instruction these instructions will be executed sequentially unless a JUMP instruction causes a transfer to a different instruction sequence of the program. If this occurs, the program execution is said to have taken a different control flow path. To determine the control structure of the program it is expedient to partition instructions into disjoint linear sequences called instruction blocks. These blocks are defined in such a way that given any two blocks B1 and B2 in a program, the condition must hold that either the execution of B1 (i.e. the execution of instructions in B1) does not necessarily imply the immediate execution of B2, or the execution of B2 does not necessarily imply the immediate execution of B1. These blocks can be viewed as the nodes of a directed graph (i.e. the control flow graph) where the directed arcs between the nodes denote the possible control flow paths of the program.

Detecting these blocks and constructing the control flow graph involves scanning the instructions starting with the entry point and inductively tracing the control flow paths until all paths (and instructions) have been found. Some difficulties arise in cases where indexed jumps occur or where the transfer address of a jump instruction is modified

by the program.

The partitioning of the program into instruction blocks enables the control structure of the program to be determined, and it facilitates the analysis of the sequences of instructions which will be coalesced into single statements of the target language. Except for the jump instruction(s) at the end of the block, these sequences will always be linear sequences of machine code such that if one instruction in the sequence is executed, then all the instructions in the sequence are executed.

In the following sections, some definitions and concepts concerning instruction blocks are developed. Then, the method for block detection is discussed.

Instruction Blocks

In machine language programs the number of blocks can be very large in relation to the size of the program, and many of these blocks may contain only a few instructions (1 or more). It is desirable to minimize the number of blocks in order to reduce the table storage and increase the execution efficiency of the decompiler. Past experience has shown that the execution time in decompilation is approximately proportional to the number of blocks raised to the 1.5 power. Thus, it is desirable to define these blocks such that the number of instructions in each block

is maximized (therefore minimizing the number of blocks), while still maintaining the integrity of the control flow graph. This maximization is accomplished by defining a block so that it may be terminated by a maximal jump instruction sequence subject to certain constraints.

An instruction block (IB) consists of a linear sequence of instructions (Ij; j=1,...,n). It should be noted that detecting instruction blocks consists of scanning the instructions and data contained in an internal array P, which is the output of phase 1 (MIXAL assembly). The instruction sequence of IB is partitioned into two sets: NJ(IB) (Ij; j=1,...,k) and J(IB) (Ij; j=k+1,...,n). NJ(IB) is a sequence of non-jump instructions, and J(IB) is a sequence of jump instructions. It is possible for either NJ(IB) or J(IB) (but not both) to be null. The first instruction in the block is called the block entry point (BEP). With the above concepts an IB can be defined in its entirety. An instruction block (IB) is a linear sequence of instructions with the following properties: a) If NJ(IB) is not empty, then if any instruction in NJ(IB) is executed then all the instructions in NJ(IB) are executed. b) If J(IB) is empty then the last instruction in IB precedes the entry point of another block.

Frailey (1971) presents the concept of logical and

physical successors and predecessors for control flow analysis in program optimization. Adapting this concept for decompiling gives rise to the following definitions. Given two instruction blocks IB and IB', IB' is a physical successor of IB if IB' contiguously follows IB. Conversely, IB is a physical predecessor of IB'. IB' is a logical successor of IB if program control can pass directly from IB to IB'. Logical predecessors are similarly defined.

It is now possible to discuss inter-block relationships of the program in terms of the instruction set J(IB). If J(IB) is null, then a physical successor of IB is also a logical successor of IB. In other words, there is an implicit jump or "fall through" from IB to IB'. When a jump instruction is encountered while scanning the instructions of a block IB, the first instruction of J(IB) has been discovered. If this instruction is the first of a linear sequence of jump instructions, it is desirable to include as many of the subsequent jumps as possible into J(IB) in order to maximize the number of instructions in IB. J(IB) may contain a sequence of jump instructions subject to the following criteria. If J(IB) consists of more than one jump instruction, all but the last must be a conditional jump, and they must conditionally test the same register. Two or more sequential conditional jump instructions are said to be in the same jump category if

they test the same register. An absolute jump is considered to be in all jump categories. In MIX (see appendix A) the possible registers are I1 (index register 1) through I6, CI (compare indicator), the A (accumulator) and X (multiplier/quotient) registers, the overflow indicator, the return jump register J, and the I/O sense registers. For the computer registers I1-I6, A, and X, the conditional jump is always based on a comparison between the given register and zero. Conditional jumps are based on the status of CI, which was set by a previous compare instruction.

Another consideration in deciding the inter-block relationship is in determining whether or not there is an implied jump to the block's physical successor. An implied jump may exist if there is a sequence of conditional jump instructions of the same category not followed by an absolute jump. This is handled with the notion of condition value, V, and total complement. Condition values are assigned according to the following table:

Condition:	$<$	$=$	$>$	$\geq$	$\neq$	$\leq$
V(condition):	1	2	4	6	5	3

Notice that the entries in the last three columns are sums

of the first three condition values. The condition indicators "less than", "equal", and "greater than" are appropriately set when two quantities are compared. The indicators can be thought of as a three bit variable which assumes the values in the above table for the specified conditions. There is a set of indicators for each register which can be tested. For registers I1 I6, A, and X, one of the comparands is always zero and is set and tested by conditional jump instructions. A total complement for a sequence of jump instructions is reached when a combination of all three conditions are tested. This occurs when the sum of the condition values in a sequence is greater than or equal to 7. Absolute jumps are given a condition value of 7. If the sum of the condition values in a sequence of jump instructions terminating a block is less than 7, it follows that there is an implied jump to the physical successor, provided that a physical successor exists. If this sum is greater than or equal to 7, then one of the jump exits must be taken, making an implied jump impossible. The above concepts are illustrated by some examples of simple MIXAL sequences.

```
1.   L1   LDA  X
2.        SUB  Y
3.        JAP  L2
4.        JAZ  L3
5.   L4   STA  T1
6.        ...
```

In the above text, lines 1-4 comprise IB^1, and line 5 is the initial instruction of IB^2. IB^2 is a physical and logical successor since both conditional jumps test the same register, A, and the sum of their condition values is less than the total complement (C(P)+C(Z)=6).

```
1.    L1      LDA   X
2.            SUB   Y
3.            JAP   L2
4.            JAZ   L3
5.            J3NZ  L4
6.    L2      STA   T1
7.    L3      LD3   LIMIT
8.            DEC3  1
...           ...
```

In the above example the following four instruction blocks are noted: IB^1 (lines 1-3 in the example), IB^2 (4,5), IB^3 (6), and IB^4 (7,...). IB^1 terminates at line 3 since the instruction at line 3 has a different jump category than that of line 4. IB^2 is a physical and logical successor of IB^1. Note also that $NJ(IB^2)$ is null. While IB^3 is a physical successor of IB^2, it is not a logical successor, since C(Z)+C(NZ)=7, which equals a total complement. Observe also that IB^3 consists of only one instruction, because line 7 is the block entry point of IB^4. $J(IB^3)$ is null, thus there is an implied jump from IB^3 to IB^4.

The primary concern is to find all the instruction blocks and all their <u>logical</u> successors (and predecessors) in order

to analyze the program's control graph. This graph is a directed graph P(N,A), where N is the set of partially ordered instruction blocks or the nodes of the graph and A is the set of directed arcs connecting the nodes of the graph. A directed arc [Ni,Nj] exists between nodes Ni and Nj if Nj is a logical successor of Ni. Once all the instruction blocks and their associated logical successors (or predecessors) have been discovered, the notion of physical successors is no longer necasarry for subsequent analyses. It is more convenient to use the terms immediate successor and immediate predecessor (Allen, 1970) in lieu of logical successor and logical predecessor respectively (i.e. Nj is an immediate successor of Ni). S[Ni] and P[Ni] denote the sets comprising all the immediate successors and immediate predecessors respectively of Ni. Henceforth, the term "instruction block" for simply "block" and "node" will be used interchangeably.

Jump Categories

The constraint that each jump instruction in J[IB] be of the same jump category is made for several reasons. Perhaps the most fundamental reason for this restriction is that in subsequent analyses (loop and data analysis) it is necessary to determine the variable upon which the exit of certain blocks depend. Multiple jump categories in J[IB] would mean that block exists would be a function

of more than one variable, making the analysis unwieldy. Furthermore, if multiple jump categories in J(IB) were allowed, the decompiler would have to keep track of multiple accumulative condition values, one for each register being tested in J(IB); this would unnecessarily complicate the analysis.

Another reason for the single category restriction on jumps relates to the generation of the target language statements, if the jump instructions J(IB) are of the same jump category, they can be analyzed as a group resulting in a single target language statement. For example:

```
JIZ  L1
JINZ L2
...
```

would result in:

```
IF REG1=0 THEN GO TO L1; ELSE GO TO L2;
```

THE BLOCK DETECTION METHOD

A machine language program P can be represented as an ordered set {P1,...,Pn}, where Pj (j=1,...,n) designates a word in core memory which serves as a computer instruction or datum (or both). Each element Pj is characterized by its core memory address (CMADDR[J]), an operation code or data type (OP[j]), and an operand field. If Pj is a MIX instruction, the operand consists of an address part or displacement (DISP[j]), an index register (IR[j]), and a word subfield specification (FLD[j]). Initially, only the program P, its entry point e, and its core memory extent are known. The goal of the block detection algorithm is to partition P into an ordered set of instruction blocks called the <u>program block set</u> N: {IB1,...,IBm}. Specific blocks are referenced by their position in N. Descriptive information is associated with each block which relates the block to the program topology (control flow graph), the original program representation P, and core memory (CM). The following attributes are associated with each block IBj.

a) <u>Block Entry Point</u> (EP[j]) - The CM address of the first instruction of IBj.

b) <u>Block Terminal Point</u> (TP[j]) - The CM address of the last instruction of IBj.

c) Immediate Successor List (IS[j]) - A list of block entry points of blocks which are immediate successors of block j.

d) Immediate Predecessor List (IP[j]) - A list of block entry points of the blocks which are immediate predecessors of block j.

e) First Block Instruction (FI[j]) - This is an index to the program P which references the first instruction of IBj in P. Note: the CM address of this instruction equals EP[j].

f) Last Block Instruction (LI[j]) - An index to P which references the last instruction of IBj. In the implementation, the result of the block detection algorithm is a block table (BLKTBL) with one entry for each block. Each entry contains the attributes previously described (BLKTBL[k] describes block k).

The algorithm commences by initializing a list called the unscanned block entry list (UBEL) to the entry point e. Generally, this list contains the block entry points of unscanned blocks. The UBEL receives subsequent entries when scanning the J(IB) portion of a block. All found transfer addresses (implicit or explicit) which reference an unscanned instruction are added to the UBEL, since these addresses must be the entry points of unscanned blocks.

The next block to be scanned (say block k) is determined by removing an item from the UBEL. The first consideration is to determine if this address references a scanned block. This can happen if when the UBEL entry for block k was made, there was a previous UBEL block entry (say for block m) whose instructions include the instruction corresponding to the entry point of the newly detected block. The <u>extent</u> of a block (i.e. its instructions in P) is determined when J(IBk) is found or when an entry point of a previously found block is detected. In the above situation, scanning for IBm did not terminate when EP[k] was encountered, because block k had gone undetected during all block scans prior to the scan for IBm. Now, however, it is realized that block m really consists of two blocks and, therefore, block m must be <u>subdivided</u> into blocks m' and k. Assuming EP[k] references a nonjump instruction, block m' will have only one immediate successor, block k; TP[m'] will equal EP[k]-1 and LI[m'] will equal FI[k]-1. The remaining attributes of block m' are the same as those of block m. The attributes TP[k] and IS[k] will be those of the former block, previously labeled m. In the implementation, this entails adding a new entry to the block table for block k (BLKTBL[k]) and altering the attributes in BLKTBL[m] to describe the newly found block m'.

Scanning Jump Sequences

If EP[k] is not within the extent of a scanned block, scanning for IBk commences with the instruction in P corresponding to EP[k]. Assuming no entry point of a previously found block is encountered, the scan must terminate with a halt or a sequence of jump instructions. Assuming the latter, the transfer address for each jump instruction (implicit or explicit) is added to IS[k]. The scan for instructions in J(IBk) is terminated when a total condition complement is reached in a sequence of conditional jumps, when a change in jump categories is detected, or when a halt or non-jump instruction is found.

Each transfer address in IS[k] is, by definition, a transfer address of a jump instruction referencing the entry point of a block. However, this block may still be undetected at the time of analysis. Given a transfer address TA in IS[k], three cases can occur: (1) if TA equals some EP[j], where j is some previously scanned block, no action is taken; (2) if TA references an instruction in a previously scanned block (say s) other than its entry point instruction, then block s must be subdivided into two blocks as described above; (3) if neither of the above cases occur, then TA is the entry point of an unscanned block and is entered in the UBEL, if it is not already present. TA is entered in sorted (ascending) order so that the next UBEL entry removed is the block entry point of

smallest address. This is done to minimize the length of the UBEL. This technique seems to maximize the number of transfer addresses (in IS[k], k is the block being scanned) which reference previously scanned blocks, thus minimizing the number of added entries to the UBEL per block processed. This is probably due to the principle of locality (Denning,1968). After all the entries in IS[k] have been analyzed, the current block table entry being constructed (BLKTBL[k]) is completed, and the process is repeated until the UBEL is empty. At this point the addresses (i.e. block entry points) in the blocks' immediate successor lists are converted to block numbers. This is an implementation consideration and is done because BLKTBL is a linear array. The block numbers serve as indicies to BLKTBL, allowing fast access to block information during analysis. Next an immediate predecessor list is constructed for each block. Given IS[k] to be the set of block numbers {Bq; q=1,...,m}, then k is added to IP[Bq] for all q.

In order to test whether or not a transfer address or the next UBEL entry references a scanned block (any instruction in the block) a bit string (INBITS) whose length is that of the number of words in the program is maintained. Every time an instruction (at address A) is scanned, the Ath bit in INBITS is turned on. Let INBITS[A] denote the bit corresponding to address A of the program, that is the

[I-(program load address)+1]th bit of INBITS. Thus, when a block entry point (EP), retrieved either from a jump instruction or the next UBEL entry, is being examined, if INBITS[EP] is on, then EP is a transfer to a scanned block, and the appropriate action is taken as described previously. It should be noted that addresses retrieved from the UBEL will never reference the entry point of a previously found block. This is due to the procedure for making the UBEL entries. Therefore, all UBEL addresses which reference an existing block always imply that the referenced block must be subdivided. This bit string serves as a memory map for the instructions. In another analysis a similar bit string is constructed for the data areas, assuming all the areas are self contained in the program area. By "ANDing" these two bit strings together, all self-modifying code is immediately recognized.

Indexed Jumps

An obvious but complex facet of the block finding procedure which has not been discussed is that of indexed jumps. When a jump instruction of J([B) contains an indexed reference, it is not immediately possible to determine the transfer addresses. The indexed reference implies as many transfers as there are unique values of the jump instruction's index register which can be realized at the jump instruction.

Indexed jumps generally represent a small percentage of the total jumps in a program. Knuth (1971) in his study of Fortran programs reports that 0.78 percent of the "go to" statements were of the "computed goto" type, and Halstead (1972) stated that for the Neliac compiler, 4.8 percent of the total transfers were indexed. These small percentages suggest that many programs would not contain any indexed jumps, and that perhaps it would be more economical to handle this feature manually. This approach would be reasonable, except that not handling an indexed jump properly during the block detection phase, even if there is only one in the program, may drastically distort the resulting decompiled program. This is due to the fact that failing to resolve the effective transfer addresses of an indexed jump can cause entire blocks of code to go undetected.

The handling of indexed jumps has not been implemented. However, because it is felt that this problem is of considerable importance, the problem is discussed.

To find these addresses a combination of analytic and heuristic techniques could be employed. The handling of indexed jumps is deferred until all blocks referenced by simple (i.e. non-indexed) jumps are found. The locations of the indexed jumps in a program P are recorded in a list as they are encountered during block detection.

A Heuristic Approach

Frequently indexed jumps are used to reference a jump table as illustrated by the following MIXAL sequence:

```
1.          JMP  JTB,2
2.   JTB    JMP  LBL1
3.          JMP  LBL2
4.          JMP  LBL3
5.          LDA  D1
            ...
```

In the above sequence, note that the displacement part in I1 (instruction in line 1) references another jump instruction which is the first of a sequence of jump instructions. It can be assumed that index register 2 (IR2) would have possible settings of 0,1,2, and possibly 3. Whether or not I1 can jump to I5 could be determined by further analysis; for example if I5 is not the entry point of some block, it can be assumed that I1 must jump to I5. When such a sequence is found it is treated as a jump table group. Each jump destination in the group is treated as a reference to an immediate successor of the block containing the indexed jump. These transfer addresses must be analyzed in the same manner described previously to determine if new blocks have been found or if existing blocks have to be subdivided. New blocks are processed as previously described. When the program is translated to the intermediate text, the jump table group is treated as if it were a single "computed goto" jump instruction.

An Analytic Approach

A more general solution to the "indexed jump" problem is to explicitly determine all possible values which the index register can have at the given jump instruction. An algorithm for backtracking through the flow of the program to compute the values of a non-indexed datum at a specified location in the program is discussed in chapter 5. Obviously the initial values for such computations must be available to the decompiler. They must be assigned in the program itself or supplied to the decompiler indirectly. With this approach in conjunction with the heuristic just described for detecting a jump table group, the previous example could be handled rigorously. For example, suppose the value list for IR2 reveals a range of 0 through 3. Then further anaylsis is unnecessary to determine if I5 is a transfer address of I1.

In general this technique is a converging, iterative process. Consider the control flow schematic shown in figure 2.A.

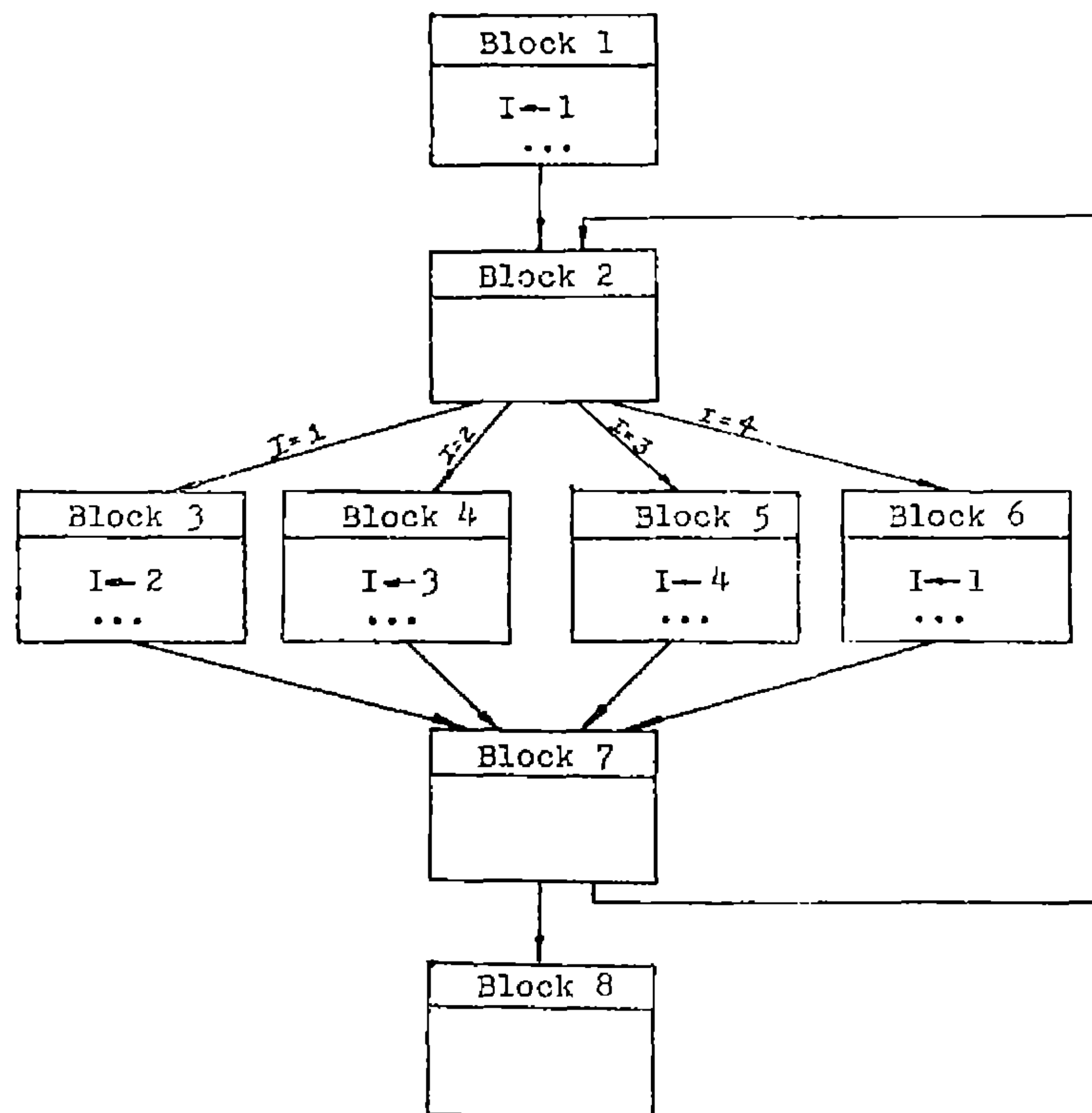


Figure 2.A - Control Flow With Indexed Jump

There is an indexed jump at block 2, which can transfer to the blocks 3,4,5, or 6 depending on the value of I as shown. However, initially only blocks 1 and 2 are known. Iteration 1 returns a value list of {1} for I. This in turn leads to the discovery of blocks 3,7, and 8, since a value of I=1 directs the transfer from block 2 to block 3. The discovery of block 3 leads to finding blocks 7 and 8 as described in the block detection algorithm. Iteration 2 returns a value list of {1,2} for I because block 3 sets I to 2, and a path from block 3 to block 2 exists; thus block 4 is discovered. The procedure continues until the value lists for iteration n and n-1 are equal.

CHAPTER 3

INTERMEDIATE TEXT GENERATION AND COMPRESSION

With most current compiling techniques the result of the syntax phase is an intermediate text which serves as a basis for subsequent code generation. In decompiling the generation of an intermediate text from the original source has also been found to be useful. The advantages of using an intermediate text are illustrated below in the discussion of some of the essential properties of the specific intermediate text, IMTEXT, designed for this decompiler study.

Property 1: All operands in the intermediate text are explicitly referenced.

That is, all "one address" instructions must be mapped into two or three operand instructions in IMTEXT. For example:

```
LDA   TWO
ADD   THREE
STA   RESULT
```

would be mapped into:

```
ASSIGN A,TWO
ADD  A,A,THREE
ASSIGN  RESULT,A
```

Example 3.8

One advantage of explicitly defining all the operands is that it is well suited for simplification (compression) of the text. In the above example, if it is assumed that the operand A is not subsequently fetched after the last ASSIGN operator (i.e. A is not busy), then the three IMTEXT statements can be replaced by:

```
ADD  RESULT,TWO,THREE
```

Making all operands explicit also provides a convenient representation for efficient interpretive execution of any segment of the source program, should it prove necessary during any of the decompilation analysis phases

Property 2: All operands are treated in a homogeneous manner.

Machine languages typically have numerous instructions for moving data between core storage and the machine registers. A hardware register often serves as a temporary work area in order to effect a computation. Such

temporaries are necessary from a hardware standpoint, but are not required to reflect the actual logic of the computation. One goal in decompilation is to eliminate all such hardware dependent temporaries. In the intermediate text all operands, whether register or core storage references in the original program, are treated similarly when trying to simplify the text. This is illustrated in example 3.A. The representation of the accumulator (A) is not differentiated from the core storage operands.

Another result from properties 1 and 2 is that the number of unique operators in IMTEXT is considerably reduced from that of the original machine language. Notice in example 3.A that LDA and STA are both mapped into the operator ASSIGN. Thus, the IMTEXT representation is an abstraction of the original program in that it preserves the original computational logic, but dispenses with the machine dependent properties involved with the assignment of data to operands. In MIX there are over 40 operators which are mapped into the IMTEXT ASSIGN operator. Similar "many to one" mappings are done with the various compare, shift, and jump instructions in MIX.

<u>Property 3</u>: The "instruction space" and "data space" of the of the IMTEXT representation are disjoint.

In the original program all instructions and data reside in the same linear address space (i.e. core memory). This representation makes it difficult to perform transformations on the program structure. As will be shown in later sections it is sometimes expedient to add and delete instructions and data and to reorder the physical placement of instruction blocks. The conversion of the source to IMTEXT is performed on an instruction block basis. The translation involves examining the instruction blocks (IB1,...,IBn) of the program in order. For every block each instruction in its linear sequence of instructions (of the original program P) is translated to the appropriate IMTEXT instruction which is then stored in an array (IT). The result is that all the instructions in IMTEXT form a linear sequence: IT[1],IT[2],...,IT[M], such that IT[K] is always adjacent to IT[K-1]. This is convenient for scanning the text during analysis.

During the translation to IMTEXT, the source program data references are analyzed and recorded in the appropriate operand data tables. An operand in an IMTEXT instruction is represented by data table pointer. By separating the instruction table (IT) and the data tables, independence of the instruction and data space is achieved. Now it is possible for the instruction text to be simplified, and reorganized without altering the relationship of the

instructions to their data.

Property 4: All operands are represented by a single unit in IMTEXT instructions, namely a pointer to an entry in an operand table.

In addition to the advantage described in property 3, having operands represented in this way simplifies the text. For example the MIX instruction:

```
ADD  T,6(2:3)
```

would be translated to an IMTEXT instruction of the form:

```
ADD  A,A,B
```

where A and B are operand table pointers designating the accumulator and the memory reference "T,6(2:3)", respectively. Often it is desirable to test for equality of operands, such as in the simplification demonstrated in example 3.A. This test is performed efficiently since it only involves comparing atomic entities (operand pointers).

Property 5: The order of any two instruction blocks within the IMTEXT instruction array (IT) is independent of their order in the original source program P.

In other words, regardless of the physical order of the instruction blocks within IT, the logical control flow of

the program is preserved. The block finding algorithm operates in such a way that the linear instruction sequences of two blocks, say IB_2 and IB_8, may be physically adjacent in the original program P. In the generated IMTEXT translation the instructions for blocks 3,...,7 would be generated between those for blocks 2 and 8. If there was an implied jump from IB_2 to IB_8 in P, direct translation to the intermediate text would be erroneous. To prevent this all implied jumps are made explicit by adding an absolute jump to every instruction block sequence which terminates with an implied jump. This technique permits the instruction blocks to be translated into the target language in any order. As will be shown the translation of the IMTEXT to PL/1 involves reorganizing the instruction blocks in order to produce a more readable, higher level translation of the original program. All redundant "jumps" introduced in IMTEXT are removed during the IMTEXT-PL/1 translation.

OVERVIEW OF THE TRANSLATION PROCESS

IMTEXT consists of an instruction table (IT) and various operand and data tables. Each entry (k) in the instruction table contains the instruction's block number (IT.BN[k]), operation code (IT.OPC[k]), and the operands (IT.N1[k],...,IT.N3[k]).

In discussing the IMTEXT operands it is convenient to introduce the notion of <u>storage structure operand classes</u>. Just as there are different data types in programming languages, one can also classify operands at the machine level. For the MIX subset considered in this study, the storage structure classes consist of immediate constants, simple (i.e. not indexed) and indexed core memory references, and simple and indexed transfer addresses. Because the data classes are treated differently, separate (physically or logically) data tables are provided to record the occurrences of each class in order to allow efficient operand processing. When the source text is translated into IMTEXT, the operand tables reflect the machine dependent storage structure of the program data. As more is learned about the program in subsequent analysis, the operand tables are augmented to reflect the implied machine independent data structures.

Initially the source text is scanned to determine all initialized (assembled constants) data areas. The data type, value, and core address of each initialized memory cell are recorded in an "Initialized Core Memory Table" (ICMT), which is used later to determine the initialized operands. This scan is straightforward, if the assembly language text is available, since it only involves detecting the appropriate assembly data declaration (e.g. ONE CON 1).

If only the object text is available, a search must be made to determine all initialized locations within the object text which are not contained in an instruction block.

After the initialized memory locations have been tabulated, the instructions are translated into the IMTEXT representation. The translation of every source instruction involves mapping its op-code into the appropriate IMTEXT op-code and then processing all the operands. For each operand to be translated the storage class of the operand is determined by its context in the source instruction. Then the operand table corresponding to the storage class is scanned to see if a previous instance of the operand has already been recorded, in which case a pointer to the existing entry is returned as the value for the operand field in the IMTEXT instruction (some IT[j]). If no match is found, the operand is entered in the table and its pointer returned as the IMTEXT operand value. If the operand is a simple core memory reference, the type of access (fetch or store) is recorded in the operand's table entry. For every simple operand entered in the simple operand table (SOT), a scan of the ICMT is performed to see if there is an initial value for the operand. This consists of comparing the core memory address and field specification of the SOT entry for the operand to those in the ICMT entries. If there is a match then an initial

value for the simple operand exists and a pointer to the initial value (an entry in ICMT) is stored in the table entry for the operand. After the translation is completed, if only the "fetch" status is recorded, the operand is in essence a literal in the source program and can be used as the operand value in subsequent analysis computation (interpretive execution). In any case the initial value may be used as the argument for the PL/1 "initial" attribute when the declaration for the operand is generated in the PL/1 target code.

The order in which the instructions are translated is determined as follows. As mentioned previously the instructions are translated on a block basis (IB1,...,IBn, where n is the last entry in BLKTBL). The extent of the source instruction sequence of a block, k, in P is given by the fields: BLKTBL.FI[k] and BLKTBL.LI[k], which point to the block's first and last instruction respectively in P. The instructions are translated in linear order for each block. Each translated instruction is stored in the next sequential location of the IMTEXT instruction array IT. After a source program instruction block has been translated, the fields BLKTBL.FI and BLKTBL.LI are updated to reflect its first and last instruction, respectively, in IT. The core memory extent of the block is still maintained in BLKTBL so that its source instructions can

be located (less efficiently) if necessary.

Once the blocks initially recorded in the block table (i.e. those receiving control from <u>simple</u> jumps) have been translated, the indexed jumps may be analyzed. Any newly found blocks which previously had gone undetected can then be translated.

<u>General IMTEXT Translation Algorithm</u>

The source (MIX) to IMTEXT translation on an instruction block basis is summarized in the following steps.
(Note: In the presentation of algorithms, a sequence of minor steps provides a more detailed description of its associated major step.)

A. Find all initialized memory locations, and enter them in the ICMT.

B. Translate all blocks initially recorded in the BLKTBL prior to analyzing any indexed jumps.

C. Get next indexed jump from the indexed jump list (IJL -created during the first block finding pass). If none then <u>TERMINATE</u>. Note: Entries in the IJL are pointers to the incomplete translation of the jump in IT.

D. Find all immediate successors of the block containing the indexed jump.

.1 Let k be an iteration index.

Let Ak be the set of jump addresses computed in the kth iteration.

Initialize: Ak←{null}, k←1.

.2 Compute Ak (described generally in chapter 2). If Ak=A(k-1), go to C.

.3 If all m in Ak are entry points to previously scanned blocks (entries in BLKTBL), go to C.

Enter all m in Ak which do not reference a previously scanned block in the unscanned block entry list (UBEL) and invoke the block finding algorithm described in chapter 2.

.4 Translate any newly discovered blocks if they have not already been translated (note: If a previously found block had to be subdivided, as a result of step D.3, its instructions would have already been translated to IMTEXT representation).

.5 k←k+1, go to D.2.

IMTEXT DESCRIPTION AND MIX-IMTEXT TRANSLATION

The next several sections describe in some detail the essential components of IMTEXT and some of the MIX-IMTEXT translation rules.

Operand Tables

The following is a partial description of the operand tables used in IMTEXT. Only those fields necessary for understanding the MIX-IMTEXT translation and simplification are described. Other fields will be described as needed.

Simple Operand Table (SOT)

This table contains entries which describe all registers, and simple core memory references. Each SOT entry contains the following fields:

SOT.LOC - Core memory address (or register number) of the operand

SOT.FLD - Field Specification of the operand (e.g. (2:4)).

SOT.INL - Pointer to ICMT entry if operand had an assembled initial value.

SOT.AC - Access (fetch and store) indicators.

Indexed Operand Table (XOT)

This table describes indexed core memory references, where the effective memory address, M, is computed as:

$M \leftarrow \pm AA + C(IDX)$. (where C means "contents of").

XOT.AA - Address part.

XOT.IDX - Index register part.

XOT.FLD - Field specification part.

Immediate Constant Table (ICT)

This table contains the values of the immediate constants used in "immediate" instructions.

Jump Address Table (JAT)

This table records all "jump" instruction operands.

JAT.IDX - Index register of jump instruction (if any).

JAT.TAL - List of Transfer addresses; if JAT.IDX is zero, this list contains only one entry, and therefore implies a simple jump instruction.

MIX-IMTEXT Translation Rules

The translation of the MIX machine instructions into

IMTEXT is generally straightforward. It involves decoding the MIX instruction operation code and selecting the appropriate IMTEXT skeleton in order to complete the translation of the operands. The operands are then decoded based on the context of the source instruction. Each operand is classified and passed to an "operand routine" which processes the operand and returns a pointer to the appropriate operand table entry.

IMTEXT Instruction Format

Most IMTEXT instructions recorded in the instruction table (IT) have the form:

<IT.OPC> <IT.N1> <IT.N2> <IT.N3>

which implies:

<IT.N1> ← <IT.N2> <IT.OPC> <IT.N3>

The main exception is the "ASSIGN" operator which has the form:

ASSIGN <IT.N1> <IT.N2>

Nomenclature for Describing MIX Instructions

AA - used for a general representation of the address

field.

F - a general representation of the field specification.

Ij - representation of index register j(j=1,...,6).

i - general designation of a register in the MIX opcode. The meaning of "i" is dictated by the MIX opcode in which it occurs.

If a symbol is omitted, it is null, and irrelevant in the computation (e.g. the use of an index register in computing an effective address).

Nomenclature for Describing the IMTEXT Translation

The symbols A, X, Ij, Ri, and CI designate entries in the SOT which correspond to the MIX A-register, X-register, index register j, the opcode register "i", and the compare indicator respectively. The symbol AX designates a SOT entry corresponding to the MIX "long" register when the A and X register are used in combination. Note that the compare indicator is treated as a simple operand whose value indicates the result (i.e. $<,=,>,\geq,\leq,\neq$) of the "compare" (CMPi) instruction. Other operands will be designated by the operand table name followed by a bracketed list of items from the MIX instruction which are included in the operand's table entry. (Notation: The format "A => B" is used to

mean: "the MIX instruction A is translated into the IMTEXT instruction B".)

The following section serves to illustrate some of the representative MIX-IMTEXT translation rules as well as discuss some of the more interesting mappings. The heading preceding translation rule, is a description of the MIX operation code. The IMTEXT codes should be largely self-explanatory.

(1) Load Register "i" with C(M).

```
a)  LDi   AA(F)           => ASSIGN Ri,SOT[AA,F]
b)  LDi   AA,Ij(F)        => ASSIGN Ri,XOT[AA,j,F]
```

(2) Store Register "i" into Memory (M).

```
a)  STi   AA(F)           => ASSIGN SOT[AA,F],Ri
b)  STi   AA,Ij(F)        => ASSIGN XOT[AA,j,F],Ri
```

(3) Enter an Immediate Value into a Register

```
a)  ENTi  AA              => ASSIGN Ri,ICT(AA]
b)  ENTi  Ij              => ASSIGN Ri,Ij
c)  ENTi  AA,Ij           => ADD    Ri,ICT(AA),Ij
```

(4) Decrement a Register by an "Immediate Value".

```
a)  DECi  AA              => SUB    Ri,Ri,ICT[AA]
b)  DECi  AA,Ij           => SUB    Ri,Ri,ICT(AA]
                          => SUB    Ri,Ri,Ij
```

The above examples illustrate the "many to one" mapping condition for opcodes which is a product of properties 1 and 2. The small "i" in the above MIX operation codes can assume eight different values (register specifications). Thus the thirty-two MIX instructions implied above translate into only three IMTEXT operators.

(5) The "Compare" Instruction (CMPi)

```
a)  CMPi  AA(F)        => CMP   CI,Ri,SOT[AA,F]
b)  CMPi  AA,Ij(F)     => CMP   CI,Ri,XOT[AA,j,F]
```

The value of Ri is compared with the value of the core memory field (indexed or simple) and the result is stored in CI.

(6) The "Jump" Instruction.

Many forms of the MIX jump instruction exist involving conditional transfers based on the status of a register or the compare indicator. There is also the absolute jump. Two typical examples are:

```
a)  JLT   AA           => JUMP   lt,CI,JAT[AA]
```

where "lt" is the condition value (numerical) associated with the a need to test CI against "lt", and execute the transfer according to the operand JAT[AA] if the comparison is successful.

```
b)  JiP   AA           => JUMP   gt,Ri,JAT[AA]
```

In this example, the jump instruction implies a comparison of Ri with zero. This test results in a condition value. If it equals "gt" (i.e. Ri > 0) then commence execution of the instruction referenced by the transfer address in the jump instruction.

Note that the "condition value" contained in the IMTEXT

is not a pointer to an operand table, but is a literal constant. The above examples (6a,6b) illustrate how all the variations in the MIX "jump" instructions can be coalesced into the operands to produce a single "jump" operator.

(7) Divide register AX by a field in core memory, store the quotient in A, and the remainder in X (DIV).

Proper translation of this instruction involves examining the structure of the program control graph in order to determine the "busy" information of A and X subsequent to the divide. (note: A variable is "busy" at some location L in the program if it is subsequently fetched, before it is redefined along some control flow path beginning at L). The divide instruction is translated in two steps:

(7a) During the initial translation of MIX to IMTEXT, only the operands are processed to produce for example:

```
DIV   AA(F)              => DIV   AX,A,SOT[AA,F]
```

(7b) During the "compression phase" of the IMTEXT (discussed later in this chapter) the "busy" status of registers A and X are examined and the following transformation is performed:

```
DIV   AX,A,SOT[AA,F]   =>

      QUOT.  A,AX,SOT[AA,F]    (if A is busy)
      REMAIN  X,AX,SOT[AA,F]   (if X is busy)
```

If register A is preset to zero prior to the divide instruction and in the same block, then register AX is replaced by A in the above IMTEXT instructions, and the instruction assigning zero to A is eliminated.

(8) An Idiom Involving the "SHIFT" Instruction.

The "SLAX n" instruction in MIX specifies to shift the AX-register left "n" bytes. In general this instruction is difficult to translate into a meaningful higher level statement without further analysis of the program. If "n" is 5 (the number of bytes per word), however, this statement can be interpreted as: {A←X, X←0}. This statement occurs quite frequently in MIX programs as a result of sequences such as:

```
LDA  X
MUL  Y
SLAX 5
ADD  Z
STA  X
```

which is the code for:

X ← X*Y+Z.

A similar translation holds for SRAX 5 (i.e. shift right). The above example illustrates the necessity and desirability for handling frequently used idioms. Proper recognition of the above idiom results in a simple translation rule which facilitates simplification of MIX arithmetic expressions.

DETERMINING THE BUSY STATUS OF VARIABLES

In decompiling as in compiler optimization techniques, it is necessary to introduce the concept of "busy status" of variables. In decompiling one of the goals is to eliminate unnecessary intermediate "loads" and "stores" and to combine groups of primitive machine language statements into a single high level statement in the target language. Take for example the following SMITEXT sequence:

```
(1)    ASSIGN A,TWO
(2)    ADD  A,A,THREE
(3)    ASSIGN RESULT,A
       ...
```

It can be seen that operand TWO can be substituted for the source operand A in (2) because it is also redefined in the same instruction. This would result in:

```
(1')   ADD  A,TWO,THREE
(2')   ASSIGN RESULT,A
       ...
```

Now if A is not used (not busy) before it is redefined subsequent to (2') then the variable RESULT in (2') could be substituted for A in (1') and (2') could be eliminated.

The concept of "busy" which is useful for decompiling will be discussed in terms of the following definitions.

Definition 3.A: Given two instructions, Ik and Im,

located at k and m respectively in a program, an instruction path from k to m exists if there is some executable instruction sequence Ik,...,Im.

Note that if more than one such path exists, then each path traverses at least one unique instruction block in the control structure of the program.

Definition 3.8: A variable V is busy at some location L in the program if it is subsequently fetched (at some instruction other than at L) before it is redefined along some instruction path beginning at L.

This definition is different than those given previously in the literature in that "busy" is defined relative to some specific location (instruction) rather than an instruction block. Whereas, in compilers busy information is used for reorganizing instruction blocks, the primary use of "busy" status in decompiling is for combining and eliminating individual instructions.

In general, to determine the busy status of a variable V, it is necessary to scan the instructions along all instruction paths beginning with L until a busy occurrence of V is found or until it is determined that V is not busy. V is not busy on an instruction path if it is redefined before it is fetched or if an exit block (i.e. a block with no immediate successors) is reached or if L itself

is reached (i.e. the path is a loop back to L). Determining the busy status of V involves a procedure which uses the immediate successor lists and other BLKTBL information to recursively scan the instruction blocks which lay on control flow paths beginning at L. It has been found convenient to be able to determine if V is busy in the block which contains L. This means that the "busy scan" terminates with the last instruction in the block containing L.

<u>Definition 3.C:</u> A variable V is <u>block</u> <u>busy</u> at some location, k, if it is fetched before it is redefined in the instruction sequence Ik,...,Im, where Im is the last instruction of the instruction block which contains Ik.

<u>Notation:</u> In discussing busy status in subsequent algorithms, two boolean functions will be used:

BUSY[V,L] - returns TRUE if a variable V is busy (definition 3.B) at some location L. Otherwise it returns FALSE.

BLKBUSY[V,L,K] - returns TRUE if V is <u>block</u> <u>busy</u> at location L; otherwise FALSE is returned. K is an output variable of the procedure which points to the last instruction scanned.

Busy Status and IMTEXT

All busy status computation is performed using the IMTEXT representation of the program. The locations of instructions referred to in the "busy" definitions are pointers to IMTEXT instructions in the instruction table IT. When examining an instruction IT[k] during the "busy scan", the variable V, whose busy status is sought, is compared first against the source operands of IT[k] to see if V is fetched at IT[k], and then against the result operand of IT[k] (i.e. IT.N1[k]) to see if it is redefined. Several observations are in order concerning these comparisons. (notation: Nij will be used to abbreviate IT.Nj[i].)

(1) In general only simple variables (V) are examined for busy status. If V is an indexed referenced its busy status is a dynamic function of the index value, and cannot be determined by a simple scan of the original program structure. One exception which is quite useful is when a redefinition of V with respect to L (L as in the busy definitions) occurs at L+1, and IT[L+1] is in the same block as IT[L]. In this case IT[L] and the instruction where V is used are adjacent within the same block and it is not possible for the index of V to have been altered.

(2) When comparing V against Nij, if Nij is simple (i.e. a pointer to SOT) then V and Nij can be compared directly and it is not necessary to reference any information in the operand's table entry.

(3) If Nij is indexed (a pointer to the XOT) then V must be compared against the index of Nij which is the field XOT.IDX recorded in the table entry for Nij.

Algorithm for Determining BUSY[V,L]

Notation:

CB - number of current instruction block being scanned.

i - index of current instruction (IT[i]) being scanned.

BN[i] - block number of instruction block containing IT[i] (i.e. field IT.BN[i]).

BL- list of block numbers of blocks which have been scanned, and which are to be scanned.

BL[NB] - next block to be scanned.

BL[NE] - next entry in BL.

{BL[1],...,BL[NB-1]} - set all blocks which have been scanned.

Some extraneous detail such as checking for "nop" operation codes is omitted for sake of clarity.

A. Initialize: {i←L,NB←0,NE←1,CB←BN[L]}

B.1 i ← i + 1.

.2 If CB = BN[i] then go to D.

C.1 For every n in IS[CB]: if n ≠ BL[k], (k=1,...,NE-1) then { BL[NE] ← n, NE ← NE + 1}.

.2 go to F.

D. For every source operand Nij (j=2,...) of IT[i]:

.1 If V = Nij then { return TRUE}.

.2 If Nij is an indexed operand and the index of Nij = V then { return TRUE}.

E.1 If Ni1 = V (i.e. V is redefined) then go to F.

.2 If i≠ L then go to B.1.

F. If NB > NE then {return FALSE}. (all paths scanned)

G.1 NB ← NB + 1.

.2 CB ← BL[NB].

.3 i ← BLKTBL.FI[CB] (first instruction of block CB)

.4 go to D.

The algorithm for computing BLKBUSY[V,L,K] is similar to that just described except that the scan is forced to terminate with the last instruction of the block containing L. Also, the index of the last instruction scanned is returned.

THE IMTEXT "COMPRESSION" ALGORITHM

As previously stated one goal in decompiling is to eliminate intermediate "loads" and "stores" which are present in the original machine language program. Another goal is to combine as many primitive instructions as possible in order to produce a single high level instruction in the target language. This discussion deals only with eliminating the intermediate loads and stores. This process will be referred to as text compression. At first glance one is tempted to combine the above goals into the single goal of "program simplification". However, advantages are realized by treating them separately. One of the essential observations concerning text compression is that it preserves the IMTEXT three address code representation. This format appears to be well suited for decompilation analysis and for the interpretive execution of the program. The advantage of compressing the text before performing further simplification is that the text compression can be performed immediately after the IMTEXT representation of the program has been completed, while still preserving the three address code format. For the samples tested, the text compression process reduces the "volume" (no. of instructions) of the program up to 40 percent, resulting in more efficient analysis of the IMTEXT representation in later phases. Since the text is scanned (at least

partially) many times in subsequent analysis phases, early compression of the text results in a cumulative savings (execution time).

The General Approach

The following example should illustrate the general idea of text compression. Consider the following program in three address code representation.

```
(1)   ← A D
(2)   + E F A
(3)   ← G E
(4)   * F G A
(5)   ← H F
```

Step 1: Replace the source operand A in (2) and (4) by the source operand D in (1). Then eliminate (1) to produce:

```
(2)   + E F D
(3)   ← G E
(4)   * F G D
(5)   ← H F
```

Step 2: Replace the source operand G in (4) by source operand E in (3), and then eliminate (3) to produce:

```
(2)   + E F D
(4)   * F E D
(5)   ← H F
```

Step 3: Replace the result operand F in (4) by the result operand H in (5), and then eliminate (5) to give:

(2) + E F D
(4) * H E D

Observe that in steps 1 and 2 that the source operand of the assignment (←) statement are substituted for source operands in subsequent instructions which are equal to the result operand in the assignment statement being considered for elimination. This is called "forward substitution" of operands. In step 3 the result operand of the assignment statement (F) is substituted for the result operand in a previous instruction in which the result operand equals the source operand of the assignment statement being considered for elimination. This is call "backward substitution" of operands. The compression algorithm consists of two major phases, one for each type of substitution. The above example is oversimplified in that a host of conditions must be met before any operand substitutions can be made. For example, if the instructions (1)-(5) are the nonjump instructions (NJ[k]) of a block (k) which has one or more immediate successors and the operand A is "busy on exit" (i.e. BUSY[A,j], where IT[j] is the last instruction in block k) in block k, then step 1 could not be performed without altering the logic of the computation. The sets of conditions which must be met in order to exercise forward and backward substitution of operands are included in the compression algorithm.

During the scan of the instruction text for the compression algorithm, the second translation step for all divide (DIV) instructions is performed (see translation rules). Depending on the values of BUSY[A,k] and BUSY[X,k], the appropriate combination of QUOT and REMAIN instructions are generated to replace the DIV instruction at IT[k]. This procedure is not included in the following algorithms.

Compression Algorithm (phase 1) Forward Substitution

A. Initialize current block (CB) to first block (entry) of the program.

.1 CB ← 1.

B. Set instruction counter to location of first instruction in block CB.

.1 i ← BLKTBL.FI[CB].

C. Scan CB for an ASSIGN operator.

.1 If IT.OPC[i] = ASSIGN, go to D.

.2 If i < BLKTBL.LI[CB] (last instruction in CB), then { i ← i+1, go to C.1 }.

.3 If all blocks in program have been scanned, then initiate phase 2.

.4 CB ← CB+1, GO TO B.

D. Now i references an ASSIGN statement which is a candidate for elimination. Let Ri and Si be temporaries designating the result and source operands, respectively, of IT[i] (i.e. Ri←Ni1 Si←Ni2). Check conditions to see if Si can be substituted for Nkj in subsequent instructions, where Nkj = Si and IT[k] is in the same block as IT[i].

.1 If Ri is an indexed operand, go to C.2.

.2 If ¬BLKBUSY[Ri,i,k], go to C.2.

.3 If Si is an indexed operand, go to F.

.4 Find last "block busy" occurrence of Ri in CB. Record the instruction locations of all instructions where Si is busy in a list B. (B1,B2,...,Bn, where Bn points to the last instruction in CB where Ri is busy).

.5 If Ri = IT.N1[Bn] (result operand of IT[Bn]), then go to D.7.

.6 If BUSY[Ri,Bn], then go to C.2. (i.e. Ri busy on exit from CB).

.7 If Si has been redefined in {IT[i+1],...,IT[Bn]}, then go to C.2.

E. Compress the Text.

.1 Replace all operands Nkj (j=2,3; k=B1,...,Bn) such that Nkj = Ri by Si.

.2 Delete IT[i].

.3 go to C.2.

F. Handle special case in which Si is an indexed operand.

.1 If BLKBUSY[Ri,k,k'], then go to C.2. (i.e. only one busy occurrence allowed).

.2 If IT[i] is not adjacent to IT[k], go to C.2.

.3 Set B = B1, where B1 ← k.

.4 Go to E.

Compression Algorithm (phase 2) Backward Substitution

A. Initialize current block (CB) to first block of the program.

.1 CB ← 1.

B. Set instruction counter (i) to location of first instruction of CB.

C. Scan CB for an instruction other than an ASSIGN instruction.

.1 If IT.OPC[i] ≠ ASSIGN then go to D.

.2 If i < BLKTBL.LI[CB], { i ← i+1, go to C.1}.

.3 If all blocks scanned, TERMINATE compression algorithm.

.4 CB ← CB+1, go to B.

D. Check Conditions for Compression.

.1 Ri ← Ni1 (the result operand of IT[i])

.2 If Ri is an indexed operand, go to C.2.

.3 If ¬BLKBUSY[Ri,i,k], go to C.2.

.4 If IT.OPC[k] ≠ ASSIGN, go to C.2.

.5 Rk ← Nk1 (the result operand of IT[k]).

.6 If Rk is an indexed operand, go to F.

.7 If BLKBUSY[Rk,i,m] and i<m<k, then go to C.2. (the result variable of the ASSIGN cannot be used between IT[i] and IT[k]).

.8 Find all "block busy" occurrences of Ri past IT[k] in CB and record their instruction locations in list B (B1,...,Bn).

.9 If BUSY[Ri,Bn], go to C.2. (i.e. If Ri is busy on exit from CB, cannot eliminate Ri in IT[i]).

.10 If Ri is redefined anywhere in the sequence {IT[k+1],...,IT[Bn]}, then go to C.2.

E. Compress the Text.

.1 Replace all operands Nmj (j=2,3; m=B1,...,Bn) such that Nmj = Ri, by Rk. (i.e. where ever Ri is busy past IT[k] , must replace Ri with Rk since IT[k] is going to be deleted.

.2 Replace Ni1 in IT[i] with Rk, and then delete IT[k].

.3 Go to C.2.

F. Special case where Rk is an indexed operand.

.1 If BUSY[Ri,k,k'], then go to C.2.

.2 If IT[i] and IT[k] are adjacent, go to E.2.

.3 Go to C.2.

CHAPTER 4

FINDING PROGRAM LOOPS

In order to obtain the necessary information to decompile a program to a higher level language it is essential for the decompiler to analyze the source program's control structure in a global way. In particular one of the primary control structures of interest is that of program loops. The main goals of loop analysis are to determine: a) the bounds of array data structures, and b) how to reorganize the instruction blocks during target language generation to produce a high level representation of the program.

A program can be viewed as a directed graph G(N,A), where N is the set of nodes of the graph which correspond to the program's blocks, and A is the set of directed arcs connecting the nodes in N. The elements in A correspond to the immediate successors of the program's blocks. In order to discuss the algorithm for finding loops the following preliminary definitions are required.

DEFINITIONS

The algorithm to be described, while it appears to be original, is based on Fran Allen's (1970) discussion of

control flow analysis from which all of the following definitions, except for 4.J and 4.K, have been taken. been taken.

Definition 4.A: A strongly connected region (SCR) is a directed subgraph of G in which there is a path between any two (not necessarily distinct) nodes of the subgraph.

Definition 4.B: An entry node (entry point) of a subgraph of G is a node in the subgraph which has either no immediate predecessor or at least one immediate predecessor which is not in the subgraph.

Definition 4.C: A path exists between nodes N[j] and N[k] in N if there exists a sequence of nodes (N[j],N[j+1],...,N[k]) and a set of arcs: {(N[q],N[q+1]): q=j,...,k-1} which is a subset of the directed arc set A.

Definition 4.D: A closed path is a path of the form: (N[j],...,N[j]).

Definition 4.E: Given a node h, an interval I(h) is the maximal, single entry subgraph for which h is the entry node and in which all closed paths contain h. The unique interval node h is called the header node.

It has been shown (Allen, 1970) that the set of unique intervals of G partitions G into a set of disjoint subgraphs. Thus, by analyzing all the intervals of the

graph, the entire graph is analyzed. The utility of partitioning the program into intervals is that if the interval contains any SCRs then control flow must pass through the header node (i.e. only one entry point in the loop).

Definition 4.F: A *latching node* of an interval is any node in the interval which has the header node as an immediate successor.

Definition 4.G: A *loop* is an SCR (not necessarily maximal) which contains only one latching node.

Definition 4.H: The original graph G is called the *first order* graph. The *second order* graph is derived from the first order graph and its intervals by making each first order interval into a node whose immediate predecessors are those of the interval header node which were not members of the interval. The immediate successors of such a node are all the immediate non-interval successors of the original exit nodes (i.e. nodes which have immediate successors which are outside the interval).

Successively higher order graphs can be derived similarly until the n-th order graph is reached such that the (n+1)st order graph results in the same number of nodes (intervals) as the n-th order graph.

Definition 4.I: A graph is reducible if its n-th order graph consists of a single node (interval); otherwise it is irreducible. An equivalent condition for an irreducible graph is that it contain an SCR with more than one entry node.

Definition 4.H: Associated with each node, k, are two sets: a) IS[k], the immediate successors of k, and b) IP[k], the immediate predecessors of k. the elements of these sets are either "starred" or "unstarred" names of nodes (i.e. node numbers) in N (e.g. {3,1*,6,7*}).

THE ALGORITHM

The algorithm commences by examining each interval of the first order graph for all its loops. These loops are called "first order" loops (or SCRs). When an SCR is found its nodes (SCR.NL[k]) and order number (SCR.ORD[k]) are recorded in the SCR table. After all first order SCRs have been found, the higher order graphs are analyzed in like manner. This procedure is repeated until the graph is completely reduced or found to be irreducible.

The algorithm is an iterative (not recursive) procedure and works in such a way that the only data structures needed (in the implementation) during analysis are those required to express N, IS[n], and IP[n] (for all n in N). In the

implementation these data structures are conveniently represented by the appropriate BLKTBL entries. Consider figure 4.A.

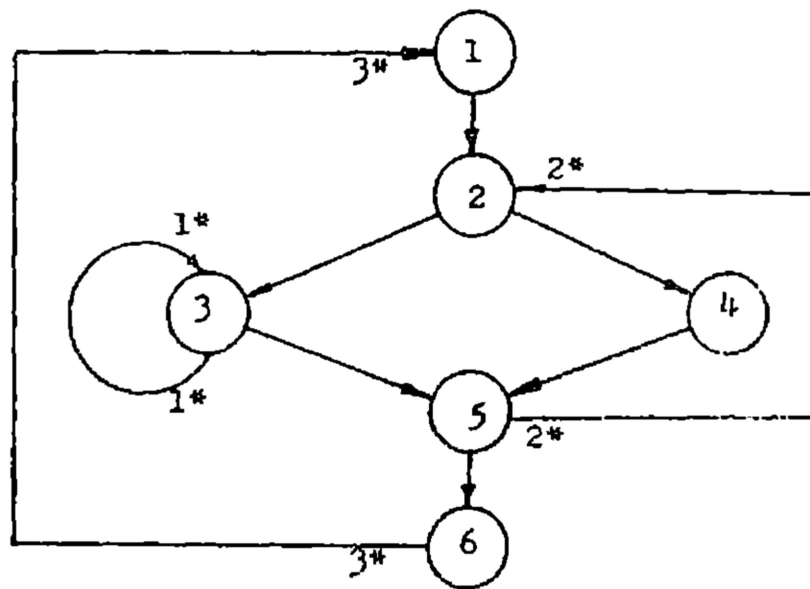


Figure 4.A - Control Flow With Nested Loops

The intervals for the above graph (1st order) are: (Notation: I[k,h] denotes the k-th order interval with header node h)

I[1,1]={1}, I[1,2]={2,4} I[1,3]={3}, I[1,5]={5,6}

The only SCR found in the first order intervals is {3} (i.e. SCR[1].NL={3}, SCR.ORD[1]=1).

To find the intervals of the next higher order graph, the directed arcs from the latching nodes to the header nodes ("latching arcs") of each SCR found in the current order graph are marked as deleted. This is done by "starring" node q in IP[h] where q is the latching node of a current order SCR, and by "starring" node h in IS[q]. Thus

IP[3] = {2,3*}, IS[3]={3*,5}

The notation "n*" illustrates the result in figure 4.A, where n is the order of the graph when the "starring" (*) occurred. Now to find the second order intervals all "*" arcs are ignored. Thus:

I[2,1]={1}, I[2,2]={2,3,4,5,6}

which results in:

SCR.NL[2]={2,3,4,5}, SCR.ORD[2]=2

Now the latching arc (5,2) is starred (2*) and the third order interval is found:

I[3,1]={1,2,3,4,5,6}

and

SCR.NL[3]={1,2,3,4,5,6}, SCR.ORD[3]=3

Determining the list SCR.NL[k] involves recording all unique nodes found on all reverse paths from the latching node to the header node. In this step no arcs are treated as deleted.

The first time a node, say n, is entered in the node list of some SCR, say k, a pointer to the SCR entry is stored in the block table for n (i.e. BLKTBL.SCR[n] ← k). This information will be used in some of the algorithms

to be discussed. In effect, this pointer designates the inner most loop which contains the given block. For example, for a program whose control flow is that of figure 4.A, BLKTBL.SCR[4] = 2 since the second entry in the SCR table is the first entry in which block number 4 is recorded as a member of some loop. If a block is not contained in some loop, the field BLKTBL.SCR is null.

Formal Algorithm specification

Notation:

CO - current order of the graph being scanned.

HL - list of header nodes; HL[1] is the first entry.

HL[NHS] - Next header node to be scanned.

HL[NHE] - Next entry in HL.

H - current header being processed.

INTRVL - list of nodes in current interval being constructed with header H.

INTRVL[CN] - current node in the interval whose immediate successors are being examined for entry in either HL or INTRVL.

NSCR - index of next entry in the SCR table (SCR.NL[NSCR], SCR.ORD[NSCR]).

NUMI[k] - the number of intervals in the k-th order graph.

Algorithm Procedure

A. Initialization for first order graph analysis.

.1 CO ← 1.

.2 NUMI[0] ← [cardinality of N].

.3 NSCR ← 0.

B. Initialize for analysis of graph of order CO.

.1 NHS ← 1, NHE ← 2.

.2 HL[NHS] ← 1. (node 1 is the entry node of G)

C. Get header of next interval to be scanned.

.1 If no unscanned entries in HL (i.e. if NHS = NHE), then go to F.

.2 H ← HL[NHS], NHS ← NHS + 1.

D. Find the next interval (INTRVL).

.1 CN ← 1, INTRVL[CN] ← H.

.2 Find the next Unstarred-Immediate-Successor (UIS) of INTRVL[CN]. If none, go to D.6.

.3 If UIS is in HL[k] (k=1,....NHE-1), or in the interval (INTRVL), go to D.2.

.4 If all unstarred k in IP[UIS] are in INTRVL, then add UIS to INTRVL, and go to D.2.

.5 Add UIS to HL, go to D.2.

.6 If CN is last entry in INTRVL, go to E.

.7 CN ← CN + 1, go to D.2.

E. Analyze newly found interval (INTRVL) for loop SCRs.

.1 Get next Unstarred-Immediate-Predecessor (UIP) of H. If none, go to C.

.2 If UIP not in INTRVL, go to E.1.

.3 Initialize new SCR table entry.

a) NSCR ← NSCR + 1.

b) SCR.NL[NSCR] ← {H}.

c) SCR.ORD[NSCR] ← CO.

.4 Complete the SCR node list (SCR.NL[NSCR]).

a) Chain through all starred and unstarred immediate predecessors, starting with UIP (latching node) and terminating with H. In this chaining process, record all unique nodes found along paths from UIP to H, in the node list (SCR.NL[NSCR]).

b) Add UIP to SCR.NL[NSCR]. Note: the first and last nodes in SCR.NL[NSCR] are the header and latching nodes respectively.

.6 Star the latching arc of the newly recorded SCR.

a) Star n in IS[UIP] where n = H.

b) Star n in IP[H], where n = UIP.

.7 Go to E.1.

F. Test for end of processing (i.e. graph completely reduced or irreducible).

.1 NUMI[CO] ← NHS - 1. (the number of intervals is equal to the number of headers scanned).

.2 If NUMI[CO] = 1, then TERMINATE.

.3 If NUMI[CO] = NUMI[CO-1], then

{write "irreducible graph", TERMINATE}.

.4 Analyze next higher order graph.

a) CO ← CO + 1.

b) Go to B.

Analysis Constraints

For purposes of further analysis it is assumed that G complies with the following constraints.

a) G is completely reducible.

If G is irreducible, then an SCR exists which has two or more entry nodes. Such a complicated control structure makes analysis extremely difficult. Separate studies by Knuth (1971) and Allen (1970) indicate that over 90 percent of Fortran programs are reducible. If it is assumed that this figure could be extrapolated to include machine language programs, this assumption would be quite reasonable. However, one would expect because of the intricasies of some machine language programs and the fact that machine language control structures are unconstrained, that the percent of reducible programs would be somewhat lower than that for Fortran programs.

Cocke and Schwartz (1970) show that any irreducible graph can be transformed into an equivalent reducible graph by a procedure known as "node-splitting". This involves duplicating some of the nodes and altering the directed paths appropriately to produce a reducible graph. Further research is needed to determine a general node-splitting (NS) method which results in the minimum number of duplicated nodes.

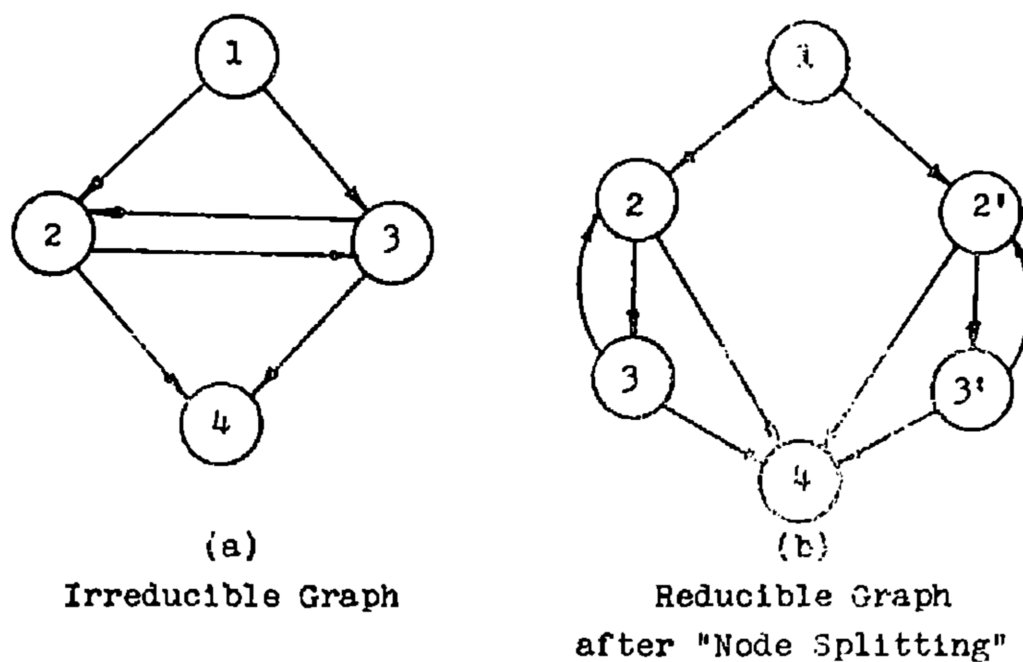
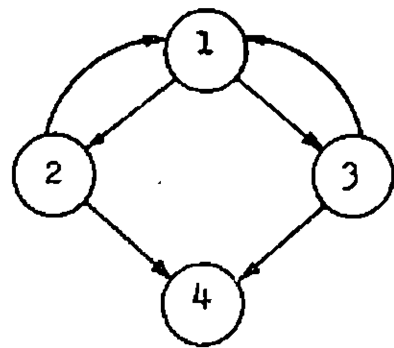


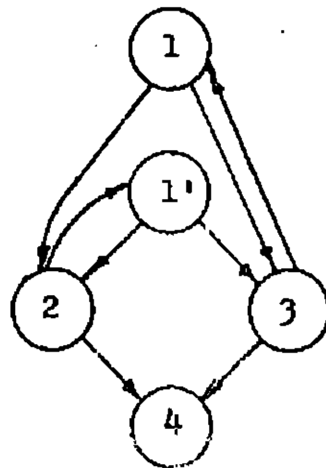
Figure 4.B - Irreducible Graph

b) G contains no "tangent" Strongly Connected Regions.

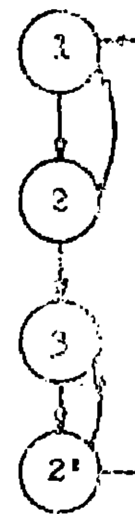
Definition 4.K: Two SCRs are tangent if: 1) they share a common header or 2) if the header node of one of the SCRs is the latching node of the other.

Tangent loops do not prevent the graph from being reduced and the algorithm will find the tangent SCRs (such SCRs will not be maximal SCRs). However, analysis becomes awkward due to the fact that tangent loops will be of the same order (i.e. found while analyzing the same order graph), but will not be disjoint. It is also true that loops in a set of nested loops are not disjoint (e.g. {3}, and {2,3,4,5} in figure 4.A); however each loop in a set of nested loops is detected while analyzing different order graphs of G, and is treated independently during portions of the analysis. Also, two nested loops have the property that one is a subset of the other. Such is not the case with tangent loops. Thus the difficulty which arises is that two or more tangent loops cannot be treated independently because the execution of the common node (i.e. the instruction block which the node represents) affects all the tangent loops involved. Like the irreducible case, these control structures would be expected to occur relatively infrequently. A node-splitting algorithm could also be developed to obviate the situation.





(Case 1)



(Case2)

Figure 4.C - Tangent Loops

BLOCK LEVELS

After the SCR finding algorithm terminates, "level" numbers are assigned to each block (node) recorded in BLKTBL (i.e. field BLKTBL.LEV). These "level" numbers are directly related to the "order" numbers of the graph. BLKTBL.LEV[k] reflects the "nesting depth" of block k, and is computed as follows:

Let HORD[k] be the order of the highest order SCR in which block k is a member.

Let LORD[k] be the lowest order SCR in which block k is a member.

Then: BLKTBL.LEV[k] ← HORD[k] - LORD[k] + 1.

The block level numbers are assigned so that all blocks which are not in any loops have a level of zero. In figure 4.A, block 3 would have a level of 3, blocks 2, 4, and 5 a level of 2, and blocks 1 and 6 a level of 1. In subsequent analysis procedures it is necessary to identify all the blocks at the same level when treating a set of nested loops.

The "level" of the k-th loop recorded in the SCR table is defined as:

SCR.LEV[k] = M MIN{BLKTBL.LEV[j], for all j in

SCR.NL[K]}.

For example, in figure 4.A, loop {3} would have a level of 3, loop {2,3,4,5} a level of 2, and loop {1,2,3,4,5} a level of 1.

CHAPTER 5

DETERMINING DISJOINT ARRAYS VIA ANALYSIS OF LOOPS

One of the more interesting problems in decompiling is that of determining all the array variables and their bounds. This clearly requires some kind of analysis involving all the program's indexed references (recorded in the indexed operand table (XOT) of INTEXT). However, merely comparing like entries in the XOT is not sufficient since two XOT operands may have different values but in fact be referencing the same array. Similarly two identical entries in the XOT may reference different data structures. From an analytical viewpoint it is necessary to determine the range of <u>effective</u> addresses of each dynamic reference. Once this is done the ranges can be analyzed to determine the set of disjoint arrays. This chapter describes analytical and heuristic methods for determining the ranges of dynamic references. The model described here is an initial attempt toward developing an abstraction of the program in order to make explicit some of the data structures which are implicitly defined by the instruction operands and their context within the topological structure of the program. As in other formal models this model

requires some assumptions and will not hold for one-hundred percent of the cases; however, it is believed to be applicable for a large percentage of "reasonable" programs. Where the model fails, the heuristic approach described later in this chapter can be employed.

The basic approach is to analyze the loop control structures in which the indexed or dynamic references occur. In particular, "iterative" loops are of interest as opposed to "conditional" loops. A conditional loop is one where all exits from the loop depend on some noniterative condition being met (e.g. IF X < .005 THEN GO TO A). With iterative loops the objective is to determine the range of indexed references, whose index values are a function of the loop index. Before any references are analyzed, properties of the loops are determined in two steps. In the first step, individual loops are analyzed to determine the "expected" maximum number of iterations of the loop *per* entry into the loop. The second step generates a data structure which reflects all the sets of nested loops. Using this representation it is possible to compute the expected range of each indexed reference occurring inside of one or more iterative loops, provided that the parameters and structure of the loop are properly constrained.

THE "VALUE-SET" OF A VARIABLE

One of the essential functions necessary to decompilation is that of being able to determine the set of initi. values of a variable (V) at a desired location (L) in the program. For example, this capability was assumed in the discussion on indexed jumps (chapter 2). Since it is possible for V to be a function of many variables which were computed along various control flow paths starting with the program entry point, it is convenient to introduce the notion of a computation graph (C-graph). Using this C-graph the "value-set" of V at L is computed by interpretively executing all sequences of instructions (along all control paths) which can return a value for V at L. It is assumed that the operand values of the initial instructions are available.

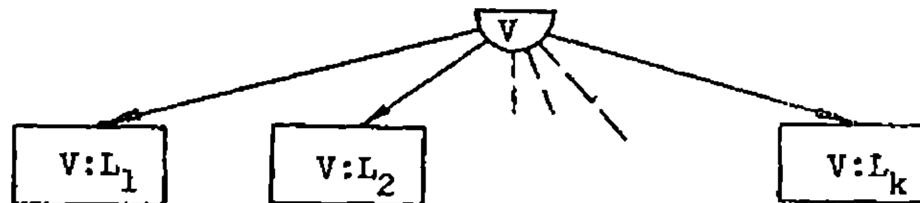
C-graph Notation and Rules

Definition 5.A: An occurrence of a definition of a variable, say V, at location L is called an OCELL and is represented as:

V:L

Definition 5.B: A set of alternative definitions of a variable V is call an OSET whose individual elements are

OCELLs. If there are k elements in the set, it would be represented in the C-graph as:

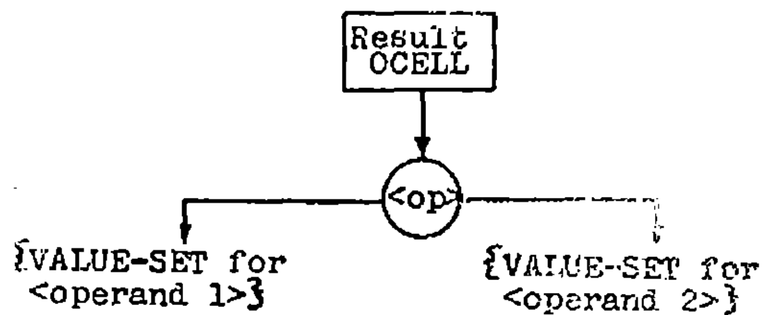


where Li is the location of the statement defining the i-th definition of V in the OSET.

Definition 5.C: A computed value-set is a set of constant values which is designated as: {C1,...Cn} in the C-graph.

Definition 5.D: An uncomputed value-set is represented as either an OSET or an OCELL.

Definition 5.E: An n-ary C-graph computation is represented by a "result" OCELL, an operator (in a circle), and a value-set (computed or uncomputed) for each of the source operands. For example, a binary operation would have the general form:



Element Connection Rules

1) An OCELL is the initial element of all C-graphs (or sub C-graphs).

2) An OCELL has one and only one successor element, namely an operator element (circle).

3) An n-ary operator element has n successor elements any of which may be a computed value-set element ({ }), an OSET element (◡), or an OCELL.

4) An OSET element has two or more successor elements, all of which are OCELLs.

5) Any OCELL which serves as an occurrence of a source operand in a C-graph computation represents an (at least partial) uncomputed value-set of a source operand. The definition of the operand occurrence in the OCELL must occur along some instruction path which can reach the instruction designated by the result OCELL of the C-graph computation.

Example of a C-graph

Consider a program whose control flow and instructions are partially represented by figure 5.A, where the label of the instructions in the boxes (blocks) designate the locations (addresses) of the instructions in the original program. Suppose it is desired to determine the initial value-set of the variable V at location 60 in block 9. The C-graph for this computation is given by figure 5.B. The "Ø" operation is a null operator used to connect the initial or request OCELL with the OCELL which describes a definition of V which can reach location 60.

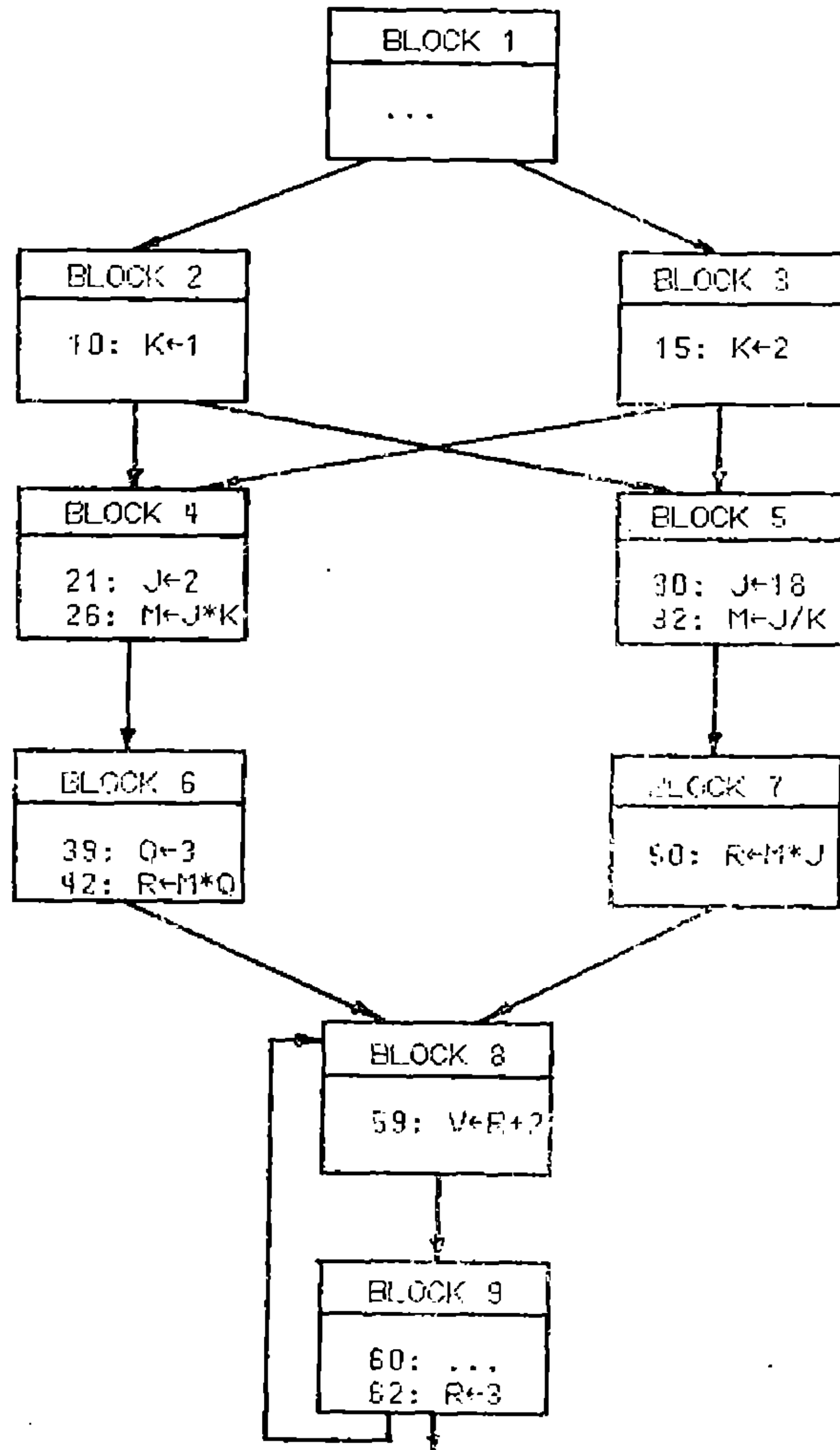


Figure 5.A - "VALUE-SET" Example Program

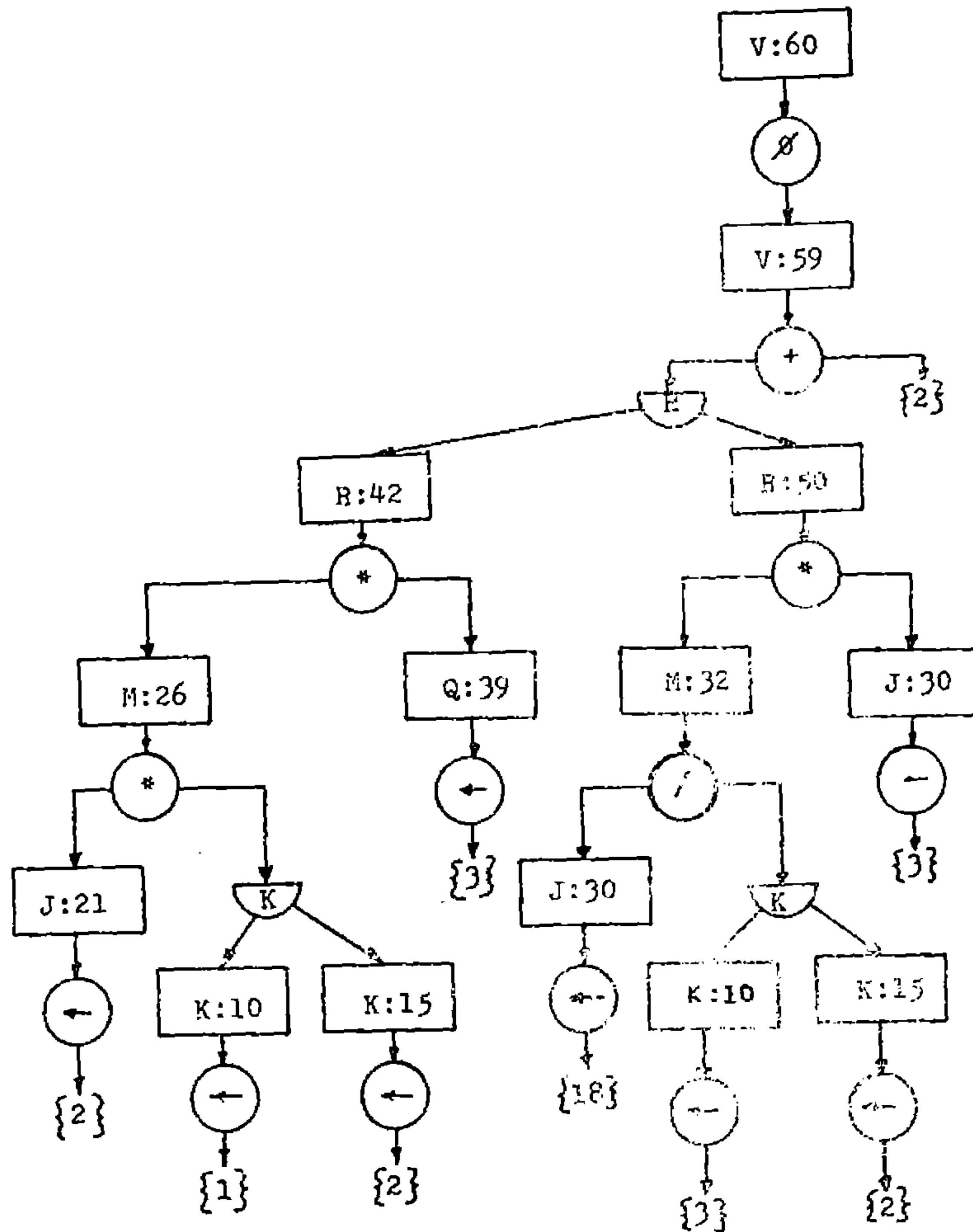
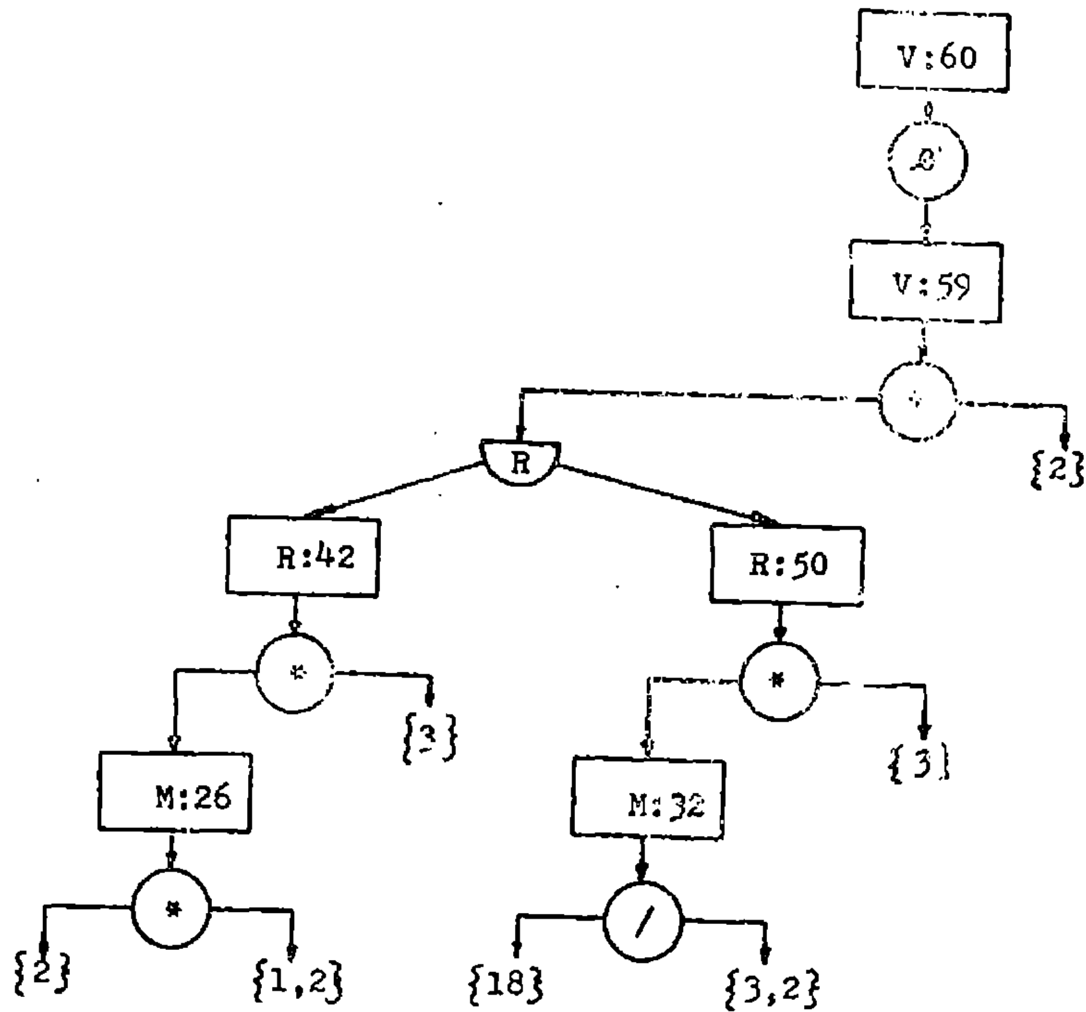


Figure 5.B - Initial C-graph Representation

The goal of the algorithm is to reduce the C-graph until the value-set of the variable in the initial OCELL is determined. This may be done by recursively applying the operators in the circles where their operand value-sets are <u>computed</u> value-sets. (i.e. at the extremities of the C-graph). For example the first set of reductions would produce:



<u>Figure 5.C</u> - C-graph Reduced One Level

The second set of reductions would give:

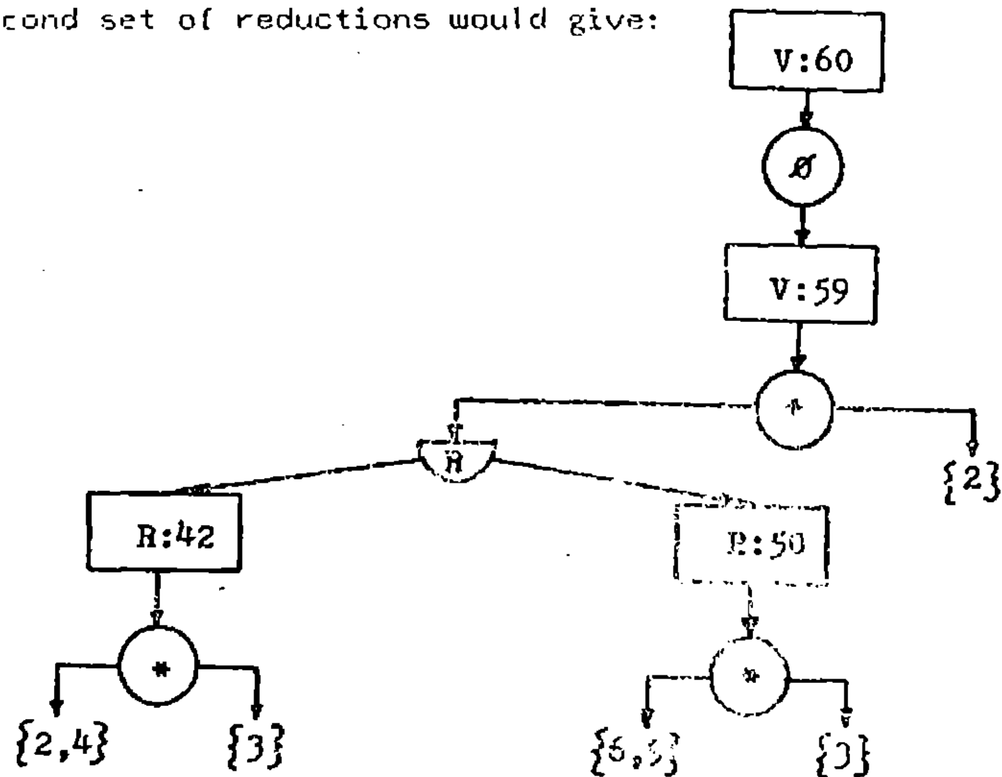


Figure 5.D - C-graph Reduced Two Levels

Applying the procedure twice more would give the final value-set of:

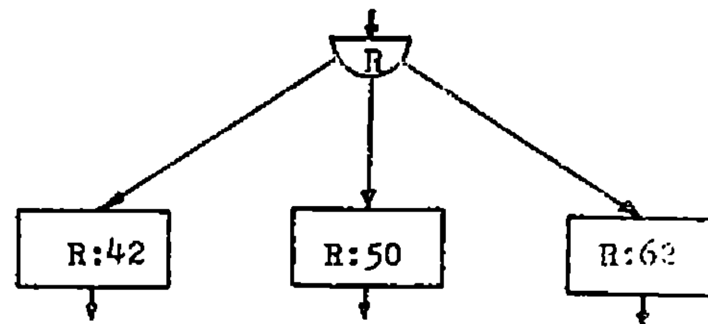
[V:59] = [V:60] = {8,14,20,[illegible]}

Notice in figure 5.B that some CCELLs are duplicated (K:10, K:15, J:30). The reason is that the reduction algorithm works in such a way that it need not maintain a history of all previously computed intermediate value-sets. This simplifies the algorithm and makes it unnecessary to keep a representation of the entire C-graph throughout the computation. In the implementation, storage

is allocated as needed to represent only those portions of the C-graph which are necessary for a particular stage of the computation.

Complete Value-Set

In the above example only the initial value-set for V at 60 was computed. That is, all values that V could assume along paths between the program entry point and location 60 were computed. Observe that the loop comprised by blocks 8 and 9 contains a definition of R at 52. If this definition were included, the OSET for R in figure 5.B would be augmented to form:



In this case the final value-set would have the additional element of 5. This value-set contains all possible values which can be determined for V at 60 by analyzing the original (static) program structure. Such a value-set is called the complete value-set. Both types of value-sets are required in various analysis procedures. To determine only the initial value-set, all latching arcs are ignored when generating the C-graph from the control

flow graph. This is easily done by ignoring the "starred" items in the immediate predecessor lists (chapter 4).

The Algorithm

A request for a value-set of V at L is represented by a procedure call of the form:

VALSET(V,L,TVS,CCG.CGP)

where the formal parameter TVS describes the type of value-set required and can have the value of "INITIAL" or "COMPLETE". The parameter CCG requests that the data structure which represents the entire C-graph (before any reductions) be returned along with the value-set of V at L. CCG may have the values: SAVE and NOSAVE. If CCG equals SAVE, a pointer to the initial OCELL of the C-graph is returned in CGP. In a subsequent section it will be seen that saving this C-graph is convenient for computing the upper bounds of V at L.

VALSET Data Structures

VALSET uses the IMTEXT representation of the program and therefore, the operators in the C-graph are restricted to unary and binary. In addition to the IMTEXT data structures, OCELL and computed value-set data structures are required. An OSET is represented as a linked list of

OCELLs. The OCELL data structure is not strictly equivalent to that in the model. Its fields are defined as follows:

IC - instruction counter which references the instruction in IT which defines the occurrence of the OCELL's operand.

VSN2 - value-set for the first source operand of IT[IC].

VSN3 - value-set for the second source operand of IT[IC]. If IT.OPC[IC] is a unary operator, this field is always null.

VS - value-set for the OCELL, which is computed by applying the operator IT.OPC[IC] to VSN2, and VSN3.

OSN2, OSN3 - pointers to the OSETs for the respective operands.

Procedure VALSET(V,L,TVS,CCG,CGP)

The VALSET procedure is defined recursively by the following algorithm.

A. Generate Occurrence Set for V at L (OS[V,L]).

.1 Scan all reverse instructions paths from L until all definitions of V are found. If TVS=INITIAL, then ignore all reverse paths involving latching arcs.

.2 For every definition of V encountered, create an OCELL

whose only defined field is IC.

B. Get next OCELL in OS[V,L]. If none, go to F.

C. Compute value-set for first operand of IT[IC].

.1 If IT.N2[IC] is an immediate constant operand or a "read only" initialized memory reference, then: {VSN2 ← VALUE[IT.N2[IC]], go to D}

.2 VSN2 ← VALSET(IT.N2[IC],IC,TVS,CCG,OSN2)

D. Compute value-set for second operand of IT[IC] if necessary.

.1 If IT.OPC[IC] is a unary operator, go to E.

.2 Compute VSN3 as in C.1, and C.2.

E. Compute Value-set.

.1 Apply operator IT.OPC[IC] to its operand's computed value-sets VSN2 and VSN3 (if binary), and store the pointer for the resulting value-set in VS (i.e. VS←VSN2 (op) VSN3).

.2 Go to B.

F. Return the final value-set (and possibly C-graph depending on the value of CCG) to caller.

.1 For every (OCELL) k in OS[V,L], where k = 1,...,n: {VALSET ← UNION(VS1,...VSn)}

.2 If CCG = SAVE:

a) CGP ← (pointer to OS[V,L] (i.e. list of OCELLs)).

b) go to F.4.

.3 If CCG = NOSAVE:

a) Free all OCELLs in OS[V,L].

b) CGP ← 0.

.4 <u>Return</u> VALSET, CGP.

INDIVIDUAL SCR (LOOP) ANALYSIS

The goal of this analysis phase is to determine important properties of each individual loop in the program. These properties are reflected in additional information stored in the SCR table for each loop analyzed. The interrelations among the individual loops (e.g. nested loops) are determined as described in the next section. In the following discussion SCR[k] will denote the current SCR or loop being analyzed.

Recursively Defined Variables

To determine bounds on dynamic or indexed references, it is essential to find all the recursively defined variables with respect to the loop being analyzed (SCR[k]). A recursively defined variable V is one which is defined in terms of itself:

$$V \leftarrow f(V)$$

Definition 5.F: A variable V is recursively defined with respect to an SCR, say SCR[k] (i.e. V is a ROV), if V is recursively defined one or more times within SCR[k] and there are no nonrecursive definitions of V in SCR[k]. In addition the block (node) in which V is defined must have the same level as SCR[k] (chapter 4). All blocks recorded

in the SCR node list (SCR.NL[k]) have a level greater than or equal to the level of the SCR. If a block has a higher level than that of SCR[k], then it is a member of some inner loop say SCR[j] and V would have already been recorded as a RDV of SCR[j] (i.e. SCR[k] covers SCR[j]).

In order to simplify the implementation, the general method has been restricted to a particular class of PDVs, namely those of the form:

$$V \leftarrow \pm V \pm DV,$$

where DV is the net change of V per iteration of the loop. If DV is a variable it is assumed that it is single valued upon entry to the loop, and that it is constant with respect to the loop (i.e. its value cannot change during an iteration).

It is interesting to note that problems would arise in computing the range of V if it is recursively defined more than once in the loop. If this occurs some method would be necessary for computing the total net change per iteration. To obviate this problem the following criteria are employed using the concept of an "SCR articulation node". If one considers the subgraph SCR[k] which is comprised of the nodes of (SCR.NL[k]) and the arcs which connect them, then the entry node of SCR[k] is the header node H of SCR[k].

Definition 5.6: A node n in SCR.NL[k] is an SCR articulation node of SCR[k] if n lies on every path from H to H in SCR[k], where H is the header node of SCR[k]. While traversing paths from H to H in SCR[k], all starred immediate successors of nodes in SCR.NL[k] whose level is greater than the level of SCR[k] are ignored.

The restriction on variables which are recursively defined with respect to an SCR is that only one such definition may occur among its SCR articulation nodes. This restriction guarantees that the RDV, say v, will be incremented (or decremented) during each iteration of the loop. The analysis could be generalized to allow more than one such RDV definition. However, for most iterative loops, it would seem that multiple recursive definitions of the same variable are relatively uncommon. If there are multiple definitions in an SCR, where only one occurs in an SCR articulation node, those definitions not in SCR articulation nodes are ignored in the bounds calculation of the RDV. It is assumed that such definitions are side effects of some condition and will not violate the bounds computed using the articulation node definition.

The final restriction on RDVs is that they be simple variables, and that the computation of their initial and final values involve only simple variables. This appears to be the case for a good number of programs involving

iterative loops. For analysis purposes it is desirable to determine the initial and final values of RDVs from the original program structure. This is generally most difficult if indexed variables are involved in these computations. One way to handle the occurrence of indexed variables which violate the above restriction is the following. Presumably a program to be decompiled was run in production in its native environment, and therefore, realistic data would be readily available. Using this data the source program could be executed (directly on source machine or by simulation) and the values of the pertinent indexed references recorded. These values could then be input to a table in the decompiler. When an indexed reference is encountered during analysis, a "typical" value (or range) could be retrieved from the table.

Determining the RDV Set of an SCR

It is of interest to determine all RDVs of each SCR. To do this all the instructions in the SCR's instruction blocks are examined, and each simple variable operand is considered for entry in either the "RDV list" (RDVL) or the "non-RDV list" (NRDVL) according to the following criteria:

Let V be a variable operand being analyzed at some location L (IT[L]).

A. If V is not recursively defined at L:

.1 If V is in NRDVL, no action.

.2 If V is in RDVL, then:

{mark it deleted in RDVL}.

.3 Add V to NRDVL if it is not already a member.

B. V is recursively defined at L:

.1 If V is in NRDVL, then no action.

.2 If V is in RDVL and IT[L] is in an SCR articulation node, then mark V deleted in RDVL; otherwise no action.

.3 If IT[L] is in an SCR articulation node then add V to RDVL. Each entry in the RDVL is comprised of three fields:

a) Variable name (i.e. pointer to the SOT).

b) The location, L, of the recursive definition.

c) The increment used in the recursive definition.

Notation: RDVL[j,k] denotes the j-th entry in the recursively defined variable list associated with SCR[k]. RDVL.NAME[j,k], RDVL.OFF[j,k], and RDVL.DEL[j,k] designate the fields a), b), and c) respectively described above.

The articulation node test must be included to handle the general case, but for this implementation the assumption that the "articulation" criteria is true has not produced

any ill effects. After all simple operands of the SCR have been analyzed, a pointer to the RDVL is stored in the SCR's table entry.

Iteration Exit Block

To compute the number of iterations of the loop per entry, it is necessary to find the exit block in which the loop iteration variable is tested for completion of the loop. This exit block is assumed to be an SCR articulation node of the SCR which describes the loop. The procedure for finding the iteration exit block first involves determining all the exit blocks of the loop. This is done by recording all nodes in SCR.NL[k] which have an immediate successor which is not in SCR.NL[k]. The list of exit nodes is recorded as part of the loop's SCR table entry (SCR.EXL[k]).

The exit test of each node in the exit node list is analyzed to see if it is a function of a recursively defined variable (i.e. an element of SCR.RDVL[k]). When such a block is found, its block number and the RDV involved in the exit test are recorded in SCR.EXB[k] and SCR.ITV[k] respectively. If there is more than one element in SCR.EXL[k], it is assumed that there is only one which meets the above criteria; all others are assumed to represent

conditional (noniterative) type exits. If an iteration exit block is not found, null values are stored for the above fields, and processing continues with the next entry in the SCR table.

Computing the Number of Iterations per Loop Entry

The number of loop iterations (NITER) per loop entry is a function of the iteration test operand used in the exit test of the iteration exit block, the test relation used in the exit test, the increment of the iteration variable, and the initial value of the iteration variable at the exit test instruction. Upon closer examination it is realized that all the blocks in a loop may not be executed the same number of times. In particular if the iteration exit block is executed n times before the exit from the loop occurs, then all successor blocks of the exit block up to and including the latching block will be executed a maximum of n-1 times. Similarly, all blocks from the entry block up to and including the iteration exit block will be executed a maximum of n times. Only in the case where the latching block and the exit block are the same, can any given block be executed a maximum of n times per entry to the loop. This difference is taken into consideration in later procedures where it is of interest to compute the maximum number of iterations per block which

is contained in one or more loops. However, when considering the number of iterations on a loop basis, it is sufficient to compute the number of times the iteration exit test is executed.

Parameters of the Iteration Exit Test

Knowing the iteration exit block, it is a simple matter to scan for the exit test instruction. If the jump instruction involves the compare indicator (CI), the test operand is possibly a variable whose value (TVAL) must be computed using the "VALSET" procedure; otherwise the exit test must involve a comparison between some operand (register) and zero. Typical exit tests are illustrated (using the IMTEXT representation) below:

a) Using Compare Instruction

```
CMP    CI,ITV,TEST
JUMP   ≤,CI,LOOP
(exit from loop if fall through)
```

The above is equivalent to:

<u>if</u> ITV ≤ TEXT <u>then</u> <u>goto</u> LOOP;

b) Implicit Test Value of Zero

```
JUMP   =,ITV,LOOPOUT
```

which is equivalent to:

<u>if</u> ITV = 0 <u>then</u> <u>goto</u> LOOPOUT;

In computing NITER the test relation (TREL) of interest is that which causes the transfer out of the loop. Thus

in a) above (assuming LOOP is the label of the loop entry block), the complement of the condition value would be used in the computation of NITER (i.e. TREL = ">"); whereas in b), the given test relation ("=") would be used since the jump is to a block outside the loop.

The Iteration Variable Parameters

The increment of the iteration variable (DITV) is determined by examining the recursive definition of the iteration variable (ITV). Recalling that both the variable name and the location of its definition are recorded in the RDVL, the defintion is readily obtained. Once the increment operand is determined from the recursive definition of ITV, its initial value is computed by invoking the VALSET procedure. The last parameter to be determined is the initial value (INVITV) of ITV at the iteration exit test instruction. This value is also determined by the appropriate invocation of the VALSET procedure.

If it is assumed that for a given loop the parameters: TVAL, TREL, INVITV, and DITV have been computed, the number of iterations of the loop (exit test) is computed as follows.

E5.A) If TREL is "=":

NITER = FLOOR[(TVAL - INVITV)/DITV] + 1.

in a) above (assuming LOOP is the label of the loop entry block), the complement of the condition value would be used in the computation of NITER (i.e. TREL = ">") whereas in b), the given test relation ("=") would be used since the jump is to a block outside the loop.

The Iteration Variable Parameters

The increment of the iteration variable (DITV) is determined by examining the recursive definition of the iteration variable (ITV). Recalling that both the variable name and the location of its definition are recorded in the RDVL, the defintion is readily obtained. Once the increment operand is determined from the recursive definition of ITV, its initial value is computed by invoking the VALSET procedure. The last parameter to be determined is the initial value (INVITV) of ITV at the iteration exit test instruction. This value is also determined by the appropriate invocation of the VALSET procedure.

If it is assumed that for a given loop the parameters: TVAL, TREL, INVITV, and DITV have been computed, the number of iterations of the loop (exit test) is computed as follows.

E5.A) If TREL is "=":

$$NITER = FLOOR[(TVAL - INVITV)/DITV] + 1.$$

E5.B) If TREL is ">" or "<":

NITER = FLOOR[(TVAL - INVITV)/DITV] + 2.

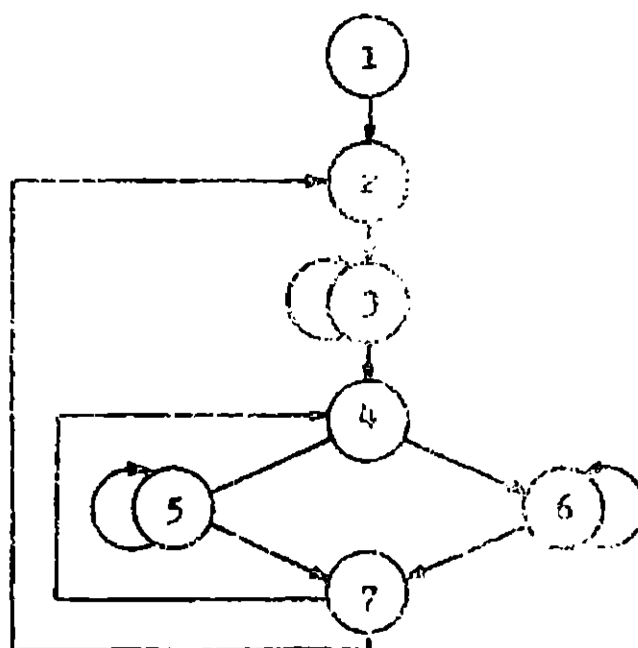
E5.C) If TREL is "≥" or "≤":

NITER = CEIL[(TVAL - INVITV)/DITV] + 1.

The above equations hold regardless of the signs of INVITV, TVAL, and DITV. Also, they are independent of whether ITV is incremented before or after the exit test.

A STRUCTURE FOR REPRESENTING NESTED LOOPS

In order to find the final bound of a subscripted or indexed reference in which the subscript is an RDV in some loop, it is necessary to determine the range of the RDV subscript. If the reference occurs within a set of nested loops, this range may depend on the number of iterations of one or more loops in the nest of loops. Therefore, it is necessary to construct a model which describes the set of nested loops in the program. To do this an entity called the Nested Region List (NRL) is defined. The NRL is an ordered list which is comprised of pointers to the SCP table. The following illustrates a control flow graph which consists of some nested loops.



Notice for example, if an indexed reference occurs in block 3, that for the purpose of computing the range of the reference, it is only necessary to consider a two loop nest, namely loops {3} and {2,3,4,5,6,7}. In other words given a reference in node n of loop SCR[k], it is only necessary to examine those loops which cover SCR[k] (i.e. all loops which contain n and whose level is less than SCR[k]). Given any two loops of the same level the computations of the bounds of the respective RDVs are independent. In order to reflect the nesting structure necessary to determine the bounds of any RDV in any block of a loop, a NRL is generated for every unique combination of nested loops (including a single region nest). In a NRL the entries are ordered according to nesting level, starting with the highest level (most nested) loop. For the above control flow graph, the SCR table entries and the nested loop lists would be constructed as follows:

Partial SCR Table

NO.	LEVEL	NODES	NRL PTR
1	2	{3}	1
2	3	{5}	2
3	3	{6}	3
4	2	{4,5,6,7}	4
5	1	{2,3,4,5,6,7}	5

NESTED REGION LISTS

NO.	SCR PTR Lists
1	{1,5}
2	{2,4,5}
3	{3,4,5}
4	{4,5}
5	{5}

Observe that a new field (SCR.NRLP) has been added to the SCR table entries. Given the k-th SCR entry, SCR.NRLP[k] references the n-th NRL where the first element of NRL[n] equals k. In other words it specifies the NRL in which SCR[k] is the innermost loop. In the given example, SCR.NRLP[4] = 4 designates that loop {4,5,6,7} is nested inside loop {2,3,4,5,6,7}.

At this point all the mechanisms have been established for determining which loops will be involved in computing the bounds of those subscripted references in the program which comply with the described conditions.

Suppose a subscripted reference in block 6 is being analyzed. Recalling from chapter 4 that the block table

field BLKTBL.SCR[k] references the SCR entry which describes the innermost loop which contains block k, it would be found that BLKTBL.SCR[6] equals 3. The field SCR.NRLP[3] references NRL[3] which indicates that the loop which contains block 6 (i.e. {6}) is contained in two outer loops, namely {4,5,6,7} (described by SCR[4]) and {2,3,4,5,6,7} (described by SCR[5]). It now remains to use the properties of each loop involved (number of iterations, etc.) in order to actually compute the bounds.

COMPUTING THE EXTENTS OF INDEXED REFERENCES

Given that the k-th operand in some instruction IT[L] (OPND[k,L]) is an indexed reference with an address part A and index IX, if the bounds of IX can be determined, then the initial effective address (IEA[OPND[k,L]]) and the final effective address (FEA[OPND[k,L]]) of the reference can be readily computed.

The initial value of IX is easily computed by invoking the VALSET procedure. During this invocation of VALSET, however, the generated C-graph is saved. The history of the initial value computation is reflected by the generated C-graph and is used along with previously described tables (SCR, BLKTBL, NRL) to determine the final value of the index. The fact that IX is a function of RDVs of outer

loops, will be discovered in the analysis process. Like the initial value (VALSET) algorithm, the intuitive idea of the final value (FVALUE) algorithm is to compute value lists for each of the operands, compute a resultant value list according to the given operator, and subsequently reduce the C-graph. The primary difference is that before a resultant value list, which was computed at some (current) OCELL in the C-graph is passed as the operand value list of the next (dominant) OCELL, the nesting levels of the instructions referenced by the current OCELL and the next OCELL respectively are compared. The difference of these levels indicates the number of nested loops which occur between the two instructions. For those loops which are iterative loops, the elements of the resultant value list computed for the current OCELL will have to be multiplied by a factor which is a function of the number of iterations per loop entry (i.e. SCR.NITER) of the individual loops involved. Consider the three nested loop programs depicted by figure 5.E (see sample program R, appendix B).

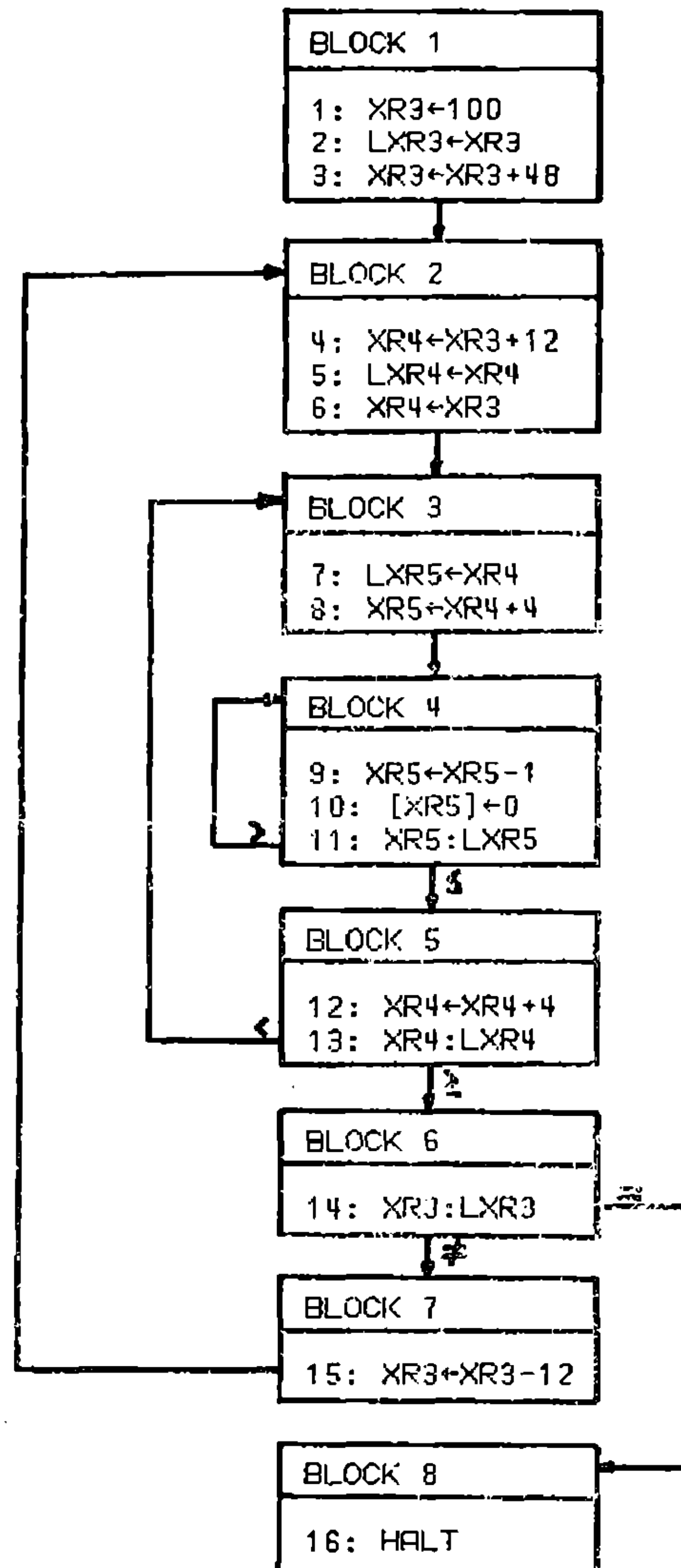


Figure 5.E - Nested Iterative Loop Example Program

SCR TABLE FOR FIGURE 5.E

NO.	LEV	NODE LIST	RDV LIST	NITER	ITV	DITV	EXB
1	3	{4}	{(XR5,9,-1)}	4	XR5	-1	4
2	2	{3,4,5}	{(XR4,12,4)}	3	XR4	4	5
3	1	{2,3,4,5,6,7}	{(XR3,15,-12)}	5	XR3	-12	6

In the above table the elements in the RDV list consist of triples of the form: (name, location, RDV increment). Suppose it is desired to compute NITER[2] (note: for simplicity the "SCR." qualifier is omitted for SCR table fields) of the above table where it is assumed that the other fields are already computed. The parameters for the equations E5.A-E5.C for this example could be computed using the VALSET procedure to give:

INVITV = VALSET(XR4,13,INITIAL,NOSAVE) = 152,

TVAL = VALSET(LXR4,13,INITIAL,NOSAVE) = 160.

Since the test relation required for exit from the loop (block 5) is "≥", equation E5.C is selected. Thus:

NITER[2] = CEIL[(160-152)/4] + 1 = 3.

The only indexed reference in figure 5.E occurs at instruction 10 (where [XR5] denotes the contents referenced by a pointer in XR5). Therefore, the goal is to compute the bounds on XR5 at 10. In this simple example, XR5 is also the loop iteration variable of SCR[1]; this need not be the case in general. To be of interest, the index should be an RDV of some loop containing the references. The first

step is to determine the initial value of XR5 at 10, and to save the associated C-graph. The C-graph is illustrated below. The initial value returned for XR5 would be 151.

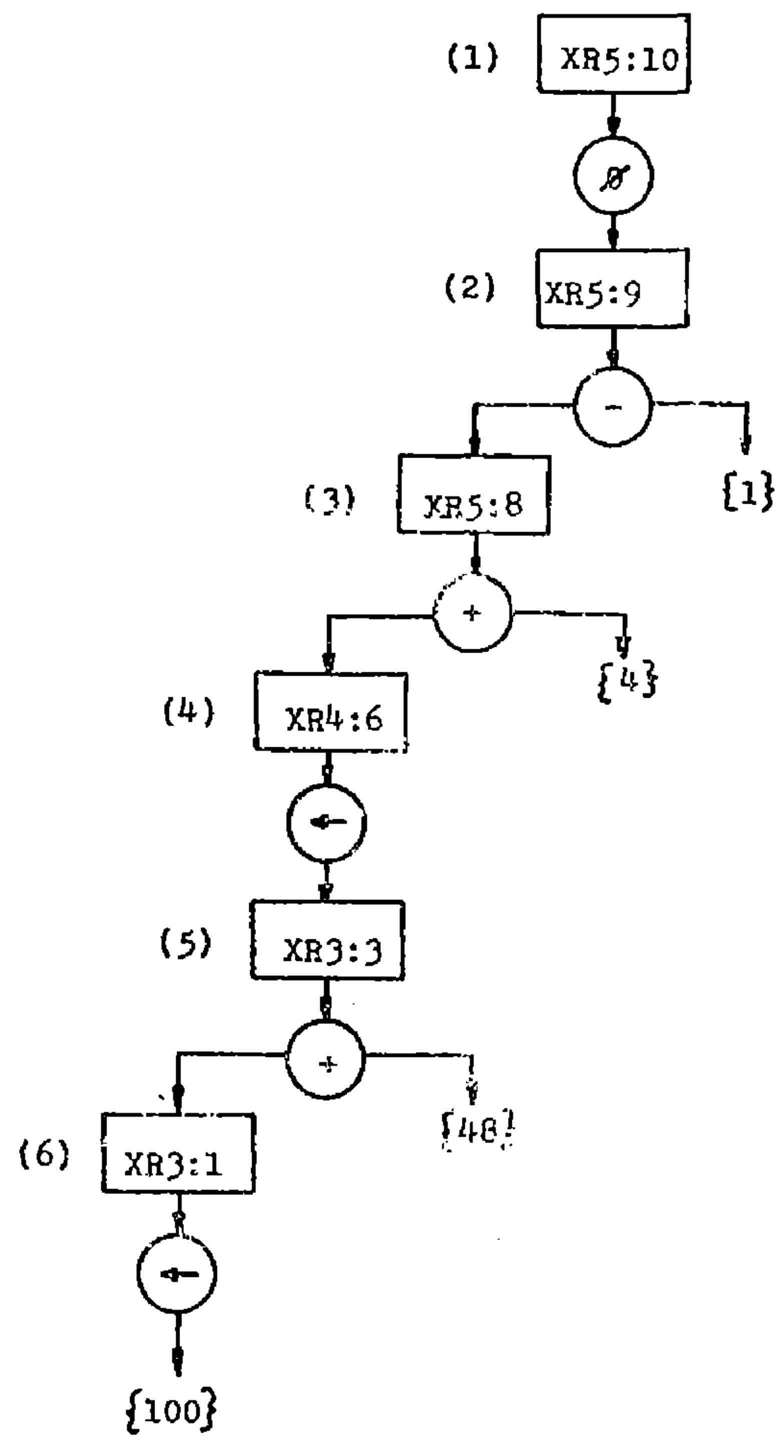


Figure 5.F - C-graph for VALSET(XR5,10,INITIAL,SAVE,CGP)

The nested region lists which would be constructed for this example would be:

```
NRL[1] = {1,2,3}
NRL[2] = {2,3}
NRL[3] = {3}
```

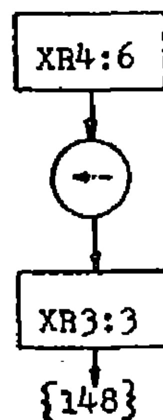
The fields NRLP[k] (k = 1,2,3) are 1, 2, and 3 respectively. Tracing through the C-graph of figure 5.F, the upper bound on XR5 is computed in the following manner.

Starting with the first (top) OCELL, the C-graph is traversed until it is determined that both operand value lists are complete. Each time an uncomputed OCELL is encountered during the traversal, it (i.e. a pointer) is pushed onto a data stack (DSTACK). The initial representation of DSTACK in this example would be:

```
(TOPDS)   XR3:1
          XR3:3
          XR4:6
          XR5:8
          XR5:9
          XR5:10
```

The value list of the top element (DSTACK(TOPDS)) can be computed (i.e. VL[XR3:1] = {100}). Before VL[XR3:1] is used as an operand value list for the computation reflected by DSTACK(TOPDS-1), the level difference must be checked. This is done in the implementation by first getting the block numbers of the two instructions (IT.BN field of IMTEXT). The block numbers are used to retrieve the level

numbers (BLKTBL.LEV) of the blocks, which can then be compared. Since the top two entries in DSTACK reflect instructions in the same block, their levels are the same and VL[XR3:1] is used directly as the operand value list for the next computation. The DSTACK is popped, and since both operand value lists are complete, the computation (+) is performed to yield a resultant value list for OCELL(5). The C-graph has been effectively reduced one level. At this stage, DSTACK[TOPDS] equals [XR3:3] and DSTACK[TOPDS-1] equals [XR4:6], and the associated C-graph elements are:



It is observed that the level of [XR3:3] is 0, while the level of [XR4:6] is 1. Thus, there is one loop separating the two computations. What is needed is the final value of XR3 at 6. Since IT[6] (instruction number 6 in figure 5.E) is in block 2, SCR[3] is the associated SCR table entry, where it is learned that XR3 is an RDV of SCR[3]. Therefore, the final value of XR3 at 6 is a function of the initial value of XR3 (i.e. 148) upon entry

to the loop, the number of iterations of SCR[3] per entry, and the increment of XR3 in its recursive definition within SCR[3]. Thus:

VL[XR3:6] = VL[XR3:3]+(NITER[3]-e[2,6]-1)*RDVL.DEL[1,3]

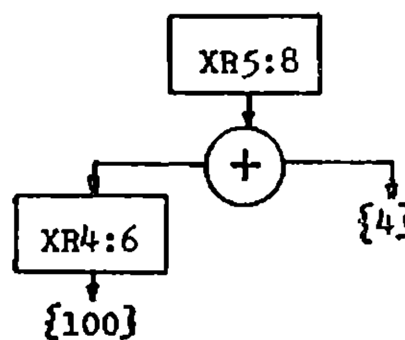
The e[i,j] term is a kronekr-delta function which is used to bias the number of iterations by -1 if the computation occurs in a successor block of the loop's iteration exit block.

e[i,j] = 1, if block i will be executed subsequently to block j.
= 0, otherwise.

Applying the "←" to effect the reduction yields:

<u>VL[XR4:6] = VL[XR3:6] = 100</u>

Now the C-graph can be reduced one level to produce the next pair of computations to be considered, which is schematically represented by the partial C-graph:



Again the difference in levels of [XR4:6] and [XR5:8] is 1, and XR4 is an RDV within a loop containing the definition of XR5 at 8. Applying the same procedure (except with a binary operator):

VL[XR5:8] = VL[XR4:8] + (4),

VL[XR4:8] = VL[XR4:6]+(NITER[2]-e[3,5]-1)*RDVL.DEL[1,2]

= 100+(3-0-1)4 = 108.

Thus: VL[XR5:8] = 108+4 = 112.

Repeating the procedure twice more yields the final result for [XR5:10].

VL[XR5:10] = VL[XR5:9]-1,

VL[XR5:9] = VL[XR5:8]+3*(-1) = 109.

Finally: VL[XR5:10] = 109-1 = 108

In this example the address part is zero, so that the initial and final effective address equal the initial and final values respectively of the index.

In general, given a subscripted reference OPND[k,L] with subscript I and address part A, if IV[I:L] and FV[I:L] denote the initial and final values of I at IT[L], then:

IEA[OPND[k,L]] = A + IV[I:L],

and FEA[OPND[k,L]] = A + FV[I:L].

The above simple example gives an intuitive understanding of the algorithm for finding the final value (FVALUE) of the index of an indexed reference. One omission was the manner in which the nested region lists are used. In general there may be n levels difference between the computations represented by two successive OCELLs. The NRLs are used to describe which loops exist between such computations. This allows the total number of iterations, seen by the RDV in question, to be computed as a function of the product of the number of iterations of the individual loops.

In the above example the C-graph is linear and the final value list is single valued. The FVALUE algorithm like the VALSET (or initial value algorithm) permits processing of a general C-graph in which a multivalued list of final values is returned.

Computing Upper and Lower Bounds

In general the initial and final effective addresses of an indexed operand do not correspond to the lower and upper bound respectively of the data area which is referenced. Upon closer examination of figure 5.E, it is discovered that the lower and upper bounds of the data area referenced by [IX5] in instruction 10 are 100 and 159

respectively.

The lower and upper bounds of the data area referenced by a subscripted operand, which is a function of one or more RDVs within one or more loops, can be computed in the following manner.

Let n denote the number of RDVs which occur in the C-graph used for computing IV[I:L] and FV[I:L] for OPND[k,L].

Let Rj denote an RDV which is involved in computing FV[I:L], where Dj is its increment. Let Nj designate the total number of times Rj is incremented per entry to the immediate loop which contains Rj, say SCR[k]. Nj equals NITER[k] if Rj is defined in a block which is a predecessor of EXB[k] (the iteration exit block); otherwise, Nj equals NITER[k]-1.

Then the subscript description list (SDL) for OPND[k,L] is defined as the list of triples:

{(R1,D1,N1),...,(Rn,Dn,Nn)}

All Rk, Dk, and Nk (k=1,...,n) are available during the computation of FV[I,L], and the SDL is easily constructed in the process. The lower bound (LB) and the upper bound (UB) of the data area referenced by the indexed operand OPND[k,L] are computed as:

$$\text{E5.D)}\quad LB = IEA[OPND[k,L]] + \sum_{i=1}^{n} (Ni-1)*Di*(Di<0)$$

$$\text{E5.E)}\quad UB = FEA[OPND[k,L]] - \sum_{i=1}^{n} (Ni-1)*Di*(Di<0)$$

(note: the term (Di<0) equals 1 if Di<0; equals 0 otherwise)

In the example:

LB = 151+3(-1)+2(4)(0)+4(-12)(1) = 100

UB = 108-(3(-1)+0+4(-12)) = 159

The above equations are based on the assumption that IV[I:L] and FV[I:L] are single valued. One interpretation of an indexed operand whose index is a function of n RDVs is that the operand is referencing an n subscripted array. Possibly this fact could be exploited in order to determine these "higher level" data structures and to translate them and their references accordingly.

The Indexed Data Area Table (IDAT)

The procedures described in the previous sections describe how to determine the bounds of all indexed operands of a certain class, namely those where the index of the reference is a single valued function of one or more RDVs in a nest of loops. The bounds for each such reference are recorded in the IDAT. Then each of these operands in

IMTEXT is set to reference its associated IDAT entry instead of the original indexed operand table (XOT) entry, thus reflecting a higher level representation of the storage structure. It should be noted that it may not be necessary to repeat the bounds finding procedure for <u>every</u> indexed reference. If two or more identical (i.e. identical XOT entries) operands occur in some local context such as in the same instruction or within a group of instructions within a block, and it can be ascertained that the index has not been redefined within the context, then the procedure need only be employed once for all like operands.

Fields for IDAT[k]:

IDAT.LB[k] - lower bound (memory address).

IDAT.UB[k] - upper bound (memory address)

IDAT.XOT[k] - pointer to XOT entry for the indexed operand(s) whose data area is described by the k-th IDAT entry.

IDAT.DCLA[k] - pointer to the array declaration table (ADT) entry. This table contains an entry for each discovered disjoint array. The limits of this array include the data area extent described by this IDAT entry.

<u>The IDAT to ADT Mapping</u>

The final step in determining the extent information necessary for a PL/1 declaration of the arrays implied by the data areas recorded in the IDAT entries is to map the IDAT extents into a sequence of disjoint areas. Each disjoint extent represents one entry in the array declaration table (ADT). These disjoint extents are computed by analyzing the IDAT extents for equality, overlap, and inclusion. An example of this process is given below:

IDAT Extents				ADT Extents		
NO.	LB	UB	DCLA	NO.	LB	UB
1	100	200	1	1	75	225
2	75	150	1	2	300	400
3	300	400	2			
4	175	225	1			
5	100	200	1			
6	325	350	2			

The other fields in ADT besides ADT.LB and ADT.UB are the array name (ADT.NAME), and the array attributes (ADT.ATR). An attempt is made to correlate all operand names to the original program names as reflected by the symbol table which is preserved from the MIX assembly process. For arrays, if ADT.LB[k] equals the value of one of the original program symbols (which references memory), then the associated symbol is used as the array name. Otherwise the array name is generated by the decompiler. The attributes of the array (character, integer, pointer)

are determined by their instruction context. Attributes will be discussed in some depth in chapter 6.

A HEURISTIC APPROACH

The method previously described analyzes iterative loops to determine data area extents; however, there are a number of situations involving data areas (arrays) where iteration does not occur. A common occurrence of this kind is when data structures involving pointer variables are employed, such as linked lists and trees. In this case indexed operands are used to reference elements within a common work area. Another case would be a hash table where the elements in the table are referenced in a pseudo random fashion by means of a hash key. The difficulty with this type of data area is that there is no analytic means for determining its extent precisely. In some cases, such a data area is _also_ used in an iterative context. For example, in a linked list application there may be an iterative loop for initially constructing a "free storage list" for dynamic storage allocation. In this case there is no problem since the _initial_ _effective_ _address_ of an indexed reference to this data area which occurs in a noniterative context, will fall within a data area which had been detected by the iterative loop analysis method. However, when an IEA of an indexed reference _does_ _not_ fall

within an existing data area another approach must be employed.

The problem is: given an IEA of an indexed reference in a noniterative context, which does not lay within an existing data area, how is the extent of the data area which contains this reference determined? A general solution to this problem involves scanning the indexed data area table (IDAT) in conjunction with the bit vector INBITS (chapter 2). Given an IEA the idea is to scan the address space in both directions from the IEA until some bounding criterion is reached. The algorithm is given below.

Nomenclature:

LCA, UCA - the lower (load address) and upper core addresses respectively of the program (input parameters to the decompiler).

IEA - the initial effective address of some noniterative indexed reference as previously described.

LDA, UDA - the tenative lower and upper bounds respectively of the data area being sought.

LIDAT - the number of entries in IDAT.

Procedure:

A. Initialize extent of data area to that of the program extent.

.1 LDA←LCA.

.2 UDA←UCA.

B. Attempt to find existing data areas which yield the smallest interval which includes IEA.

.1 j←k←0 (initialize IDAT pointers).

.2 i←1.

.3 If IEA ≥ IDAT.LBD[i], go to B.5.

.4 If IDAT.LBD[i] < UDA, {UDA←IDAT.LBD[i], j←i, go to B.6}.

.5 If IDAT.UBD[i] > LDA, {LDA←IDAT.UBD[i], k←i}

.6 i←i+1, if i < LIDAT, go to B.3.

C. Scan to see if there is an instruction block within IEA+1,...,UDA-1.

.1 L←IEA+1.

.2 If L = UDA, go to D.

.3 If INBITS[L] = 1, {UDA←L-1, j←0, go to D}.

.4 L←L+1, go to C.2.

D. Scan to see if there is an instruction block within IEA-1, IEA-2, ... ,LDA+1.

.1 L←IEA-1.

.2 If L = LDA, go to E.

.3 If INBITS[L] = 1, {LDA←L+1, k←0, go to E}.

E. Determine type of bound and if another data area borders the tenative data area found (on either extreme), then merge the data areas into one (i.e. if j≠0 or k≠0).

.1 If j ≠ 0, UDA←IDAT.UBD[j].

.2 If k ≠ 0, LDA←IDAT.UBD[k].

.3 Make new IDAT entry with extent LDA:UDA.

(Note: during the IDAT-ADT mapping process, the data areas IDAT[j] (if j≠0), IDAT[k] (if k≠0) and the new entry will map to a single array)

Several observations are in order concerning the above algorithm. Notice that bounds are determined by detecting a neighboring data area or instruction block (which ever comes first). It is possible that simple datum (recorded in SOT) will be included within the extent of the newly found data area. When a word which is simply referenced is encountered during the scan for bounds of a data area, it is not clear whether the word represents a simple variable or whether it is in fact a part of the data area and is referenced absolutely (i.e. constant subscript). Absolute referencing of array elements can occur when specific elements of the array are initialized in the program. The effect of the data area scan is to subsume

individual data items into the data area. If such a simply referenced datum served as a simple variable in the source program, its identity is lost. References to the variable will be ultimately translated as an array reference with a constant subscript (see chapter 6). While this would not be what the source program author had in mind, the referenced array element will in essence serve as a simple variable in the target program and will produce computationally correct code. The effect of this data merging is to lower the level of the target program translation. When the decompiler cannot absolutely guarantee that contiguous data are explicit (no overlap), they are merged. The result is a computationally correct translation, but one which may be difficult to understand.

The data merging principle is also applicable to arrays. Consider the arrays A, B, and an IEA in the following example.

```
  AAAA...A     IEA     BBBBB.....B
   ↑      ↑             ↑         ↑
 1000   1500    1600  1650      1800
```

The effect of the algorithm would be to produce one large array with bounds of 1000:1800.

CHAPTER 6

TARGET LANGUAGE GENERATION

The techniques used for target language generation are heavily dependent on the desired goals of decompiling. For example, if program conversion is the aim, then completeness and machine independence are primary considerations, while the level of the target translation is secondary. For purposes of documentation the reverse may be true. The approach of this study is to decompile to a reasonably high level in cases where relatively efficient algorithms can be developed for combining and simplifying a class of source statement sequences in a generalized fashion. In cases which do not readily lend themselves to regular treatment, the level of translation is sacrificed in order to achieve a more complete translation. The extreme case of this approach is when "crutch" code is generated in the target translation.

The target language generation process is comprised of two major phases: 1) the generation of data declarations, and 2) the generation of executable code. Before dealing with the data declarations it is important to understand how the storage structures of the source machine are mapped

into data structures of the target language.

DATA DECLARATIONS

Source Machine Storage Structure Mapping

In previous chapters methods have been described for determining the data areas in the memory of the source machine. The bounds of data areas and the names of these data are now available. However, nothing has been said concerning how the storage elements which comprise these data are mapped into data elements of the target language.

In order to formulate the rules for translating the source program storage structures into the target language data structures, several factors must be considered. Perhaps the primary criterion in this process is that the relationships among data elements in the source program must be preserved by the corresponding data items in the target program. This requires intimate knowledge of how the compiler of the target language maps data in the target program to the memory of the target machine. Ideally it would be desirable to have a language in which all data could be described in a completely abstract and machine independent manner. In addition, to achieve machine independence, all the compilers of this language for various

machines should map data to (memory) storage in a manner which guarantees that the relationships among interdependent data elements are preserved. In PL/1 this consideration applies largely to elements within PL/1 structures. In other words, while storage allocation for data is necessarily machine dependent, perhaps the portion of the mapping algorithms which determine the interrelationships among data could be made machine independent. It is difficult to reach the above goal because of two primary problems which arise with the common languages in use today: 1) the languages contain constructs which are explicitly or implicitly machine dependent, and 2) for reasons of efficiency, compilers tend to introduce machine dependencies. For instance a declaration of BINARY FIXED(23) in PL/1 specifies 23 bits (plus sign) of precision, which requires 3 bytes of IBM/360/370 memory. However, the compiler generates 31 bits of storage for any precision greater than 15, because it is more efficient to manipulate a full word. As will be demonstrated in this section, such machine dependent considerations must be allowed for when formulating the rules for mapping the source program data into the target language data declarations.

One interesting mapping is the case when partial words are referenced in the source program. Each source program

"word" is translated to a corresponding _major_ element in PL/1. A structure (or structures) is generated to provide a template which reflects all the partial word accesses to the word. Each data element in the structure is a _minor_ element. MIX is a "word" and "byte" machine, where each word is comprised of five bytes and a sign. The MIX architecture permits accessing of an entire word or an arbitrary byte subfield of a word. Thus, the accesses of some word in MIX may imply a rather involved structuring of the word. Consider the field references of: 0:5, 1:3, 4:5, and 5:5. These references of a MIX word establish a heirarchy of fields and subfields which are not independent. An essential ingredient of the data mapping is that the data structures which are generated in the target language _must_ preserve the original structuring.

In the data translation process it has also been found necessary to consider the data types of the source program data elements. Suppose some MIX word is accessed and the contents of this word represents five characters of some alphameric string. Such a word would translate directly into a five byte character string in PL/1. However, if this word is used as a variable in some arithmetic computation, the base, scale, and precision of its PL/1 counterpart must be determined. In this implementation the translation of MIX floating point computations is not

handled, so all arithmetic variables are integer. An integer variable (whole word) in MIX has a precision of forty bits (assuming 8 bits/byte). In order to avoid data alignment difficulties in the IBM/370, a decimal base is used. Thus, an entire MIX word of type integer would translate to a PL/1 declaration with the attributes: DECIMAL FIXED(13,0). This translation preserves the precision of the original MIX variable (3.32 bits/dec * 13 dec = 43.16 bits). Further analysis could be performed to learn more about the data in order to conserve storage in the target machine. For example, if a "read only" constant 2 resides in a full MIX word in the source machine, it could be represented in the target machine with a much smaller precision declaration than thirteen decimal digits. Such storage optimization analysis is not implemented in this decompiler. A declaration precision of thirteen decimal digits generates seven bytes of IBM/370 storage. If there are any partial field accesses to this MIX word, they are mapped isomorphically onto the first five bytes of the storage generated for the "decimal fixed" declaration. This is done by using PL/1 "based" variables and will be described later. In this mapping it must be assumed that the referencing of the entire word as an integer is independent of the subfield references. This assumption should be valid except in pathological cases.

In the above discussion it is seen that the attributes of the source data, the features of the target language, and the storage characteristics of the source and target machines influence how the source data storage structures are translated into the target language data declarations.

Data Attributes

The attributes of the source data can be determined by the data usage in the computation. For example, if an I/O operation is performed the attribute of the I/O buffer is "character". Consider the sequence:

```
1      LDA  S
2      CHAR
3      SLAX 0,1
4      SLA  1
5      INCA 40
6      STA  BUF,2
7      STX  BUF+1,2
```

Example 6.A

The CHAR instruction in MIX converts the integer contained in the A-register into a ten byte character representation of the number. The result replaces the contents of the AX-register. Thus the A-register in 1 is of type "integer", and the AX, A, and X registers are of type "character" beginning with instruction 2. Instructions 3 and 4 can be viewed as shift operations on the resultant character (sub) strings. Instruction 5 presents an

interesting situation. Notice that if this instruction were considered out of context it would appear as the arithmetic computation: A←A+40. However, at this point it is known that A has type "character". The question arises: how can an arithmetic operator be applied to a character string? To resolve such anomolies, the local context of the instruction must be analyzed. In this example, instructions 4 and 5 can be treated as a "shift and mask" operation, where the constant 40 is viewed as a MIX character code (period).

Before discussing how this sequence would be translated into PL/1, another problem must be dealt with, namely that of mapping storage elements which have multiple data type attributes. Notice that the attributes "integer" and "character" are both applied to the A-register in the above sequence. In machine language this multiple usage of storage elements is frequently encountered with the registers, although it can occur with core memory data as well. Translating multiple usage data elements into PL/1 data declarations can be handled conveniently by simply generating a separate declaration for each usage context. The corresponding operands are translated to reference the appropriate declaration depending on its context (current attribute) in the source program. In order to preserve storage in the target machine, these declarations may be

equivalenced by using based variables. In the implementation the MIX A, X, and AX registers are translated into the following declarations:

```
DECLARE 1 RAPX,
          2 RA DEC FIXED(13),
          2 RX DEC FIXED(13),
          2 FILLAX CHAR(2);

DECLARE RAX CHAR(16) BASED(PRAPX);
```

The pointer PRAPX is initialized to the address of RAPX.

The CHAR and NUM operations translate into assignment statements in PL/1, since the data conversion is done automatically. The "substr" pseudo function is used to handle the character string shift operations. Thus, the sequence in example 6.A would translate into:

```
RAX=SUBSTR(RAX,1+IXR1);
BUF(100+IXR2)=SUBSTR(RAX,8,4)||'.';
BUF(101+IXR2)=SUBSTR(RAX,12);
```

The contextual analysis of instructions involving multiple attributes is not done automatically in this decompiler, and attribute analysis is performed in a limited way. However, it appears that these functions could be automated without serious difficulty.

The above discussion is inherently dependent on the usage of PL/1 as a target language. Much of the attribute analysis described above is necessary only because of the complex automatic data conversion rules among data types

in PL/I. In order to achieve an accurate translation, it is critical that the attributes of each operand are determined correctly. The generation of the correct data declaration can become very complex. Also, the semantics for translating computations is not only a function of the source instruction operator, but also the attributes of the instruction operands. If the target language were a "type ignorant" language (Lang, 1969) the data type analysis phase of a decompiler could be completely eliminated. Such a language allows the storage attributes (BIT, BYTE, WORD, etc.) and size to be declared but lets the operator determine how the data operands are to be manipulated. This approach is used in the design of some of today's systems languages (Lang, 1969).

Declarations of Unstructured Data

The storage structure of program data in the source machine can be viewed as consisting of major and minor storage elements. The major storage element is defined as a word of memory or one of the machines registers. A minor storage element is defined as some proper byte subfield of a major or minor storage element. An _unstructured_ datum is a major storage element such that no subfields within the major storage element are referenced in the program.

All the information needed to declare unstructured data is available in the operand and symbol table. If possible the datum name is derived from the original symbols in the source program. For simple core data a hashing technique is used to map the address of a datum to its original source program name. If such a name does not exist, a name of the form: C<datum address> is generated. This technique guarantees name uniqueness since two data cannot occupy the same storage location and also relates the target language name to its source program location. The name IXRi (i+1, . . . ,6) is generated for index register declaration.

For array variables the name is retrieved from the array declaration table (ADT – Chapter 5). The bound of an array are taken as the memory extent of the array as recorded in the ADT. This technique allows all subscript computations of the array references to be mapped directly into the target language.

The data types for unstructured data result in one of the following:

TABLE 6.A – Unstructured Storage Element Mappings

	CHARACTER	INTEGER
Index Register	CHAR(2)	DEC FIXED (7)
Memory Word	CHAR(5)	DEC FIXED (13)

Declarations of Structured Data

In machine language it is a common practice to subdivide memory words into subfields in order to conserve memory. This is especially prevalent in programs involving pointer manipulation. As mentioned previously, source program words (major elements) which contain minor elements are translated into PL/1 "structures". It is a straightforward procedure to scan the operand tables to determine all the subfield references for a particular storage location (or range of locations if it is an array). Suppose such a scan for some datum, say Y, results in the following subfield reference list (SRL):

SRL = {5:5,0:3,2:2,4:5,2:3}

The first step is to reorder the list to reflect the structure. In this reordering process a structure level number is assigned to each element. The field 0:5 is added to the list to reflect the major storage element. The reordered list would result in:

SRL' = { 0:5, 0:3, 2:3, 2:2, 4:5, 5:5}
(level) 1 2 3 4 2 3

This reordered list can now be used to translate the original structured datum into a corresponding PL/1 data structure. Due to the manner in which PL/1 structures are

defined, this translation is quite awkward. Ideally one would like to be able to generate a structure of the form:

```
DECLARE 1 Y ...  ,
     2 FLD03 ...  ,
       3 FLD23 ...  ,
         4 FLD22 ...  ,
     2 FLD45 ...  ,
       3 FLD55:
```

where the "..." represent attributes. Such a translation, for example, is acceptable with IBM's PL/S (Weiderhold, 1972). In PL/1, however, only the lowest level of a data structure may have data attributes. All other levels act simply as nodes of the structure. This deficiency is handled by generating a separate declaration for each level and equivalencing them via a based variable. In the above example, the following declarations would be generated.

```
1)        DECLARE Y ...  ;

2)        DECLARE 1 Y2 BASED(PY),
               2 FLD03 ...  ,
               2 FLD45 ...  ,

3)        DECLARE 1 Y3 BASED(PY),
               2 FILL1 CHAR(1),
               2 FLD23 ...  ,
               2 FILL2 CHAR(1),
               2 FLD55 ...  ;

4)        DECLARE 1 Y4 BASED(PY),
               2 FILL3 CHAR(1),
               2 FLD22 ...  ,
               2 FILL4 CHAR(3);
```

<u>Example 6.B</u>

The pointer PY is initialized to the address of Y. The fields FILLk are "filler" fields which serve to maintain the proper relationships among the original fields.

The data elements within based structures are given default attributes according to set conventions. The correctness of these attributes is determined manually. Experience has shown this to be a straightforward procedure. In most of the cases tried, the default attributes have been sufficient. When errors are found it is usually a clerical task to determine the appropriate attribute and edit the corrections into the target program. Given a MIX subfield reference j:k, the default attribute of the corresponding PL/1 data item is determined according to the following table.

TABLE 6.B - Default Attributes for Structured Data

j	k	Attributes
0	1	BIT(8)
0	2	BIN FIXED(15,0) UNALIGNED
0	3	BIT(24)
0	4	BIN FIXED(31,0) UNALIGNED
>0	≥j	BIT(8*(k-j+1))

The above defaults obviate all alignment difficulties in the IBM/370 and always cause a data element to occupy the exact number of bytes which is implied by the original MIX reference. For instance a MIX field reference of 0:3 would occupy three bytes of a MIX word and would cause a

corresponding PL/1 declaration of the form:

<structure-name>.F1003 BIT(24),

to be generated, which specifies that three bytes of IBM/370 storage are to be allocated for this data item. Although the field specifications 0:1 and 0:3 include the sign of the MIX word, a BIT attribute instead of BIN FIXED is assigned to its PL/1 counterpart. This is because the PL/1 compiler does not permit the allocation of an odd number of bytes for BIN FIXED data, which is necessary if the integrity of the data structure mapping is to be maintained. If the program uses these fields to hold negative numbers, then manual correction is necessary.

Data Initialization

The declarations of data which correspond to storage locations in the source program which contain assembled constants must also be initialized in the corresponding target language program. This initialization is done in the generated PL/1 program either statically (INITIAL attribute) or dynamically (assignment statement) depending on the nature of the initialization.

The initialized memory locations are determined by scanning the Initialized Core Memory Table (ICMT). Two

conditions may cause the generation of a dynamic initialization statement:

1) When a <u>partial word is initialized</u> in the source program, its corresponding representation in the target program will be that of a subfield of some based structure. Initialization of based variables using the PL/1 INITIAL attribute is not allowed in this case, and therefore, an executable assignment statement must be used to accomplish the initialization.

2) If some locations within a detected array are initialized, dynamic initialization is also used. Here, convenience in target code code generation is the rationale rather than necessity. Once the declarations have been generated, it is straight forward to scan the ICMT and generate the appropriate initialization assignment statements.

Static initialization is used for simple operand variables which are not within some array extent (discussed below) and are not part of a based structure. If the SOT entry for the simple variable contains a pointer to the ICMT, the initial value in the ICMT entry is used as the parameter in the PL/1 INITIAL attribute.

Operand Name Generation

Once the data for the source program have been mapped into the target language declarations, there is still the problem of translating the IMTEXT operand references to their proper PL/1 symbolic representation during the IMTEXT-PL/1 translation phase.

If an operand is unstructured and does not reference a datum with multiple data types, the procedure is straightforward in that there is a one to one mapping between the source datum and the PL/1 name. If, however, a datum has n data types in the source program, then n target declarations have to be produced in the mapping process. Similarly, if m levels of structuring of a datum are present due to partial word referencing, then m based structures must be generated. Consequently, there may be up to n*m names in the PL/1 translation which correspond to the one source program datum. When translating an (IMTEXT) operand, the problem is to select the proper name. This involves analyzing the context of the operand in order to determine its data type. If the operand references a partial word, its data type and field specification can be used to determine the appropriate qualified PL/1 name. Experience indicates that cases which involve a combination of the multiple attribute problem and multilevel referencing for the same datum occur infrequently. The generation of

based structures and their corresponding qualified references is done automatically in the implementation.

For an operand which references a subfield of a structured datum, the properly qualified reference for its PL/1 translation must be constructed. This reference is comprised of a major part and a minor part. In example 6.B the names "Yj" (j=1,2,3,4) correspond to the major part, and "FLDsf" corresponds to the minor part, where "j" designates the structure level and "sf" indicates the subfield being referenced. Given an INDEX operand, the qualified name is constructed from the base name (in example 6.B the base name is "Y") and the level of and the subfield specification. The subfield of an operand reference is available in the operand tables (XOT,SOT). When a structured datum is processed for its PL/1 declaration, its subfield reference list (SFRL) is saved. The structure level of the operand referenced is retrieved from the appropriate SFRL whose entry corresponds to the operand field specification. With respect to example 6.B, an operand reference of Y(2:3) would translate into the qualified name of Y3.FLD23.

Simple References To Decompiled Array Variables

A simple reference to an array variable is one in which

the subscript is a constant. This type of reference occurs when initializing various elements of an array either implicitly or explicitly. Simple references to a decompiled array can also result from the decompilation analysis. If the array bounds were not determined precisely, then the array may represent a group of data elements which have been merged (chapter 5). That is to say the author of the source program may have treated the original storage cells corresponding to the one decompiled array as n distinct data elements (either arrays or simple variables). Due to a lack of information the decompiler is sometimes forced to merge a group of contiguous data elements into one composite array element. In fact, some of these array elements may have corresponded to simple operands in the source program. This implies that a variable recorded in the simple operand table (SOT) is also an element of a decompiled array. The problem then is how to alter the tables of the decompiler so that these types of variables are represented properly in the target language, namely as array variables with the correct subscript and attributes. The naming problem is handled by merely setting the symbolic name of the SOT entry to the proper array element (i.e. array name and constant subscript). Using the name of the decompiled array found in the array declaration table (ADT) and the memory location of the data element, this name is easily constructed. Henceforth,

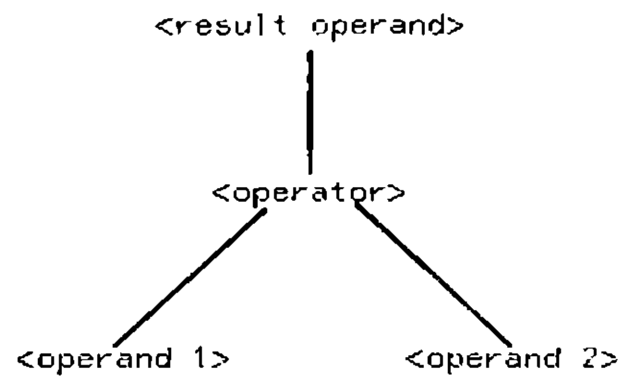
during code generation these variables are treated as simple variables whose symbolic name corresponds to the appropriate array element. Initialization of all simple references to array elements is done dynamically.

ARITHMETIC EXPRESSION TRANSLATION

One class of sequences which is amenable to decompilation to a high level representation in the target language is that of arithmetic expressions.

Expression Tree Generation

This phase of the analysis effectively transforms sequences of 3-tuples of the intermediate text into a tree representation, which will later be translated into an infix arithmetic expression. The concept involved in the algorithm is to scan sequences of instructions within each block. If it is discovered that the result operand of some instruction is a source operand in a subsequent instruction, then these two instructions can be combined, providing certain conditions are satisfied. This results in a single instruction. This process is called "reduction". Each original 3-tuple (arithmetic) can be thought of as an expression tree of the form:



When two instructions are reduced into one, the effect is that the expression trees of the original instructions are coalesced to produce an expanded tree for the resulting instruction.

The reduction algorithm utilizes the original 3-tuple (IMTEXT) data structures by chaining 3-tuples together to form an n-tuple which represents an expression tree. As in the compression algorithms (chapter 3), busy status of variables is used to determine if two n-tuples (expression trees) may be reduced. Only intra-block sequences are considered during the reduction process. The reduction algorithm can be summarized by the following steps. This procedure is applied to all blocks in the program.

Reduction Algorithm

Nomenclature:

A *definition n-tuple* is an n-tuple which defines some result operand R.

The *busy n-tuple* is the first n-tuple subsequent to the definition n-tuple (within the same block) in which R is a source operand.

A. Scan for the first definition n-tuple (call the result operand R). If none, TERMINATE.

B. Find busy n-tuple for R. If none, go to F.

C. If R is busy past the busy n-tuple, go to F.

D. If any source operands of the definition n-tuple have been redefined between the definition and busy n-tuples, go to F.

E. Replace all R in the busy n-tuple by a pointer to the definition n-tuple, and set the result operand in the definition n-tuple to null.

F. Scan for the next n-tuple.

.1 If not end of block and a definition n-tuple has been found, go to B.

.2 If any reductions have been made in this pass, go to A.

.3 TERMINATE.

The beginning of an n-tuple in IMTEXT is detected by

scanning for a non-null result operand in IT (the intermediate text array). The pointers which replace the intermediate operands (step E) can be thought of as "nonterminal" operands, where the terminal operands are the original operands. Determining busy status is done as described in chapter 3 with the exception that the n-tuple may have more than three operands. Determining all these operands involves chaining through the nonterminal operands in order to retrieve all the terminal operands. The following example serves to illustrate the algorithm just described. It is assumed that no operands are busy on exit.

k	IT[k]
1	ADD T1,B,C
2	MUL T2,T1,D
3	SUB T1,T1,E
4	DIV T3,T2,G
5	ADD X,T3,T1

The first definition n-tuple found occurs at 1 and its busy n-tuple is at 2; however, another busy occurrence of T1 is found at 3, thus violating the condition in step C. In otherwords T1 cannot be eliminated in 1 because it would invalidate the source operand T1 in IT[3]. The next definition and busy n-tuples occur in IT[2] and IT[4] respectively. Here, a source operand (T1) used in computing T2 at 2 is redefined at 3 which violates the condition in step D. Thus if IT[2] and IT[4] were reduced, the computed

value of T2 in the resultant n-tuple (which would begin at 4) would be in error. The first pair of n-tuples which satisfy the prescribed conditions is found at 3 (definition) and 5 respectively (busy) respectively. The subsequent reduction would yield:

```
1          ADD T1,B,C
2          MUL T2,T1,D
3          SUB null,T1,E
4          DIV T3,T2,G
5          ADD X,T3,(3)
```

The next reduction is made by combining 4 and 5 to produce:

```
1          ADD T1,B,C
2          MUL T2,T1,D
3          SUB null,T1,E
4          DIV null,T2,G
5          ADD X,(4),(3)
```

This terminates the first pass. Notice that the number of n-tuples has been reduced from 5 to 3. Commencing the second pass (step F.2), it is seen that the n-tuple beginning at 2 can be combined with that beginning at 5, since T1 is no longer redefined between the definition and busy n-tuple. This reduction results in:

```
1          ADD T1,B,C
2          MUL null,T1,D
3          SUB null,T1,E
4          DIV null,(2),G
5          ADD X,(4),(3)
```

This is the only reduction possible in pass 2. Now there are only 2 remaining n-tuples occurring at 1 and 5 respectively. The n-tuple 1 can now be combined with 5.

Although T1 occurs twice in the n-tuple beginning at 5, there is only one busy occurrence for the purpose of reduction. The final result then is:

```
1        ADD null,B,C
2        MUL null,(1),D
3        SUB null,(1),E
4        DIV null,(2),G
5        ADD X,(4),(3)
```

The tree representation for this result is shown in figure 6.A.

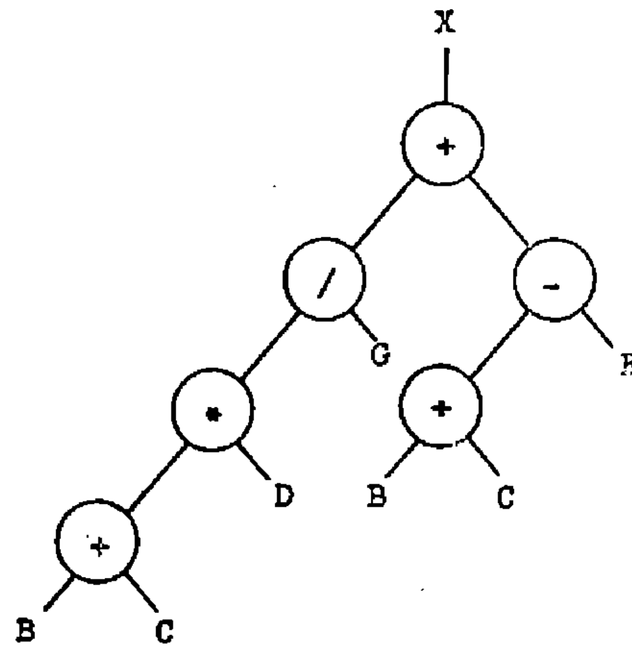


Figure 6.A - Expression Tree For an n-tuple

Generation of Infix Arithmetic Expressions

During target language generation, when an n-tuple is encountered, it is translated into a target language statement in two phases: 1) n-tuple to polish postfix, and 2) polish to infix notation with no redundant parenthesization. Given a tree representation, the polish representation is easily derived.

Converting a given polish string to an infix expression with a minimum number of parentheses involves the precedence and the algebraic properties of the operators. The idea is to scan the polish string from left to right until a triple of the form:

<operator> <operand>1 <operand>2

is detected. This triple is converted to an infix expression and the triple is replaced by a pointer to the infix expression. Associated with the pointer is the infix operator of the expression. The pointer is then treated as an (nonatomic) operand which can then be used as part of a polish triple in the reduced polish string. The process is repeated until the entire polish string is converted to infix. If a polish triple contains a nonatomic operand, then analysis must be performed to determine if the infix expression designated by the operand must be parenthesized before the triple is converted to infix.

This is done by examining the precedence and algebraic properties of the polish operator and the operator associated with the infix operand.

If only the operators: +, -, *, and / are considered, where {+,-} and {*,/} are assigned a precedence of 1 and 2 respectively (i.e. Pr(+)=Pr(-)=1, Pr(*)=Pr(/)=2), then the following parenthesization rules can be applied.

Let <inop>i designate the infix operator of the infix expression associated with <operand>i. If <operand>i is an atomic operand then <inop>i is null.

<u>Rule 1:</u> If <operand>i (i=1,2) is a nonatomic operand and Pr(<operator>) > Pr(<inop>i) then parenthesize <operand>i when translating the polish triple to infix.

When the precedence of <operator> and <inop>i are equal, the operand expression may have to be parenthesized depending on the operators and the value of i. Since expressions are evaluated left to right and due to the fact that the associative property holds for addition and multiplication and does not hold for subtraction and division, the following relations hold for the operands A, B, and C:

a) A+(B±C) = A+B±C

```
b)   A*(B*C) = A*B*C
c)   A*(B/C) = A*B/C
d)   A-(B+C) ≠ A-B+C
e)   A/(B*C) ≠ A/B*C
f)   A/(B/C) ≠ A/B/C
```

If the terms in the parentheses are thought of as nonatomic operands in a polish triple, it is evident that the value of the polish operator and the position of the nonatomic operand in the triple determines the need for parentheses. For example:

```
-[A+B]C => A+B-C
-A[B+C] => A-(B+C)
/[A*B]C => A*B/C
/A[B*C] => A/(B*C)
```

(Note: the brackets [] above are not parentheses, but simply denote a nonatomic operand in the polish triple)

<u>Rule 2:</u> If Pr(<operator>) equals Pr(<inop>2) and <operator> is "-" or "/" then parenthesize <operand>2.

The above rules are illustrated in converting the following polish string to infix notation. The underlined operator within a bracketed expression indicates the most recent infix operator (i.e. <inop>i) generated in the infix expression. Consider the polish string:

```
+//+ABCD--EF/G*HK
```

Initially all atomic triples can be converted to infix to produce the following polish string with three nonatomic

operands:

+//[A+B]CD-[E-F]/G[H*K]

Conversion of the triple: /[A+B]C requires "A+B" to be parenthesized according to rule 1.

=> +/[(A+B)/C]D-[E-F]/G[H*K]

Reducing the triple: /G[H*K], and applying rule 2:

=> +/[(A+B)/C]D-[E-F][G/(H*K)]

Reducing the triple /[(A+B)/C]D, yields:

=> +[(A+B)/C/D]-[E-F][G/(H*K)]

Reducing triple -[E-F][G/(H*K)], produces:

=> +[(A+B)/C/D][E-F-G/(H*K)]

Finally reducing the above triple yields:

(A+B)/C/D+E-F-G/(H*K).

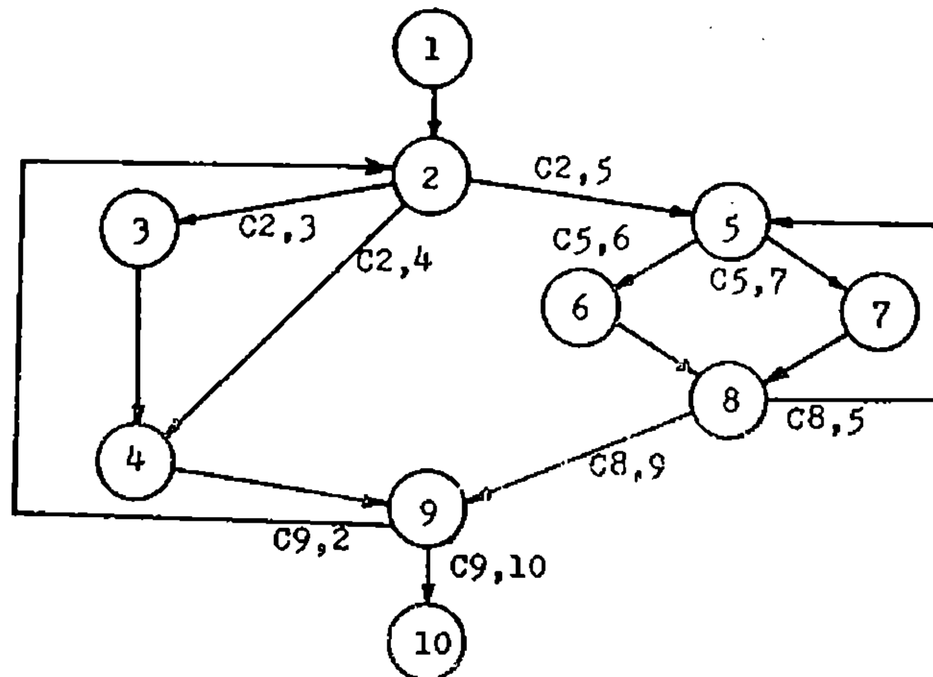
TRANSLATION OF CONTROL STATEMENTS

When a block terminates with a conditional jump sequence then the block must have more than one immediate successor. A jump sequence is analyzed as a group and is translated into some type of "if-then-else" construct in PL/I. If

a block terminates with a single (absolute) jump instruction, the implied immediate successor block is determined. If this block is not the next block to be translated, then a "goto" instruction is generated in the target program. Otherwise nothing is generated; this in effect eliminates the redundant "jumps" which were introduced in the intermediate text (chapter 2). A "halt" instruction merely generates a PL/1 "RETURN" statement.

The translation of the JUMP instructions not only results in target code, but also determines the order in which blocks are translated. The immediate successor of the current JUMP instruction being processed is determined and if a conditional expression is being processed, an analysis is performed to determine whether or not the immediate successor can be treated as the initial block of a DO-group. If this is the case the DO-group is generated as the body of code for one of the alternatives in the conditional expression. Otherwise the immediate successor is placed on the Next Block list. This procedure is recursive since blocks within a DO-group being translated may result in subsequent DO-group generation.

Consider the following program graph, where Ci,j is the condition to be satisfied for block i to transfer to block j.



The corresponding PL/1 structure for the above graph is given below, where Li is the label of block i and Bi is the noncontrol instructions for block i.

```
1    MAIN:B1;
2    L2:  B2;
3          if C2,3 then do; B3 end;
4          else if C2,4 then goto L4;
5            else
6          L5: do; B5
7               if C5,6 then do; B6 end;
8               else do; B7 end;
9               B8
10              if C8,5 then goto L5;
11              else goto L9;
12              end L5;
13   L4:   B4
14   L9:   B9
15         if C9,2 then goto L2;
16         else do; B10 end;
17         end MAIN;
```

Figure 6.8 - "DO-group" Translation

DO-Groups

When analyzing the immediate successor block referenced by a JUMP within a conditional jump sequence, the control flow graph must be analyzed to ascertain if a DO-group can be generated as the code to be executed if the given condition is satisfied. Namely, if n is the block referenced by such a JUMP instruction, within block k then either:

1) IP[n] = k, or

2) k is in IP[n], and n is the header node of some SCR[j] such that all m≠k in IP[n] must be a member of SCR[j].

If the first criterion is satisfied, the DO-group will consist of only one block. With the second criterion the DO-group is comprised of a single entry SCR and all blocks in the SCR will be translated as part of the DO-group. In figure 6.8 blocks 3, 6, 7, and 10 qualify as DO-group constructs according to the first criterion, while the SCR consisting of {5,6,7,8} comprises a DO-group under the second condition. Blocks 4 and 9 do not qualify since they have more than one immediate predecessor. Blocks 6 and 7 form DO-groups within the DO-group {5,6,7,8}. The corresponding PL/1 construct is illustrated in lines 6-10 of figure 6.8. In order to handle the recursion encountered

when processing a "DO-group nest", a push down stack (DOGSTK) is provided. Information such as the current instruction counter (to INTEXT), the locations of the first and last instructions of the current jump sequence being processed, the label of the new DO-group, and line indentation is stored in each DOGSTK entry. When a DO-group is detected, the processing of the current conditional jump sequence is interrupted, and translation of the initial block of the new DO-group is initiated. When all the blocks in the DO-group have been translated, the bracketing "END" statement is generated, the DOGSTK is popped, and processing of the previous jump sequence being translated is reinstated.

DO-groups and The Order of Block Translation

The Next Block List (NBL) serves as a FIFO queue which contains block numbers of blocks which have yet to be translated. The list is initialized to {1} since block 1 is the program entry block. Elements are added to the NBL as a result of analyzing the jump instruction sequence of the block being translated. Namely, if the immediate successor of the current block does not qualify as the initial block of a DO-group, then it is added to the next block list. In this case if a conditional jump instruction is being processed then a PL/1 "goto" statement is generated

as the body of the appropriate alternative of the *if-then-else* construct being generated. However, if an absolute jump instruction is being analyzed (either alone or terminating a conditional jump sequence), a comparison is made between its (implied) immediate successor and the next block to be selected for translation in the NBL. If they are equal then a "goto" is not generated because execution will fall through from the current block to that referenced by the absolute jump.

If a conditional jump instruction references an implied DO-group construct, two situations can occur. If the DO-group consists of a single block, then that block is selected as the next block to be translated and, therefore, it is not necessary to add it to the NBL. If a multiple node DO-group (single entry SCR) is referenced, it is mandatory that all the nodes in the SCR be translated before the next entry in the NBL is selected. In effect these nodes are treated as a composite node in order that the entire body of the DO-group can be translated before other portions of the program. To handle this, a new NBL is created. This NBL is initialized to the *header* block of the DO-group SCR and is used to determine the next block to be translated within the DO-group subgraph. When all the blocks of the SCR have been translated the list is eliminated (popped) and the previous NBL is used as the

current NBL. This procedure results in a <u>stack</u> of <u>next block lists</u> NBL[1],...,NBL[n], where NBL[n] is the current NBL, and NBL[1] is the list initially created. A pointer to the proper NBL is stored in the appropriate DOGSTK entry. A scheme similar to that described previously is used to determine if generation of the final GOTO statement is necessary. If the absolute jump of the last block in the DO-group which is translated equals the next block to be translated in the <u>previous</u> NBL, then a GOTO is not generated.

<u>Comments</u>

The way in which DO-groups are recursively generated during the analysis of conditional jump sequences usually results in a target language program which is a more highly structured representation of the algorithm than that of the original program. The output is formatted to reflect the nesting level of each DO-group.

The algorithm for determining the order of block translation produces an interesting effect. The code produced for blocks which are strongly related because of control flow (e.g. multiple node DO-groups) tends to be <u>physically</u> close together in the target program regardless of their physical placement in the source machine. This

has the effect of producing a more readable program. Also, the degree of "locality of reference" (Denning,1968) is increased, which may result in greater execution efficiency in a paging environment.

Another effect, due to the generation of DO-groups and the elimination of redundant transfers, is that the number of explicit "gotos" is in some cases substantially reduced. The subject of "goto-less" programming has received much attention in the literature (Leavenworth,1972). The concensus appears to be that reducing the number of explicit transfers (at least to some degree) results in programs which are more understandable and easier to maintain.

In summary, the techniques described in this section for translating the control statements result in a level of translation which takes advantage of some of the PL/1 structuring features. Some extensions of these techniques would be to generate more complex DO-group constructs such as iterative and conditional DO-groups.

CHAPTER 7

EXPERIMENTAL PROCEDURE AND RESULTS

In order to test the algorithms described in previous chapters a decompiler, written in Fortran, was implemented on the CDC6500. Using this experimental decompiler six of Knuth's (1969) MIX algorithms were decompiled and executed on an IBM/370. These MIX programs represent a variety of applications and coding techniques which serve to demonstrate the features contained in the implemented decompiler and the scope of the methods used. These test cases are found in appendix B. In addition to these test cases, a number of others were coded for testing individual components of the decompiler during development.

The experimental procedure is depicted by the following block diagram.

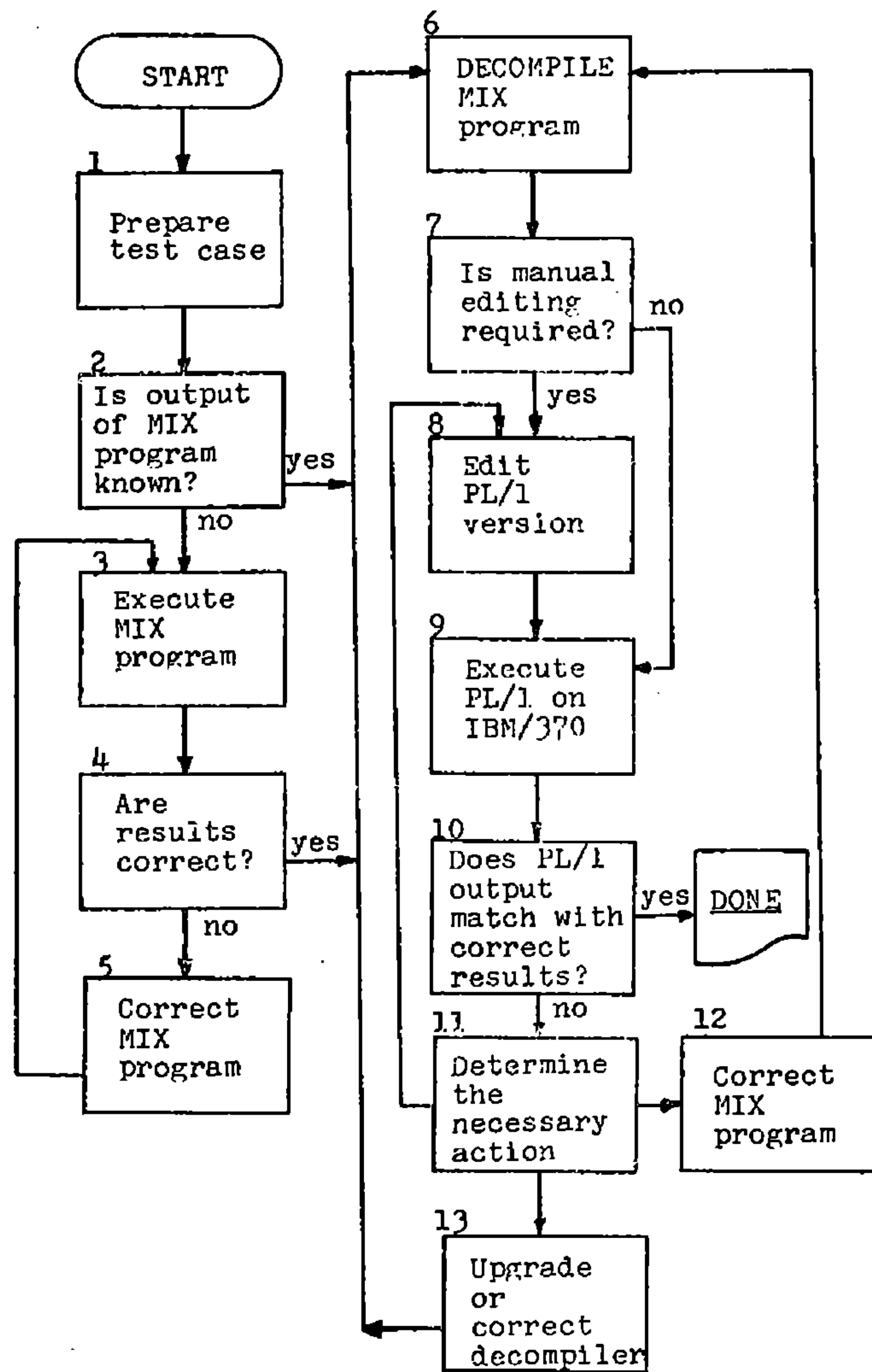


Figure 7.A - Experimental Decompilation Procedure

The initial step of the procedure is to prepare the MIX test case. In most cases this involved only punching the MIX program directly from Knuth (1969); however, in some instances it was necessary to create a driver program for the test case.

To verify that the decompiled program is correct, the output of the source program must be known for specific test data. In some of the test cases data and results are published by Knuth. In other cases a description of the algorithm is given and results can be predicted. With the above situation the "yes" branch is taken in block 2; and if the executed PL/1 program returns the expected result, (block 8) it was assumed that the translation was correct. If the result was in doubt, the MIX source was executed via a MIX assembler-interpreter written for the CDC6500. The output of this execution is examined for correctness (block 4) and the appropriate action is taken (block 5 or 6). Errors here were due either to keypunching or missprints.

Once it is decided that the source program is correct it is ready for decompilation (block 6). The output is examined manually (block 7) to determine the need for editing. Some editing of the target program was required for each test case, namely that of providing the necessary I/O. This is a clerical procedure which consisted of coding

the appropriate PL/1 GET and PUT statements in lieu of the MIX IN and OUT instructions or adding I/O statements, if required. The remainder of the editing, if any, involves recognizing cases which are not handled by the decompiler and making the appropriate corrections. Frequently, these statements are flagged and the corrections are straightforward. The more subtle errors are not detected until an incorrect result is found after the target program is executed (block 10). The appropriate IBM/370 JCL is generated by the decompiler.

After the initial editing is completed, the resulting PL/1 program is executed. If an incorrect result occurs, a number of interesting actions can ensue (block 11). In one case (test case 4 - appendix B) an error in the MIX program was found to be due to an omission in the published algorithm. This was found by reading the higher level version of the algorithm in PL/1. When it was determined that it did not describe the intended algorithm, the original MIX source was reviewed and found to contain the corresponding error. It would seem that decompilation can serve as a debugging aid in some cases. If the source program is correct; then, normally, additional manual editing of the PL/1 program is required (block 8). These subtle errors were usually a result of overlooking an erroneous data attribute which caused improper data

conversions. However, in some instances it was either necessary or desirable to modify the decompiler (block 13). It was pointed out in chapter 6 that some of the translation rules, especially in data structure mapping, involved intricate PL/1 constructs. Part of this may be due to the fact that in some instances the level of the translation was lower than that which could be easily accommodated by PL/1. Some of these translations contained errors due to the author's lack of complete understanding of the myriad of data conversion and mapping rules. In other cases it was seen that a more elegant translation could be realized, and the decompiler was appropriately upgraded.

IMPLEMENTATION

The experimental decompiler is subject to a number of limitations. In some instances it was felt that the implementation of a translation rule did not significantly add to demonstrating the fundamental concepts being explored. The manual translation of such a rule is generally straightforward.

This type of limitation is viewed as a practical limitation. Examples of this would be the manual translation of the MIX I/O and floating point instructions, and the handling of erroneous data attributes. In other

words, given the justification, it is clear that these rules could be automated without difficulty.

Other limitations are conceptual in nature and require further theoretical study. Some of these problems are discussed in chapter 8.

PERFORMANCE CONSIDERATIONS

The implementation consists of two main programs, the preprocessor (partial assembler) and the analysis and code generation phases. The preprocessor is viewed as a convenience since theoretically an object deck could be input to the latter program. Henceforth, the term "decompiler" will refer only to the analysis and code generation program.

When considering performance of this implementation, it should be realized that the decompiler was designed to serve as a flexible research vehicle in which algorithms could easily be incorporated and tested. A considerable amount of redundant processing could be eliminated by combining algorithms, thus eliminating some multiple scans of the program.

The decompiler requires 55000 octal words of memory. For the test cases run the mean decompile time is .056

seconds per instruction (1080 instructions per minute) on the CDC6500 running under the Purdue Mace operating system.

Because of the limited number of test cases and their brevity, it is difficult to make a general statement concerning execution time. One would expect that the decompile time would be in proportion to the number of instructions (I) in the program. Another factor which significantly affects the execution time is the complexity of the control flow graph of the program. One measure of this complexity is the number of blocks (B) in the program. After examining the available data, it would appear that there is some interaction between I and B. This could be attributed to the fact that the time spent scanning the control flow graph increases as the block size (I/B) increases (due to busy analysis). One model involving I and B which reflects this interaction and gives reasonable results for the available data is of the form:

$$(1) \quad T = kI\sqrt{B}$$

where k is a machine dependent constant (k=.018 for CDC6500).

It is of interest to note that an unpublished result by Caudle in connection with the Lockheed decompiling effort suggests that

(2) $T = cB^{1.5}$ (c is a proportionality constant).

If it is assumed that the number of instructions per block is constant, then equation (1) reduces to Caudle's result. That is, expressing I as nB and substituting into (1) yields (2), where c equals kn.

A summary of test case performance data and a comparison between actual and predicted decompile times are given in Appendix C.

TEST CASE EDITING

A summary of the manual editing (block 7 of figure 7.A) required for the six test cases successfully run of the IBM/370 is given below.

TABLE 7.A - Summary of Test Case Editing

CASE NO.	NO. INST.	DT	DI	IO	IC	IT	TOTAL-IO
1	30	1	1	2	2	0	4
2	45	1	1	1	2	2	6
3	26	0	0	1	0	0	0
4	32	0	0	4	0	0	0
5	25	1	0	1	1	0	2
6	45	4	2	18	10	0	16*

(DT, DI, IO, IC and IT are the EDIT CATEGORIES.)

LEGEND:

DT - data type attribute modifications.
DI - data initialization modifications.
IO - lines of code for I/O.
IC - corrections for translations which were attempted but found to contain errors.
IT - instruction translation not attempted; entirely translated by hand.

* Note - this figure includes the same edits applied to several lines. The number of unique edits is about 8.

Sometimes more than one edit was required for a line of code, where a line is delimited by two successive semicolons. Appendix B gives the original and edited version of each test case.

The DI type modifications were necessary either because

the MIX program assumed memory to be initialized to zero or because the data attribute was in error and the corresponding initialization had to be appropriately altered. The DT modifications designate that the default attribute generation was insufficient for reasons previously discussed. The IC edits were often a result of the data type errors. For example, in test case 6 several constants had to be recoded to give the proper initialization in assignment statements. IC edits also occurred when it was necessary to examine the local context of an instruction, such as in test case 2. This category presented the most difficulties. The required IT edits are detected immediately because they are flagged by the decompiler. Translating these manually into PL/1 consisted of examining their context in the target program and coding the appropriate PL/1 equivalent. These manual translations were facilitated by the fact that the surrounding instructions were already in a higher level language, usually making the meaning of the untranslated instruction clear.

The IO edits are not included in the total column because this would distort the editing required to make the result correct. In most cases I/O statements were added in order to provide test case verification and did not have a MIX counterpart.

As seen in the table, the required editing was not excessive in spite of the limitations of the decompiler and the diversity of the test cases.

Of particular interest is the fact that in no case was it necessary to alter the dimensions of an array or modify the control flow logic. In some cases the dimensions of arrays were larger than necessary but did not impede correct execution of the target program.

CHAPTER 8

EXTENSIONS AND CONCLUSIONS

Two of the more critical areas which have not been treated in this study are subroutines and self-modifying code. The following discussion outlines some proposed solutions to these problems. Other problems such as indirect addressing, interrupt handling (systems and real time applications), the decompilation of programs with overlay structures, are but a few of the areas which require further study.

SUBROUTINES

Typical machine languages have instructions especially designed to facilitate subroutine linkages. In MIX when a JMP instruction is executed the address of the instruction following the JMP is saved in the "J-register". A MIX subroutine is usually detected when a STJ (store J-register) instruction is detected as the first instruction in the block referenced by the JMP. The field referenced by the STJ is generally the address part of a JMP instruction. This JMP acts as a subroutine RETURN statement and can be

translated as such during target code generation. The initial JMP can be translated into a "CALL", and the STJ replaced by "<subr-name>: PROCEDURE;" in the target translation. To avoid the problem of distinguishing local and global variables of a subroutine, all data can be considered global (i.e. declared in the main PL/I procedure). This also has the effect of making all subroutines "parameterless".

Another problem in handling subroutines is that of incorporating the control graph of the subroutine into the main control flow graph of the program. Assuming the subroutine has only one entry and exit (return) block, this can be handled nicely by treating the control flow graph of the subroutine as a two terminal subgraph (Nylin, 1972) of the main control flow graph, where the entry block is that which receives control from the calling program and the exit block is that which returns control to the caller. Thus, if a subroutine is called n times its subgraph would have n immediate predecessors and n immediate successors. The JMP instructions which serve to call a subroutine, S, would constitute the <u>last</u> instructions of IP[SG(S)] (where SG(S) is the subgraph of S) and the instructions which received control upon exit from the subroutine would comprise the initial instruction of the instruction blocks represented by IS[SG(S)].

This approach would allow all the results previously described to be employed. A transfer from the subroutine to a definite location outside the subroutine could be considered as a jump to a global (PL/1) block. Determining the blocks which comprise the subroutine may be done by tracing the control flow from the entry block. SG(S) is initialized to the entry block (e). When considering a block, b, for inclusion into SG(S), if some k in SG(S) is a predecessor of b and the return block (x) is a successor of b then b can be added to SG(S). When no more blocks can be included, x is add to SG(S).

SELF MODIFYING CODE

Since self-modifying code is not permitted in higher level languages, the decompiler must transform these constructs into equivalent ones which can be mapped into the target language.

Two philosophies can be used in attacking this problem. One is to classify ways in which self-modifying code is commonly used in the source language and then develop a specific technique to handle each class. A common coding technique is the modification of the address part of instructions to effect indexing of data in machine languages which have an insufficient number of index registers. A

possible solution to this problem is outlined in a subsequent section.

A General Approach

Another approach is to try to handle code modification in a general way. The problem here is that the resulting translation may be exorbitantly inefficient. As pointed out by Halstead (1970), practically speaking, hand translation of these cases would generally be preferable from an economic viewpoint.

The theme of the following discussion is that if a minor constraint is imposed, a general solution to the code modification problem appears possible within the context of static decompiling (no simulation required). Thus, the the mysterious nemesis, "the self-modifying code problem", which has often been thought a deterrent to decompiling can be dealt with theoretically.

If it is assumed that instructions which are modified do not subsequently modify other instructions, a general approach appears possible. This restriction defines first order self-modifying code. One would expect most debugged programs to be subject to this constraint. Higher orders of code modification are time dependent and must be handled by simulation.

A randomly modified instruction could be translated as a subroutine call which serves a function analogous to a machine language "execute" statement. The parameters passed to this subroutine would define the "state" of the modified instruction. Accordingly, the subroutine would decode the state parameters and execute the appropriate "version" of the original modified instruction.

The next question is how to define and maintain these state variables for a modified instruction. A unique state variable could be assigned for each altered field in the machine language instruction. The translation of a machine language instruction which modifies a field of another instruction would be a statement which updates the appropriate state variable to its new state. Determining all the unique states may involve analysis using the control flow graph and some of the methods previously discussed.

The relevant assertion here is that if the first order restriction holds, then all the unique states can be determined at decompile time and target code can be produced to change the states appropriately during execution. Thus, whenever the "execute" subroutine is called, a determination of the appropriate action can be made.

Address Modification

The technique suggested here is to modify those portions of the intermediate text (INTEXT) representation which relate to address code modification in the original program, resulting in a INTEXT version which does not reflect code modification. This could be accomplished by generating simple temporary operands (in the SOT) which serve as "pseudo" index registers and then produce an INTEXT translation which references the pseudo registers in lieu of instructions.

The procedure for generating the intermediate text for MIX programs which contain address modification is given as follows.

A. Generate a temporary $Tj (entry in SOT).

B. Generate an instruction in the program initialization (entry) block which sets $Tj to the assembled address of the instruction being modified.

C. Alter the INTEXT instruction being modified.

.1 Make the memory reference operand a reference to the indexed operand table with the entry XOT[0,$TJ].

.2 If the index field is nonzero (say Rk), insert the instruction:

ADD $Tj,$Tj,Rk

immediately preceding the modified instruction.

D. Replace all references to the address part of the modified instruction by $Tj.

The following example serves to illustrate the concept. Consider the following MIX program which initializes an array to zero without using index registers to locate elements of the array.

```
1    ARRAY       EQU   1000
2    MOD1        ENT1  ARRAY
3                ST1   MOD2(0:2)
4    MOD2        STZ   0
5                CMP1  LASTELMT
6                JGT   CONTINUE
7                LDA   MOD1(0:2)
8                ADD   =1=
9                STA   MOD1(0:2)
10               JMP   MOD1
11   LASTELMT    CON   ARRAY+499
12   CONTINUE    ...
```

Following the given procedure would produce the following IMTEXT representation of the above code. The temporaries used for the modified instructions 2 and 4 are $T1 and $T2 respectively. It is assumed that these temporaries are simple operand table entries (see chapter 3 for IMTEXT notation).

```
1                   ASSIGN   $T1,ICT[ARRAY]
2                   ASSIGN   $T2,ICT[0]
3     MOD1          ASSIGN   R1,XOT[0,$T1]
4                   ASSIGN   $T2,R1
5     MOD2          ASSIGN   XOT[0,$T2],ICT[0]
6                   CMP      CI,R1,LASTELMT
7                   JUMP     >,CI,CONTINUE
8                   ASSIGN   RA,$T1
9                   ADD      RA,RA,ICMT[1]
10                  ASSIGN   $T1,RA
11                  JUMP     ,MOD1
12    CONTINUE      ...
```

All redundant assigments introduced would be removed during the text compression phase.

CONCLUSIONS

Many of the concepts developed in this study appear to be generally applicable to the decompilation of typical machine languages. In a current research project, Friedman (1973) has used this decompiler as a basis for decompiling IBM 1130 operating system code to a systems programming language for mini-computers (Friedman,Schneider,1973). It is evident that the technology of decompiling need not be categorized as ad hoc and machine dependant, and that it offers an interesting and challenging area for intellectual study.

The basis for many of the algorithms seems to be that of providing a high level, abstract representation of the program during the analysis phase, where this representation

consists of the intermediate text along with the control flow graph of the program. This "up over and down" mapping process appears to be a key concept for systematically attacking the problem.

Another conclusion from this study is that the complexity of the decompiler is directly related to the target language. As previously mentioned, PL/1 is not well suited as a target language if ease of decompiler implementation is sought.

It appears that if decompilation is subject to a systematic approach, that the manual completion of the task is not difficult. A conscious effort was made not to become intimately familiar with the test programs. The manual intervention required to complete the conversion of the test cases usually did not require becoming intimately acquainted with the "meaning" of the program. This fact underscores the notion that decompilation is still very beneficial even though the translation may be incomplete.

While the potential of decompiling has been commercially exploited by a few, the area has been generally underestimated. It is clear that it offers a viable tool toward reaching the described objectives. Since this study represents only an initial step in the development of systematic decompiling technology, and in view of the

tremendous economic implications, it is apparent that this area will be one of continued interest.

LIST OF REFERENCES

LIST OF REFERENCES

ALLEN, J. J., et. al., 1963. "Share Internal Fortran Translator," Datamation, March 1963, pp. 43-46.

Allen, F. E., 1970. "Control Flow Analysis," Sigplan Notices, Vol. 5, No. 7 (July, 1970), pp. 1-19.

Barbe, P., 1970. "Techniques for Automatic Program Translation," Software Engineering, Vol. 1, New York: Academic Press Inc., 1970, pp. 151-165.

Dellert, G. T., 1965. "A Use of Macros in Translation of Symbolic Assembly Language of One Computer to Another," CACM, Vol. 8, No. 12 (December, 1965), pp. 742-748.

Denning, P. J., 1968. "The Working Set Model for Program Behavior," CACM, Vol. 11, No. 5 (may, 1968), pp. 323-333.

Friedman, F. L., 1973. Private Communications, Purdue University.

Friedman, F. L., Schneider, V. S., 1973. "A Programming Language for Mini-computer Systems," (Submitted for publication).

Gaines, R. S., 1965. "On the Translation of Machine Language Programs," CACM, Vol. 8, No. 12 (December, 1965), pp. 736-741.

Graham, M. L., Ingerman, P. Z., 1965. "An Assembly Language for Reprogramming," CACM, Vol. 8, No. 12 (December, 1965), pp. 769-773.

Gunn, J. H., 1962. "Problems in Program Interchangeability," Symbolic Languages in Data Processing. New York: Gordon and Beach Science Publishers, 1962, pp. 777-790.

Halstead, M. H., 1962. Machine-Independent Computer Programming, Chapter 11, Washington D. C: Spartan

Books, 1962.

Halstead, M. H., 1967. "Machine Independence and Third Generation Computers," Proceedings FJCC, pp. 587-592.

Halstead, M. H., 1970. "Using the Computer for Program Conversion," Datamation, May 1970, pp. 125-129.

IBM, 1967. 1400 Autocoder to COBOL Conversion Aid Program (360A-SE-19X), Version 2 Application Description Manual (GH20-0352-2), White Plains, New York: IBM, 1967.

Knuth, D. E., 1969. The Art of Computer Programming, Vol. 1 (Fundamental Algorithms), Reading Massechusetts: Addison-Wesley, 1969.

Lang, C. A., 1969. "SAL - Systems Assembly Languages," Proceedings FJCC, pp. 587-592.

Leavenworth, B. M. (Editor), 1972. "The GOTO Controversy," Sigplan Notices, Vol. 7, No. 11, Novermber, 1972, pp. 54-91.

Lichstein, Henry A., 1969. "When Should You Emulate?," Datamation, November 1969, pp. 205-210.

Nylin, W. C., 1972. "Structural Reorganization of Multipass Computer Programs," Ph.d. Thesis, Dept. of Computer Science, Purdue University, June 1972.

Olsen, T. M., 1965. "Philco/IBM at Problem-Oriented, Symbolic and Binary Levels," CACM, Vol. 8, No. 12 (December, 1965), pp. 762-768.

Opler Ascher, et. al., 1962. "Automatic Translation of Programs From One Computer to Another," Proceedings IFIP Congress, 1962, pp. 550-553.

Opler, Ascher, 1963. "Automatic Program Translation," Datamation, May 1963, pp. 45-48.

Sassaman, William A., 1966. "A Computer Program to Translate Machine Language into Fortran," Proceedings SJCC, 1966, pp. 235-239.

Weiderhold, G., Ehrman, J., 1971. "Inferred SYNTAX and SEMANTICS of PL/S," Sigplan Notices, Vol. 6, No. 9, October, 1971, pp. 111-121.

GENERAL REFERENCES

Allen, F. E., 1969. "Program Optimization," Annual Review in Automatic Programming, Vol. 5, New York: Pergamon, 1969, pp. 239-307.

Bayer, R., et. al., 1967. "The ALCOR Illinois 7090/7094 Post Mortem Dump," CACM, Vol. 10, No. 12 (December, 1967), pp. 804-808.

Buxton and Randell (Editors), 1970. A Report on a Conference Sponsored by the NATO Science Committee, Brussels Belgium: NATO Science Committee, April, 1970, pp. 28-40.

Gordon, W. L., 1965. "Liberator, the Concept and the Hardware," ACM Symp. Reprogramming Problem, Princeton, New Jersy, june 1965.

Graham, S., 1965. "The Semi-automatic Computer Conversion System (SACCS)," ACM Symp. Reprogramming Problem, Princeton, New Jersy, June 1965.

IBM, 1970. PL/1(F) Language Reference Manual (GC28-8201-3), White Plains, New York: IBM, 1970.

Lowry, E. S., Medlock, C. W., 1962. "Object Code Optimization," CACM, Vol. 12, No. 1 (January, 1962), pp. 13-22.

Mendicino, Sam F., et. al., 1968. "The LRLTRAN Compiler," CACM, Vol. 11, No. 11 (Novermber, 1968), pp. 747-755.

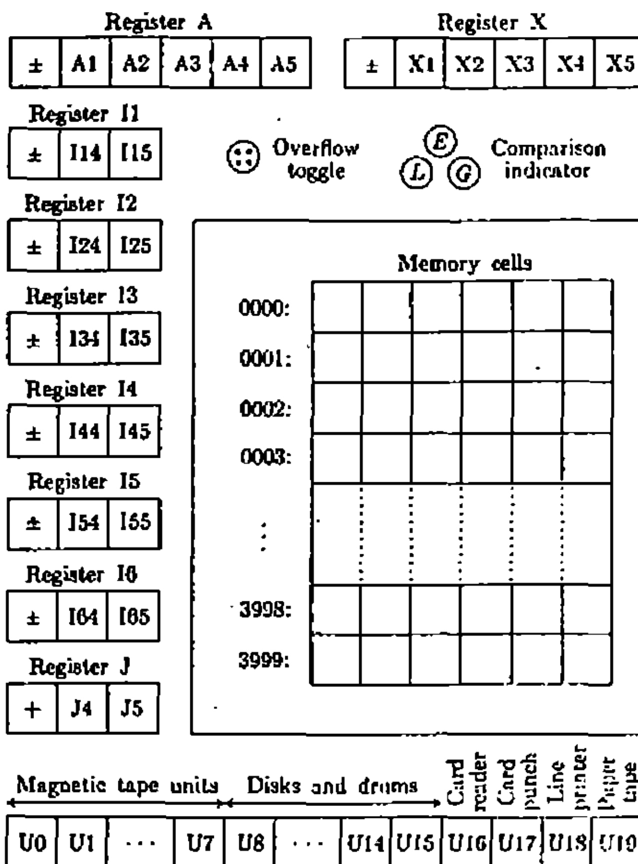
Nievergelt, J., 1965. "On Simplification of Computer Programs," CACM, Vol. 8, No. 6 (June, 1965), pp. 366-370.

APPENDICIES

APPENDIX A - SUMMARY OF THE "MIX" MACHINE

This appendix gives a brief description of the MIX architecture and instruction set.

The following material is reprinted by special permission from Knuth, THE ART OF COMPUTER PROGRAMMING, Volume 1, Fundamental Algorithms, 1968, Addison-Wesley, Reading, Mass.



The MIX computer.

A computer word is five bytes plus a sign. The sign position has only two possible values, + and −.

Registers. There are nine registers in MIX:

The A-register (Accumulator) is five bytes plus sign.

The X-register (Extension) is also five bytes plus sign.

The I-registers (Index registers) I1, I2, I3, I4, I5, and I6 each hold two bytes plus sign.

The J-register (Jump address) holds two bytes, and its sign is always +.

We shall use a small letter "r" prefixed to the name, to identify a MIX register. Thus, "rA" means "register A."

The A-register has many uses, especially for arithmetic and operating on data. The X-register is an extension on the "right-hand side" of rA, and it is used in connection with rA to hold ten bytes of a product or dividend, or it can be used to hold information shifted to the right out of rA. The index registers rI1, rI2, rI3, rI4, rI5, and rI6 are used primarily for counting and for referencing variable memory addresses. The J-register always holds the address of the instruction following the preceding "JUMP" instruction, and it is primarily used in connection with subroutines.

Besides these registers, MIX contains

an *overflow toggle* (a single bit which is either "on" or "off"),
a *comparison indicator* (which has three values: less, equal, or greater),
memory (4000 words of storage, each word with five bytes plus sign),
and *input-output devices* (card, tape, etc.).

Partial fields of words. The five bytes and sign of a computer word are numbered as follows:

0	1	2	3	4	5
±	Byte	Byte	Byte	Byte	Byte

(2)

Most of the instructions allow the programmer to use only part of a word if he chooses. In this case a "field specification" is given. The allowable fields are those which are adjacent in a computer word, and they are represented by (L:R), where L is the number of the left-hand part and R is the number of the right-hand part of the field. Examples of field specifications are:

(0:0), the sign only.
(0:2), the sign and the first two bytes.
(0:5), the whole word. This is the most common field specification.
(1:5), the whole word except for the sign.
(4:4), the fourth byte only.
(4:5), the two least significant bytes.

The use of these field specifications varies slightly from instruction to instruction, and it will be explained in detail for each instruction where it applies.

Although it is generally not important to the programmer, the field $(L:R)$ is denoted in the machine by the single number $8L + R$, and this number will fit in one byte.

Instruction format. Computer words used for instructions have the following form:

0	1	2	3	4	5
±	A	A	I	F	C

(3)

The rightmost byte, C, is the operation code telling what operation is to be performed. For example, $C = 8$ is the operation LDA, "load the A register."

The F-byte holds a modification of the operation code. F is usually a field specification $(L:R) = 8L + R$; for example, if $C = 8$ and $F = 11$, the operation is "load the A-register with the (1:3) field." Sometimes F is used for other purposes; on input-output instructions, for example, F is the number of the affected input or output unit.

The left-hand portion of the instruction, $\pm AA$, is the "address." (Note that the sign is part of the address.) The I-field, which comes next to the address, is the "index specification," which may be used to modify the address of an instruction. If $I = 0$, the address $\pm AA$ is used without change; otherwise I should contain a number i between 1 and 6, and the contents of index register Ii are added algebraically to $\pm AA$; the result is used as the address of the instruction. This indexing process takes place on *every* instruction. We will use the letter M to indicate the address after any specified indexing has occurred. (If the addition of the index register to the address $\pm AA$ yields a result which does not fit in two bytes, the value of M is undefined.)

In most instructions, M will refer to a memory cell. The terms "memory cell" and "memory location" are used almost interchangeably in this book. We assume that there are 4000 memory cells, numbered from 0 to 3999; hence every memory location can be addressed with two bytes. For every instruction in which M is to refer to a memory cell we must have $0 \leq M \leq 3999$, and in this case we will write CONTENTS(M) to denote the value stored in memory location M.

On certain instructions, the "address" M has another significance, and it may even be negative. Thus one instruction adds M to an index register, and this takes account of the sign of M.

00 1	01 2	02 2	03 10
No operation NOP(0)	rA ← rA + V ADD(0:5) FADD(6)	rA ← rA − V SUB(0:5) FSUB(6)	rAX ← rA × V MUL(0:5) FMUL(6)
08 2	09 2	10 2	11 2
rA ← V LDA(0:5)	rI1 ← V LD1(0:5)	rI2 ← V LD2(0:5)	rI3 ← V LD3(0:5)
16 2	17 2	18 2	19 2
rA ← −V LDAN(0:5)	rI1 ← −V LD1N(0:5)	rI2 ← −V LD2N(0:5)	rI3 ← −V LD3N(0:5)
24 2	25 2	26 2	27 2
F(M) ← rA STA(0:5)	F(M) ← rI1 ST1(0:5)	F(M) ← rI2 ST2(0:5)	F(M) ← rI3 ST3(0:5)
32 2	33 2	34 1	35 1 + T
F(M) ← rJ STJ(0:2)	F(M) ← 0 STZ(0:5)	Unit F busy? JBUS(0)	Control, unit F IOC(0)
40 1	41 1	42 1	43 1
rA:0, jump JA[+]	rI1:0, jump J1[+]	rI2:0, jump J2[+]	rI3:0, jump J3[+]
48 1	49 1	50 1	51 1
rA ← [rA]? ± M INCA(0)DECA(1) ENTA(2)ENNA(3)	rI1 ← [rI1]? ± M INC1(0)DEC1(1) ENT1(2)ENN1(3)	rI2 ← [rI2]? ± M INC2(0)DEC2(1) ENT2(2)ENN2(3)	rI3 ← [rI3]? ± M INC3(0)DEC3(1) ENT3(2)ENN3(3)
56 2	57 2	58 2	59 2
rA(F):V → CI CMPA(0:5) FCMP(6)	rI1(F):V → CI CMP1(0:5)	rI2(F):V → CI CMP2(0:5)	rI3(F):V → CI CMP3(0:5)

General form:

C   t
Description OP(F)

C = operation code, (5:5) field of instruction
F = op variant, (4:4) field of instruction
M = address of instruction after indexing
V = F(M) = contents of F field of location M
OP = symbolic name for operation
(F) = standard F setting
t = execution time; T = interlock time

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9	.	,	(	)	+	-	*	/	=	$	<	>	@	;	:	'

04	*12*	05	*1*	06	*2*	07	*1 + 2F*
rA ← rAX/V rX ← remainder DIV(0:5) FDIV(6)		Special NUM(0) CHAR(1) HLT(2)		Shift M bytes SLA(0) SRA(1) SLAX(2) SRAX(3) SLC(4) SRC(5)		Move F words from M to rI1 MOVE(1)	
12	***2***	**13**	***2***	**14**	***2***	**15**	***2***
rI4 ← V LD4(0:5)		rI5 ← V LD5(0:5)		rI6 ← V LD6(0:5)		rX ← V LDX(0:5)	
20	***2***	**21**	***2***	**22**	***2***	**23**	***2***
rI4 ← −V LD4N(0:5)		rI5 ← −V LD5N(0:5)		rI6 ← −V LD6N(0:5)		rX ← −V LDXN(0:5)	
28	***2***	**29**	***2***	**30**	***2***	**31**	***2***
F(M) ← rI4 ST4(0:5)		F(M) ← rI5 ST5(0:5)		F(M) ← rI6 ST6(0:5)		F(M) ← rX STX(0:5)	
36	***1 + T***	**37**	***1 + T***	**38**	***1***	**39**	***1***
Input, unit F IN(0)		Output, unit F OUT(0)		Unit F ready? JRED(0)		Jumps JMP(0) JSJ(1) JOV(2) JNOV(3) also [*] below	
44	***1***	**45**	***1***	**46**	***1***	**47**	***1***
rI4:0, jump J4[+]		rI5:0, jump J5[+]		rI6:0, jump J6[+]		rX:0, jump JX[+]	
52	***1***	**53**	***1***	**54**	***1***	**55**	***1***
rI4 ← [rI4]? ± M INC4(0) DEC4(1) ENT4(2) ENN4(3)		rI5 ← [rI5]? ± M INC5(0) DEC5(1) ENT5(2) ENN5(3)		rI6 ← [rI6]? ± M INC6(0) DEC6(1) ENT6(2) ENN6(3)		rX ← [rX]? ± M INCX(0) DECX(1) ENTX(2) ENNX(3)	
60	***2***	**61**	***2***	**62**	***2***	**63**	***2***
rI4(F) : V → CI CMP4(0:5)		rI5(F) : V → CI CMP5(0:5)		rI6(F) : V → CI CMP6(0:5)		rX(F) : V → CI CMPX(0:5)	

rA = register A
rX = register X
rAX = registers AX as one
rI*i* = index reg. *i*, $1 \le i \le 6$
rJ = register J
CI = comparison indicator

[*]:		[+]:
JL(4)	<	N(0)
JE(5)	=	Z(1)
JG(6)	>	P(2)
JGE(7)	≥	NN(3)
JNE(8)	≠	NZ(4)
JLE(9)	≤	NP(5)

APPENDIX B - TEST CASE RESULTS

This appendix lists eight programs. The first two are sample programs, written by the author, and were not executed on the IBM/370. The last six programs comprise test cases 1 - 6 as referenced in the text.

Included with the sample programs are some intermediate output generated by the decompiler, showing the results of some of the analysis phases discussed in the text. Sample program A is depicted in figure 5.E and is discussed in depth in chapter 5.

The data tables are labeled: DATAB1, DATAB2, ... , DATAB5. In reference to the text notation, the following are equivalent: (DATAB1,SOT), (DATAB2,XOT), (DATAB3,ICMT), (DATAB4,ICT), and (DATAB5,JAT). The tables: SCR, IDAT, and ADT are labeled as such in the intermediate output. The IMTEXT heading labels: BLKNO, OPCODE, OPRND1, OPRND2, and OPRND3, correspond to: IT.BN, IT.OPC, IT.N1, IT.N2, and IT.N3, respectively (see chapter 3).

Except for the first and third operands of the JUMP instructions, the IMTEXT operand pointers are of the form:

<table index><table number>

For example, an operand value of 301 would designate the thirtieth entry in DATAB1. A table number of 2 designates DATAB2 or IDAT, depending on the phase of analysis. Those entries in DATAB1 with negative locations correspond to various registers.

All statements in the edited PL/1 programs which required manual correction (except for I/O) are flagged by an "*". For each test case, the MIXAL listing, the initial generated output, and a listing of the edited version are presented. The result produced by executing the edited program is attached. (Note: In the PL/1 listings the "'" (apostrophe) is represented by "≠".)

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

* SAMPLE PROGRAM A - THE GENERATED PL/I PROGRAM WAS NOT EXECUTED ON
*                    THE IBM/370
*
*  EXAMPLE OF TRIPLE SUBSCRIPTED ARRAY , ARRAY(5:1,1:3,4:1).
*  THE ENTIRE ARRAY IS INITIALIZED TO 1.
*
ARRAY      EQU  100
XR3        EQU  3
XR4        EQU  4
XR5        EQU  5
           ORIG 1000
START      ENT3 ARRAY
           ST3  LIMXR3              STORE BOUND ON SUBSCRIPT XR3
           INC3 48                  INITIALIZE XR3
LOOP3      ENT4 12,XR3              COMPUTE
           ST4  LIMXR4              BOUND OF XR4
           ENT4 0,XR3               INITIALIZE XR4
LOOP2      ST4  LIMXR5              STORE LIMIT OF XR5
           ENT5 4,XR4               INITIALIZE XR5
LOOP1      DEC5 1                   DECREMENT XR5
           LD6  ONE                 STORE 1 AT THE LOCATION
           ST6  0,XR5               CONTAINED IN XR5
           CMP5 LIMXR5              TEST FOR END OF INNER LOOP
           JG   LOOP1               *
           INC4 4                   BUMP MIDDLE LOOP SUBSCRIPT
           CMP4 LIMXR4              TEST FOR END OF MIDDLE SUBSCRIPT
           JL   LOOP2
           CMP3 LIMXR3              TEST FOR END OF OUTER MOST SUBSC
           JE   DONE
           DEC3 12                  DECR OUTER SUBSCRIPT
           JMP  LOOP3
DONE       HLT
ONE        CON  1
LIMXR3     CON  0
LIMXR4     CON  0
LIMXR5     CON  0
           END  START
```

OUTPUT CODE

NO. SYMBOLS=	13	
ARRAY	100	1
XR3	3	1
XR4	4	1
XR5	5	1
START	1000	4
LIMXR3	1022	4
LOOP3	1003	4
LIMXR4	1023	4
LOOP2	1006	4
LIMXR5	1024	4
LOOP1	1008	4
ONE	1021	4
DONE	1020	4

LOCTN	OPC	ADDRESS	INDEX	FIELD
1000	ORIG	1000	0	0
1000	ENT3	100	0	2
1001	ST3	1022	0	5
1002	INC3	48	0	0
1003	ENT4	12	3	2
1004	ST4	1023	0	5
1005	ENT4	0	3	2
1006	ST4	1024	0	5
1007	ENT5	4	4	2
1008	DEC5	1	0	1
1009	LD6	1021	0	5
1010	ST6	0	5	5
1011	CMP5	1024	0	5
1012	JG	1008	0	6
1013	INC4	4	0	0
1014	CMP4	1023	0	5
1015	JL	1006	0	4
1016	CMP3	1022	0	5
1017	JE	1020	0	5
1018	DEC3	12	0	1
1019	JMP	1003	0	0
1020	HLT	0	0	2
1021	CON	1	0	5
1022	CON	0	0	5
1023	CON	0	0	5
1024	CON	0	0	5
1025	END	1000	0	0

*** ORIGINAL INPUT (P) BY BLOCK ***

NO.	LOCTN	OPC	ADDRESS	INDEX	FIELD	BLOCK
1	1000	ENT3	100	0	2	1
2	1001	ST3	1022	0	5	1
3	1002	INC3	48	0	0	1
4	1003	ENT4	12	3	2	2
5	1004	ST4	1023	0	5	2
6	1005	ENT4	0	3	2	2
7	1006	ST4	1024	0	5	3
8	1007	ENT5	4	4	2	3
9	1008	DEC5	1	0	1	4
10	1009	LD6	1021	0	5	4
11	1010	ST6	0	5	5	4
12	1011	CMP5	1024	0	5	4
13	1012	JG	1008	0	6	4
14	1013	INC4	4	0	0	5
15	1014	CMP4	1023	0	5	5
16	1015	JL	1006	0	4	5
17	1016	CMP3	1022	0	5	6
18	1017	JE	1020	0	5	6
19	1018	DEC3	12	0	1	7
20	1019	JMP	1003	0	0	7
21	1020	HLT	0	0	2	8

**** ANALYSIS FOR STRONGLY CONNECTED REGIONS ****

```
LEVEL=   1   INTERVAL =   1
LEVEL=   1   INTERVAL =   2
LEVEL=   1   INTERVAL =   3
LEVEL=   1   INTERVAL =   4   5   6   8   7

SCR, LEVEL=   1   NODES=   4
LEVEL=   2   INTERVAL =   1
LEVEL=   2   INTERVAL =   2
LEVEL=   2   INTERVAL =   3   4   5   6   8   7

SCR, LEVEL=   2   NODES=   3   4   5
LEVEL=   3   INTERVAL =   1
LEVEL=   3   INTERVAL =   2   3   4   5   6   8   7

SCR, LEVEL=   3   NODES=   2   3   4   5   6   7
LEVEL=   4   INTERVAL =   1   2   3   4   5   6   8   7
```

```
INSTRUCTION BLOCK TABLE

********** BLOCK NO.    1 **********

BLOCK LEVEL=  0     SCR TABLE ENTRY=  0
START LOC=  1000  END LOC=  1002
 IMM SUCC INDEX=  29  IMM PRED INDEX=   0 CODE PTRS = (    1,    4)
IMMEDIATE SUCCESSORS =   2

********** BLOCK NO.    2 **********

BLOCK LEVEL=  1     SCR TABLE ENTRY=  3
START LOC=  1003  END LOC=  1005
 IMM SUCC INDEX=  15  IMM PRED INDEX=  31 CODE PTRS = (    5,    8)
IMMEDIATE SUCCESSORS =   3
IMMEDIATE PREDECESSORS=   1   7

********** BLOCK NO.    3 **********

BLOCK LEVEL=  2     SCR TABLE ENTRY=  2
START LOC=  1006  END LOC=  1007
 IMM SUCC INDEX=   7  IMM PRED INDEX=  33 CODE PTRS = (    9,   11)
IMMEDIATE SUCCESSORS =   4
IMMEDIATE PREDECESSORS=   2   5

********** BLOCK NO.    4 **********

BLOCK LEVEL=  3     SCR TABLE ENTRY=  1
START LOC=  1008  END LOC=  1012
 IMM SUCC INDEX=   3  IMM PRED INDEX=  35 CODE PTRS = (   12,   17)
IMMEDIATE SUCCESSORS =   4   5
IMMEDIATE PREDECESSORS=   3   4

********** BLOCK NO.    5 **********

BLOCK LEVEL=  2     SCR TABLE ENTRY=  2
START LOC=  1013  END LOC=  1015
 IMM SUCC INDEX=  11  IMM PRED INDEX=  39 CODE PTRS = (   18,   21)
IMMEDIATE SUCCESSORS =   3   6
IMMEDIATE PREDECESSORS=   4

********** BLOCK NO.    6 **********

BLOCK LEVEL=  1     SCR TABLE ENTRY=  3
START LOC=  1016  END LOC=  1017
 IMM SUCC INDEX=  19  IMM PRED INDEX=  43 CODE PTRS = (   22,   24)
IMMEDIATE SUCCESSORS =   8   7
IMMEDIATE PREDECESSORS=   5

********** BLOCK NO.    7 **********

BLOCK LEVEL=  1     SCR TABLE ENTRY=  3
START LOC=  1018  END LOC=  1019
 IMM SUCC INDEX=  27  IMM PRED INDEX=  47 CODE PTRS = (   25,   26)
IMMEDIATE SUCCESSORS =   2
IMMEDIATE PREDECESSORS=   6

********** BLOCK NO.    8 **********

BLOCK LEVEL=  0     SCR TABLE ENTRY=  0
START LOC=  1020  END LOC=  1020
 IMM SUCC INDEX=   0  IMM PRED INDEX=  45 CODE PTRS = (   27,   27)
IMMEDIATE PREDECESSORS=   6
```

*** INITIAL ABSTRACT REPRESENTATION (IAR) OF INTERMEDIATE TEXT ***

NO.	BLKNO	OPCODE	OPRND1	OPRND2	OPRND3
1	1	ASSIGN	31	23	0
2	1	ASSIGN	1271	31	0
3	1	ADD	31	31	33
4	1	JUMP	7	0	1
5	2	ADD	41	43	31
6	2	ASSIGN	1281	41	0
7	2	ASSIGN	41	31	0
8	2	JUMP	7	0	2
9	3	ASSIGN	141	41	0
10	3	ADD	51	53	41
11	3	JUMP	7	0	3
12	4	SUB	51	51	63
13	4	ASSIGN	61	1261	0
14	4	ASSIGN	12	61	0
15	4	CMP	91	51	141
16	4	JUMP	4	91	3
17	4	JUMP	7	0	4
18	5	ADD	41	41	53
19	5	CMP	91	41	1281
20	5	JUMP	1	91	2
21	5	JUMP	7	0	5
22	6	CMP	91	31	1271
23	6	JUMP	2	91	6
24	6	JUMP	7	0	7
25	7	SUB	31	31	43
26	7	JUMP	7	0	1
27	8	HLT	1	0	0

DATAB1

NO.	DLOCTN	DFIELD	FLNK CLNK STATUS-TA	INLVAL
1	-1	0	00000000000000000000	0
2	-2	0	00000000000000000000	0
3	-3	0	00000000000000000003	0
4	-4	0	00000000000000000003	0
5	-5	0	00000000000000000003	0
6	-6	0	00000000000000000003	0
7	-7	0	00000000000000000000	0
8	-8	0	00000000000000000000	0
9	-9	0	00000000000000000003	0
10	-10	0	00000000000000000000	0
11	-11	0	00000000000000000000	0
12	-12	0	00000000000000000000	0
13	-13	0	00000000000000000000	0
14	1024	5	00000000000000000002	4
126	1021	5	00000000000000000001	1
127	1022	5	00000000000000000002	2
128	1023	5	00000000000000000002	3

DATAB2

NO.	ADDRESS	IDX/FLD
1	0	505
2	0	0

DATAB3

NO.	DLOCTN	DFIELD	ATTRIB	INLVAL
1	1021	5	000010	1
2	1022	5	000010	0
3	1023	5	000010	0
4	1024	5	000010	0

DATAB4 (IMMEDIATE CONSTANTS)

NO.	CVALUE
1	0
2	100
3	48
4	12
5	4
6	1

DATAB5 (JUMP TABLE)

NO.	JMPLOC
1	1003
2	1006
3	1008
4	1013
5	1016
6	1020
7	1018

```
*** COMPRESSION PHASE ***

IMTXT INSTR (   1) DELETED       1  ASSIGN   31   23    0
IMTXT INSTR (  13) DELETED       4  ASSIGN   61 1261    0
IMTXT INSTR (   6) DELETED       2  ASSIGN 1281   41    0
```

```
*** BUILD NESTED REGION LISTS (NRL) ***

EXIT BLOCKS FOR SCR( 1)=     4

VARIABLE LISTS FOR SCR( 1)
  RECURS DEF VBL LIST=     51
  NON-RECURS DEF VBL LIST=     91

EXIT BLOCKS FOR SCR( 2)=     5

VARIABLE LISTS FOR SCR( 2)
  RECURS DEF VBL LIST=     41
  NON-RECURS DEF VBL LIST=   141     51     91

EXIT BLOCKS FOR SCR( 3)=     6

VARIABLE LISTS FOR SCR( 3)
  RECURS DEF VBL LIST=     31
  NON-RECURS DEF VBL LIST=  1281     41     91
```

```
*** SCR (LOOP) TABLE ***

SCR NO.  LEV   NRLP NDLP    NITER   INITV   TESTV    PSCR    DITV
     1     3  0022300111        4     151     148       0      -1
     2     2  0022300143        3     152     160       0       4
     3     1  0022300201        5     148     100       0     -12
```

```
INDEX DATA ACCESS TABLE (IDAT)

NO.,        ICTR DT2P DCLP     LBND    UPBD    SDLP CGPH          TA

   1   000000000130000100001    100     159    0066700605000000000
ARRAY DECLARATION TABLE (ADT)

NO.    ARRAY NME     LWR BND    UPR BND    STATUS

  1    ARRAY             100        159    00000000000000000000
```

*** FINAL ABSTRACT REPRESENTATION (FAR) OF INTERMEDIATE TEXT ***

NO.	BLKNO	OPCODE	OPRND1	OPRND2	OPRND3
1	1	ASSIGN	1271	23	0
2	1	ADD	31	23	33
3	1	JUMP	7	0	1
4	2	ADD	1281	43	31
5	2	ASSIGN	41	31	0
6	2	JUMP	7	0	2
7	3	ASSIGN	141	41	0
8	3	ADD	51	53	41
9	3	JUMP	7	0	3
10	4	SUB	51	51	63
11	4	ASSIGN	12	1261	0
12	4	CMP	91	51	141
13	4	JUMP	4	91	3
14	4	JUMP	7	0	4
15	5	ADD	41	41	53
16	5	CMP	91	41	1281
17	5	JUMP	1	91	2
18	5	JUMP	7	0	5
19	6	CMP	91	31	1271
20	6	JUMP	2	91	6
21	6	JUMP	7	0	7
22	7	SUB	31	31	43
23	7	JUMP	7	0	1
24	8	HLT	1	0	0

Note: After the array analysis, indexed operands which previously referenced the XOT now point to the IDAT. Thus, OPRND1 in line 11 references the first entry in the IDAT, not DATAB2.

```
*** GENERATED PL/1 PROGRAM ***
   1    MAIN: PROCEDURE OPTIONS(MAIN);
   2    DCL (IXR3,IXR4,IXR5,IXR6)  DEC FIXED(7);
   3    DCL 1 RARX,
               2 RA DEC FIXED(13),
               2 RX DEC FIXED(13),
               2 FILLAX CHAR(2);
   5    DCL RAX CHAR(16) BASED(PRARX);
   5    DCL LIMXR5 INIT(0) DEC FIXED(13);
   6    DCL ONE INIT(1) DEC FIXED(13);
   7    DCL LIMXR3 INIT(0) DEC FIXED(13);
   8    DCL LIMXR4 INIT(0) DEC FIXED(13);
   9    DCL ARRAY(100:159)  DEC FIXED(13);


  10                PRARX=ADDR(RARX);
  11                LIMXR3=100;
  12                IXR3=100+48;
  13    LOOP3:      LIMXR4=12+IXR3;
  14                IXR4=IXR3;
  15    LOOP2:      LIMXR5=IXR4;
  16                IXR5=4+IXR4;
  17    LOOP1:      IXR5=IXR5-1;
  18                ARRAY(IXR5)=ONE;
  19                IF IXR5 > LIMXR5 THEN GOTO LOOP1;
  20                ELSE DO;
  21                  IXR4=IXR4+4;
  22                  IF IXR4 < LIMXR4 THEN GOTO LOOP2;
  23                  ELSE DO;
  24                    IF IXR3 = LIMXR3 THEN DO;
  25                      RETURN;
  26                      END;
  27                    ELSE DO;
  28                      IXR3=IXR3-12;
  29                      GOTO LOOP3;
  30                      END;
  31                    END;
  32                  END;
  33                END MAIN;
```

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

* SAMPLE PROGRAM B - THE GENERATED PL/1 PROGRAM HAS NOT EXECUTED ON
*                    THE IBM/370
*
* EXCHANGE SORT WITH *NOWPE* WORDS PER ENTRY, WORD 1 IS THE
* COMPARE FIELD.
*
ARRAY      EQU  500
           ORIG 2000
SORT       LDA  NOWPE                 LOAD NO. ELTS PER ENTRY
           INCA ARRAY                 STORE INDEX TO 2ND
           STA  T1                    ENTRY IN T1
           LD2  T1                    SET I2 TO INDEX TO NEXT ENTRY
           ENT1 ARRAY                 SET I1 = INDEX TO CURRENT ENTRY
           ENT3 0
           LD6  NOWPE                 SET I6=ENTRY INCR
SORT1      LDA  0,1                   COMPARE
           CMPA 0,2                   WITH ENTRIES K,K+1
           JLE  SORT6                 IF ALREADY ORDERED, DONT EXCHANGE
* MUST EXCHANGE
           ENT3 0,1                   I3,I4 USED FOR EXCHANGE FLDS.
           ENT4 0,2
           ENT5 1                     I5=ENTRY COUNTER
SORT2      LDA  0,3                   SAVE K(W)
           STA  T1                    *
           LDA  0,4                   SET K(W)=K+1(W)
           STA  0,3                   *
           LDA  T1                    SET K+1(W)=K(W)
           STA  0,4                   *
           CMP5 NOWPE                 ALL WORDS IN ENTRY PROCESSED.
           JE   SORT6                 IF SO JUMP
           INC3 1                     BUMP ALL RELEVANT
           INC4 1                     INDICIES
           INC5 1                     *
           JMP  SORT2                 GO EXCHANGE NEXT WORD OF ENTRIES
* TEST FOR END, INCREMENT IF NOT
SORT6      INC1 0,6                   BUMP K INDEX
           INC2 0,6                   BUMP K+1 INDEX
           CMP2 LNTRY                 END OF ARRAY TEST
           JLE  SORT1                 IF NOT GO TEST NXT 2 ENTRIES
           J3P  SORT                  IF NOT ALL SORTED, RESCAN ARRAY
           HLT                        DONE
T1         CON  0                     TEMPORARY
NOWPE      CON  4                     NUMBER OF WORDS PER ENTRY
LNTRY      CON  696                   ADDRESS OF LAST ENTRY +1
           END  SORT
```

```
**OUTPUT CODE**

NO. SYMBOLS=      8
ARRAY            500          1
SORT            2000          4
NOWPE           2032          4
T1              2031          4
SORT1           2007          4
SORT6           2025          4
SORT2           2013          4
LNTRY           2033          4

LOCTN     OPC     ADDRESS      INDEX      FIELD

 2000     ORIG      2000          0          0
 2000     LDA       2032          0          5
 2001     INCA       500          0          0
 2002     STA       2031          0          5
 2003     LD2       2031          0          5
 2004     ENT1       500          0          2
 2005     ENT3         0          0          2
 2006     LD6       2032          0          5
 2007     LDA          0          1          5
 2008     CMPA         0          2          5
 2009     JLE       2025          0          9
 2010     ENT3         0          1          2
 2011     ENT4         0          2          2
 2012     ENT5         1          0          2
 2013     LDA          0          3          5
 2014     STA       2031          0          5
 2015     LDA          0          4          5
 2016     STA          0          3          5
 2017     LDA       2031          0          5
 2018     STA          0          4          5
 2019     CMP5      2032          0          5
 2020     JE        2025          0          5
 2021     INC3         1          0          0
 2022     INC4         1          0          0
 2023     INC5         1          0          0
 2024     JMP       2013          0          0
 2025     INC1         0          6          0
 2026     INC2         0          6          0
 2027     CMP2      2033          0          5
 2028     JLE       2007          0          9
 2029     J3P       2000          0          2
 2030     HLT          0          0          2
 2031     CON          0          0          5
 2032     CON          4          0          5
 2033     CON        696          0          5
 2034     END       2000          0          0
```

*** ORIGINAL INPUT (P) BY BLOCK ***

NO.	LOCTN	OPC	ADDRESS	INDEX	FIELD	BLOCK
1	2000	LDA	2032	0	5	1
2	2001	INCA	500	0	0	1
3	2002	STA	2031	0	5	1
4	2003	LD2	2031	0	5	1
5	2004	ENT1	500	0	2	1
6	2005	ENT3	0	0	2	1
7	2006	LD6	2032	0	5	1
8	2007	LDA	0	1	5	2
9	2008	CMPA	0	2	5	2
10	2009	JLE	2025	0	9	2
11	2010	ENT3	0	1	2	3
12	2011	ENT4	0	2	2	3
13	2012	ENT5	1	0	2	3
14	2013	LDA	0	3	5	4
15	2014	STA	2031	0	5	4
16	2015	LDA	0	4	5	4
17	2016	STA	0	3	5	4
18	2017	LDA	2031	0	5	4
19	2018	STA	0	4	5	4
20	2019	CMP5	2032	0	5	4
21	2020	JE	2025	0	5	4
22	2021	INC3	1	0	0	5
23	2022	INC4	1	0	0	5
24	2023	INC5	1	0	0	5
25	2024	JMP	2013	0	0	5
26	2025	INC1	0	6	0	6
27	2026	INC2	0	6	0	6
28	2027	CMP2	2033	0	5	6
29	2028	JLE	2007	0	9	6
30	2029	J3P	2000	0	2	7
31	2030	HLT	0	0	2	8

**** ANALYSIS FOR STRONGLY CONNECTED REGIONS ****

```
LEVEL=   1   INTERVAL =   1
LEVEL=   1   INTERVAL =   2    3
LEVEL=   1   INTERVAL =   6    7    8
LEVEL=   1   INTERVAL =   4    5

SCR, LEVEL=   1   NODES=   4    5
LEVEL=   2   INTERVAL =   1
LEVEL=   2   INTERVAL =   2    3    4    6    5    7    8

SCR, LEVEL=   2   NODES=   2    3    5    4    6
LEVEL=   3   INTERVAL =   1    2    3    4    6    5    7    8

SCR, LEVEL=   3   NODES=   1    2    3    5    4    6    7
```

```
INSTRUCTION BLOCK TABLE

********** BLOCK NO.   1 **********

BLOCK LEVEL=  1     SCR TABLE ENTRY=  3
START LOC=  2000  END LOC=  2006
 IMM SUCC INDEX=  15  IMM PRED INDEX=  53 CODE PTRS = (   1,   8)
IMMEDIATE SUCCESSORS =   2
IMMEDIATE PREDECESSORS=   7

********** BLOCK NO.   2 **********

BLOCK LEVEL=  2     SCR TABLE ENTRY=  2
START LOC=  2007  END LOC=  2009
 IMM SUCC INDEX=   3  IMM PRED INDEX=  35 CODE PTRS = (   9,  12)
IMMEDIATE SUCCESSORS =   6   3
IMMEDIATE PREDECESSORS=   1   6

********** BLOCK NO.   3 **********

BLOCK LEVEL=  2     SCR TABLE ENTRY=  2
START LOC=  2010  END LOC=  2012
 IMM SUCC INDEX=  33  IMM PRED INDEX=  39 CODE PTRS = (  13,  16)
IMMEDIATE SUCCESSORS =   4
IMMEDIATE PREDECESSORS=   2

********** BLOCK NO.   4 **********

BLOCK LEVEL=  3     SCR TABLE ENTRY=  1
START LOC=  2013  END LOC=  2020
 IMM SUCC INDEX=  19  IMM PRED INDEX=  41 CODE PTRS = (  17,  25)
IMMEDIATE SUCCESSORS =   6   5
IMMEDIATE PREDECESSORS=   3   5

********** BLOCK NO.   5 **********

BLOCK LEVEL=  3     SCR TABLE ENTRY=  1
START LOC=  2021  END LOC=  2024
 IMM SUCC INDEX=  31  IMM PRED INDEX=  45 CODE PTRS = (  26,  29)
IMMEDIATE SUCCESSORS =   4
IMMEDIATE PREDECESSORS=   4

********** BLOCK NO.   6 **********

BLOCK LEVEL=  2     SCR TABLE ENTRY=  2
START LOC=  2025  END LOC=  2028
 IMM SUCC INDEX=  11  IMM PRED INDEX=  37 CODE PTRS = (  30,  34)
IMMEDIATE SUCCESSORS =   2   7
IMMEDIATE PREDECESSORS=   2   4

********** BLOCK NO.   7 **********

BLOCK LEVEL=  1     SCR TABLE ENTRY=  3
START LOC=  2029  END LOC=  2029
 IMM SUCC INDEX=  25  IMM PRED INDEX=  51 CODE PTRS = (  35,  36)
IMMEDIATE SUCCESSORS =   1   8
IMMEDIATE PREDECESSORS=   6

********** BLOCK NO.   8 **********

BLOCK LEVEL=  0     SCR TABLE ENTRY=  0
START LOC=  2030  END LOC=  2030
 IMM SUCC INDEX=   0  IMM PRED INDEX=  55 CODE PTRS = (  37,  37)
IMMEDIATE PREDECESSORS=   7
```

*** INITIAL ABSTRACT REPRESENTATION (IAR) OF INTERMEDIATE TEXT ***

NO.	BLKNO	OPCODE	OPRND1	OPRND2	OPRND3
1	1	ASSIGN	71	1131	0
2	1	ADD	71	71	23
3	1	ASSIGN	1121	71	0
4	1	ASSIGN	21	1121	0
5	1	ASSIGN	11	23	0
6	1	ASSIGN	31	13	0
7	1	ASSIGN	61	1131	0
8	1	JUMP	7	0	1
9	2	ASSIGN	71	12	0
10	2	CMP	91	71	22
11	2	JUMP	3	91	2
12	2	JUMP	7	0	3
13	3	ASSIGN	31	11	0
14	3	ASSIGN	41	21	0
15	3	ASSIGN	51	33	0
16	3	JUMP	7	0	4
17	4	ASSIGN	71	32	0
18	4	ASSIGN	1121	71	0
19	4	ASSIGN	71	42	0
20	4	ASSIGN	32	71	0
21	4	ASSIGN	71	1121	0
22	4	ASSIGN	42	71	0
23	4	CMP	91	51	1131
24	4	JUMP	2	91	2
25	4	JUMP	7	0	5
26	5	ADD	31	31	33
27	5	ADD	41	41	33
28	5	ADD	51	51	33
29	5	JUMP	7	0	4
30	6	ADD	11	11	61
31	6	ADD	21	21	61
32	6	CMP	91	21	1141
33	6	JUMP	3	91	1
34	6	JUMP	7	0	6
35	7	JUMP	4	31	7
36	7	JUMP	7	0	8
37	8	HLT	1	0	0

DATAB1

NO.	DLOCTN	DFIELD	FLNK CLNK STATUS-TA	INLVAL
1	-1	0	00000000000000000003	0
2	-2	0	00000000000000000003	0
3	-3	0	00000000000000000003	0
4	-4	0	00000000000000000003	0
5	-5	0	00000000000000000003	0
6	-6	0	00000000000000000003	0
7	-7	0	00000000000000000003	0
8	-8	0	00000000000000000000	0
9	-9	0	00000000000000000003	0
10	-10	0	00000000000000000000	0
11	-11	0	00000000000000000000	0
12	-12	0	00000000000000000000	0
13	-13	0	00000000000000000000	0
112	2031	5	00000000000000000002	1
113	2032	5	00000000000000000001	2
114	2033	5	00000000000000000001	3

DATAB2

NO.	ADDRESS	IDX/FLD
1	0	105
2	0	205
3	0	305
4	0	405
5	0	0

DATAB3

NO.	DLOCTN	DFIELD	ATTRIB	INLVAL
1	2031	5	000010	0
2	2032	5	000010	4
3	2033	5	000010	696

DATAB4 (IMMEDIATE CONSTANTS)

NO.	CVALUE
1	0
2	500
3	1

DATAB5 (JUMP TABLE)

NO.	JMPLOC
1	2007
2	2025
3	2010
4	2013
5	2021
6	2029
7	2000
8	2030

```
*** COMPRESSION PHASE ***

IMTXT INSTR (    1) DELETED        1  ASSIGN   71 1131      0
IMTXT INSTR (    3) DELETED        1  ASSIGN 1121   71      0
IMTXT INSTR (    9) DELETED        2  ASSIGN   71   12      0
IMTXT INSTR (   17) DELETED        4  ASSIGN   71   32      0
IMTXT INSTR (   19) DELETED        4  ASSIGN   71   42      0
IMTXT INSTR (   21) DELETED        4  ASSIGN   71 1121      0
IMTXT INSTR (    4) DELETED        1  ASSIGN   21   71      0
```

```
*** BUILD NESTED REGION LISTS (NRL) ***

EXIT BLOCKS FOR SCR( 1)=      4

VARIABLE LISTS FOR SCR( 1)
  RECURS DEF VBL LIST=     31     41     51
  NON-RECURS DEF VBL LIST=  1121     91

EXIT BLOCKS FOR SCR( 2)=      6

VARIABLE LISTS FOR SCR( 2)
  RECURS DEF VBL LIST=     11     21
  NON-RECURS DEF VBL LIST=     91     31     41     51     91

EXIT BLOCKS FOR SCR( 3)=      7

VARIABLE LISTS FOR SCR( 3)
  RECURS DEF VBL LIST=      7
  NON-RECURS DEF VBL LIST=     21     11     31     61
```

```
*** SCR (LOOP) TABLE ***

SCR NO.   LEV    NRLP NDLP    NITER    INITV    TESTV    PSCR    DITV
      1     3   0022100121        4        1        4       0       1
      2     2   0022100157       49      508      696       0       4
      3     1   0022100217        0        0        0       0       0
```

```
INDEX DATA ACCESS TABLE (IDAT)

NO.,        ICTR DT2P DCLP     LBND    UPBD    SDLP CGPH           TA

   1     00000000060000100001    500     692    00503004730000000000
   2     00000000060000200001    504     696    00533005130000000000
   3     00000000150000300001    500     695    00567005430000000000
   4     00000000160000300001    500     695    00625006010000000000
   5     00000000160000400001    504     699    00673006370000000000
   6     00000000170000400001    504     699    00741007050000000000
ARRAY DECLARATION TABLE (ADT)

NO.    ARRAY NME      LWR BND    UPR BND    STATUS

  1    ARRAY              500        699    00000000000000000000
```

*** FINAL ABSTRACT REPRESENTATION (FAR) OF INTERMEDIATE TEXT ***

NO.	BLKNO	OPCODE	OPRND1	OPRND2	OPRND3
1	1	ADD	21	1131	23
2	1	ASSIGN	11	23	0
3	1	ASSIGN	31	13	0
4	1	ASSIGN	61	1131	0
5	1	JUMP	7	0	1
6	2	CMP	91	12	22
7	2	JUMP	3	91	2
8	2	JUMP	7	0	3
9	3	ASSIGN	31	11	0
10	3	ASSIGN	41	21	0
11	3	ASSIGN	51	33	0
12	3	JUMP	7	0	4
13	4	ASSIGN	1121	32	0
14	4	ASSIGN	42	52	0
15	4	ASSIGN	62	1121	0
16	4	CMP	91	51	1131
17	4	JUMP	2	91	2
18	4	JUMP	7	0	5
19	5	ADD	31	31	33
20	5	ADD	41	41	33
21	5	ADD	51	51	33
22	5	JUMP	7	0	4
23	6	ADD	11	11	61
24	6	ADD	21	21	61
25	6	CMP	91	21	1141
26	6	JUMP	3	91	1
27	6	JUMP	7	0	6
28	7	JUMP	4	31	7
29	7	JUMP	7	0	8
30	8	HLT	1	0	0

```
*** GENERATED PL/1 PROGRAM ***
   1     MAIN: PROCEDURE OPTIONS(MAIN);
   2     DCL (IXR1,IXR2,IXR3,IXR4,IXR5,IXR6)  DEC FIXED(7);
   3     DCL 1 RARX,
                2 RA DEC FIXED(13),
                2 RX DEC FIXED(13),
                2 FILLAX CHAR(2);
   5     DCL RAX CHAR(16) BASED(PRARX);
   5     DCL T1 INIT(0) DEC FIXED(13);
   6     DCL NOWPE INIT(4) DEC FIXED(13);
   7     DCL LNTRY INIT(696) DEC FIXED(13);
   8     DCL ARRAY(500:699)  DEC FIXED(13);


   9                 PRARX=ADDR(RARX);
  10                 IXR2=NOWPE+500;
  11                 IXR1=500;
  12                 IXR3=0;
  13                 IXR6=NOWPE;
  14     SORT1:      IF ARRAY(IXR1) <= ARRAY(IXR2) THEN GOTO SORT6;
  15                 ELSE DO;
  16                   IXR3=IXR1;
  17                   IXR4=IXR2;
  18                   IXR5=1;
  19                   GOTO SORT2;
  20                   END;
  21     SORT6:      IXR1=IXR1+IXR6;
  22                 IXR2=IXR2+IXR6;
  23                 IF IXR2 <= LNTRY THEN GOTO SORT1;
  24                 ELSE DO;
  25                   IF IXR3 > 0 THEN GOTO SORT;
  26                   ELSE DO;
  27                     RETURN;
  28                     END;
  29                   END;
  30     SORT2:      T1=ARRAY(IXR3);
  31                 ARRAY(IXR3)=ARRAY(IXR4);
  32                 ARRAY(IXR4)=T1;
  33                 IF IXR5 = NOWPE THEN GOTO SORT6;
  34                 ELSE DO;
  35                   IXR3=IXR3+1;
  36                   IXR4=IXR4+1;
  37                   IXR5=IXR5+1;
  38                   GOTO SORT2;
  39                   END;
  40                 END MAIN;
```

*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

```
*
*  TEST CASE NUMBER 1.
*
*  KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*  FIRST 500 PRIMES, P. 144-145.
*
L          EQU  500
PRINTER    EQU  18
PRIME      EQU  -1
BUF0       EQU  2000
BUF1       EQU  BUF0+25
X1         EQU  1$2
X2         EQU  3$4
X3         EQU  5$5
           ORIG 3000
START      IOC  0(PRINTER)
           LD1  LIT1
           LD2  LIT2
2H         INC1 1
           ST2  PRIME+L,1
           J1Z  2F
4H         INC2 2
           ENT3 2
6H         ENTA 0
           ENTX 0,2
           DIV  PRIME,3
           JXZ  4B
           CMPA PRIME,3
           INC3 1
           JG   6B
           JMP  2B
2H         OUT  TITLE(PRINTER)
           ENT4 BUF1+10
           ENT5 -50
2H         INC5 L+1
4H         LDA  PRIME,5
           CHAR
           STX  0,4(1$4)
           DEC4 1
           DEC5 50
           J5P  4B
           OUT  0,4(PRINTER)
           LD4  24,4
           J5N  2B
           HLT
* INITIAL CONTESTS OF TABLES AND BUFFER
           ORIG PRIME+1
           CON  2
           ORIG BUF0-5
TITLE      ALF  FIRST
           ALF   FIVE
           ALF   HUND
           ALF  RED P
           ALF  RIMES
           ORIG BUF0+24
           CON  BUF1+10
           ORIG BUF1+24
           CON  BUF0+10
LIT1       CON  1-L
LIT2       CON  3
           END  START
```

```
*** GENERATED PL/1 PROGRAM ***
   1    MAIN: PROCEDURE OPTIONS(MAIN);
   2    DCL (IXR1,IXR2,IXR3,IXR4,IXR5)  DEC FIXED(7);
   3    DCL 1 RARX,
                2 RA DEC FIXED(13),
                2 RX DEC FIXED(13),
                2 FILLAX CHAR(2);
   5    DCL RAX CHAR(16) BASED(PRARX);
   5    DCL (TEST,LT INIT(1)) BIT(8);
   6    DCL (EQ INIT(2),GT INIT(4)) BIT(8);
   7    DCL (LE INIT(3),NE INIT(5),GE INIT(6)) BIT(8);
   8    DCL ARRAY01(0:951)  DEC FIXED(13);
   9    DCL ARRAY02(1935:2058)  DEC FIXED(13);
  10    DCL 1 ARRAY022(1935:2056) UNAL BASED(PARRAY02),
               2 FLD14 BIT(32),
  11           2 FILL1 CHAR(3);
  12

  13                 PRARX=ADDR(RARX);
  14                 PARRAY02=ADDR(ARRAY02);
  15                 ARRAY01(0)=2;
  16                 ARRAY02(1995)=≠FIRST≠;
  17                 ARRAY02(1996)=≠ FIVE≠;
  18                 ARRAY02(1997)=≠ HUND≠;
  19                 ARRAY02(1998)=≠RED P≠;
  20                 ARRAY02(1999)=≠RIMES≠;
  21                 ARRAY02(2024)=2035;
  22                 ARRAY02(2049)=2010;
  23                 ARRAY02(2050)=-499;
  24                 ARRAY02(2051)=3;
  25                 IXR1=ARRAY02(2050);
  26                 IXR2=ARRAY02(2051);
  27    L3003:       IXR1=IXR1+1;
  28                 ARRAY01(499+IXR1)=IXR2;
  29                 IF IXR1 = 0 THEN DO;
  30    /*  THE FOLLOWING INSTR NOT HANDLED */
  31                   OUT ARRAY02(1995),;
  32                   IXR4=2035;
  33                   IXR5=-50;
  34                   END;
  35                 ELSE L3006: DO;
  36                   IXR2=IXR2+2;
  37                   IXR3=2;
  38      L3008:       RA=IXR2/ARRAY01(-1+IXR3);
  39                   RX=MOD(IXR2,ARRAY01(-1+IXR3));
  40                   IF RX = 0 THEN GOTO L3006;
  41                   ELSE DO;
  42                     IF RA<ARRAY01(-1+IXR3) THEN TEST=LT;
  43                        ELSE IF RA=ARRAY01(-1+IXR3) THEN TEST=EQ;
  44                           ELSE TEST=GT;
  45                   IXR3=IXR3+1;
  46                   IF TEST=GT THEN GOTO L3008;
  47                   ELSE GOTO L3003;
  48                   END;
  49                 END L3006;
  50    L3019:       IXR5=IXR5+501;
  51    /* THE FOLLOWING ASSIGNMENT IMPLIES CHARACTER  CONVERSION*/
  52    L3020:       RAX=ARRAY01(-1+IXR5);
  53                 ARRAY022.FLD14(IXR4)=RX;
  54                 IXR4=IXR4-1;
  55                 IXR5=IXR5-50;
  56                 IF IXR5 > 0 THEN GOTO L3020;
  57                 ELSE DO;
  58    /*  THE FOLLOWING INSTR NOT HANDLED */
  59                   OUT ARRAY02(IXR4),;
  60                   IXR4=ARRAY02(24+IXR4);
  61                   IF IXR5 < 0 THEN GOTO L3019;
  62                   ELSE DO;
  63                     RETURN;
  64                     END;
  65                   END;
  66                 END MAIN;
```

*** EDITED PL/1 PROGRAM FOR TEST CASE 1 ***

```
 1       MAIN: PROCEDURE OPTIONS(MAIN);
 2       DCL (IXR1,IXR2,IXR3,IXR4,IXR5)  DEC FIXED(7);
 3       DCL 1 RARX,
 4              2 RA DEC FIXED(13),
 5              2 RX DEC FIXED(13),
 6              2 FILLAX CHAR(2);
 7       DCL RAX CHAR(16) BASED(PRARX);
 8       DCL (TEST,LT INIT(1)) BIT(8);
 9       DCL (EQ INIT(2),GT INIT(4)) BIT(8);
10       DCL (LE INIT(3),NE INIT(5),GE INIT(6)) BIT(8);
11       DCL ARRAY01(0:951)  DEC FIXED(13);
12*      DCL ARRAY02(1935:2058)  CHAR(5) INIT((124)(5) ≠ ≠);
13       DCL 1 ARRAY022(1935:2058) UNAL BASED(PARRAY02),
14              2 FLD14 CHAR(4),
15              2 FILL1 CHAR(1);
16
17
18                    PRARX=ADDR(RARX);
19                    PARRAY02=ADDR(ARRAY02);
20                    ARRAY01(0)=2;
21                    ARRAY02(1995)=≠FIRST≠;
22                    ARRAY02(1996)=≠ FIVE≠;
23                    ARRAY02(1997)=≠ HUND≠;
24                    ARRAY02(1998)=≠RED P≠;
25                    ARRAY02(1999)=≠RIMES≠;
26*                   ARRAY02(2024)=≠2035≠;
27*                   ARRAY02(2049)=≠2010≠;
28*                   ARRAY02(2050)=≠-499≠;
29*                   ARRAY02(2051)=≠3≠;
30                    IXR1=ARRAY02(2050);
31                    IXR2=ARRAY02(2051);
32       L3003:       IXR1=IXR1+1;
33                    ARRAY01(499+IXR1)=IXR2;
34                    IF IXR1 = 0 THEN DO;
35                      PUT EDIT((ARRAY02(I) DO I=1995 TO 1999)) ((5)A(5));
36                      IXR4=2035;
37                      IXR5=-50;
38                      END;
39                    ELSE L3006: DO;
40                      IXR2=IXR2+2;
41                      IXR3=2;
42         L3008:       RA=IXR2/ARRAY01(-1+IXR3);
43                      RX=MOD(IXR2,ARRAY01(-1+IXR3));
44                      IF RX = 0 THEN GOTO L3006;
45                      ELSE DO;
46                        IF RA<ARRAY01(-1+IXR3) THEN TEST=LT;
47                           ELSE IF RA=ARRAY01(-1+IXR3) THEN TEST=EQ;
48                              ELSE TEST=GT;
49                      IXR3=IXR3+1;
50                      IF TEST=GT THEN GOTO L3008;
51                      ELSE GOTO L3003;
52                      END;
53                    END L3006;
54       L3019:       IXR5=IXR5+501;
55       L3020:       RAX=ARRAY01(-1+IXR5);
56*                   ARRAY022.FLD14(IXR4)=SUBSTR(RAX,13);
57                    IXR4=IXR4-1;
58                    IXR5=IXR5-50;
59                    IF IXR5 > 0 THEN GOTO L3020;
60                    ELSE DO;
61                      PUT SKIP EDIT((ARRAY02(I) DO I=IXR4 TO IXR4+10))
62                                  ((11)A(5));
63                      IXR4=ARRAY02(24+IXR4);
64                      IF IXR5 < 0 THEN GOTO L3019;
65                      ELSE DO;
66                        RETURN;
67                        END;
68                      END;
69                    END MAIN;
```

Results from executing edited PL/1 program:

IRST FIVE HUNDRED PRIMES

2	233	547	877	1229	1597	1993	2371	2749	3187
3	239	557	881	1231	1601	1997	2377	2753	3191
5	241	563	883	1237	1607	1999	2381	2767	3203
7	251	569	887	1249	1609	2003	2383	2777	3209
11	257	571	907	1259	1613	2011	2389	2789	3217
13	263	577	911	1277	1619	2017	2393	2791	3221
17	269	587	919	1279	1621	2027	2399	2797	3229
19	271	593	929	1283	1627	2029	2411	2801	3251
23	277	599	937	1289	1637	2039	2417	2803	3253
29	281	601	941	1291	1657	2053	2423	2819	3257
31	283	607	947	1297	1663	2063	2437	2833	3259
37	293	613	953	1301	1667	2069	2441	2837	3271
41	307	617	967	1303	1669	2081	2447	2843	3299
43	311	619	971	1307	1693	2083	2459	2851	3301
47	313	631	977	1319	1697	2087	2467	2857	3307
53	317	641	983	1321	1699	2089	2473	2861	3313
59	331	643	991	1327	1709	2099	2477	2879	3319
61	337	647	997	1361	1721	2111	2503	2887	3323
67	347	653	1009	1367	1723	2113	2521	2897	3329
71	349	659	1013	1373	1733	2129	2531	2903	3331
73	353	661	1019	1381	1741	2131	2539	2909	3343
79	359	673	1021	1399	1747	2137	2543	2917	3347
83	367	677	1031	1409	1753	2141	2549	2927	3359
89	373	683	1033	1423	1759	2143	2551	2939	3361
97	379	691	1039	1427	1777	2153	2557	2953	3371
101	383	701	1049	1429	1783	2161	2579	2957	3373
103	389	709	1051	1433	1787	2179	2591	2963	3389
107	397	719	1061	1439	1789	2203	2593	2969	3391
109	401	727	1063	1447	1801	2207	2609	2971	3407
113	409	733	1069	1451	1811	2213	2617	2999	3413
127	419	739	1087	1453	1823	2221	2621	3001	3433
131	421	743	1091	1459	1831	2237	2633	3011	3449
137	431	751	1093	1471	1847	2239	2647	3019	3457
139	433	757	1097	1481	1861	2243	2657	3023	3461
149	439	761	1103	1483	1867	2251	2659	3037	3463
151	443	769	1109	1487	1871	2267	2663	3041	3467
157	449	773	1117	1489	1873	2269	2671	3049	3469
163	457	787	1123	1493	1877	2273	2677	3061	3491
167	461	797	1129	1499	1879	2281	2683	3067	3499
173	463	809	1151	1511	1889	2287	2687	3079	3511
179	467	811	1153	1523	1901	2293	2689	3083	3517
181	479	821	1163	1531	1907	2297	2693	3089	3527
191	487	823	1171	1543	1913	2309	2699	3109	3529
193	491	827	1181	1549	1931	2311	2707	3119	3533
197	499	829	1187	1553	1933	2333	2711	3121	3539
199	503	839	1193	1559	1949	2339	2713	3137	3541
211	509	853	1201	1567	1951	2341	2719	3163	3547
223	521	857	1213	1571	1973	2347	2729	3167	3557
227	523	859	1217	1579	1979	2351	2731	3169	3559
229	541	863	1223	1583	1987	2357	2741	3181	3571

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

*
*  TEST CASE NUMBER 2.
*
*  KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*  SUM OF HARMONIC SERIES, P. 513.
*
              ORIG 100
BUF           EQU  *
              ORIG *+24
START         ENT2 0
              ENT1 3
              ENTA 20
OUTER         MUL  LIT10
              STX  CONST
              DIV  LIT2
              ENTX 2
              JMP  1F
INNER         STA  R
              ADD  R
              DECA 1
              STA  TEMP
              LDX  CONST
              ENTA 0
              DIV  TEMP
              INCA 1
              STA  TEMP
              SUB  M
              MUL  R
              SLAX 5
              ADD  S
              LDX  TEMP
1H            STA  S
              STX  M
              LDA  M
              ADD  M
              STA  TEMP
              LDA  CONST
              ADD  M
              SRAX 5
              DIV  TEMP
              JAP  INNER
              LDA  S
              CHAR
              SLAX 0,1
              SLA  1
              INCA 40
              STA  BUF,2
              STX  BUF+1,2
              INC2 3
              DEC1 1
              LDA  CONST
              JINN OUTER
              OUT  BUF(18)
              HLT
CONST         CON  0
R             CON  0
TEMP          CON  0
M             CON  0
S             CON  0
LIT2          CON  2
LIT10         CON  10
              END  START
```

```
*** GENERATED PL/1 PROGRAM ***
 1    MAIN: PROCEDURE OPTIONS(MAIN);
 2    DCL (IXR1,IXR2)  DEC FIXED(7);
 3    DCL 1 RARX,
             2 RA DEC FIXED(13),
             2 RX DEC FIXED(13),
             2 FILLAX CHAR(2);
 5    DCL RAX CHAR(16) BASED(PRARX);
 5    DCL CONST INIT(0) DEC FIXED(13);
 6    DCL R INIT(0) DEC FIXED(13);
 7    DCL TEMP INIT(0) DEC FIXED(13);
 8    DCL M INIT(0) DEC FIXED(13);
 9    DCL S INIT(0) DEC FIXED(13);
10    DCL LIT2 INIT(2) DEC FIXED(13);
11    DCL LIT10 INIT(10) DEC FIXED(13);
12    DCL BUF(100:110)  DEC FIXED(13);


13                PRARX=ADDR(RARX);
14                IXR2=0;
15                IXR1=3;
16                RA=20;
17    OUTER:      CONST=RA*LIT10;
18                RA=CONST/LIT2;
19                RX=2;
20    L146:       S=RA;
21                M=RX;
22                RA=(CONST+M)/(M+M);
23                IF RA > 0 THEN DO;
24                  R=RA;
25                  TEMP=CONST/(RA*R-1)+1;
26                  RA=(TEMP-M)*R+S;
27                  RX=TEMP;
28                  GOTO L146;
29                  END;
30                ELSE DO;
31    /* THE FOLLOWING ASSIGNMENT IMPLIES CHARACTER  CONVERSION*/
32                  RAX=3;
33    /*  THE FOLLOWING INSTR NOT HANDLED */
34                  SHIFTL RAX,RAX,IXR1;
35    /*  THE FOLLOWING INSTR NOT HANDLED */
36                  SHIFTL RA,RA,1;
37                  BUF(100+IXR2)=RA+40;
38                  BUF(101+IXR2)=RX;
39                  IXR2=IXR2+3;
40                  IXR1=IXR1-1;
41                  RA=CONST;
42                  IF IXR1 >= 0 THEN GOTO OUTER;
43                  ELSE DO;
44    /*  THE FOLLOWING INSTR NOT HANDLED */
45                    OUT BUF(100),;
46                    RETURN;
47                    END;
48                  END;
49                END MAIN;
```

```
*** EDITED PL/1 PROGRAM FOR TEST CASE  2  ***

  1      MAIN: PROCEDURE OPTIONS(MAIN);
  2      DCL (IXR1,IXR2)  DEC FIXED(7);
  3      DCL 1 RARX,
  4            2 RA DEC FIXED(13),
  5            2 RX DEC FIXED(13),
  6            2 FILLAX CHAR(2);
  7      DCL RAX CHAR(16) BASED(PRARX);
  8      DCL CONST INIT(0) DEC FIXED(13);
  9      DCL R INIT(0) DEC FIXED(13);
 10      DCL TEMP INIT(0) DEC FIXED(13);
 11      DCL M INIT(0) DEC FIXED(13);
 12      DCL S INIT(0) DEC FIXED(13);
 13      DCL LIT2 INIT(2) DEC FIXED(13);
 14      DCL LIT10 INIT(10) DEC FIXED(13);
 15*     DCL BUF(100:110) CHAR(5) INIT((11) (5) ≠ ≠);
 16
 17
 18                  PRARX=ADDR(RARX);
 19                  IXR2=0;
 20                  IXR1=3;
 21                  RA=20;
 22      OUTER:      CONST=RA*LIT10;
 23                  RA=CONST/LIT2;
 24                  RX=2;
 25      L146:       S=RA;
 26                  M=RX;
 27                  RA=(CONST+M)/(M+M);
 28                  IF RA > 0 THEN DO;
 29                    R=RA;
 30                    TEMP=CONST/(RA+R-1)+1;
 31                    RA=(TEMP-M)*R+S;
 32                    RX=TEMP;
 33                    GOTO L146;
 34                    END;
 35                  ELSE DO;
 36                    RAX=S;
 37*                   RAX=SUBSTR(RAX,1+IXR1);
 38*                   BUF(100+IXR2)=SUBSTR(RAX,8,4)↑↑≠.≠;
 39*                   BUF(101+IXR2)=SUBSTR(RAX,12);
 40                    IXR2=IXR2+3;
 41                    IXR1=IXR1-1;
 42                    RA=CONST;
 43                    IF IXR1 >= 0 THEN GOTO OUTER;
 44                    ELSE DO;
 45                    PUT EDIT (BUF) (12 A(5));
 46                      RETURN;
 47                      END;
 48                    END;
 49                  END MAIN;
```

Results from editing edited PL/1 program:

6.16	8.449	10.7509	13.05363

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION CARDS

*
*   TEST CASE NUMBER 3.
*
*   KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*   FAREY SERIES, P. 514.
*
X          EQU  1
Y          EQU  100
N          EQU  7
           ORIG 200
FAREY      ENT1 N
           ENTA 0
           STA  X
           ENTX 1
           STX  Y
           STX  X+1
           ST1  Y+1
           ENT2 2
1H         LDX  Y-2,2
           INCX 0,1
           ENTA 0
           DIV  Y-1,2
           STA  TEMP
           MUL  Y-1,2
           SLAX 5
           SUB  Y-2,2
           STA  Y,2
           LDA  TEMP
           MUL  X-1,2
           SLAX 5
           SUB  X-2,2
           STA  X,2
           CMPA Y,2
           INC2 1
           JL   1B
           HLT
TEMP       CON  0
           END  FAREY
```

```
*** GENERATED PL/I PROGRAM ***
   1    MAIN: PROCEDURE OPTIONS(MAIN);
   2    DCL (IXR1,IXR2) DEC FIXED(2);
   3    DCL 1 RARX,
                 2 RA DEC FIXED(13),
                 2 RX DEC FIXED(13),
                 2 FILLRX CHAR(2);
   5    DCL RAX CHAR(16) BASED(PRARX);
   5    DCL (TEST,LT INIT(1)) BIT(8);
   6    DCL (EQ INIT(2),GT INIT(4)) BIT(8);
   7    DCL (LE INIT(3),NE INIT(5),GE INIT(6)) BIT(8);
   8    DCL TEMP INIT(0) DEC FIXED(13);
   9    DCL ARRAY01(0:199) DEC FIXED(13);


  10                    PRARX=ADDR(RARX);
  11                    IXR1=7;
  12                    ARRAY01(1)=0;
  13                    ARRAY01(100)=1;
  14                    ARRAY01(2)=1;
  15                    ARRAY01(101)=IXR1;
  16                    IXR2=2;
  17    L208:           TEMP=(ARRAY01(98+IXR2)+IXR1)/ARRAY01(99+IXR2);
  18                    ARRAY01(100+IXR2)=TEMP*ARRAY01(99+IXR2)-ARRAY01(98+IXR2);
  19                    RA=TEMP*ARRAY01(IXR2)-ARRAY01(-1+IXR2);
  20                    ARRAY01(1+IXR2)=RA;
  21                    IF RA<ARRAY01(100+IXR2) THEN TEST=LT;
  22                       ELSE IF RA=ARRAY01(100+IXR2) THEN TEST=EQ;
  23                          ELSE TEST=GT;
  24                    IXR2=IXR2+1;
  25                    IF TEST=LT THEN GOTO L208;
  26                    ELSE DO;
  27                      RETURN;
  28                      END;
  29                    END MAIN;
```

```
*** EDITED PL/1 PROGRAM FOR TEST CASE  3  ***

 1       MAIN: PROCEDURE OPTIONS(MAIN);
 2       DCL (IXR1,IXR2)  DEC FIXED(7);
 3       DCL 1 RARX,
 4             2 RA DEC FIXED(13),
 5             2 RX DEC FIXED(13),
 6             2 FILLAX CHAR(2);
 7       DCL RAX CHAR(16) BASED(PRARX);
 8       DCL (TEST,LT INIT(1)) BIT(8);
 9       DCL (EQ INIT(2),GT INIT(4)) BIT(8);
10       DCL (LE INIT(3),NE INIT(5),GE INIT(6)) BIT(8);
11       DCL TEMP INIT(0) DEC FIXED(13);
12       DCL ARRAY01(0:199)  DEC FIXED(13);
13
14
15                   PRARX=ADDR(RARX);
16                   IXR1=7;
17                   ARRAY01(1)=0;
18                   ARRAY01(100)=1;
19                   ARRAY01(2)=1;
20                   ARRAY01(101)=IXR1;
21                   IXR2=2;
22       L208:       TEMP=(ARRAY01(98+IXR2)+IXR1)/ARRAY01(99+IXR2);
23                   ARRAY01(100+IXR2)=TEMP*ARRAY01(99+IXR2)-ARRAY01(98+IXR2);
24                   RA=TEMP*ARRAY01(IXR2)-ARRAY01(-1+IXR2);
25                   ARRAY01(1+IXR2)=RA;
26                   IF RA<ARRAY01(100+IXR2) THEN TEST=LT;
27                      ELSE IF RA=ARRAY01(100+IXR2) THEN TEST=EQ;
28                         ELSE TEST=GT;
29                   IXR2=IXR2+1;
30                   IF TEST=LT THEN GOTO L208;
31                   ELSE DO;
32                     PUT LIST(≠ FAREY SERIES, N=7≠);
33                     PUT SKIP(2) EDIT((ARRAY01(I),≠/≠,ARRAY01(I+99)
34                         DO I=1 TO IXR2)) (10(F(2),A(1),F(2),X(3)),SKIP(2));
35                     RETURN;
36                     END;
37                   END MAIN;
```

Results from executing edited PL/1 program:

```
FAREY SERIES, N=7

0/ 1    1/ 7    1/ 6    1/ 5    1/ 4    2/ 7    1/ 3    2/ 5    3/ 7    1/ 2

4/ 7    3/ 5    2/ 3    5/ 7    3/ 4    4/ 5    5/ 6    6/ 7    1/ 1
```

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

*
*   TEST CASE NUMBER 4.
*
*   KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*   MATRIX SADDLE POINT PROBLEM, SOLUTION 2, P. 506.
*
A10         EQU  1008
CMAX        EQU  1000
PHASE1      ENT1 8
3H          ENT2 64,1
            JMP  2F
1H          CMPX A10,2
            JGE  *+2
2H          LDX  A10,2
            DEC2 8
            J2P  1B
            STX  CMAX+8,2
            J2Z  1F
            CMPA A10,2
            JLE  *+2
1H          LDA  A10,2
            DEC1 1
            J1P  3B
PHASE2      ENT3 64
3H          ENT2 8,3
            ENT4 8
1H          CMPA A10,2
            JG   NO
            JL   2F
            CMPA CMAX,4
            JNE  2F
            ENT1 A10,2
2H          DEC4 1
            DEC2 1
            J4P  1B
            HLT
NO          DEC3 8
            ENT1 0
            J3P  3B
            HLT
            END  PHASE1
```

```
*** GENERATED PL/1 PROGRAM ***
 1    MAIN: PROCEDURE OPTIONS(MAIN);
 2    DCL (IXR1,IXR2,IXR3,IXR4)  DEC FIXED(7);
 3    DCL 1 RARX,
              2 RA DEC FIXED(13),
              2 RX DEC FIXED(13),
              2 FILLAX CHAR(2);
 5    DCL RAX CHAR(16) BASED(PRARX);
 5    DCL ARRAY01(1001:1080)  DEC FIXED(13);


 6                  PRARX=ADDR(RARX);
 7                  IXR1=8;
 8    L1:           IXR2=64+IXR1;
 9    L5:           RX=ARRAY01(1008+IXR2);
10    L6:           IXR2=IXR2-8;
11                  IF IXR2 > 0 THEN DO;
12                    IF RX >= ARRAY01(1008+IXR2) THEN GOTO L6;
13                    ELSE GOTO L5;
14                    END;
15                  ELSE DO;
16                    ARRAY01(1008+IXR2)=RX;
17                    IF IXR2 = 0 THEN GOTO L12;
18                    ELSE DO;
19                      IF RA <= ARRAY01(1008+IXR2) THEN GOTO L13;
20                      END;
21                    END;
22    L12:          RA=ARRAY01(1008+IXR2);
23    L13:          IXR1=IXR1-1;
24                  IF IXR1 > 0 THEN GOTO L1;
25                  ELSE DO;
26                    IXR3=64;
27                    END;
28    L16:          IXR2=8+IXR3;
29                  IXR4=8;
30    L18:          IF RA > ARRAY01(1008+IXR2) THEN DO;
31                    IXR3=IXR3-8;
32                    IXR1=0;
33                    IF IXR3 > 0 THEN GOTO L16;
34                    ELSE DO;
35                      RETURN;
36                      END;
37                    END;
38                  ELSE IF RA < ARRAY01(1008+IXR2) THEN GOTO L24;
39                  ELSE DO;
40                    IF RA ¬= ARRAY01(1000+IXR4) THEN GOTO L24;
41                    ELSE DO;
42                      IXR1=1008+IXR2;
43                      END;
44                    END;
45    L24:          IXR4=IXR4-1;
46                  IXR2=IXR2-1;
47                  IF IXR4 > 0 THEN GOTO L18;
48                  ELSE DO;
49                    RETURN;
50                    END;
51                  END MAIN;
```

```
*** EDITED PL/1 PROGRAM FOR TEST CASE  4  ***

  1      MAIN: PROCEDURE OPTIONS(MAIN);
  2      DCL (IXR1,IXR2,IXR3,IXR4)  DEC FIXED(7);
  3      DCL 1 RARX,
  4            2 RA DEC FIXED(13),
  5            2 RX DEC FIXED(13),
  6            2 FILLAX CHAR(2);
  7      DCL RAX CHAR(16) BASED(PRARX);
  8      DCL ARRAY01(1001:1080)  DEC FIXED(13);
  9
 10
 11                 GET LIST(ARRAY01); /* INSERTED /*
 12                 /* FOLLOWING STATEMENT IS INSERTED BY HAND */
 13                 PUT EDIT((ARRAY01(I) DO I=1009 TO 1080)) (SKIP,(8)F(8));
 14                 PRARX=ADDR(RARX);
 15                 IXR1=8;
 16      L1:        IXR2=64+IXR1;
 17      L5:        RX=ARRAY01(1008+IXR2);
 18      L6:        IXR2=IXR2-8;
 19                 IF IXR2 > 0 THEN DO;
 20                   IF RX >= ARRAY01(1008+IXR2) THEN GOTO L6;
 21                   ELSE GOTO L5;
 22                   END;
 23                 ELSE DO;
 24                   ARRAY01(1008+IXR2)=RX;
 25                   IF IXR2 = 0 THEN GOTO L12;
 26                   ELSE DO;
 27                     IF RA <= ARRAY01(1009+IXR2) THEN GOTO L13;
 28                     END;
 29                   END;
 30      L12:       RA=ARRAY01(1008+IXR2);
 31      L13:       IXR1=IXR1-1;
 32                 IF IXR1 > 0 THEN GOTO L1;
 33                 ELSE DO;
 34                 PUT SKIP(2);
 35                 PUT DATA((ARRAY01(I) DO I=1001 TO 1008));
 36                 PUT SKIP(2);
 37                   IXR3=64;
 38                   END;
 39      L16:       IXR2=8+IXR3;
 40                 IXR4=8;
 41      L18:       IF RA > ARRAY01(1008+IXR2) THEN DO;
 42                   IXR3=IXR3-8;
 43                   IXR1=0;
 44                   IF IXR3 > 0 THEN GOTO L16;
 45                   ELSE DO;
 46                     RETURN;
 47                     END;
 48                   END;
 49                 ELSE IF RA < ARRAY01(1008+IXR2) THEN GOTO L24;
 50                 ELSE DO;
 51                   IF RA ¬= ARRAY01(1000+IXR4) THEN GOTO L24;
 52                   ELSE DO;
 53                     IXR1=1008+IXR2;
 54                     END;
 55                   END;
 56      L24:       IXR4=IXR4-1;
 57                 IXR2=IXR2-1;
 58                 IF IXR4 > 0 THEN GOTO L18;
 59                 ELSE DO;
 60                   PUT DATA(IXR1,IXR2,IXR3,IXR4,RA) SKIP(2); /* INSERTED */
 61                   RETURN;
 62                   END;
 63                 END MAIN;
```

Results from executing edited PL/1 program:

```
     8        10        20        46        10      1000        92        96
     5         3         8       900         1        44        67        98
    18        67        93       102        80       106        57        [illegible]
   110       115       180       162       100       200       250       275
    20        27        26        22        40        99        64        93
    36        41        42        86        45        99        91         4
    35         3         9         6        93         7        11        [illegible]
  1050      2000        31         4        62         8        22        33
    20        60        90        14        66         1         9        17

ARRAY01(1001)=          1050                    ARRAY01(1002)=
ARRAY01(1003)=           180                    ARRAY01(1004)=
ARRAY01(1005)=           [illegible]            ARRAY01([illegible])=
ARRAY01(1007)=           250                    ARRAY01([illegible])=

IXP1=       1037          IXP2=         74          IXP3=         74
RAPX.RA=          100:
```

```
*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

*
*   TEST CASE NUMBER 5.
*
*   KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*   JOSEPHUS PROBLEM, P. 516.
*
N            EQU  24
M            EQU  11
Y            EQU  1
             ORIG 100
JOE          ENT1 N-1
             STZ  Y,1
             ST1  Y-1,1
             DEC1 1
             J1P  *-2
             ENT1 0
             ENTA 1
UT           ENT2 M-2
             LD1  Y,1
             DEC2 1
             J2P  *-2
             LD2  Y,1
             LD3  Y,2
             CHAR
             STX  Y,2(4:5)
             NUM
             INCA 1
             ST3  Y,1
             ENT1 0,3
             CMPA LITN
             JL   16
             CHAR
             STX  Y,1(4:5)
             OUT  Y(18)
             HLT
LITN         CON  24
             END  JOE
```

```
*** GENERATED PL/1 PROGRAM ***
  1    MAIN: PROCEDURE OPTIONS(MAIN);
  2    DCL (IXR1,IXR2,IXR3)  DEC FIXED(7);
  3    DCL 1 RARX,
             2 RA DEC FIXED(13),
             2 RX DEC FIXED(13),
             2 FILLAX CHAR(2);
  5    DCL RAX CHAR(16) BASED(PRARX);
  5    DCL LITN INIT(24) DEC FIXED(13);
  6    DCL Y(1:99)  DEC FIXED(13);
  7    DCL 1 Y2(1:99) UNAL BASED(PY),
  8         2 FILL1 CHAR(3),
  9         2 FLD45 BIT(16),
 10         2 FILL2 CHAR(2);


 11                PRARX=ADDR(RARX);
 12                PY=ADDR(Y);
 13                IXR1=23;
 14                Y(1+IXR1)=0;
 15    L102:       Y(IXR1)=IXR1;
 16                IXR1=IXR1-1;
 17                IF IXR1 > 0 THEN GOTO L102;
 18                ELSE DO;
 19                  IXR1=0;
 20                  RA=1;
 21                  END;
 22    L107:       IXR2=9;
 23    L108:       IXR1=Y(1+IXR1);
 24                IXR2=IXR2-1;
 25                IF IXR2 > 0 THEN GOTO L108;
 26                ELSE DO;
 27                  IXR2=Y(1+IXR1);
 28                  IXR3=Y(1+IXR2);
 29    /* THE FOLLOWING ASSIGNMENT IMPLIES CHARACTER  CONVERSION*/
 30                  RAX=RA;
 31                  Y2.FLD45(1+IXR2)=RX;
 32    /* THE FOLLOWING ASSIGNMENT IMPLIES CHARACTER  CONVERSION*/
 33                  RA=RAX;
 34                  RA=RA+1;
 35                  Y(1+IXR1)=IXR3;
 36                  IXR1=IXR3;
 37                  IF RA < LITN THEN GOTO L107;
 38                  ELSE DO;
 39    /* THE FOLLOWING ASSIGNMENT IMPLIES CHARACTER  CONVERSION*/
 40                    RAX=RA;
 41                    Y2.FLD45(1+IXR1)=RX;
 42    /*  THE FOLLOWING INSTR NOT HANDLED */
 43                    OUT Y(1),;
 44                    RETURN;
 45                    END;
 46                  END;
 47                END MAIN;
```

```
*** EDITED PL/1 PROGRAM FOR TEST CASE  5  ***

 1       MAIN: PROCEDURE OPTIONS(MAIN);
 2       DCL (IXR1,IXR2,IXR3)  DEC FIXED(7);
 3       DCL 1 RARX,
 4             2 RA DEC FIXED(13),
 5             2 RX DEC FIXED(13),
 6             2 FILLAX CHAR(2);
 7       DCL RAX CHAR(16) BASED(PRARX);
 8       DCL LITN INIT(24) DEC FIXED(13);
 9       DCL Y(1:99)  DEC FIXED(13);
10       DCL 1 Y2(1:99) UNAL BASED(PY),
11             2 FILL1 CHAR(3),
12*            2 FLD45 CHAR(2),
13             2 FILL2 CHAR(2);
14
15
16                  PRARX=ADDR(RARX);
17                  PY=ADDR(Y);
18                  IXR1=23;
19                  Y(1+IXR1)=0;
20       L102:      Y(IXR1)=IXR1;
21                  IXR1=IXR1-1;
22                  IF IXR1 > 0 THEN GOTO L102;
23                  ELSE DO;
24                    IXR1=0;
25                    RA=1;
26                    END;
27       L107:      IXR2=9;
28       L108:      IXR1=Y(1+IXR1);
29                  IXR2=IXR2-1;
30                  IF IXR2 > 0 THEN GOTO L108;
31                  ELSE DO;
32                    IXR2=Y(1+IXR1);
33                    IXR3=Y(1+IXR2);
34                    RAX=RA;
35*                   Y2.FLD45(1+IXR2)=SUBSTR(RAX,15,2);
36                    RA=RAX;
37                    RA=RA+1;
38                    Y(1+IXR1)=IXR3;
39                    IXR1=IXR3;
40                    IF RA < LITN THEN GOTO L107;
41                    ELSE DO;
42                      RAX=RA;
43*                     Y2.FLD45(1+IXR1)=SUBSTR(RAX,15,2);
44                      PUT EDIT((Y2.FLD45(I) DO I=1 TO 24))
45                              ((24)A(3));
46                      RETURN;
47                      END;
48                    END;
49                  END MAIN;
```

Results from executing edited PL/1 program:

15 12 8 16 11 23 21 3 5 1 17 10 7 24 19 20
18 9 14 4 2 13 6

*** INPUT TO MIX PREPROCESSOR ***, OPTION=CARDS

```
*
*  TEST CASE NUMBER 6.
*
*  KNUTH, FUNDAMENTAL ALGORITHMS, VOLUME 1, ADDISON WESLEY, 1969.
*  POLYNOMIAL ADDITION, P.275.
*
*
* INITIALIZE WORK AREA AVAIL CHAIN
*
BEGIN       ENT2 FREECELLS
NEXTCELL    STZ  0,2
            CMP2 LASTCELL
            JGE  ADDPOLY
            INC2 2
            ST2  -1,2(4:5)
            JMP  NEXTCELL
*
* INITIALIZE INPUT REGISTERS AND BEGIN
*
ADDPOLY     STZ  1,2            ZERO LAST FREE CELL LINK
            ENT1 P
            ENT2 Q
            ENTA FREECELLS
            STA  AVAIL
*
*
LINK        EQU  4:5
ABC         EQU  0:3
ADD         ENT3 0,2
0H          LD1  1,1(LINK)
SW1         LDA  1,1
1H          LD2  1,3(LINK)
2H          CMPA 1,2(ABC)
            JE   3F
            JG   5F
            ENT3 0,2
            JMP  1B
3H          JAN  DONE
SW2         LDA  0,1
            ADD  0,2
            STA  0,2
            JANZ 6F
            ENT6 0,2
            LD2  1,2(LINK)
            LDX  AVAIL
            STX  1,6(LINK)
            ST6  AVAIL
            ST2  1,3(LINK)
            JMP  0B
6H          ENT3 0,2
            JMP  0B
5H          LD6  AVAIL
            J6Z  OVERFLOW
            LDX  1,6(LINK)
            STX  AVAIL
            STA  1,6
SW3         LDA  0,1
            STA  0,6
            ST2  1,6(LINK)
            ST6  1,3(LINK)
            ENT3 0,6
            JMP  0B
DONE        HLT
OVERFLOW    HLT
*
* INITIALIZED WORK AREA FOR POLYNOMIAL P
*
CELLS       CON  1
            CON  1(1:1),0(2:2),0(3:3),*+1(4:5)
            CON  1
            CON  0(1:1),1(2:2),0(3:3),*+1(4:5)
```

```
            CON  1
            CON  0(1:1),0(2:2),1(3:3),*+1(4:5)
P           CON  0
            CON  -1(0:3),CELLS(4:5)
*
* INITIALIZED WORK AREA FOR POLYNOMIAL Q.
*
Q1          CON  1
            CON  2(1:1),0(2:2),0(3:3),*+1(4:5)
            CON  -2
            CON  0(1:1),1(2:2),0(3:3),*+1(4:5)
            CON  -1
            CON  0(1:1),0(2:2),1(3:3),*+1(4:5)
Q           CON  0
            CON  -1(0:3),Q1(4:5)
FREECELLS   CON  0
            ORIG CELLS+500
LASTCELL    CON  CELLS+498
AVAIL       CON  0
            END  BEGIN
```

```
*** GENERATED PL/1 PROGRAM ***
  1    MAIN: PROCEDURE OPTIONS(MAIN);
  2    DCL (IXR1,IXR2,IXR3,IXR6)  DEC FIXED(7);
  3    DCL 1 RARX,
             2 RA DEC FIXED(13),
             2 RX DEC FIXED(13),
             2 FILLAX CHAR(2);
  5    DCL RAX CHAR(16) BASED(PRARX);
  5    DCL LASTCELL INIT(546) DEC FIXED(13);
  6    DCL AVAIL INIT(0) DEC FIXED(13);
  7    DCL CELLS(48:546)  DEC FIXED(13);
  8    DCL 1 CELLS2(48:546) UNAL BASED(PCELLS),
  9          2 FLD03 BIN FIXED(23),
 10          2 FLD45 BIT(16),
 11          2 FILL1 CHAR(2);
 12    DCL 1 CELLS3(48:546) UNAL BASED(PCELLS),
 13          2 FLD11 BIT(8),
 14          2 FLD22 BIT(8),
 15          2 FLD33 BIT(8),
 16          2 FILL2 CHAR(4);


 17                 PRARX=ADDR(RARX);
 18                 PCELLS=ADDR(CELLS);
 19                 CELLS(48)=1;
 20                 CELLS3.FLD11(49)=1;
 21                 CELLS3.FLD22(49)=0;
 22                 CELLS3.FLD33(49)=0;
 23                 CELLS2.FLD45(49)=50;
 24                 CELLS(50)=1;
 25                 CELLS3.FLD11(51)=0;
 26                 CELLS3.FLD22(51)=1;
 27                 CELLS3.FLD33(51)=0;
 28                 CELLS2.FLD45(51)=52;
 29                 CELLS(52)=1;
 30                 CELLS3.FLD11(53)=0;
 31                 CELLS3.FLD22(53)=0;
 32                 CELLS3.FLD33(53)=1;
 33                 CELLS2.FLD45(53)=54;
 34                 CELLS(54)=0;
 35                 CELLS2.FLD03(55)=-1;
 36                 CELLS2.FLD45(55)=48;
 37                 CELLS(56)=1;
 38                 CELLS3.FLD11(57)=2;
 39                 CELLS3.FLD22(57)=0;
 40                 CELLS3.FLD33(57)=0;
 41                 CELLS2.FLD45(57)=58;
 42                 CELLS(58)=-2;
 43                 CELLS3.FLD11(59)=0;
 44                 CELLS3.FLD22(59)=1;
 45                 CELLS3.FLD33(59)=0;
 46                 CELLS2.FLD45(59)=60;
 47                 CELLS(60)=-1;
 48                 CELLS3.FLD11(61)=0;
 49                 CELLS3.FLD22(61)=0;
 50                 CELLS3.FLD33(61)=1;
 51                 CELLS2.FLD45(61)=62;
 52                 CELLS(62)=0;
 53                 CELLS2.FLD03(63)=-1;
 54                 CELLS2.FLD45(63)=56;
 55                 CELLS(64)=0;
 56                 IXR2=64;
 57    NEXTCELL:    CELLS(IXR2)=0;
 58                 IF IXR2 >= LASTCELL THEN DO;
 59                   CELLS(1+IXR2)=0;
 60                   IXR1=54;
 61                   AVAIL=64;
 62                   IXR3=62;
 63                   END;
 64                 ELSE DO;
 65                   IXR2=IXR2+2;
 66                   CELLS2.FLD45(-1+IXR2)=IXR2;
 67                   GOTO NEXTCELL;
 68                   END;
```

```
 69   L13:     IXR1=CELLS2.FLD45(1+IXR1);
 70            RA=CELLS(1+IXR1);
 71   L15:     IXR2=CELLS2.FLD45(1+IXR3);
 72            IF RA = CELLS2.FLD03(1+IXR2) THEN DO;
 73              IF RA < 0 THEN DO;
 74                RETURN;
 75                END;
 76              ELSE DO;
 77                RA=CELLS(IXR1)+CELLS(IXR2);
 78                CELLS(IXR2)=RA;
 79                IF RA ¬= 0 THEN DO;
 80                  IXR3=IXR2;
 81                  GOTO L13;
 82                  END;
 83                ELSE DO;
 84                  IXR6=IXR2;
 85                  IXR2=CELLS2.FLD45(1+IXR2);
 86                  CELLS2.FLD45(1+IXR6)=AVAIL;
 87                  AVAIL=IXR6;
 88                  CELLS2.FLD45(1+IXR3)=IXR2;
 89                  GOTO L13;
 90                  END;
 91                END;
 92              END;
 93            ELSE IF RA > CELLS2.FLD03(1+IXR2) THEN DO;
 94              IXR6=AVAIL;
 95              IF IXR6 = 0 THEN DO;
 96                RETURN;
 97                END;
 98              ELSE DO;
 99                AVAIL=CELLS2.FLD45(1+IXR6);
100                CELLS(1+IXR6)=RA;
101                CELLS(IXR6)=CELLS(IXR1);
102                CELLS2.FLD45(1+IXR6)=IXR2;
103                CELLS2.FLD45(1+IXR3)=IXR6;
104                IXR3=IXR6;
105                GOTO L13;
106                END;
107              END;
108            ELSE DO;
109              IXR3=IXR2;
110              GOTO L15;
111              END;
112            END MAIN;
```

*** EDITED PL/1 PROGRAM FOR TEST CASE 6 ***

```
  1      MAIN: PROCEDURE OPTIONS(MAIN);
  2      DCL (IXR1,IXR2,IXR3,IXR6)  DEC FIXED(7);
  3      DCL 1 RARX,
  4            2 RA DEC FIXED(13),
  5            2 RX DEC FIXED(13),
  6            2 FILLAX CHAR(2);
  7      DCL RAX CHAR(16) BASED(PRARX);
  8      DCL LASTCELL INIT(546) DEC FIXED(13);
  9      DCL AVAIL INIT(0) DEC FIXED(13);
 10      DCL CELLS(48:546)  DEC FIXED(13);
 11      DCL 1 CELLS2(48:546) UNAL BASED(PCELLS),
 12            2 FLD03 BIT(24),
 13*           2 FLD45 BIN FIXED(15),
 14            2 FILL1 CHAR(2);
 15      DCL 1 CELLS3(48:546) UNAL BASED(PCELLS),
 16            2 FLD11 BIT(8),
 17            2 FLD22 BIT(8),
 18            2 FLD33 BIT(8),
 19*           2 FLD45 BIN FIXED(15),
 20            2 FILL2 CHAR(2);
 21
 22*     DCL ONE BIT(8) INIT(≠00000001≠B);
 23*     DCL TWO BIT(8) INIT(≠00000010≠B);
 24
 25                  PRARX=ADDR(RARX);
 26                  PCELLS=ADDR(CELLS);
 27                  CELLS(48)=1;
 28*                 CELLS3.FLD11(49)=ONE;
 29                  CELLS3.FLD22(49)=0;
 30                  CELLS3.FLD33(49)=0;
 31                  CELLS2.FLD45(49)=50;
 32                  CELLS(50)=1;
 33                  CELLS3.FLD11(51)=0;
 34*                 CELLS3.FLD22(51)=ONE;
 35                  CELLS3.FLD33(51)=0;
 36                  CELLS2.FLD45(51)=52;
 37                  CELLS(52)=1;
 38                  CELLS3.FLD11(53)=0;
 39                  CELLS3.FLD22(53)=0;
 40*                 CELLS3.FLD33(53)=ONE;
 41                  CELLS2.FLD45(53)=54;
 42                  CELLS(54)=0;
 43*                 CELLS2.FLD03(55)=(24)≠1≠B;
 44                  CELLS2.FLD45(55)=48;
 45                  CELLS(56)=1;
 46*                 CELLS3.FLD11(57)=TWO;
 47                  CELLS3.FLD22(57)=0;
 48                  CELLS3.FLD33(57)=0;
 49                  CELLS2.FLD45(57)=58;
 50                  CELLS(58)=-2;
 51                  CELLS3.FLD11(59)=0;
 52*                 CELLS3.FLD22(59)=ONE;
 53                  CELLS3.FLD33(59)=0;
 54                  CELLS2.FLD45(59)=60;
 55                  CELLS(60)=-1;
 56                  CELLS3.FLD11(61)=0;
 57                  CELLS3.FLD22(61)=0;
 58*                 CELLS3.FLD33(61)=ONE;
 59                  CELLS2.FLD45(61)=62;
 60                  CELLS(62)=0;
 61*                 CELLS2.FLD03(63)=(24)≠1≠B;
 62                  CELLS2.FLD45(63)=56;
 63                  CELLS(64)=0;
 64                  IXR2=64;
 65     NEXTCELL:    CELLS(IXR2)=0;
 66                  IF IXR2 >= LASTCELL THEN DO;
 67                    CELLS(1+IXR2)=0;
 68                    IXR1=54;
 69                    AVAIL=64;
 70                    IXR3=62;
```

```
 71                 END;
 72              ELSE DO;
 73                 IXR2=IXR2+2;
 74                 CELLS2.FLD45(-1+IXR2)=IXR2;
 75                 GOTO NEXTCELL;
 76                 END;
 77     L13:     IXR1=CELLS2.FLD45(1+IXR1);
 78*             RA=CELLS2.FLD03(1+IXR1);
 79     L15:     IXR2=CELLS2.FLD45(1+IXR3);
 80              IF RA = CELLS2.FLD03(1+IXR2) THEN DO;
 81*               IF RA = (24)≠1≠B THEN DO;
 82         /* HAND CODED I/O TO PRINT OUT CELLS OF RESULT */
 83                   PUT LIST(≠POLYNOMIAL ADDITION≠);
 84                   PUT SKIP LIST(≠P=X+Y+Z≠);
 85                   PUT SKIP LIST(≠Q=X**2-2Y-Z≠);
 86                   LINK=56;
 87                   I=1;
 88                   DO WHILE(CELLS(LINK) ¬=0);
 89                   PUT SKIP(2) EDIT(≠TERM(≠,I,≠):≠,≠COEFFICIENT=≠,
 90                   CELLS(LINK)) (A(5),F(2),A(2),X(3),A(12),F(4));
 91                   PUT SKIP EDIT(≠A=≠,CELLS3.FLD11(LINK+1))
 92                   (X(22),A(2),F(3));
 93                   PUT SKIP EDIT(≠B=≠,CELLS3.FLD22(LINK+1))
 94                   (X(22),A(2),F(3));
 95                   PUT SKIP EDIT(≠C=≠,CELLS3.FLD33(LINK+1))
 96                   (X(22),A(2),F(3));
 97                   LINK=CELLS3.FLD45(LINK+1);
 98                   PUT SKIP EDIT(≠LINK=≠,LINK) (X(19),A(5),F(4));
 99                   I=I+1;
100                   END;
101                   PUT SKIP(2) LIST(≠(P+Q)=X**2+X-Y≠);
102                   RETURN;
103                   END;
104                ELSE DO;
105                   RA=CELLS(IXR1)+CELLS(IXR2);
106                   CELLS(IXR2)=RA;
107                   IF RA ¬= 0 THEN DO;
108                      IXR3=IXR2;
109                      GOTO L13;
110                      END;
111                   ELSE DO;
112                      IXR6=IXR2;
113                      IXR2=CELLS2.FLD45(1+IXR2);
114                      CELLS2.FLD45(1+IXR6)=AVAIL;
115                      AVAIL=IXR6;
116                      CELLS2.FLD45(1+IXR3)=IXR2;
117                      GOTO L13;
118                      END;
119                   END;
120                END;
121             ELSE IF RA > CELLS2.FLD03(1+IXR2) THEN DO;
122                IXR6=AVAIL;
123                IF IXR6 = 0 THEN DO;
124                   RETURN;
125                   END;
126                ELSE DO;
127                   AVAIL=CELLS2.FLD45(1+IXR6);
128*                  CELLS2.FLD03(1+IXR6)=CELLS2.FLD03(IXR1+1);
129                   CELLS(IXR6)=CELLS(IXR1);
130                   CELLS2.FLD45(1+IXR6)=IXR2;
131                   CELLS2.FLD45(1+IXR3)=IXR6;
132                   IXR3=IXR6;
133                   GOTO L13;
134                   END;
135                END;
136             ELSE DO;
137                IXR3=IXR2;
138                GOTO L15;
139                END;
140             END MAIN;
```

Results from executing edited PL/1 program:

```
POLYNOMIAL ADDITION
P=X+Y+Z
Q=X**2-2Y-Z

TERM( 1):   COEFFICIENT=  1
                      A=  2
                      B=  0
                      C=  0
                   LINK=  64

TERM( 2):   COEFFICIENT=  1
                      A=  1
                      B=  0
                      C=  0
                   LINK=  58

TERM( 3):   COEFFICIENT= -1
                      A=  0
                      B=  1
                      C=  0
                   LINK=  62

(P+Q)=X**2+X-Y
```

```
IEF142I - STEP WAS EXECUTED - COND CODE 0000
IEF285I   SYS1.PLILIB                                      KEPT
IEF285I   VOL SER NOS= PACK11.
IEF285I   SYS73114.T043212.RV000.ZZZPR600.LOADSET          DELETED
IEF285I   VOL SER NOS= PUCC03.
IEF285I   SYS73114.T043212.SV000.ZZZPR600.R0000492         SYSOUT
IEF285I   VOL SER NOS= PUCC03.
IEF285I   SYS73114.T043212.SV000.ZZZPR600.R0000493         SYSOUT
IEF285I   VOL SER NOS= PUCC03.
IEF285I   SYS73114.T043212.SV000.ZZZPR600.R0000494         DELETED
IEF285I   VOL SER NOS= PUCC03.
IEF373I STEP /GO       / START 73115.0357
IEF374I STEP /GO       / STOP  73115.0359 CPU    0MIN 01.36SEC M
IEF375I  JOB /ZZZPR600/ START 73115.0354
IEF376I  JOB /ZZZPR600/ STOP  73115.0359 CPU    0MIN 07.77SEC
```

APPENDIX C - TEST CASE PERFORMANCE DATA

Legend:

I - Number of instructions in test case.

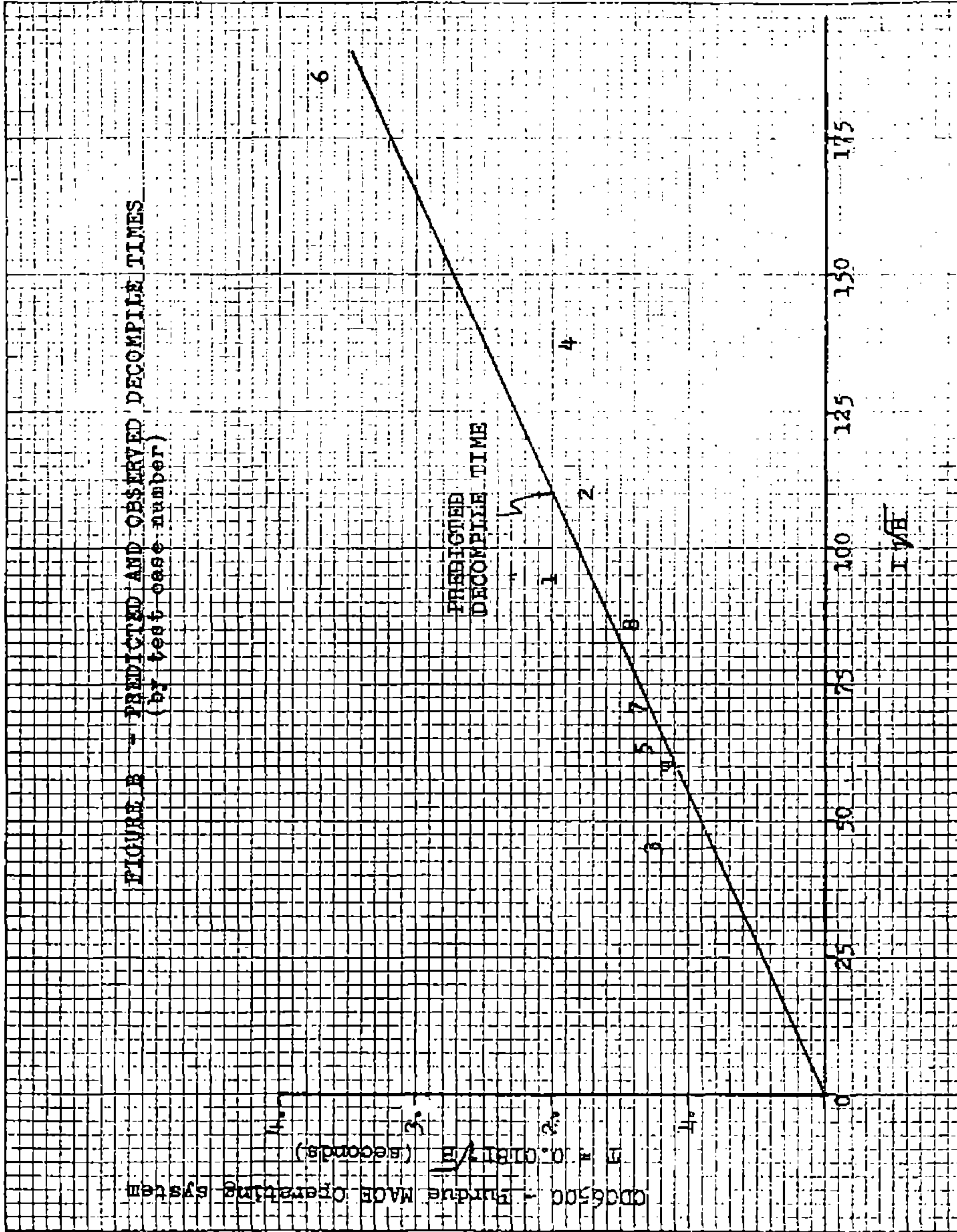
B - Number of blocks in test case.

PDT - Predicted decompile time ($.018I \sqrt{B}$) (seconds).

ODT - Observed decompile time (seconds).

Test Case Number	I	B	PDT	ODT	(ODT-PDT)
1	30	10	1.72	2.00	0.281
2	45	6	2.00	1.73	-0.267
3	26	3	0.816	1.21	0.394
4	32	18	2.46	1.90	-0.560
5	25	7	1.20	1.33	0.132
6	48	15	3.37	3.69	0.322
7	23	10	1.32	1.37	0.0522
8	31	8	1.59	1.41	-0.179
9	21	8	1.08	1.15	0.0738

PURDUE UNIVERSITY



VITA

VITA

Barron C. Housel, III was born in Oklahoma City, Oklahoma on September 14, 1940. He attended Sir Francis Drake High School in San Anselmo, California. After graduating in 1958 he attended the University of Oklahoma and earned a B.S. degree in mechanical engineering in 1963, and an M.S. in engineering sciences in 1964. After this, he was employed by the IBM Corporation, during which time he enrolled in the Department of Computer Science at Stanford University under the IBM work-study program and received an M.S. degree in 1968. In the fall of 1969, under the IBM resident study program, he enrolled in the Computer Science Department of Purdue University.